

HERRAMIENTAS PARA CONSTRUIR MUNDOS VIDA ARTIFICIAL I

• ÁNGEL DE LA ENCARNACIÓN GARCÍA BAÑOS •



Programa Editorial

Es un libro de texto sobre temas que explico habitualmente en las asignaturas Vida Artificial y Computación Evolutiva, de la carrera Ingeniería de Sistemas; compilado de una manera personal, pues lo oriento a explicar herramientas conocidas de matemáticas y computación que sirven para crear complejidad, y añado experiencias propias y de mis estudiantes.

Las herramientas que se explican en el libro son:

Realimentación: al conectar las salidas de un sistema para que afecten a sus propias entradas se producen bucles de realimentación que cambian por completo el comportamiento del sistema.

Fractales: son objetos matemáticos de muy alta complejidad aparente, pero cuyo algoritmo subyacente es muy simple.

Caos: sistemas dinámicos cuyo algoritmo es determinista y perfectamente conocido pero que, a pesar de ello, su comportamiento futuro no se puede predecir.

Leyes de potencias: sistemas que producen eventos con una distribución de probabilidad de cola gruesa, donde típicamente un 20% de los eventos contribuyen en un 80% al fenómeno bajo estudio.

Estos cuatro conceptos (realimentaciones, fractales, caos y leyes de potencia) están fuertemente asociados entre sí, y son los generadores básicos de complejidad.

Algoritmos evolutivos: si un sistema alcanza la complejidad suficiente (usando las herramientas anteriores) para ser capaz de sacar copias de sí mismo, entonces es inevitable que también aparezca la evolución.

Teoría de juegos: solo se da una introducción suficiente para entender que la cooperación entre individuos puede emerger incluso cuando las interacciones entre ellos se dan en términos competitivos.

Autómatas celulares: cuando hay una población de individuos similares que cooperan entre sí comunicándose localmente, entonces emergen fenómenos a nivel social, que son mucho más complejos todavía, como la capacidad de cómputo universal y la capacidad de autocopia.



HERRAMIENTAS PARA CONSTRUIR MUNDOS VIDA ARTIFICIAL I

ÁNGEL DE LA ENCARNACIÓN GARCÍA BAÑOS



Colección Ciencias Naturales y Exactas

García Baños, Ángel de la Encarnación
Herramientas para construir mundos. Vida artificial I /
Ángel de la Encarnación García Baños. – Cali : Programa
Editorial Universidad del Valle, 2019.
316 páginas ; ilustraciones ; 24 cm. -- (Colección ciencias
naturales y exactas)
Incluye bibliografías.
1. Caos (Ciencia) 2. Fractales 3. Algoritmos evolutivos
4. Teoría de los juegos 5. Autómatas celulares I. Tít. II. Serie.
003.857 cd 22 ed.
A1644706

CEP-Banco de la República-Biblioteca Luis Ángel Arango

Esta es una revisión del autor, no oficial y sin ISBN, lo que en cine se llama el "director cut". La razón de hacerla es que la editorial no responde a mis peticiones para corregir las erratas que hay todavía en el libro. Y como este libro se usa en varias asignaturas, es importante que los estudiantes reciban el mejor material posible.

Esta es la versión v0.5 que fue impresa el 2021-09-16.

Si está interesado en la versión oficial, la puede encontrar con ISBN 978-958-765-985-6
y DOI: 10.25100/peu.150.72
en: <http://hdl.handle.net/10893/19985>

Universidad del Valle

Programa Editorial

Título: *Herramientas para construir mundos. Vida artificial I.*

Autor: Ángel de la Encarnación García Baños

ISBN PDF: [REDACTED]

Colección: Ciencias Naturales y Exactas

Primera edición

Rector de la Universidad del Valle: Édgar Varela Barrios

Vicerrector de Investigaciones: Jaime R. Cantera Kintz

Director del Programa Editorial: Omar Díaz Saldaña

© Universidad del Valle

© Ángel de la Encarnación García Baños

Diseño y diagramación: Alejandro Soto Perez

Corrección de estilo: María Camila Cuenca O.

Este libro, o parte de él, no puede ser reproducido por ningún medio sin autorización escrita de la Universidad del Valle.

El contenido de esta obra corresponde al derecho de expresión del autor y no compromete el pensamiento institucional de la Universidad del Valle, ni genera responsabilidad frente a terceros. El autor es el responsable del respeto a los derechos de autor y del material contenido en la publicación, razón por la cual la Universidad no puede asumir ninguna responsabilidad en caso de omisiones o errores.

Cali, Colombia, septiembre de 2019

HERRAMIENTAS PARA CONSTRUIR MUNDOS VIDA ARTIFICIAL I



Colección Ciencias Naturales y Exactas

Ángel de la Encarnación García B.

Madrid, España en 1960. Completó sus estudios de Ingeniero de Telecomunicación en 1985 en la Universidad Politécnica de Madrid. En 1987 se afincó en Colombia trabajando independientemente en su propia empresa. En 1993 ingresa a la Universidad del Valle, como profesor de Electrónica en temas que van desde los microprocesadores, interfaces, lógica digital, FPGAs, VHDL y tiempo real. En esa época realiza su doctorado en la Universidad Politécnica de Valencia, España, graduándose en 1999. A partir de allí se traslada al departamento de Ingeniería de Sistemas y Computación donde trabaja en temas de programación orientada a objetos, sistemas operativos, vida artificial y computación evolutiva. Fundó el laboratorio EVA-LAB (Evolución y Vida Artificiales) y el grupo de investigación GUIA en Inteligencia Artificial. Hizo parte durante varios años del grupo de filosofía de la mente MENTIS. Autor de múltiples artículos publicados en revistas y actas de congresos.

Dedico este libro a mi mamá, Carmen, y a mi papá, Ángel.

Índice

PREFACIO.....	11
INTRODUCCIÓN.....	20
Resumen.....	22
Para saber más.....	22
REALIMENTACIÓN.....	24
Realimentación negativa.....	27
Realimentación positiva.....	31
Realimentación distribuida.....	43
Reproducción.....	44
Congestión en el tráfico.....	48
Aglomeración de personas.....	49
Gaia.....	50
Daisyworld.....	50
Sincronización.....	52
Resumen.....	54
Referencias.....	58
FRACTALES.....	60
Fractales iterativos.....	60
Fractales de tiempo de escape.....	73
Fractales estocásticos.....	80
Fractales en la naturaleza.....	84
Espectro multifractal.....	88
Resumen.....	89
Para saber más.....	90
Referencias.....	92
CAOS.....	94
Un sistema caótico muy sencillo: la curva logística.....	99
Otros sistemas que exhiben caos.....	116
Llave de agua goteando.....	121
Péndulo doble.....	123
Péndulo con imanes.....	124
Sistemas fisiológicos.....	124
Definiciones.....	125
Resumen.....	134
Para saber más.....	135
Referencias.....	138
LEYES DE POTENCIAS.....	140
Resumen.....	151
Para saber más.....	151
Referencias.....	152
ALGORITMOS EVOLUTIVOS.....	154
Algoritmos genéticos.....	156
Diagrama de flujo de datos.....	159
Población de cromosomas.....	159
Cálculo de la aptitud.....	163
Selección.....	165
Reproducción.....	169

Reemplazo.....	172
Ejemplo 1: salas de cine.....	173
Ejemplo 2: invertir una matriz.....	175
Problemas que pueden aparecer.....	176
Paralelización.....	183
Aplicaciones de los algoritmos genéticos a problemas combinatorios.....	185
Aplicaciones de los algoritmos genéticos a problemas multiobjetivo.....	187
Recomendaciones.....	199
Otras aplicaciones.....	201
¿El final de las ingenierías?.....	204
Mitos ampliamente divulgados sobre los algoritmos genéticos.....	206
Sistemas clasificadores evolutivos.....	209
Programación evolutiva.....	214
Estrategias evolutivas.....	219
Enfriamiento simulado.....	220
Evolución diferencial.....	225
Algoritmo genético híbrido de Taguchi.....	226
Programación genética.....	232
Evolución gramatical.....	242
Programación por expresión genética.....	246
Coevolución.....	251
Algoritmo evolutivo general básico.....	252
Resumen.....	258
Para saber más.....	259
Referencias.....	260
TEORÍA DE JUEGOS.....	265
Dilema del prisionero.....	284
Tragedia de los comunes.....	287
Paradoja de Braess.....	288
Paradoja del votante.....	289
Señalización.....	289
Resumen.....	294
Para saber más.....	295
Referencias.....	296
AUTÓMATAS CELULARES.....	298
Definiciones.....	299
LIFE.....	302
Autómatas celulares 1D.....	308
Autoduplicación.....	317
Resumen.....	333
Para saber más.....	334
Referencias.....	335
SOLUCIÓN A LOS PROBLEMAS DE INGENIO.....	338
Torres de Hanoi.....	338
Dinero de bolsillo.....	339
Nave espacial.....	340
Velocidad de la luz.....	340
Bandera francesa.....	341
Para saber más.....	344

Referencias.....	346
INTRODUCCIÓN A RUBY.....	347
Similitudes entre Ruby y Python.....	348
Diferencias entre Ruby y Python, a este nivel.....	349
Comentarios.....	349
Funciones.....	350
Paso de argumentos a funciones.....	351
Bucles.....	352
Verdadero y falso.....	353
Condicionales.....	355
Otra forma del condicional.....	355
Operadores lógicos.....	356
Arrays.....	357
Índices de los arrays.....	357
Arrays de dos dimensiones.....	358
Biblioteca estándar.....	358
Cómo imprimir en la pantalla.....	359
Interpolación.....	359
Cómo leer el teclado.....	360
Cómo convertir un string a entero, a flotante o a string.....	360
Ruby es OO puro.....	361
Iteradores.....	361
Clases y objetos.....	363
Composición de objetos.....	366
Metaprogramación.....	366
Para saber más.....	367
Referencias.....	367
INTRODUCCIÓN A CUCUMBER.....	370
Para saber más.....	374
Referencias.....	375
Glosario.....	376

PREFACIO

La vida artificial es una disciplina relativamente nueva que surgió en 1987 en el primer congreso sobre el tema. Fue definida por su organizador Christopher Langton como “la vida, no como es, sino como podría ser”. A pesar de que llevamos trabajando durante tres décadas, considero que todavía es una disciplina poco madura pues le falta alcanzar su objetivo principal: crear vida artificial en el computador. Esto lo menciono sin pretender menoscabar los grandes avances que se han dado en el campo y que veremos a lo largo de este libro, pero lo cierto es que todavía no hemos logrado un gran éxito, como desarrollar un robot autónomo que se desenvuelva por su cuenta y que podamos decir que “es como si estuviese vivo”.

Pero ya casi.

El interés por esta disciplina ha ido en aumento y desde su comienzo se han escrito algunos libros muy buenos, aunque también podríamos decir que no han sido muchos. Cuando uno los lee se queda deslumbrado ante una variedad de técnicas, conceptos y ejemplos sorprendentes, pero que no se sabe muy bien a qué apuntan o por qué están allí. ¿Qué tiene que ver el caos con los autómatas celulares? ¿Usando ambos se logrará vida artificial? ¿Por qué? Y, por cierto, ¿qué es la vida artificial y cuáles son sus aplicaciones e importancia, aparte de hacer dibujos animados muy realistas? Estas y muchas más son preguntas que surgen en los interesados en la vida artificial.

Y ahora, en lo que atañe más directamente al texto que propongo, ¿en qué se diferencia de otros que tocan el mismo tema? El tema es tan amplio que se hizo necesario separarlo en dos libros (aunque podrían ser más en el futuro): en el primero, titulado *Herramientas para construir mundos*, contemplo todas las técnicas en detalle para que sirvan como ayuda a cualquier programador, informático o científico de la computación al desarrollar su propio *software*. A

continuación presentaré una visión general de estos tópicos.

En primer lugar se habla de la realimentación, que ocurre cuando las acciones que realizas sobre un objeto de alguna manera se devuelven sobre ti, con lo cual la idea de causa y efecto se desvanece, a la vez que se hace más difícil predecir lo que cada acción va a desencadenar.

En el siguiente capítulo se presentan los fractales, unos objetos muy bonitos, con muchos detalles y colores, que se fabrican a partir de una fórmula matemática tan simple que nadie lo creería. Además, no parecen ser objetos comunes de una, dos o tres dimensiones —que corresponderían aproximadamente a una cuerda, una hoja de papel o una mesa—, sino que pueden tener otras dimensiones que no son números enteros, como 0.8 o 2.45.

Luego hacen su aparición los fenómenos caóticos, para los que es imposible predecir a largo plazo cuál va a ser su comportamiento, a pesar de conocerse todos sus detalles de funcionamiento y no contener ningún proceso que dependa del azar. El ejemplo más conocido es el clima y, gracias a ello, tenemos tema de conversación sobre cómo se han equivocado en las predicciones de ayer de la televisión. Sin embargo, existen muchos otros fenómenos caóticos que van desde las cotizaciones de acciones en la bolsa hasta nuestros propios movimientos al caminar, e incluso fenómenos fisiológicos como los latidos del corazón.

Pasamos luego a las leyes de potencia, las cuales son distribuciones de probabilidad que ocurren tanto en el ámbito humano como en las ciencias naturales. Se caracterizan porque hay unos pocos fenómenos dominantes que consumen la mayor parte de los recursos de un sistema. Muchas veces se enuncia de esta forma: el 20% de los clientes de un negocio generan el 80% de los beneficios; el 20% de las líneas de código de un programa son las causantes del 80% de los *bugs*; el 20% de las páginas web son visitadas por el 80% de los navegantes; el 20% de los terremotos producen el 80% de las víctimas. Son innumerables.

Estos cuatro conceptos, realimentación, fractales, caos y leyes de potencia, están entrelazados. Siempre que se da uno aparecen los demás —aun cuando la realimentación es el más fundamental—, y todos generan mucha complejidad a partir de muy poco. Son procesos creativos.

Los algoritmos evolutivos descritos en el sexto capítulo, son una generalización del proceso de la evolución descubierto por Darwin, y se aplican a todo tipo de problemas donde haya que buscar una solución u optimizarla. Y cuando digo “todo tipo” no estoy exagerando, pues sirven para problemas de *software*, de

economía, de matemáticas, de diseño industrial, para jugar y ganar cualquier juego e incluso para crear música o pintura. Posiblemente sean los algoritmos más simples que existen, pero a cambio son muy versátiles y robustos, tanto que se consideran uno de los pilares de la inteligencia artificial. Y aunque hay que admitir su lentitud, eso no es una gran desventaja gracias a la potencia de los computadores actuales. Después de todo, la evolución biológica necesitó varios miles de millones de años para llegar al cerebro humano. Con los computadores actuales se pueden lograr resultados interesantes en cuestión de minutos. Los algoritmos evolutivos son grandes creadores de complejidad, mucho más que los cuatro procesos anteriores. Sin embargo, exigen una condición difícil de lograr de forma espontánea: que los entes que van a evolucionar sean capaces de realizar copias de sí mismos. Todo ello lo analizaremos en detalle.

En el séptimo capítulo se presenta una somera introducción a la teoría de juegos, con el objetivo de entender cómo emerge la cooperación a partir de los procesos evolutivos que, en principio, son todo lo contrario (son competitivos). Veremos que la cooperación genera complejidad en cantidades enormes, mucho más que los algoritmos evolutivos por sí solos.

Por último, veremos los autómatas celulares, entendidos como una gran cantidad de individuos idénticos y muy simples, donde cada uno se comunica solo con sus vecinos. No hay ningún dato global ni tampoco un control central. A pesar de ello, y a partir de comportamientos tan simples, la sociedad en su conjunto puede desarrollar una complejidad todavía mayor a las que hemos enumerado. En particular, estudiaremos que una sociedad así es capaz de sacar copias de sí misma y de hacer cualquier tipo de cómputo.

Todo lo anterior forma parte del bagaje bien conocido en el mundo de la vida artificial, pero quiero presentarlo de una manera nueva que ya estoy mencionando a cada momento: como generadores de complejidad. La importancia de esto radica en la idea intuitiva que tenemos acerca de que los sistemas simples no pueden hacer mucho, mientras que los sistemas complejos pueden moverse, reaccionar a estímulos, adaptarse a su entorno, pensar e incluso ser conscientes de sí mismos (que son temas que veremos en el segundo libro). Por ello necesitamos primero conocer estas técnicas que fabrican sistemas complejos.

También para los programadores presento un ejemplo de algoritmo evolutivo, escrito por mí en lenguaje Ruby y usando metodología BDD (*Behavior Driven Development*) con la herramienta Cucumber. Uso Ruby porque es un lenguaje sencillo y elegante, con mínima sintaxis y gran expresividad. Después de 30 años

programando principalmente en C++, que es perfecto pero complicado, y luchando contra muchos otros lenguajes incoherentes, descubrir Ruby ha sido una sorpresa refrescante pues me permite escribir código conciso con poco esfuerzo. Pero sé que no es un lenguaje muy popular, de modo que en los apéndices también ofrezco una introducción básica a Ruby y Cucumber¹.

En el segundo libro, *La escalera de la complejidad*, exploró los aspectos filosóficos de la vida artificial bajo un hilo conductor que es, como indica el título, la complejidad. Veremos que, conforme aumenta, emergen propiedades nuevas por las que podremos decir que hemos creado un mundo computacional que disfruta de libertad, que está vivo, que es inteligente, o incluso que es consciente. A cada uno de estos temas les dedico un capítulo. Doy un enfoque nuevo a lo que significa libertad e inteligencia. Y presento mi visión de lo que es la conciencia bajo un punto de vista computacional, despojándola de todo misterio. La conciencia tiene varios aspectos que han desconcertado desde siempre a los filósofos, pero todos son entendibles desde esta perspectiva. Quizás sea este el aporte más importante de los dos libros.

Como veremos también, todas estas propiedades son graduales. Hay una cierta inteligencia en una ameba e incluso en un destornillador². Y aunque consideramos que tanto plantas como animales están vivos, en los animales la vida alcanza más posibilidades, como moverse rápidamente. Lo mismo se puede decir de la conciencia que existe en diversos grados en los animales y tal vez incluso en las plantas. De modo que es natural la aparición simultánea de varias propiedades. Si las presento en este orden es porque la conciencia más sofisticada que conocemos requiere un poco más de complejidad que la inteligencia más sofisticada que conocemos, que a su vez requiere más complejidad que la vida, y así sucesivamente. Explicaré que el único salto notable en complejidad se da al pasar de objetos que pueden computar a objetos vivos. Los demás saltos son relativamente pequeños. Y por eso utilizo como hilo conductor la complejidad, que es simplemente una fórmula. Al final, todos estos temas se reducen a matemáticas y, más concretamente, a computación.

1 En el siguiente enlace al depósito de software Github se encuentra la información actualizada del libro con las erratas, el software y hojas de cálculo desarrolladas para este su realización: <https://github.com/angarciaiba/libroVA>

2 La mera existencia de los destornilladores y sus formas insinúan la existencia de los tornillos. Los tornillos no se podrían manipular sin el destornillador, de modo que el destornillador es como un artefacto de inteligencia congelada, que ayuda a realizar ciertos trabajos. En este sentido, posee una cierta inteligencia aun cuando no esté vivo.

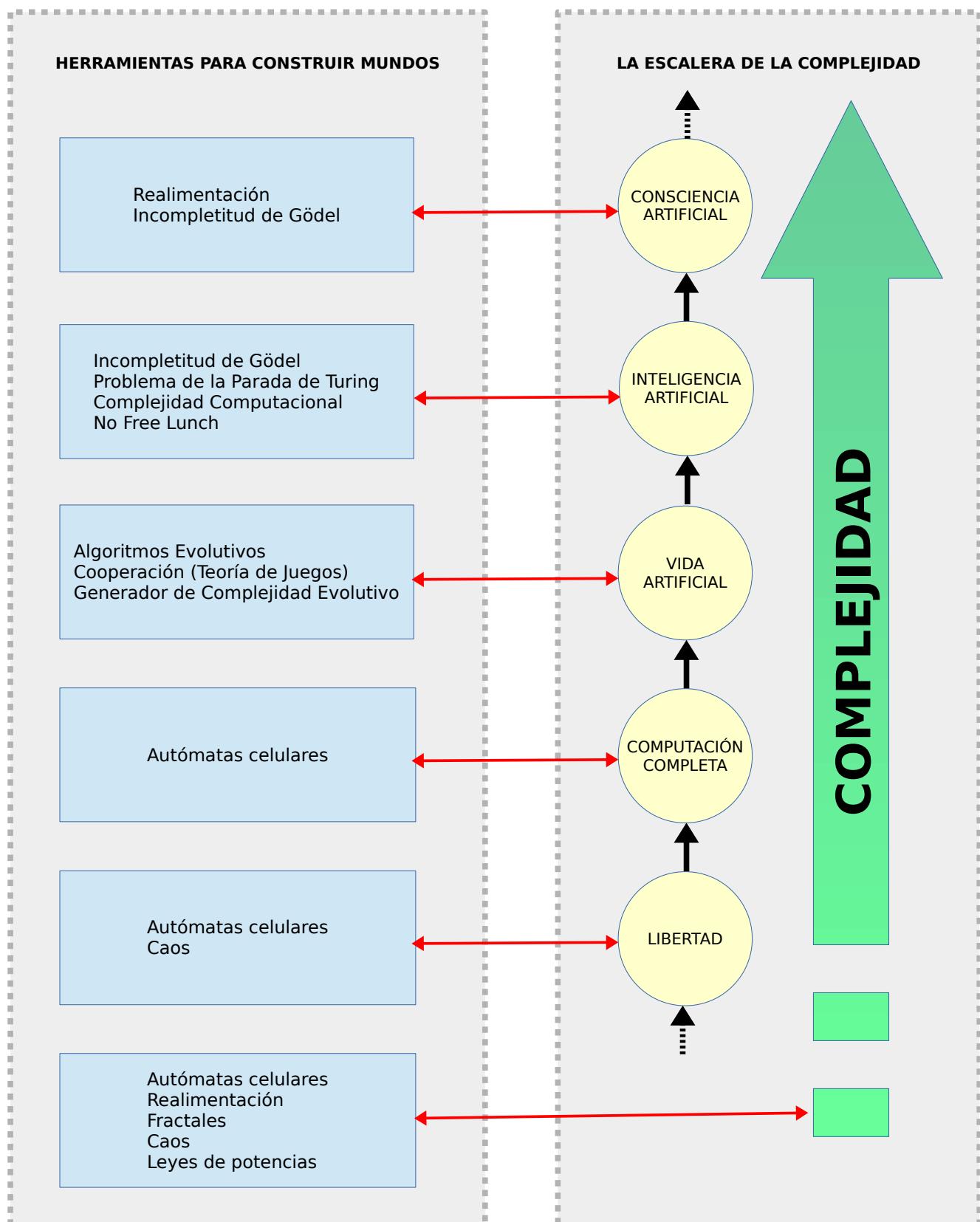


Figura 1: Estructura de los dos libros y relaciones entre ellos.

Además, presento una idea nueva: un generador de complejidad evolutivo, que

toma varios de los conceptos expuestos y los combina para crear una herramienta potente y general para fabricar sistemas complejos.

En resumen, el primer libro es de sistematización del conocimiento para programadores, mientras que el segundo libro es de investigación, más filosófico. En la figura 1 vemos de abajo hacia arriba los temas que desarrollo en cada libro y la correspondencia entre ellos, es decir, qué herramientas hace falta conocer para entender cómo se caracteriza la complejidad de cada nivel de emergencia.

En ambos libros también incluyo recuadros que contienen información paralela: en color azul están unas pequeñas notas bibliográficas de los personajes más importantes que han contribuido en estos temas; en color verde hay problemas de ingenio que ilustran alguna idea recién presentada o que se va a presentar; y en amarillo están las ideas más importantes que deseo resaltar. Al final de cada libro incluyo las soluciones a esos problemas de ingenio que voy proponiendo para estimular la reflexión sobre ciertos temas.

Verás que con cierta frecuencia menciono cuentos y películas de ciencia ficción. Eso no hace que los libros sean menos rigurosos. Lo que hay que entender es que la investigación tiene dos fases, y cuando formulas una hipótesis, diseñas experimentos y verificas si los resultados están o no de acuerdo con la hipótesis, estás en la segunda fase. No se suele hablar de la primera fase —las ideas iniciales que llevan a plantear la hipótesis—, seguramente porque nadie sabe cómo hacerlo, no hay una metodología que te lleve a las buenas ideas.

Simplemente se requiere inspiración. Y es allí donde las novelas y películas que exploran libremente el futuro sin censuras, sin revisión de pares ni otra atadura, pueden servir como chispa creativa para generar esas buenas ideas. Un momento acertado para capturar las ideas generadas por el inconsciente se da al despertar, cuando aún no se ha logrado vencer del todo el sueño. El lóbulo frontal del cerebro no parece estar activado todavía, censurando los pensamientos para que sobrevivan únicamente los que están acordes con la realidad comúnmente aceptada, en una especie de proceso evolutivo de las ideas. Y, con la censura dormida, eso permite que surjan las ideas más locas. A veces incluso nos sirve estar distraídos en una actividad manual —como lavar los platos— para bajar el umbral de censura. Si quieras continuar investigando en estos temas te animo entonces a que leas mucho y veas cine, especialmente de ciencia ficción dura, y a que laves los platos con mayor frecuencia.

Además, en las referencias incluyo diversos videos, algunos de TED y otros de YouTube. Los primeros son reconocidos por su alta calidad y en ellos participan

investigadores de primera línea. Se puede criticar a los segundos, debido a que son publicados libremente sin pasar por una revisión de pares. Y es verdad. Pero lo cierto es que a veces un video o una animación son más esclarecedores que todo el texto que yo pueda escribir aquí, de modo que corro gustoso el riesgo.

El público objetivo son los estudiantes que habitualmente toman mis asignaturas electivas de “Computación evolutiva” y “Vida artificial”, donde el requisito principal es tener experiencia en programación de computadores. No obstante, dada la época en que vivimos donde cualquier persona debería saber leer, escribir, la aritmética básica y programar, no es descabellado decir que estos dos volúmenes son también de divulgación, asequibles a personas con una mínima formación científica. En cualquier caso, cada libro se puede leer de manera independiente de acuerdo al interés del lector.

Desde el año 2000 trabajo como profesor en la Escuela de Ingeniería de Sistemas y Computación de la Universidad del Valle, dictando varias asignaturas incluidas las ya mencionadas. En estos libros pretendo reunir todo el material de clase, darle una coherencia y mayor profundidad, así como añadir algunos temas nuevos. Para entender rápidamente en qué consisten estas asignaturas te diré que la computación evolutiva abarca muchos algoritmos de optimización inspirados en la evolución de Darwin, aunque aquí los mostraré desde otra perspectiva: algoritmos creativos. Y con la vida artificial se intenta construir mundos artificiales en el computador donde emergen objetos con comportamientos tan complejos que pueden considerarse vivos, es decir, que se adaptan a su medio y se reproducen, colaborando o compitiendo con otros objetos.

Al pensar en el diseño de mundos artificiales es inevitable la pregunta de si nuestro propio mundo también lo es, o sea, si vivimos dentro de un computador, *a la Matrix*, o en alguna otra variante. Intentaré justificar una respuesta afirmativa a esta pregunta, bastante relacionada con una nueva concepción de la Física llamada *Digital Physics*, de la que haré algunos breves comentarios. El mundo es solo información.

Quiero agradecer a mi esposa Helga Rocío por todo su apoyo durante la escritura de estos libros, por hacer la primera revisión, ayudarme a organizar las referencias y hacer algunos bonitos dibujos. A mi hija Maya por ayudarme con la revisión del inglés de algunos artículos relacionados con este trabajo. Y a mi hermano Rafa por encontrar algunos de los errores más insospechados. El libro lo escribí en España durante mi año sabático del 2016-2017.

Y por ello también agradezco a la Universidad del Valle, pues nos ofrece a los profesores un tiempo para desarrollar ideas sin las presiones académicas y administrativas de todos los días. En la Escuela de Ingeniería de Sistemas y Computación disponemos del laboratorio EVALAB (evolución y vida artificiales), uno de varios donde opera el grupo GUIA de investigación en inteligencia artificial al que pertenezco. En ese laboratorio trabajan o han trabajado varios profesores (Irene Tischer, Víctor Andrés Bucheli Guerrero, Raúl Gutiérrez de Piñérez Reyes, Henry Antonio Saltaren Quiñones, Gabriel Conde Arango) y muchos estudiantes de pregrado, maestría y doctorado cuyos proyectos de fin de carrera se mencionan en el libro. Entre estos estudiantes, ahora ingenieros, quiero agradecer a Cristian Leonardo Ríos López que me haya cedido muy amablemente varias figuras de su trabajo de grado, y a Eider Falla que me haya permitido mostrar su software de la bandera francesa. Las discusiones con todos estos profesores y estudiantes han sido muy enriquecedoras a lo largo de tantos años, y recuerdo con especial agrado las reuniones de los martes en la tarde en el laboratorio y las generales en Piedralinda, debatiendo ideas y compartiendo un tiempo con personas sensacionales. En ese ambiente fue posible escribir el presente libro, por lo que les agradezco mucho a todos ellos su esfuerzo.

Así mismo, durante mi paso por el grupo de filosofía MENTIS de la misma universidad, recuerdo con mucho agrado a profesores y estudiantes de filosofía quienes me iniciaron en el tema de la conciencia, Juan Manuel Cuartas Restrepo, Ómar Díaz Saldaña, Ernesto Enrique Combariza Cruz, Juan Carlos Vélez Rengifo y Germán Guerrero Pino.

También agradezco al profesor Jesús Alexander Aranda Bueno por revisarme el teorema de Gödel. Al profesor Carlos Alberto Mayora Pernía, por revisarme el inglés del primer artículo relacionado con este trabajo. Y al profesor Fabio Germán Guerrero Moreno siempre muy interesado en estos temas y que me hizo una pregunta que me ha mantenido activo buscando su respuesta por varias décadas: ¿cuáles son los límites computacionales fundamentales? A los evaluadores quiero agradecerles por sus generosos comentarios y útiles sugerencias.

Para terminar, doy las gracias a mis amigos José Arturo, Diana Lorena y Víctor Manuel por su apoyo continuado en este y otros proyectos, y por tantas horas dedicadas a conversar sobre ideas interesantes.

Desde luego que si quedan errores deben achacármelos a mí, como siempre se dice en estos casos, pero no por repetirlo deja de ser verdad. Mi principal afán fue presentar una disciplina coherente en las asignaturas que ofrezco y, especialmente, las ganas de aprender más. Espero despertar las mismas ganas a

los lectores, aunque seguramente todos nos quedaremos con más preguntas que respuestas, pues es un tema abierto a más investigaciones.

El *software* y las figuras son míos excepto donde se indique lo contrario. Una parte procede de los cursos electivos que dicto habitualmente en la Universidad del Valle y el resto fueron diseñados *ex profeso* para estos libros. Los documentos de texto están escritos con *Libre Office 5.1.6.2* con tipos de letra FFF_Tusj de Magnus Cederholm, DejaVu Sans y Liberation Serif (usando *itálicas* para ecuaciones en el texto, tecnicismos y extranjerismos), corriendo sobre *Ubuntu 16.04LTS* y haciendo *backups* varias veces al día en diferentes medios, usando *Dropbox*, *rdiff-backup* y *git*. Como pueden imaginarse, en mi escuela estamos orgullosos de usar y apoyar el *software libre*.

INTRODUCCIÓN

“EVALAB es un pequeño laboratorio de una modesta universidad ubicada en una ruidosa ciudad en un país tropical situado en un olvidado planeta de una remota galaxia”

Ana Cristina Calderón Castrillón

La vida artificial es el estudio de las propiedades abstractas de los seres vivos, independientemente del sistema físico donde estén implementados. El sustrato de la vida natural es la química del carbono, pero podrían existir otros sustratos químicos, mecánicos, electrónicos e informáticos donde también sea posible crear sistemas que muestren las mismas propiedades. En general, la vida artificial intenta entender de dónde surge la complejidad de estos sistemas, creando herramientas matemáticas y computacionales para poder analizarla, manipularla y sintetizarla.

Desde un punto de vista computacional estamos buscando crear un sistema muy sencillo que sea capaz por sí mismo de aumentar su complejidad de forma autónoma, que decida su propio futuro, que se adapte a su entorno y sepa resolver los problemas que le vayan apareciendo. Para lograrlo prácticamente solo conocemos una manera: la realimentación, es decir, que la historia del sistema afecte a su futuro.

La realimentación es algo así como lo contrario a la independencia. En los sistemas donde lo que haga un agente no influye en lo que hagan los demás, jamás surgirá un comportamiento complejo. Y un agente que actúe independientemente de lo que le haya ocurrido en el pasado tampoco tendrá un comportamiento complejo. Veremos que hay dos tipos de realimentación, positiva y negativa, y que cualquier sistema suele tener muchas realimentaciones simultáneamente. Por ello no solo es que la realimentación produce sistemas complejos, sino que también son complejos de analizar.

La realimentación produce una serie de efectos observables muy llamativos. La ciencia ha identificado y puesto nombre a tres de ellos, aunque seguramente hay

muchos más. Son los fractales, el caos y las leyes de potencia. A ellos dedicaremos los tres siguientes capítulos.

Los fractales son figuras geométricas con infinitos detalles que muestran autosemejanza cuando se amplían o reducen de tamaño. Es lo que se denomina invariancia frente a cambios de escala. Pero a pesar de esa autosemejanza, al cambiar la escala pueden aparecer nuevas formas que nos sorprenden. Por otro lado, el fenómeno del caos ocurre en series de datos que transcurren a lo largo del tiempo y que pueden parecer periódicas pero realmente no lo son. También tienen y dependen de infinitos detalles y su atractor³ es un fractal. Y son impredecibles a largo plazo, a pesar de que no hay nada estocástico en ellas. Como tercer fenómeno aparecen las leyes de potencia que se caracterizan porque los eventos más grandes ocurren pocas veces (por ejemplo, grandes terremotos), mientras que los más pequeños ocurren muchas veces (por ejemplo, pequeños temblores). También se llaman distribuciones de densidad de probabilidad de tipo “cola gorda” porque ningún fenómeno tiene una probabilidad despreciable de ocurrir, y ello hace que no se puedan calcular promedios y que fallen las herramientas de predicción clásicas.

En sistemas que exhiban alguno de los tres fenómenos anteriores debidos a la realimentación, no es posible predecir exactamente su futuro. Son sistemas complejos de entender. Y nuestro objetivo será crearlos usando estas herramientas computacionales.

La realimentación puede ser distribuida, es decir, que el sistema esté formado por muchos agentes similares y que el futuro de cada uno de ellos dependa de la historia de todos. En estos sistemas no hay un control central que decida qué va a pasar, sino que el futuro depende de la interacción local entre agentes. Los agentes muy lejanos no se afectan entre sí, mientras que los cercanos sí lo pueden hacer. Esto implica la creación de una estructura espacial rudimentaria, siendo la más sencilla los autómatas celulares, que veremos en el último capítulo.

Pero los autómatas celulares son también una herramienta simple para entender cómo surge la capacidad de cómputo universal en un agente, que la necesita para desenvolverse en el mundo y tomar acciones para sobrevivir. Además, estos autómatas permiten analizar cómo surge la capacidad de autocopia, es decir, cómo puede un agente sacar una copia de sí mismo. En biología a ello se le llama tener hijos. Esta capacidad es primordial porque una vez que aparece también lo hace la evolución, entendida no solo desde el contexto biológico sino como un poderoso algoritmo computacional. Por ello, en el correspondiente capítulo

³ Concepto técnico que se verá en el correspondiente capítulo.

mostraremos cómo funcionan los algoritmos evolutivos más importantes y una implementación concreta de uno de ellos en Ruby usando BDD.

Cuando tenemos una población de agentes sometidos a la evolución, la complejidad de todo el sistema aumenta bastante pues los agentes interactúan entre sí buscando su propio beneficio. Ello se estudia en el penúltimo capítulo, usando una herramienta matemática llamada teoría de juegos. Lo más interesante de ello es que la propia fuerza de la competencia genera finalmente cooperación.

La realimentación, la evolución y las interacciones entre agentes producen la aparición o el reforzamiento de otros fenómenos interesantes como la libertad, la inteligencia o la conciencia que se verán en el segundo libro.

Resumen

Para crear vida artificial en el computador se emplean una serie de herramientas que generan complejidad espontáneamente. La principal es la realimentación, que genera fenómenos fractales, caóticos y leyes de potencias. La realimentación distribuida se puede modelar muy bien usando autómatas celulares, que también sirven para entender cómo emerge la computación completa y la capacidad de autocopia. Una vez que se da esta última, aparece la evolución. Los agentes que evolucionan interactúan entre sí para su propio beneficio, y ello se puede modelar usando la teoría de juegos.

Para saber más

Para ampliar el conocimiento en el tema de la vida artificial se incluye a continuación una lista de libros entretenidos, en varios idiomas.

- **Cristoph Adami. (1998). *Introduction to Artificial Life*. New York: Springer-Verlag.**
- **Dante Augusto Couto Barone, Ana Lúcia Cetertich Bazzan, Anderson Priebe Ferrugem, Cecília Dias Flores, Diego Correa Lucas, Eduardo do Valle Simões, Evandro Franzen, Henrique Oliveira da Silva, Igor Yepes, João Paulo Schwarz Schüler, Karla Fedrizzi Machado, Lucia Maria Martins Giraffa, Luis Otavio Campos Alvares,**

Maria Ribeiro Rodrigues, Paulo Martins Engel, Rosa Maria Vicari, Sidnei Renato Silveira e Vanessa Lindemann. (2003). *Sociedades Artificiais: A Nova Fronteira da Inteligência nas Máquinas*. São Paulo: Artmed Editora.

- **Claus Emmeche.** (1998). *Vida simulada en el ordenador*. Barcelona: Gedisa Editorial.
- **Julio Fernández Otozala y Álvaro Moreno Bergareche.** (1992). *Vida Artificial*. Madrid: Eudema.
- **José Santos Reyes.** (2007). *Vida Artificial: realizaciones computacionales*. A Coruña: Universidad de la Coruña.
- **José Santos Reyes y Richard J. Duró Fernández.** (2005). *Evolución artificial y robótica autónoma*. Madrid: RA-MA.

REALIMENTACIÓN

“Prediction is difficult, especially about the future”
Niels Bohr

La complejidad nace únicamente de la interacción entre objetos. Si no hay objetos, o si los hay pero no interactúan entre sí, entonces todo es sencillo y predecible. No hay complejidad. Lo que ocurre es la superposición (suma) de todo lo que hace cada uno de los objetos y, en ese caso, se hace posible calcular promedios y desviaciones típicas, esto es, entender el fenómeno por medio de estadísticas.

Para lograr complejidad necesitamos interacciones entre objetos, lo cual se modela de varias formas (por ejemplo, con grafos), pero aquí hemos elegido otra que nos va a servir para entender lo que viene después. Esa forma como nace la complejidad es la realimentación de procesos.

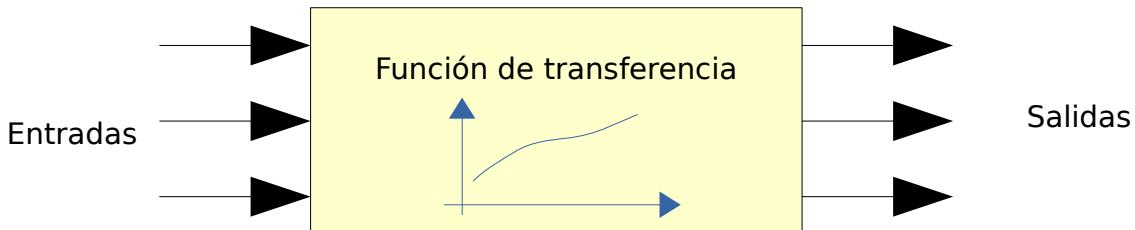


Figura 2: Sistema sin realimentación.

Los sistemas sencillos son procesos resumidos en una función de transferencia, que transforman entradas en salidas (figura 2). La función de transferencia puede ser lineal o no lineal. En cualquier caso, estos sistemas se comportan de una manera simple y, por tanto, fácil de predecir. Un ejemplo de ello puede ser... ejem... puede ser... la verdad es que casi no existen los sistemas lineales. Pero imaginemos una bicicleta de piñón fijo (en la que puedes pedalear hacia atrás). En ella, el número de pedaleadas que des (entrada) se traduce al número de metros que recorres (salida) y la relación entre uno y otro es una constante que viene dada por la geometría de los engranajes (radios de los pedales, rueda y

piñón). Esa bicicleta es un sistema lineal.

Los sistemas lineales se definen matemáticamente como aquellos donde hay superposición de soluciones, es decir, donde

$$\begin{aligned} f(x+y) &= f(x) + f(y) \\ f(Kx) &= Kf(x) \end{aligned} \quad Ec. 1$$

siendo x, y entradas del sistema, $f()$ la función de transferencia y K una constante, que también es equivalente a

$$f(K_1x + K_2y) = K_1f(x) + K_2f(y) \quad Ec. 2$$

siendo K_1 y K_2 constantes.

La primera línea de la ecuación 1 nos dice que, si tenemos dos entradas, la salida de la suma de las entradas es igual a la suma de las salidas producidas por cada entrada por separado. Y la segunda línea nos dice que la salida de un múltiplo de la entrada es igual al mismo múltiplo de la salida producida por esa entrada.

Esto es muy útil en ingeniería porque si un sistema es lineal podemos analizarlo en varios casos particulares, o sea, varias entradas, y si superponemos entradas, obtendremos superposición de salidas.

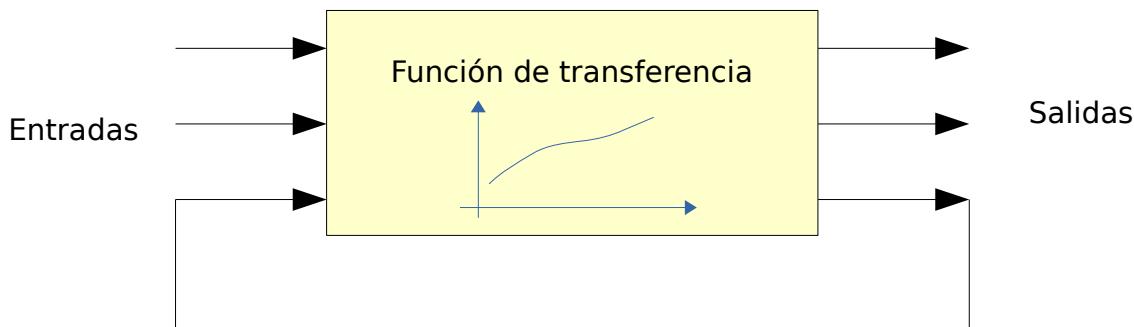


Figura 3: Sistema con realimentación.

Un caso especial ocurre cuando a uno de estos sistemas se le conecta alguna salida con alguna entrada (figura 3) y entonces ya con ello tenemos una realimentación. La salida que creímos fácilmente calculable ahora va a cambiar una de las entradas, que a su vez va a afectar a la salida, que cambiará la entrada, y así sucesivamente. Las cosas ya no son tan sencillas, ni desde una perspectiva intuitiva ni formal. El análisis de sistemas realimentados es una de las áreas más difíciles de la ingeniería, como puede verse en Phillips y Harbor (1988) o en Zilouchian y Jamshidi (2001).

La verdad es que es raro encontrar sistemas sin realimentación. Iba a proponer como ejemplo una nevera donde girando un dial seleccionas la temperatura que deseas (entrada) y como salida obtienes esa temperatura en el interior de la nevera. La función de transferencia convierte grados geométricos del dial en grados centígrados dentro de la nevera. Pero ello no es del todo cierto, pues resulta que las neveras modernas tienen internamente un control de temperatura, que es una realimentación negativa, con el que logran mantener la temperatura deseada independientemente de si se abre la puerta o si se colocan alimentos calientes dentro.

Iba también a proponer el ejemplo de un automóvil con los pedales de aceleración y freno y con el timón como entradas, siendo la salida su posición. Si el automóvil estuviera suspendido en el aire, podría valer como ejemplo, pero una vez que las ruedas tocan tierra, el espacio que recorre es una especie de acumulador de velocidad, es decir una memoria. No es lo mismo girar a la derecha en un punto de la ciudad que en otro (puede haber una dirección prohibida, una pared o una obra). La consecuencia de las acciones que tomas ahora dependen de la secuencia de acciones pasadas. Definitivamente un automóvil recorriendo una ciudad tiene memoria y, por tanto, hay una realimentación (figura 4).



Figura 4: Cualquier objeto físico recuerda su posición, lo cual es un tipo de realimentación.

Incluso muchos fenómenos de la física que nos enseñan que son simples, realmente tienen algún tipo de realimentación. Por ejemplo, el movimiento del péndulo de longitud l , sigue la conocida ecuación

$$\frac{d^2\alpha}{dt^2} + \frac{g}{l} \sin \alpha = 0 \quad \text{Ec. 3}$$

que me dice cómo varía el ángulo de desviación α respecto a la vertical sabiendo la aceleración de la gravedad g . Después se suele decir que para pequeños valores de α es posible aproximar $\sin(\alpha) \approx \alpha$ y entonces obtenemos la solución, que

corresponde a una oscilación sinusoidal. La ecuación 3 obviamente no es lineal, pero tampoco es exacta como nos suelen decir. Y ello es debido a una realimentación ya que g no es constante. Dado que el péndulo oscila, en los puntos más altos de su movimiento se encuentra más alejado de la Tierra, por lo que g es menor. El efecto es muy pequeño, claro, pero existe. Y es una realimentación porque la aceleración gravitatoria g hace que el péndulo oscile, y la oscilación del péndulo hace que cambie el valor de la aceleración gravitatoria.

A su vez, hay dos tipos de bucles: de realimentación negativa y de realimentación positiva. Veamos ejemplos y características de cada uno.

Realimentación negativa

La realimentación negativa se usa cuando se desea mantener estable la salida de un sistema, incluso frente a perturbaciones externas. Algunos ejemplos de ello son el control de la altura de un avión, el control de velocidad de crucero de un automóvil automático y el control de temperatura de un horno para hacer pan. Si hay una turbulencia que empuja el avión hacia abajo, veremos que la realimentación negativa se encarga de volverlo a llevar a la altura deseada. Si el automóvil comienza a subir una carretera muy empinada —con lo que de forma natural se reduce su velocidad—, el control de crucero lo volverá a llevar a la velocidad seleccionada. Si abrimos la puerta del horno para introducir pan y se nos escapa aire caliente entrando aire frío, la realimentación negativa del control del horno volverá a estabilizar la temperatura en el valor correcto para hacer el pan sin que quede crudo ni se queme.

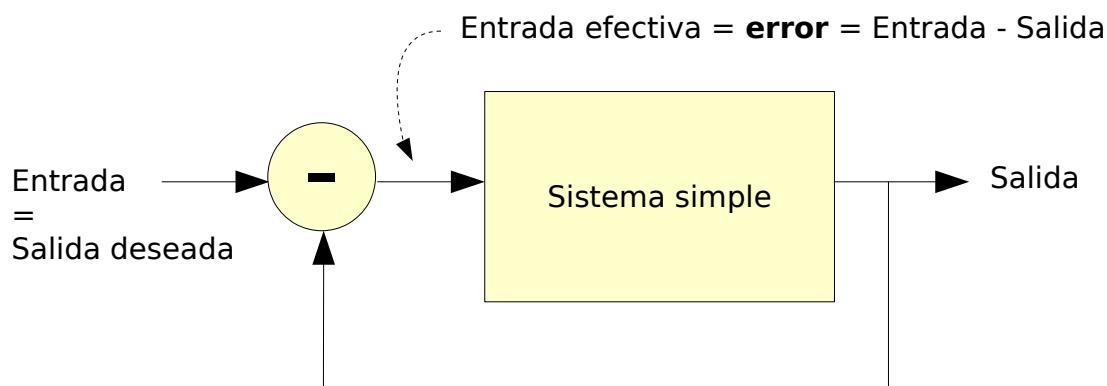


Figura 5: Realimentación negativa.

Para lograr realimentación negativa, la variable de salida se vuelve a inyectar a la entrada de modo que se oponga a los cambios en la salida. Esto se puede realizar

de varias formas, pero la más sencilla de entender es la de la figura 5, donde simplemente la entrada se resta de la salida y esa es la nueva entrada al sistema.

Para ver cómo funciona, lo habitual es considerar la entrada como la salida deseada. En el ejemplo del horno de la figura 6, tengo un dial donde selecciono la temperatura que deseo. Por ejemplo, supongamos que es 250 grados centígrados. Esa es la “salida deseada” del horno. La salida real (o, simplemente, salida) del horno, dado que lo acabo de conectar, será la temperatura ambiente, por ejemplo 27 grados centígrados. Entonces el sistema resta la salida deseada menos la salida, lo cual da $250-27=223$. Ese número 223 es la potencia calorífica que se inyecta al horno. Como es mucha, el horno se calentará rápidamente. Pero conforme la temperatura de salida aumenta, el resultado de la resta disminuye, y la potencia que se inyecta al horno también lo hace. Esto ocurre hasta el momento en que la salida coincide con la salida deseada, donde la potencia aplicada será $250-250=0$. Es justo lo que necesitamos: cuando la salida ha alcanzado la temperatura deseada, ya no hay que inyectar más calor al horno.

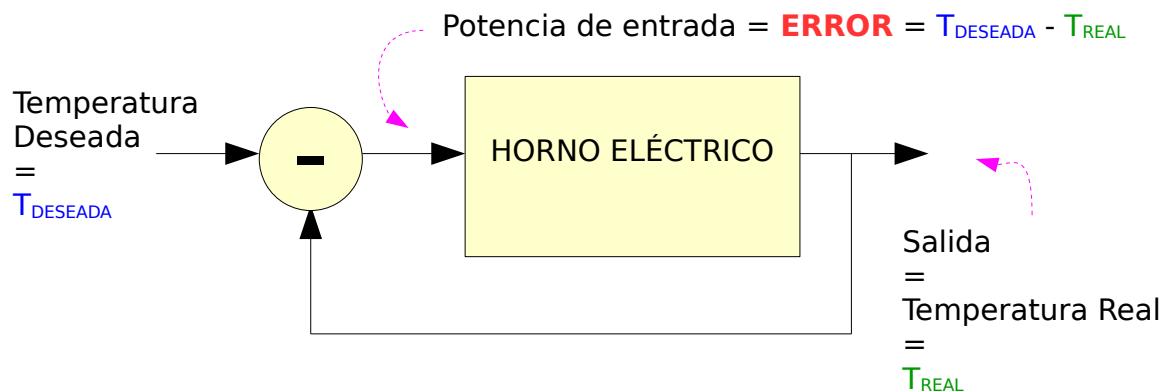


Figura 6: Horno con realimentación negativa.

En la práctica, y dependiendo de la ganancia y de los retrasos (lo que demora la potencia calorífica de entrada en lograr un aumento de temperatura), puede que la estabilización de la salida ocurra demasiado tarde, es decir, se acerque asintóticamente al valor deseado, pero solo se alcance cuando pase infinito tiempo (figura 7-a). Para evitarlo existen varias técnicas, pero la más sencilla es aumentar la ganancia. Con ello se aplicará más calor del necesario y el horno alcanzará rápidamente la temperatura deseada. Pero a cambio —debido a los retrasos (las inercias térmicas)— esa temperatura deseada será superada. Por ejemplo, supongamos que la salida llega a 270. Entonces ahora la entrada valdrá $250-270=-20$. La entrada es negativa y eso significa que el circuito de control extraerá 20 unidades de calor del horno empleando, por ejemplo, ventiladores. De este modo la temperatura comenzará a bajar hasta que la salida coincida con la

temperatura deseada. En ese momento se le aplicará una potencia de $250-250=0$. Y como hemos dicho que la ganancia es más alta de lo debido, ocurrirá ahora el mismo fenómeno pero hacia abajo: debido a las inercias térmicas, el horno bajará su temperatura un poco más de lo esperado, digamos a 247, con lo cual la entrada será $250-247=3$ de potencia calorífica. Y así seguirá, ajustándose poco a poco. Al final, la temperatura se estabilizará en el valor deseado. A este fenómeno de sobrepasar hacia arriba y hacia abajo el valor deseado se le llama *ringing*, rizado o sobreoscilación (figura 7-b).

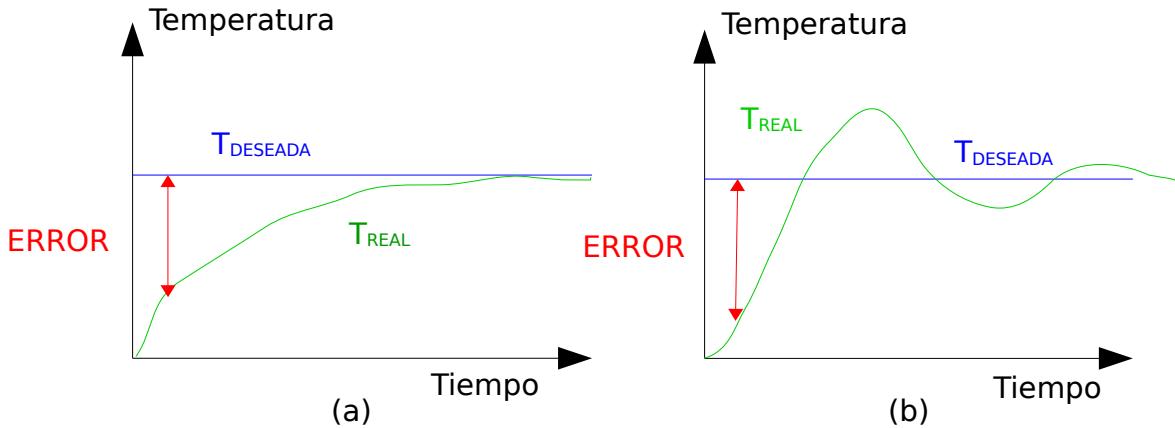


Figura 7: Evolución en el tiempo de una realimentación negativa a) subamortiguada; b) sobreamortiguada.

Mediante el uso de técnicas adecuadas (como poner circuitos de control que compensen las inercias térmicas del horno) se puede reducir la sobreoscilación al mínimo⁴.

La realimentación negativa se usa mucho en ingeniería porque sirve para:

- Estabilizar un sistema.
- Hacer robusto un sistema frente a diversas formas de funcionamiento, ruidos o perturbaciones externas.
- Controlar un sistema para que se comporte como uno deseé.

Otros ejemplos cotidianos donde se usa la realimentación negativa son:

- Conducir un automóvil mirando de frente y a los lados. Lo hacemos todos los días: si me aproximo mucho a un lado, giro el timón en dirección

⁴ Hay otras técnicas de control donde se evitan estos problemas (como el control predictivo), pero no son objetivo de este libro. La técnica explicada aquí, el control PID, es suficiente para ilustrar lo que significa un bucle de realimentación negativa.

contraria, y eso es una realimentación negativa que sirve para estabilizar el automóvil en el centro de la carretera.

- El tanque del agua en el inodoro. Su nivel de agua se mantiene constante gracias a un flotador conectado a una válvula de paso del agua. Cuando hay poca agua, el flotador está muy abajo y eso hace que la válvula de agua se abra completamente para llenar el tanque. Conforme el nivel de agua sube, empuja el flotador hacia arriba que, a su vez, va cerrando la válvula porque ya no se requiere tanto flujo de entrada. Al llegar a un cierto nivel prefijado, el flotador cierra la válvula por completo y deja de entrar agua. El tanque está lleno. El objetivo de este mecanismo es mantener el tanque lleno incluso cuando haya perturbaciones externas (cuando vaciamos el inodoro). El perfil en el tiempo para lograr anular la perturbación es el de la figura 7-a.
- Prácticamente en todos los sistemas metabólicos de los cuerpos vivos, como el control del azúcar, del pH o del oxígeno en la sangre. Por ejemplo, al comer el nivel de azúcar aumenta y entonces el páncreas libera insulina en el torrente sanguíneo que captura el azúcar y lo lleva al interior de las células donde se usa para generar energía o se almacena como grasa o como glucógeno, volviendo a dejar ese nivel en su valor normal. Y si el nivel de azúcar en la sangre disminuye, entonces el páncreas libera glucagón, que convierte el glucógeno en azúcar. Esta realimentación negativa evita que los niveles de azúcar sean demasiado altos o demasiado bajos, y cuando falla es causa de enfermedades como la diabetes o la hipoglucemia.

En sistemas complejos a veces aparecen bucles de realimentación debido a casualidades aleatorias. Por ejemplo, en el mar de los Sargazos se encuentran muchas algas en una zona de aguas limpias donde no se supone que haya alimento para ellas. Se ha descubierto que esas algas generan vapores de compuestos de azufre que actúan como núcleos de condensación de lluvias, que arrastran hacia abajo el polvo continental, rico en minerales que necesitan las algas. ¿Por qué hay algas mar adentro, en aguas limpias donde se supone que no hay alimento? ¿Saben las algas que deben emitir azufre para producir lluvia y que les llueva el alimento del cielo? ¿Sabe el viento que debe soplar hacia ese lado, ya que allí hay algas esperando la comida que les lleva? ¿Sabe el continente que debe “producir” cierto tipo de minerales para que los arrastre el viento y los lleve a las algas? La respuesta a esas preguntas es un “no” rotundo. Se trata de una casualidad, que es aprovechada por la evolución. Y el cierre del bucle (es decir, que las algas se encuentren en el lugar correcto del mar) consiste en una realimentación negativa: las algas que se alejan del sitio correcto, mueren

(Lovelock, 2000, p. 164).

Realimentación positiva

En los sistemas con realimentación positiva, la salida se suma a la entrada (figura 8). Estos sistemas son inestables, fluctúan, cambian incluso aunque no haya ningún estímulo de entrada. Es muy difícil controlar sistemas así. Típicamente la salida crece exponencialmente (explota) hasta llegar a algún límite (consumir todos los recursos), por lo que se suelen producir catástrofes, como explosiones, y extinciones.

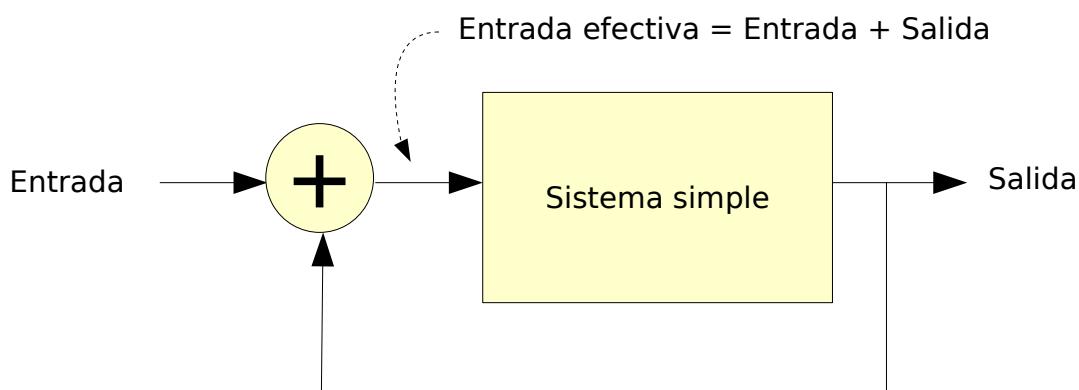


Figura 8: Realimentación positiva.

Ejemplos de estos sistemas son:

- La reproducción de conejos o cualquier otro ser vivo que disponga de recursos (alimento, oxígeno, etc.) de forma ilimitada.
- Los agentes económicos (cuanto más tienes, más ganas).
- Mantener una escoba en la palma de la mano (cuanto mayor es el desvío respecto a la vertical, más rápido se cae).
- Los meandros de los ríos (figura 9). Cuando un río se mueve por terrenos montañosos, busca siempre la dirección de máxima pendiente hacia abajo. Pero cuando lo hace por terrenos planos, la dirección que toma es bastante arbitraria. En este último caso, el cauce se decide por pequeños accidentes del terreno o por las rocas más o menos duras con las que se encuentre, y a causa de ello el río puede hacer una pequeña curva. Lo interesante es que la curva se amplificará por un fenómeno de realimentación positiva, dado

que el agua va más rápido en la parte externa (figura 10), con lo cual ese lado se erosiona más y se alarga el meandro, mientras que en el lado interno el agua se mueve lentamente favoreciendo el depósito de sedimentos, empujando todo el río hacia el otro lado. Hay un momento donde el meandro es tan pronunciado que se estrangula consigo mismo, dejando un pequeño lago aislado (llamado madrevieja) y retornando el río a un cauce rectilíneo, hasta que el proceso comience de nuevo. Y algo sorprendente es que cuando se expresa gráficamente la tendencia a formar curvas de un río conforme pasa el tiempo, este valor tiende en el límite a π (Stølum, 1996). ¡Un número irracional encontrado en la evolución de un proceso natural! Y no es este el único caso.

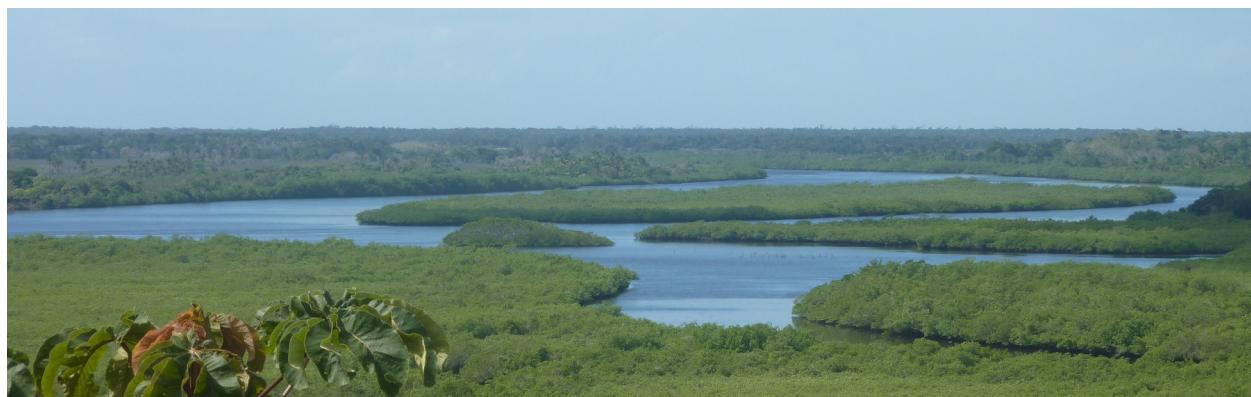


Figura 9: Meandros en río Paraguaçú, Brasil.

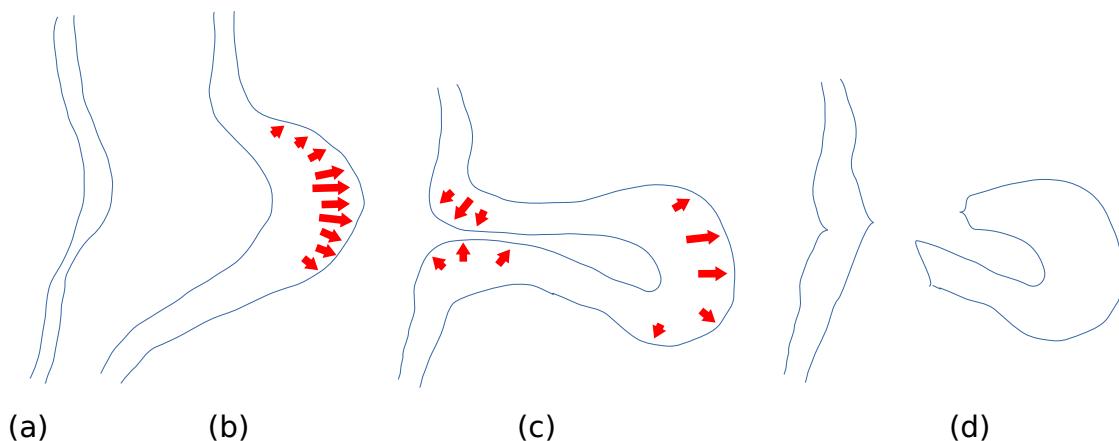


Figura 10: Formación de meandros en ríos. Las flechas indican cómo se desplaza la orilla.

- Un ejemplo de sistema lineal puede ser conducir un automóvil por una ciudad con muy poco tráfico. Si salgo de casa a la hora de siempre, llego al trabajo sin retrasos. Si salgo con un minuto de atraso, llego un minuto

tarde. Si salgo con cinco minutos de atraso, llego cinco minutos tarde. Es una situación ideal donde fácilmente puedo predecir qué va a pasar porque el sistema es lineal. Pero si pensamos ahora en la misma ciudad cuando se aproxima la hora punta, si salgo con un minuto de retraso puedo llegar dos minutos tarde, y si salgo con cinco minutos de retraso puedo llegar veinte minutos tarde. ¡No es lineal! Todos lo hemos experimentado y sabemos la causa: al salir un poco más tarde nos encontraremos con más tráfico que hará que cada una de las calles que atravesemos lo hagamos más lentamente y tengamos también mayor probabilidad de que nos afecte cualquier mínimo incidente, que a su vez producirá que lleguemos más tarde a la calle siguiente donde ya habrá aún más tráfico debido a que la hora punta está más cerca. En conclusión, los retrasos producen más retrasos. Es una realimentación positiva.

- La gravedad es un ejemplo interesante de realimentación positiva. Después de los primeros trescientos mil años a partir del *Big Bang*, el universo debió tener una densidad de materia uniforme (compuesta principalmente por átomos de hidrógeno y helio). Las mediciones del fondo cósmico de microondas de los proyectos COBE y WMAP de la NASA nos muestran que hubo pequeñas desviaciones respecto a esa densidad uniforme: las zonas ligeramente más densas atrajeron hacia sí más materia, aumentando su fuerza de atracción, y así sucesivamente. Este es el mecanismo de formación de los supercúmulos, cúmulos, galaxias, estrellas y planetas (Cristianfcao, 2010).

Debido a que la salida vuelve a inyectarse a la entrada para reforzarla, su crecimiento siempre es exponencial (figura 11). Por eso estas realimentaciones no se suelen usar mucho en ingeniería: conllevan a catástrofes. Y tampoco hay tanta teoría detrás de ellas como la que hay para las realimentaciones negativas. Además, cualquier ruido a la entrada es amplificado exponencialmente, por lo cual los valores exactos de salida son difíciles de predecir. Todo lo contrario de lo que un ingeniero desea.

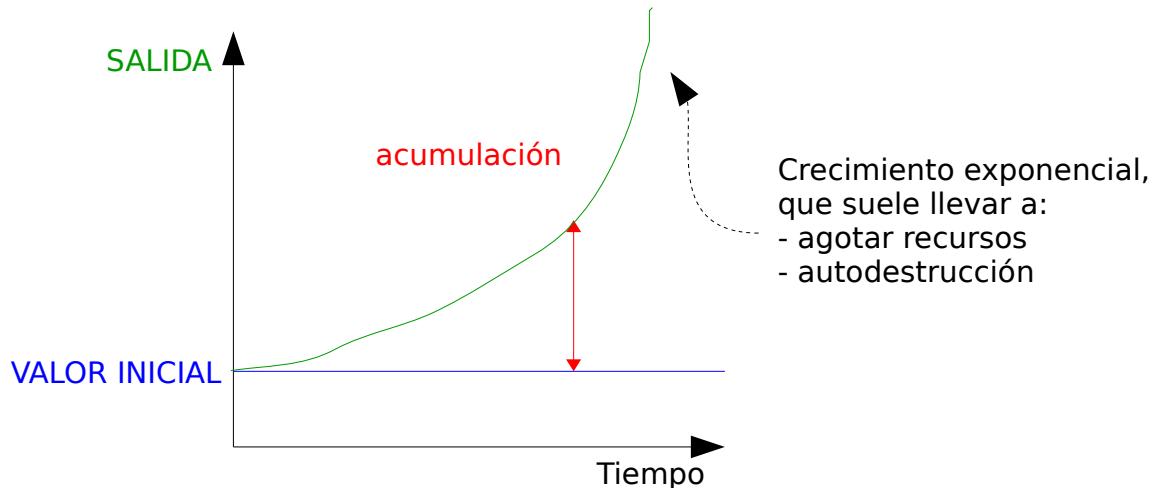


Figura 11: Evolución en el tiempo de una realimentación positiva.

Cualquier proceso exponencial no se puede sostener indefinidamente. De modo que las realimentaciones positivas suelen acabar tarde o temprano de una de estas dos maneras: si alguno de los recursos de los que dependen para crecer se agota, terminan en un proceso de crecimiento cada vez más lento, hasta la saturación (figura 12-a); también puede ocurrir que ello desencadene un cambio de signo en el proceso de crecimiento, es decir, que la exponencial sea ahora decreciente al alcanzarse algún umbral, lo cual producirá un proceso de oscilación⁵ sostenido (figura 12-b).

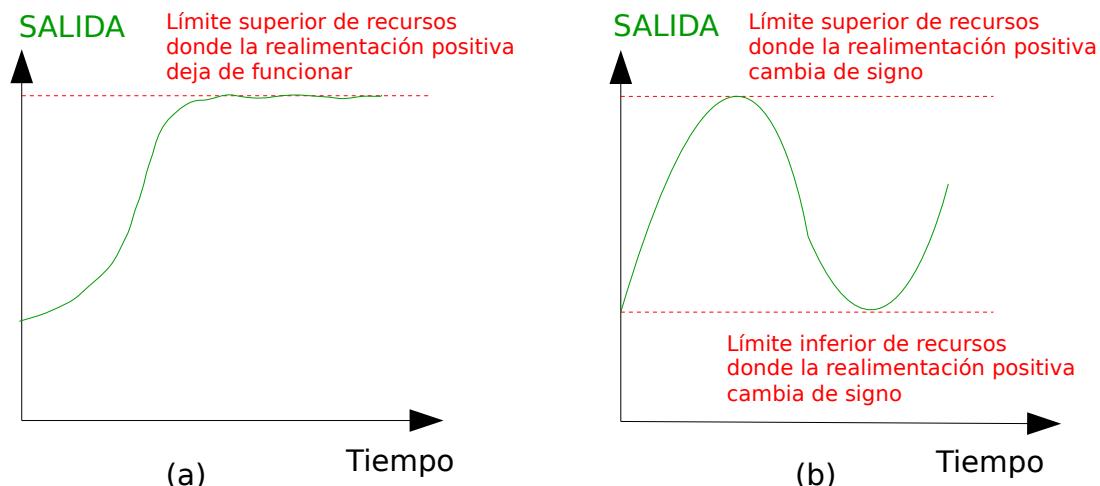


Figura 12: Como termina la realimentación positiva: a) saturación; b) oscilación.

En estos ejemplos hemos visto realimentaciones por superposición (suma o resta) de la salida sobre la entrada. Pero la realidad suele ser más complicada ya que puede haber realimentaciones más complejas con multiplicaciones, divisiones o

⁵ Recordemos que las oscilaciones están muy relacionadas con las exponenciales: $\cos(x) = (e^{ix} + e^{-ix})/2$

incluso operaciones no lineales, y no necesariamente de las salidas hacia las entradas, sino también desde/hasta estados internos. Los sistemas reales son difíciles de analizar. Además, hay otros efectos a considerar:

- La realimentación negativa mal hecha, con mucha ganancia o mucho retardo, puede convertirse en positiva. Los retrasos en las realimentaciones producen efectos indeseados en la mayoría de los casos. Por ejemplo, si conducimos un automóvil mirando únicamente por el retrovisor tendremos rápidamente un accidente, ya que la entrada llega con retraso. Y cuando por el espejo vemos una curva hacia la derecha es demasiado tarde para tomarla pues estaríamos fuera de la carretera. No vemos lo que está ocurriendo, sino lo que ya ocurrió. Por cierto, casi todos los sistemas políticos y económicos operan mirando el pasado, por lo que no es de extrañar que suframos graves problemas.
- En otros casos los retrasos producen efectos de estabilización. Por ejemplo, cuando se hacen compras las tiendas de barrio sobreviven porque están más cerca de ti. Si otras tiendas con mejores precios están más lejos y la información sobre sus ofertas te llega, pero tarde, o si es complicado desplazarse hasta ellas, o si compras por Internet, pero pagar es complicado, entonces todos estos factores introducen retrasos que estabilizan el sistema. Por el contrario, si tienes tiendas en Internet bien establecidas, eficientes con los pagos y la entrega, y donde la información sobre precios y calidades fluye al instante —sin retrasos—, entonces cualquier tienda que venda algo con alguna ligera ventaja sobre las demás se llevará todos los clientes. El que gana, lo gana todo; y los demás pierden y quiebran. Esto puede ser bueno para los clientes pero produce sistemas muy inestables, donde incluso el mejor vendedor en un momento dado puede ser reemplazado por otro que le supere ligeramente. A largo plazo tampoco es bueno para los clientes porque se pierde variedad en la oferta y calidad en las garantías.
- La realimentación positiva, cuando el sistema se satura sin llegar a dañarse, puede convertirse en negativa, es decir, en pequeñas oscilaciones alrededor del punto de saturación, que es el nuevo punto de estabilidad.
- No es el objetivo de este libro entrar en tecnicismos, pero el análisis riguroso de las realimentaciones requiere una matemática sofisticada como las transformadas de Fourier, de Laplace, Z y *wavelets*, por mencionar las más conocidas. Usando las dos primeras podemos calcular la ganancia y el desfase en el bucle en función de la frecuencia (figura 13). Un desfase de

180 grados convierte una realimentación positiva en negativa y viceversa. Y para algunas frecuencias puede haber realimentación negativa y para otras positiva.

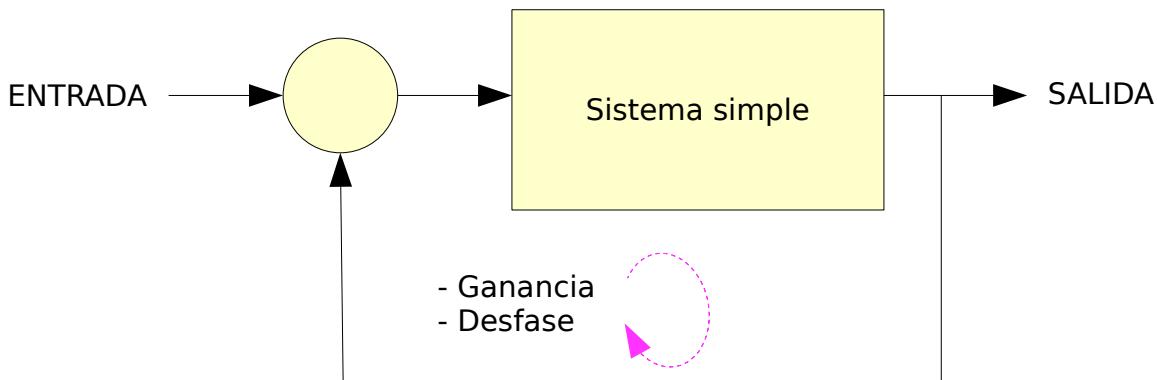


Figura 13: La ganancia y la fase en un bucle de realimentación.

Algunos ejemplos de lo complejo que es entender y controlar la realimentación:

- Si tenemos un amplificador de sonido conectado a un micrófono y un parlante, y acercamos el micrófono al parlante, habitualmente hay una frecuencia dominante para la cual la ganancia es 1 (es decir, lo que se introduce por el micrófono se amplifica de modo que, al ser reproducido por el parlante, sale el mismo sonido con la misma amplitud que el original) y un desfase de 0 grados o múltiplos enteros de 360 grados, con lo cual la onda de salida se suma a la de la entrada. El resultado de ello es el llamado “acople”, o sea, un pitido bastante molesto.



Figura 14: Cámara rotada apuntando a la pantalla, usando el programa Cheese en Linux.

- Si se conecta una *webcam* a un computador de modo que reproduzca el video capturado en su pantalla, y se apunta la *webcam* a la propia pantalla, tenemos un bucle de realimentación. Acercando más la *webcam* a la

pantalla y girándola en diversos ángulos obtendremos imágenes estáticas fractales (figura 14), que se explican en el correspondiente capítulo. Por otro lado, si se introduce algún tipo de filtro no lineal en la imagen, podemos obtener imágenes dinámicas (figura 15) con una complejidad emergente que no es obvio entender de dónde surge pues la imagen se mueve a pesar de que todo lo que está grabando permanece quieto.

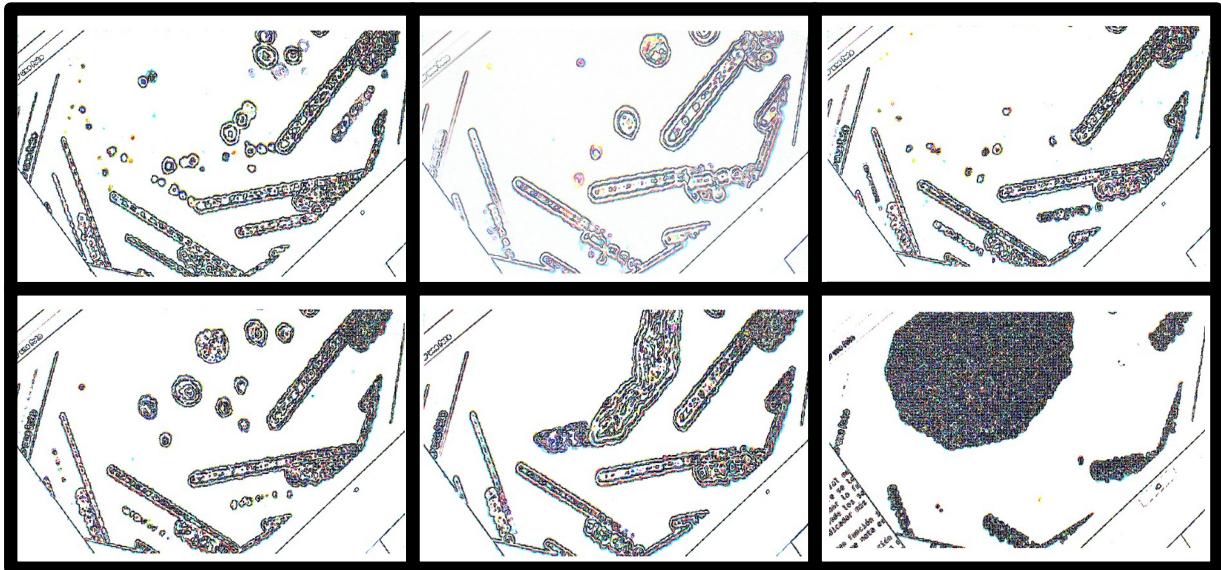


Figura 15: El mismo montaje anterior, pero usando un filtro de contorno. Se capturaron 6 instantáneas.

- Las encuestas previas a una votación introducen realimentaciones con resultados difíciles de prever. Si una encuesta indica un “Sí” rotundo a alguna cuestión, los votantes partidarios de esa opción podrían relajarse y dejar de ir a las urnas, pues el resultado les da una victoria holgada y, unos pocos votos de más o de menos no van a importar. Pero si muchos razonan así, los votos por el “Sí” disminuirán y el resultado puede terminar siendo lo contrario de lo que se esperaba.
- Habitualmente no se tiene control de todas las variables involucradas, por lo que al intentar controlar por realimentación solo una, puede que se obtenga el efecto contrario al deseado. Imaginemos un alcalde que necesita ingresos para mejorar la infraestructura de su ciudad. Lo usual es que aumente para ello los impuestos. El total de dinero recaudado es $T=NC$ siendo N el número de habitantes y C la contribución que paga cada uno (figura 16).

El alcalde imagina este modelo, pero no tiene control sobre la variable N . Lo más probable que ocurra al subir las contribuciones es que la gente se vaya a otras ciudades o aumente la evasión, disminuyendo drásticamente el total

del ingreso que obtiene en realidad. Para conseguir lo que desea debe hacer justo lo contrario: disminuir impuestos para atraer gente que quiera vivir allí. El modelo es más complicado, porque si todas las ciudades hacen lo mismo, competirán entre sí por conseguir más habitantes.

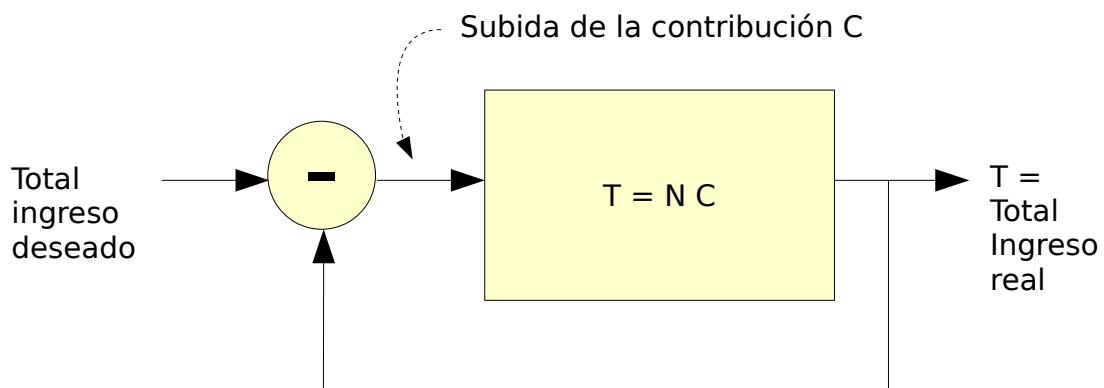


Figura 16: Modelo ingenuo de recaudación de impuestos.

Lo que pretendo con estos ejemplos es dejar claro que las realimentaciones cambian radicalmente el comportamiento de los sistemas, muchas veces de manera impredecible.

Las exponenciales son difíciles de detectar al principio, porque casi no hay cambios o, si llegan a ser detectables, solemos pensar que son lineales. La mejor forma de detectarlas es buscando la presencia de un bucle. La pregunta que hay que hacerse es: ¿existe un fenómeno que cuanto más se dé, más acelera su propio crecimiento?

Por ejemplo, mi abuelo, campesino, vivió el paso del azadón al tractor. Si yo le hubiera dicho hace 50 años que ello desembocaría en tractores autónomos, que aran el campo sin intervención humana, se habría reído de mí. Pero eso era razonable de esperar por cuanto existía y existe un bucle de realimentación positiva: la primera etapa —el paso del azadón al tractor— tuvo éxito porque permitía hacer los trabajos más rápido, produciendo mayores ganancias monetarias que podrían invertirse en mejorar las tecnologías, que a su vez generarían más ganancias. El bucle está allí. A pesar de ello es imposible predecir si todo el proceso se va a dar, pues pueden aparecer inconvenientes como que los costos tecnológicos aumenten (con lo cual desaparece la realimentación positiva). Tampoco es sencillo saber cuál va a ser el resultado final. Al respecto recordemos que actualmente ya hay tractores autónomos, pero seguramente aparecerán cosas mejores, dado que la realimentación positiva sigue en marcha. Ni siquiera sabemos si aparecerán realimentaciones negativas que estabilicen el proceso en algún punto, o realimentaciones positivas que entren en competencia con la

automatización, como mano de obra humana cada vez más barata. Predecir el futuro es difícil, como bien decía Niels Bohr, pero si se va a hacer, hay que estar atento a detectar la formación de bucles de realimentación, más que a los efectos inmediatos que produzcan.

En ingeniería se busca la realimentación negativa para que el sistema sea estable, predecible y controlable, y se intenta anular la positiva. Pero nosotros, para crear complejidad en un sistema necesitamos una combinación de ambas realimentaciones. Y hay que volver a insistir en que la negativa es fácil de estudiar, mientras que la positiva no, porque depende fuertemente de ruidos (errores) que haya en el ambiente. En la negativa los errores desaparecen, mientras que en la positiva los errores se quedan o incluso se amplifican.

Es importante que quede claro que la única manera de generar complejidad en un sistema es usando bucles de realimentación entre sus partes constituyentes. Y cuando hay realimentación suelen aparecer otros fenómenos asociados a la complejidad: fractales, caos y leyes de potencia. Pero la realimentación es lo básico. Típicamente existe una realimentación negativa que estabiliza el sistema hasta que aparece una positiva, que lo desborda y lo lleva a trabajar de una forma nueva. Esto sería mortal para el sistema —pues las exponenciales terminan mal— a no ser que aparezca una nueva realimentación negativa que lo estabilice en un nuevo punto de trabajo. El sistema ha cambiado. Entonces se dice que ha habido emergencia de algo nuevo y este es un mecanismo clave para generar complejidad.

Realimentación distribuida

La teoría de control estudia las realimentaciones que ya hemos abordado, y supone que el sistema es único y está ubicado espacialmente en un solo sitio. Pero la realidad es otra: los sistemas de vida artificial están compuestos por una multitud de objetos distribuidos espacialmente, donde no hay ninguno que tenga el control sobre los demás, y donde el comportamiento de cada uno afecta y es afectado únicamente por sus vecinos. Entonces, aunque *grosso modo* todo lo dicho hasta ahora sigue siendo válido, el estudio en detalle de un sistema de estos es terriblemente complejo e incluso imposible por métodos analíticos. Por ello se recurre a simuladores en *software*.

Estos simuladores pueden ser específicos para cada sistema, o pueden ser más generales, como los autómatas celulares en los que se incorpora directamente la

espacialidad y la limitación de la interacción entre objetos, que solo puede darse cuando son vecinos.

Reproducción

Un ejemplo de realimentación positiva distribuida espacialmente es la reproducción. Si tienes una pareja de conejos, en poco tiempo tendrán hijos que, a su vez tendrán más hijos. Si la tasa de fertilidad es constante y no muere ningún conejo, la curva que representa el número de parejas de conejos conforme pasa el tiempo es, aproximadamente⁶, una exponencial. Para ello podemos inventarnos dos modelos:

Cada cierto periodo, cada pareja de conejos tiene una pareja de hijos, duplicándose la población (figura 17). La secuencia es entonces: 1, 2, 4, 8, 16, 32, 64, 128...

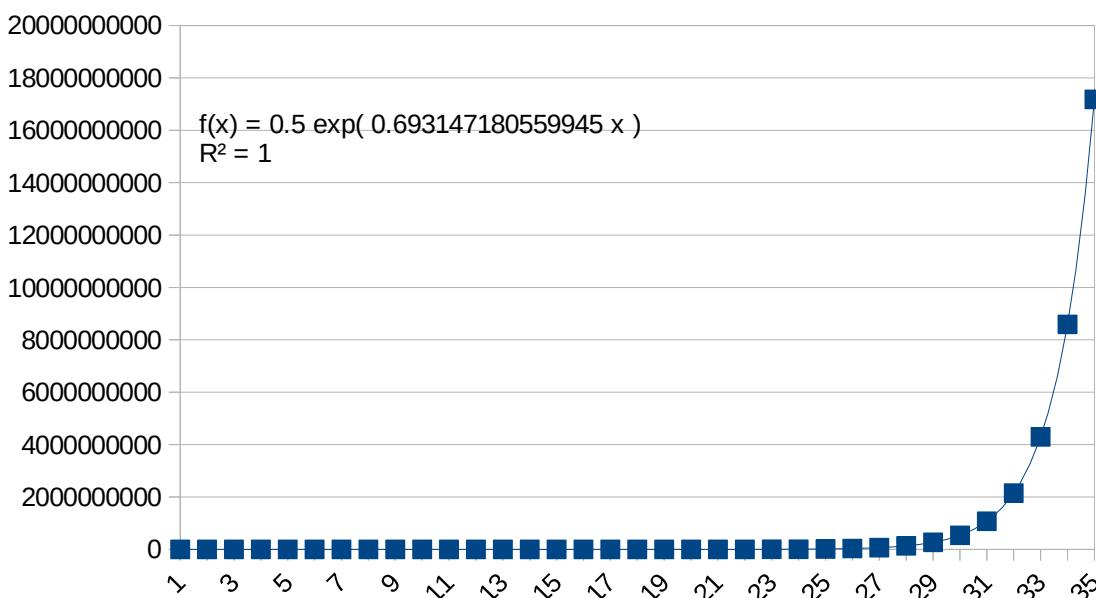


Figura 17: Reproducción exponencial.

O podemos poner un retraso de maduración sexual de un periodo. Todas las parejas de conejos tienen hijos. Inicialmente hay una pareja y, después del periodo de maduración, sigue habiendo una pareja. En el siguiente periodo tienen una pareja de hijos. En el siguiente, los hijos maduran, pero los padres ya lo están por lo que continúan con otra pareja de hijos, de modo que la cantidad de parejas de conejos en cada periodo es: 1, 1, 2, 3, 5, 8, 13, 21, 34... Esta es la archiconocida secuencia de Fibonacci (figura 18), donde cada término se calcula

⁶ Ya que la secuencia de conejos es discreta, mientras que la función exponencial es continua.

sumando los dos anteriores (ecuación 4). Y es también exponencial, aunque más lenta que la anterior debido al retraso de maduración.

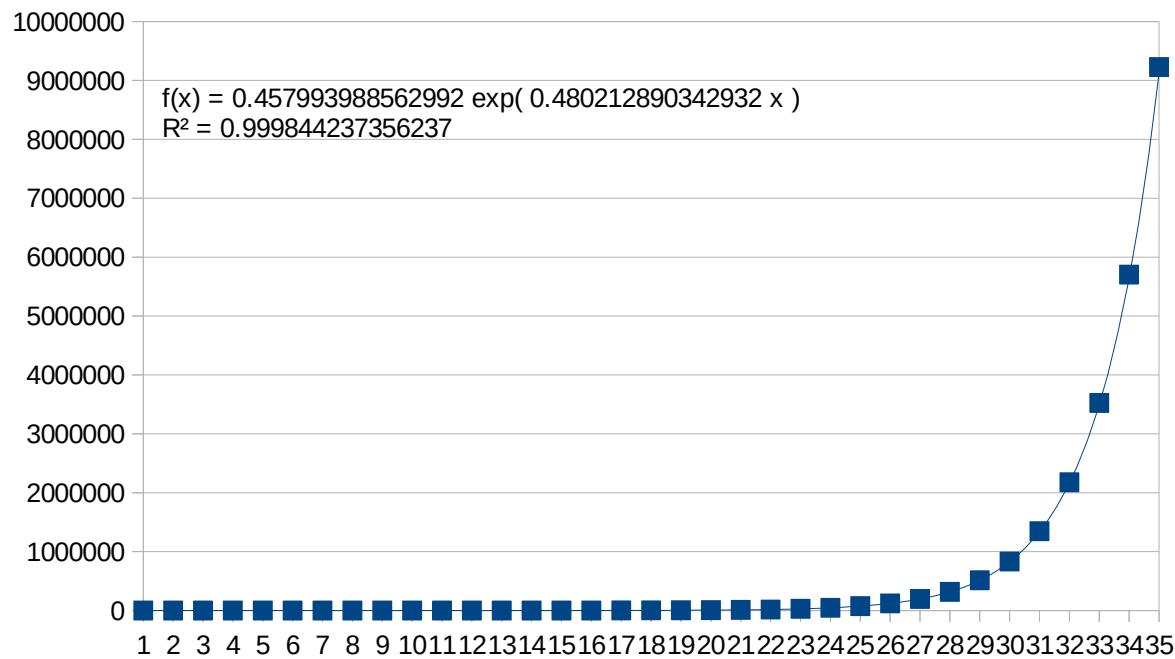


Figura 18: Reproducción exponencial con retrasos (secuencia de Fibonacci).

$$f_1=1$$

$$f_2=1$$

$$f_n=f_{n-1}+f_{n-2}$$

Ec. 4

La secuencia de Fibonacci se puede expresar también como:

$$f_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \quad \text{Ec. 5}$$

$$\text{siendo } \varphi = \frac{1+\sqrt{5}}{2}$$

que, para valores grandes de n es aproximadamente:

$$f_n \approx \frac{\varphi^n}{\sqrt{5}} \quad \text{Ec. 6}$$

donde se ve claramente que es una exponencial. Al número irracional $\varphi \approx 1.61803398874\dots$ se le llama relación áurea o número áureo y también puede ser calculado aproximadamente como la relación entre dos términos consecutivos de la serie de Fibonacci. Conforme los términos son más altos, la aproximación es

mejor:

$$\varphi = \lim_{n \rightarrow \infty} \frac{f_n}{f_{n-1}}$$

Ec. 7



Figura 19: Margarita con 13 espirales dextrögiras (en rojo) y 21 levögiras (en verde).

El número áureo se encuentra en muchas situaciones en la naturaleza. Un ejemplo bastante utilizado es el de los girasoles y las margaritas, que tienen sus semillas empaquetadas de tal forma que se pueden identificar dos tipos de espirales, dextrögiras y levögiras. En muchos libros nos dicen que si contamos el número de espirales que hay en cada sentido suelen salir dos números de Fibonacci consecutivos (como en la figura 19) que, al dividirlos entre sí, se aproximan al número áureo.

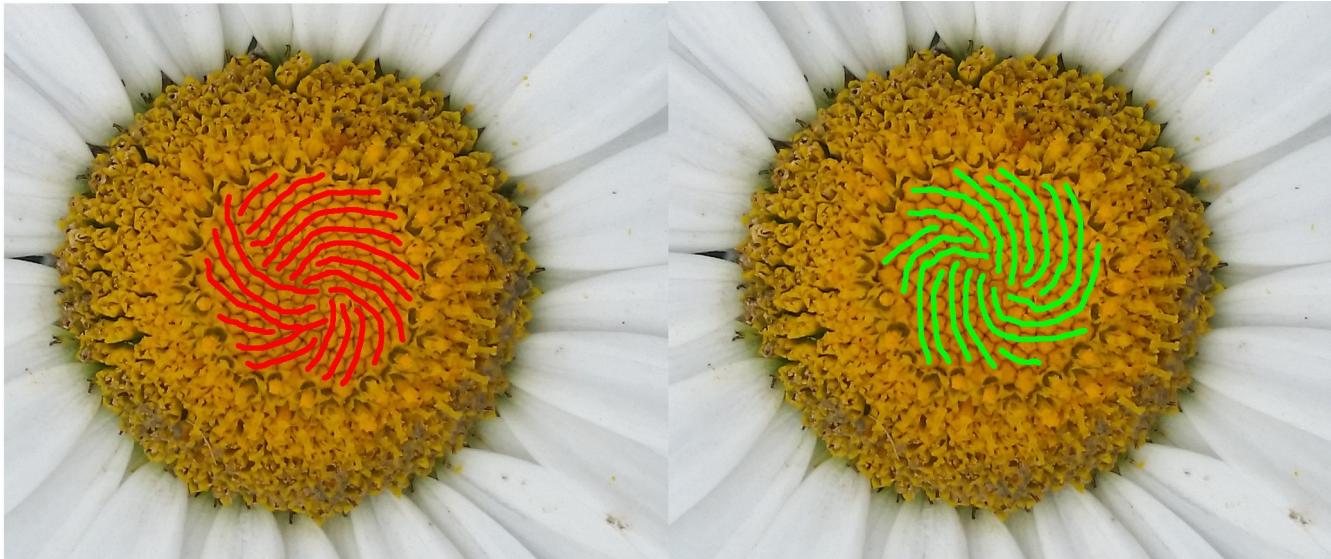


Figura 20: Margarita con 18 espirales dextrögiras (en rojo) y 16 levögiras (en verde).

En la práctica es difícil encontrar espirales tan perfectas, pues están en crecimiento, han sufrido problemas ambientales y tendrán imperfecciones genéticas, como las margaritas de la figura 20 o el girasol de la figura 21.



Figura 21: ¿Cuántas espirales tiene un girasol?

Congestión en el tráfico

Se ha realizado un divertido experimento de tráfico de vehículos pidiendo a varios conductores seguir una pequeña carretera circular. Aunque al principio comienzan equiespaciados, inevitablemente hay alguno que acelera un poco más acercándose demasiado al de delante. El de atrás, al ver un espacio mayor,

también acelerará, y así sucesivamente. El inconveniente es que conforme se aproximen a los de delante, se verán obligados a frenar y cada uno lo hará cada vez más bruscamente. Y el que está delante de todos acelerará, al ver atrás de sí un vehículo muy próximo. El algoritmo es así: *si hay más espacio delante o menos atrás, acelerar; si hay menos espacio delante o más atrás, frenar*. Aparentemente es una realimentación negativa pero, como es distribuida, cada conductor la ejecuta asíncronamente y bajo sus propios criterios, y ello produce fenómenos impredecibles de congestión. Concretamente, lo que se observa es una onda de compresión propagándose hacia adelante en la carretera (figura 22).

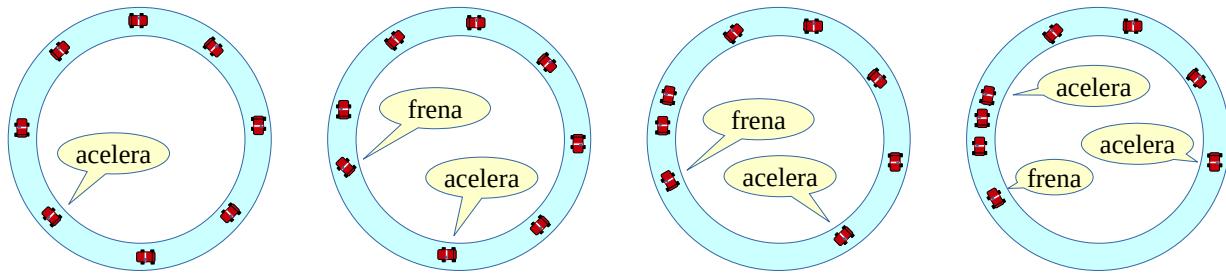


Figura 22: Carretera circular en instantes de tiempo sucesivos.

El trabajo completo puede leerse en Sugiyama et ál. (2008) que contiene varios videos.

Aglomeración de personas

Algo similar ocurrió en el 2006 en Arabia Saudí, cuando cientos de peregrinos murieron aplastados por causas desconocidas. El incidente volvió a repetirse un año después con la diferencia de que esta vez habían puesto cámaras en los recorridos y pudieron registrar lo que ocurrió. Y descubrieron que era similar al ejemplo anterior de tráfico. Cualquier incidente menor hacía que la fila se detuviera, con lo cual las personas de atrás frenaban y se apretaban más, produciendo una onda de presión que se propagaba hacia atrás. Era realmente una onda, aunque de densidad de personas, lo que significa que se reflejaba en los recodos e interfería consigo misma. Donde la interferencia era constructiva aparecían grandes máximos y mínimos de presión. En los máximos ocurrían los problemas.

Cada persona tomaba decisiones locales razonables: avanzar más deprisa si había mucho espacio vacío por delante, y frenar si no lo había. Pero ese conjunto de bucles de control locales producía globalmente fenómenos no deseados.

Al año siguiente rediseñaron la ruta para evitar estos problemas. La historia

completa, con otros ejemplos de comportamientos distribuidos, la relata Peter Miller (2010).

Gaia

James Lovelock planteó la hipótesis Gaia al afirmar que nuestro planeta Tierra es un conjunto enorme de bucles de realimentación. El tema está desarrollado en muchos de sus libros, con varios niveles de profundidad desde el técnico al divulgativo, pero recomiendo en particular *Las edades de Gaia* (2010), donde explica que hay fuertes interrelaciones entre todas las partes de nuestro planeta. Lo que ocurre en la atmósfera afecta a los océanos, lo que ocurre en los bosques afecta a la atmósfera, lo que ocurre en los océanos afecta a los bosques, de modo que todo está interrelacionado. Incluso las placas tectónicas tienen que ver con la vida, pues en sus bordes se acumulan bolsas de petróleo que facilitan su movimiento.

Lovelock aporta varios ejemplos interesantes, de los cuales quiero resaltar uno: los árboles y las algas emiten hacia la atmósfera bisulfuro de metilo, un compuesto químico que sirve como núcleo de condensación del vapor de agua de las nubes. Por eso suele llover donde hay árboles. Y hay árboles donde suele llover. Es un bucle de realimentación positivo y distribuido, sin un control central que decida todo.

Otro ejemplo que no está en este libro, es más sofisticado. Involucra la rotación del núcleo terrestre metálico, que produce un campo magnético que evita que el flujo de partículas procedente del sol impacte en la atmósfera sacándola hacia el espacio. Gracias a esto hay vida en la Tierra y, en particular, el *Homo Sapiens*, que se encuentra en estos momentos haciendo planes para mantener ese campo magnético en caso de que falle por causas naturales. E incluso está planeando cómo crear un escudo magnético similar en Marte para regenerar su atmósfera, que se perdió por la falta de uno (Green, 2017). El objetivo es colonizar Marte, es decir, reproducir la civilización humana terrestre allí. Este es un ejemplo de evolución con un bucle de realimentación.

Daisyworld

Es uno de los primeros sistemas de control distribuido artificial, diseñado por James Lovelock para comprobar la teoría Gaia. A él le intrigaba que la temperatura de la Tierra fuese estable durante muchos millones de años, a pesar de haber fluctuaciones en la energía que emite el sol. Las estrellas al principio de

su ciclo son más frías, y se calientan al envejecer. Sin embargo, la temperatura de la Tierra no ha sufrido cambios tan grandes en todo este tiempo, lo cual ha favorecido el nacimiento y existencia de los seres vivos que la habitamos.

El modelo consiste en un planeta cubierto completamente por dos tipos de margaritas, blancas y negras. Las negras absorben más luz y se calientan (y calientan su entorno). Las blancas reflejan la luz hacia afuera del planeta. De esta forma, el albedo⁷ total del planeta depende de la proporción entre estos dos tipos de margaritas.

Además, las margaritas tienen un rango de temperaturas donde pueden sobrevivir, típicamente entre 5°C y 40°C, así como una temperatura óptima donde se reproducen más deprisa, típicamente 22°C.

Cuando en un principio el sol generaba poca energía, había pocas margaritas blancas y muchas negras, de modo que el albedo del planeta era bajo y absorbía mucha de la energía que le llegaba, aumentando su temperatura. Conforme el sol calentaba, aparecían más margaritas blancas y morían más negras, de modo que el albedo aumentaba, reflejando una cantidad considerable de la energía solar recibida y logrando así mantener estable la temperatura del planeta.

El resultado es un bucle de realimentación distribuido, que captura más energía cuando llega poca, y refleja mucha energía cuando llega mucha, manteniendo así estable la temperatura del planeta⁸.

Hay modelos más sofisticados con margaritas de diversos colores, e incluso con conejos que se comen las margaritas y zorros que se comen los conejos. En ellos también se logra estabilizar la temperatura del planeta.

El último libro de Lovelock, más comercial y menos científico que los anteriores, se titula *La venganza de Gaia* y viene a decirnos que hemos destruido muchas cosas naturales y que ahora el planeta va a vengarse de nosotros, aunque el final es feliz como en los cuentos de hadas. La realidad es bastante distinta. El planeta no necesita tener intenciones ni tomar decisiones voluntarias vengativas. El problema es solo de bucles de realimentación positivos y negativos que han dado lugar al soporte adecuado para que nazca la vida y, con ella, nosotros. Si rompemos esos bucles nos va a ir mal, porque todos los organismos hemos evolucionado adaptándonos a estas realimentaciones. Desgraciadamente eso es lo que estamos haciendo con el cambio climático, que produce oscilaciones de temperaturas cada vez más bruscas. Este es el comportamiento típico de bucles

7 Coeficiente de reflexión de la luz.

8 Alexander (2012) nos ofrece un modelo interactivo con el que podemos jugar.

de realimentación positiva que están superando a los negativos que tenían estabilizado al planeta.

Algo similar ocurrió en el Precámbrico, una época en la que casi no había oxígeno en la atmósfera de la Tierra. Las bacterias vivían muy bien en un ambiente de anhídrido carbónico y metano, porque el oxígeno es muy reactivo, prácticamente un veneno para ellas. Entonces le empezó a ir muy bien a una bacteria que comía mucho y generaba oxígeno como deshecho. Se reprodujo mucho, demasiado, y llenó la atmósfera de contaminación venenosa: el oxígeno. Como consecuencia, ocurrió una extinción masiva de seres vivos que no estaban adaptados a tanto oxígeno.

Sincronización

Una propiedad muy importante en sistemas de vida natural y artificial es la sincronización distribuida, es decir, sin control centralizado, de muchísimos individuos. Ocurre de forma natural con los cantos de muchos animales, como las cigarras que uno escucha en las carreteras y que dan la sensación de ser un único zumbido que siempre está allí, como si se tratara de un animal de varios kilómetros de longitud. Ocurre con las células del corazón, que pueden oscilar a distintas frecuencias cuando se las cultiva *in vitro*, pero que se sincronizan en cuanto se ponen en contacto unas con otras.

Cuando se trata de seres vivos uno podría pensar que actúan coordinadamente por su propia voluntad, pero probablemente no sea así, ya que este fenómeno existe incluso en sistemas muy alejados de los biológicos. Recuerdo hace muchos años haber hecho un experimento similar al de las células del corazón, con osciladores electrónicos analógicos a transistores, tratando de diseñar un sintetizador de sonidos. Cada circuito oscilaba a una frecuencia distinta y yo pensaba ingenuamente que al unir todas las salidas obtendría la suma de las frecuencias. No fue así. Obtuve una sola frecuencia. En electrónica, esto se debe a que las salidas de los circuitos analógicos con transistores son también entradas. Y entonces, la salida de cada oscilador se colaba al interior de los otros osciladores, hasta que se pusieron de acuerdo, por así decir, en la frecuencia a la cual oscilar todos a la vez. Con los modernos amplificadores operacionales que vinieron después, esto no volvió a suceder debido a que las salidas eran unidireccionales, es decir, no permitían que por ellas entrase ninguna perturbación al circuito⁹.

⁹ Técnicamente hablando, en el modelo híbrido del amplificador operacional $h_{12} \approx 0$ y $h_{22} \approx \infty$, lo cual no ocurre con los transistores.

Una forma muy sencilla de visualizar el fenómeno de la sincronización es ubicando muchos metrónomos¹⁰ en una plataforma flotante que se puede mover libremente, por ejemplo, suspendida de cuerdas: aunque los metrónomos inicialmente van cada uno a su ritmo, en pocos minutos se sincronizan todos. No hay nada mágico en ello. Por la segunda ley de Newton, a cada impulso que dan al péndulo hacia un lado corresponde un impulso de la plataforma hacia el lado contrario, de manera que los metrónomos intercambian energía a través de la plataforma (Santos, 2013). Es muy fácil entender por qué la transferencia de energía produce sincronización gracias a que la gravedad interviene de una forma no lineal, donde aportando poco se puede obtener mucho. Recordemos lo que ocurre cuando empujamos a una niña en un columpio: si la empujamos cuando está en el extremo más alto y cercano a mí (figura 23-a), basta con un empujoncito suave para que se transforme en mucha velocidad cuando llega al punto más bajo. Pero si la empujamos en cualquier otro punto nos cuesta más esfuerzo. E incluso si tratamos de hacerlo cuando el columpio está en el lado derecho regresando hacia a mí (figura 23-b) va a pasar justo lo contrario: el columpio me transferirá energía, golpeándome y empujándome hacia atrás, hacia donde yo debería estar. De alguna forma, ese golpe me está sincronizando.

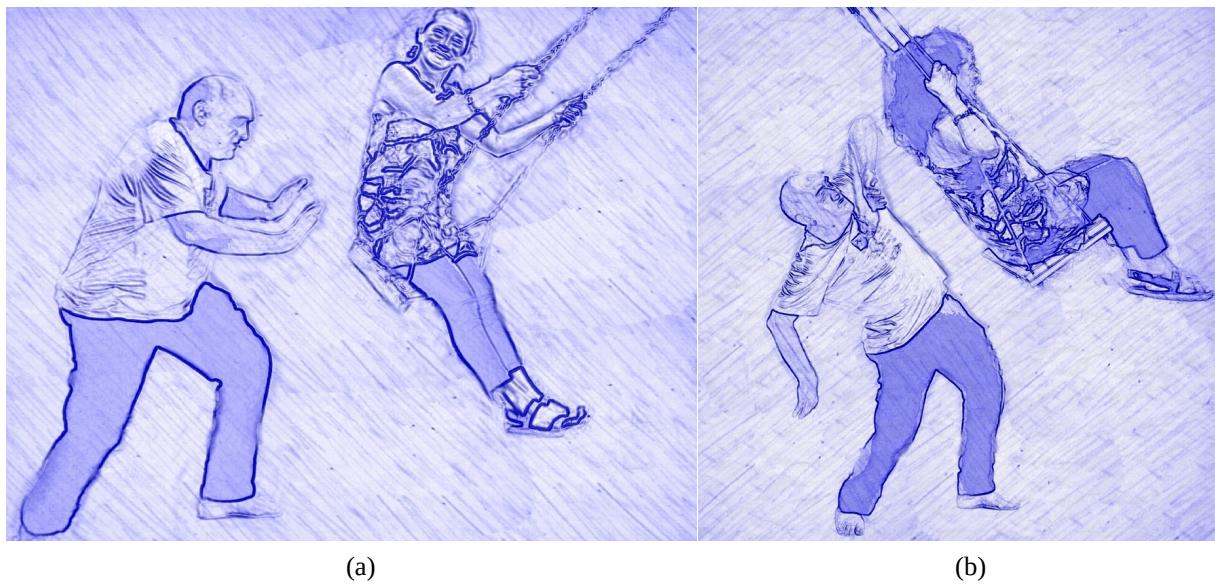


Figura 23: Columpio: (a) momento correcto para transferir energía; (b) momento equivocado.

De hecho, cuando de niño uno aprende a columpiarse solo, enviando las piernas hacia adelante cuando el columpio avanza y encogiendo las piernas cuando el columpio retrocede, sigue el mismo principio. Se trata de un péndulo doble sincronizado, de modo que la energía muscular del niño se transfiere al columpio.

Una de las personas que más tiempo le ha dedicado a este fenómeno es Steven

¹⁰ Pequeños péndulos que usaban los músicos del siglo pasado.

Strogatz. Puedes leer su libro (2004) aunque antes te puede motivar un corto video de resumen (2008), donde incluye imágenes muy bonitas de bandadas de pájaros y bancos de peces.

La sincronización es una mezcla de varias realimentaciones. Por un lado suele haber una realimentación positiva en cada individuo que lo hace oscilar. Por otro lado, suele haber un medio de comunicación entre ellos que permite otra realimentación positiva: cuanto más se empuje en una dirección, más se verán arrastrados los otros individuos a moverse de la misma manera. La sincronización es, seguramente, una forma de construir superindividuos como un conjunto de individuos que funcionan de manera coordinada.

Resumen

Cuando tenemos una cantidad de objetos similares, si no interactúan entre sí el resultado final se puede analizar como la suma de los comportamientos de cada objeto. Pero si interactúan entre sí, el sistema no es lineal y se puede modelar de varias formas. La que hemos elegido aquí es a través de realimentaciones, donde la salida del sistema modifica la entrada.

Hay dos tipos de realimentaciones: las negativas, que producen estabilidad pues se oponen a las perturbaciones que puedan llegar por la entrada del sistema, y las positivas, que producen inestabilidades fuertes típicamente traducidas a crecimientos o decrecimientos exponenciales, oscilaciones, saturación (donde el sistema deja de responder a las entradas) e incluso la propia destrucción del sistema. Los ingenieros diseñan sus sistemas (mecánicos, electrónicos, hidráulicos, etc.) usando realimentaciones negativas. Pero la naturaleza usa también las positivas pues son fuente de creatividad.

Los humanos no tenemos una intuición clara de lo que significa una exponencial, pues al principio su crecimiento es tan lento que ni siquiera lo detectamos. Cuando ya se hace obvio, suponemos que es un crecimiento lineal. Y para cuando todas las predicciones fallan y comenzamos a entender lo que ocurre, el fenómeno puede ser ya imposible de manejar. Por ejemplo, en ciencia, ingeniería o cualquier rama del saber que genere nuevo conocimiento, los científicos suelen ser muy conservadores en sus predicciones. Estoy trabajando en un proyecto que comienza a entregar resultados prometedores. Sé que la mayoría de proyectos de investigación requieren más tiempo del presupuestado por lo que, si me preguntan, diré que en un par de años tendremos los resultados definitivos y que,

seguramente en cincuenta años habrá aplicaciones industriales útiles para la humanidad. Pero no me doy cuenta que en la misma área en que yo trabajo hay millones de otros científicos e ingenieros, trabajando en proyectos similares, por lo que es poco probable que yo sea el primero en resolver el problema planteado. Puede haber alguien que lo encuentre antes. Y para esa persona la perspectiva es la misma. De modo que seguramente se encontrará el resultado antes de los dos años que vaticino. Pero, además, hay una red distribuida de investigadores usando unos los resultados de los otros. Hay bucles. Cualquier avance en un campo puede producir avances en otros, rápidamente y en cascada. Esos cincuenta años que creo que se necesitan para que el producto sea viable comercialmente, y que he calculado bajo la perspectiva de mi propio trabajo aislado, se pueden convertir en diez, en cinco o en menos, porque no soy consciente de la red de bucles que hay a mi alrededor que van a producir un crecimiento exponencial de conocimientos.

Habitualmente un sistema complejo se encuentra estabilizado por una realimentación negativa, hasta que ocurre un fenómeno imprevisto que produce una realimentación positiva o destruye la negativa que lo tenía estabilizado. A partir de allí, la realimentación positiva dominante producirá un (de)crecimiento exponencial de alguna de sus variables. El sistema podría autodestruirse a no ser que aparezca una realimentación negativa que lo vuelva a estabilizar en otro estado distinto. En ese momento se dice que ha habido emergencia de algo nuevo.

Entender que los sistemas complejos tienen realimentaciones puede ayudarnos a evitar acciones ingenuas al intentar resolver problemas. Por ejemplo, si pensamos en las cada vez más bruscas oscilaciones de temperatura que sufrimos, es razonable suponer que son producidas por un bucle de realimentación positiva que tomó el control debido a que hemos roto un bucle de realimentación negativa que estabilizaba el sistema. Existe un proyecto para poner en órbita una gran pantalla que haga sombra a la Tierra. Con ello se logrará bajar la temperatura promedio pero no se evitarán las fuertes oscilaciones, pues bajará tanto el valor máximo como el mínimo de temperatura (figura 24-b). Será un alivio bajar la temperatura máxima, pero al bajar también la mínima se producirá otro tipo de catástrofes. Si deseamos disminuir las oscilaciones (figura 24-c), hay que restaurar el bucle de realimentación negativa perdido, o hay que disminuir la ganancia del bucle de realimentación positiva que provoca las oscilaciones, quién sabe cómo.

Además, la temperatura promedio del planeta va en aumento, lo que provoca deshielos en el Ártico y la Antártida. El hielo, al desprenderse y caer al mar, enfriá

su temperatura. Es un bucle de realimentación negativa, lo cual, contra la opinión general, de momento es bueno. Lo malo vendrá cuando desaparezca todo el hielo porque dejará de existir ese bucle que trataba de estabilizar la temperatura, y con ello las consecuencias serán peores.

De la misma manera, el aumento de la temperatura del océano produce huracanes que son catastróficos para la infraestructura y la vida humana. Pero desde un punto de vista sistémico son solo una realimentación negativa que intenta estabilizar esa temperatura, succionando el agua caliente hacia la parte alta de la atmósfera, donde se enfriá, e incluso mezclando las profundas aguas frías con las superficiales, más calientes.

Apenas comenzamos a tener conciencia de nuestro propio planeta con sus bucles de realimentación, lo cual nos está permitiendo entender que las acciones locales pueden tener consecuencias globales. La NASA (2017) acaba de publicar un bonito video al respecto.

Lo mismo ocurre con tantos otros sistemas, como el cuerpo humano. Si tenemos fiebre alta, hay que bajar la temperatura con baños de agua o antipiréticos para evitar daños en el cerebro (figura 24-a). Pero si lo que tenemos es una fuerte oscilación, por ejemplo de presión sanguínea o de glucosa en la sangre, de poco sirve aplicar remedios similares. Si se disminuyen todos los niveles de glucemia lograremos mantener el más alto en un rango aceptable, pero el más bajo puede caer tanto que nos produzca un desmayo (figura 24-b). Lo que hay que hacer es regular los bucles de control, aumentando la ganancia de las realimentaciones negativas, para estabilizar o disminuir la de las positivas, con el objetivo de menguar el rango de las oscilaciones (figura 24-c).

En resumen, cuando hay oscilaciones no es lo mismo disminuir el valor máximo (figura 24-b) que disminuir el rango (figura 24-c), mientras que cuando no hay oscilaciones sí es lo mismo (figura 24-a). Esto debería ser cultura general, porque se puede hacer mucho daño realizando regulaciones incorrectas.

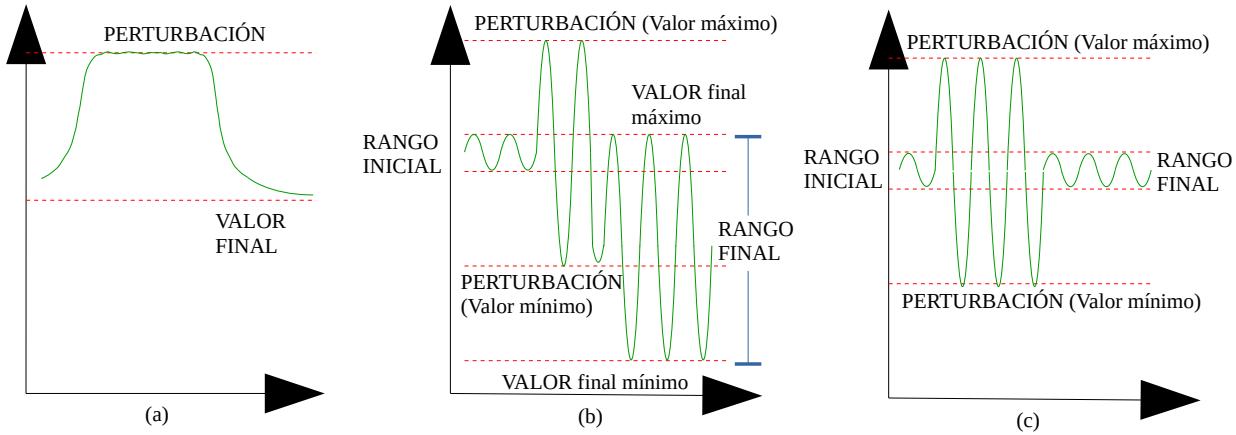


Figura 24: Perturbación y su posterior estabilización: (a) y (b) a un valor final máximo; (c) a un rango final.

Las realimentaciones en sistemas de vida artificial suelen ser distribuidas. Ningún objeto del sistema controla a todos los demás. Y cada objeto interactúa apenas con sus vecinos. El resultado de ello es mucho más difícil de prever y para lograrlo hay que utilizar algún simulador en *software*.

Las realimentaciones son el camino para crecer en complejidad, cuando todavía no existe la evolución. Cuando ya existe la evolución, tiende a estudiarse de forma separada, pues es un mecanismo mucho más potente para generar complejidad. Y, por si fuera poco, también puede haber interacciones entre las realimentaciones y la evolución.

Las realimentaciones producen sistemas complejos no lineales donde típicamente se harán presentes tres fenómenos conocidos: el caos, los fractales y las leyes de potencia, que estudiaremos a continuación.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Alexander, Ch. (2012). *Welcome to DaisyWorld*. Recuperado el 27 de agosto de 2017. Disponible en: <https://github.com/siu07cja/DaisyWorldWeb>

Green, J. L., Hollingsworth, J., Brain, D., Airapetian, V., Pulkkinen, A., Dong, C. y Bamford, R. (2017). A Future Mars Environment for Science and Exploration. *Planetary Science Vision 2050 Workshop*. Washington: NASA.

Lovelock, J. (2000). *Las edades de Gaia*. Barcelona: Tusquets Editores.

Miller, P. (2010). *The Smart Swarm: How Understanding Flocks, Schools, and Colonies Can Make Us Better at Communicating, Decision Making, and Getting Things Done*. New Jersey: Avery Publishing Group, Inc.

Phillips, C. L. y Harbor, R. D. (1988). *Feedback Control Systems*. New Jersey: Prentice Hall.

Stølum, H.-H. (1996). River Meandering as a Self-Organization Process. *Science* 271(5256), pp. 1710-1713. DOI: <https://doi.org/10.1126/science.271.5256.1710>

Strogatz, S. (2004). *SYNC: The emerging Science of Spontaneous Order*. London: Penguin Books.

Sugiyama, Y., Fukui, M., Kikuchi, M., Hasebe, K., Nakayama, A., Nishinari, K. Tadaki, S. y Yukawa, S. (2008). Traffic jams without bottlenecks: experimental evidence for the physical mechanism of the formation of a jam. *New Journal of Physics*, 10.

Zilouchian, A. y Jamshidi, M. (2001). *Intelligent Control Systems Using Soft Computing Methodologies*. Boca Ratón: CRC Press.

PELÍCULAS Y VIDEOS

Clore, J. (2010). *Feedback loops*. Recuperado el 2 de septiembre de 2017. Disponible en: https://www.youtube.com/watch?v=_QbD92p_EVs

Cristianfcao (2010). *The Cosmic Web, or: What does the universe look like at a very large*. Recuperado el 18 de julio de 2017. Disponible en: <https://www.youtube.com/watch?v=74lsySs3RGU>

Dedoimedo (2015). *Camera + monitor video feedback loop experiment*. Recuperado el 12 de junio de 2017. Disponible en: <https://www.youtube.com/watch?v=OvijFbFsulQ>

Emerton, M. (2007). *Fractal Video Feedback Effects*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=uzzktnjKkVg>

NASA (2017). *Our Living Planet From Space*. Recuperado el 21 de noviembre de 2017. Disponible en: <https://www.youtube.com/watch?v=3olcJBiyvw&sns=em>

Santos, W. (2013). *32 Metrônimos desorganizados entram em sincronismo Como?* Recuperado el 27 de agosto de 2017. Disponible en:

<https://www.youtube.com/watch?v=CR-yLEX-D4>

Strogatz, S. (2008). *How things in nature tend to sync up.* TED. Recuperado el 27 de agosto de 2017. Disponible en: <https://www.youtube.com/watch?v=aSNrKS-sCE0>

Villatoro, F. (2008). *Traffic Jam without bottleneck: experimental evidence.* Recuperado el 23 de agosto de 2017. Disponible en: https://www.youtube.com/watch?v=7wm-pZp_mi0

FRACTALES

“Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line”

Benoit Mandelbrot

Hay varios tipos de fractales, por lo que comenzaremos con los más sencillos de entender. Más adelante definiremos la dimensión fractal, así como sus propiedades más importantes: infinita rugosidad, autosimilitud e invariancia frente a cambios de escala.

Fractales iterativos

Se llaman iterativos en la literatura fractal, pero deberían llamarse recursivos teniendo en cuenta la forma como se implementan en el computador. Veamos un ejemplo: dibuja un segmento de línea vertical y divídela en dos partes iguales. Borra la parte de arriba y en su lugar dibuja dos segmentos de la mitad de longitud de la línea original, uno inclinado hacia la derecha y el otro hacia la izquierda, separados 90 grados. Con cada uno de los nuevos dos segmentos repite lo anterior, es decir, divídelos en dos partes iguales, borra el extremo final, sustituyéndolo por otros dos segmentos, y así sucesivamente (figura 25).

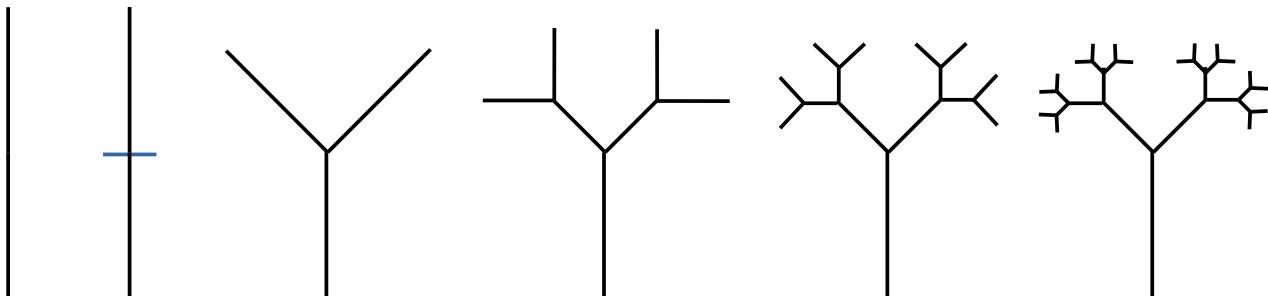


Figura 25: Fractal arbóreo (híbrido).

El objeto que obtenemos tiene infinitos detalles. En teoría, el proceso recursivo

con el que lo fabricamos es infinito, aunque en la práctica uno se detiene cuando la resolución de la pantalla no permite ver más. Es autosimilar, pues las dos nuevas ramas se parecen a las anteriores, pero giradas y a mitad de tamaño. Y es muy habitual que se parezca a algún objeto del mundo real, en este caso a un árbol.



Figura 26: Fractal de Koch.

De forma parecida, si dibujamos un segmento de línea horizontal y lo dividimos en tres partes iguales, eliminando la de mitad y poniendo en su lugar dos de estas partes con ángulo hacia arriba, y repetimos muchas veces, obtenemos el objeto de la figura 26.

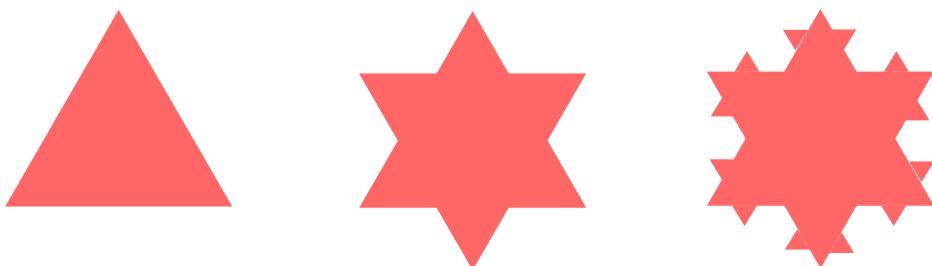


Figura 27: Fractal copo de nieve.

Por cierto que si usamos esa línea de base para los tres lados de un triángulo, obtenemos el conocido fractal que se asemeja a un copo de nieve (figura 27).



Figura 28: Fractal triángulo de Sierpinski.

Otro fractal muy conocido es el triángulo de Sierpinski. Para construirlo, se parte de un triángulo equilátero que se divide en cuatro triángulos iguales, eliminándose el central y repitiendo lo mismo sobre los tres triángulos restantes (figura 28).

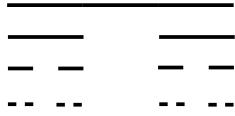


Figura 29: Fractal conjunto de Cantor.

El primer fractal seguramente lo propuso Cantor, aunque en aquella época no se denominaba fractal. Era más bien una especie de monstruosidad matemática. Consiste, como se ve en la figura 29, en un segmento de línea que se divide en tres partes eliminando la de en medio e iterando. Después de infinitas iteraciones, el resultado es un conjunto de puntos completamente inconexo: son puntos sueltos.

En este último ejemplo podemos ver que aunque partimos de un objeto de dimensión 1 (el segmento de línea), el resultado final es una colección de puntos aislados (de dimensión 0). Por aquí vendrá la definición de fractal. Pero antes que nada, es bueno advertir que en muchos libros definen un fractal como un objeto geométrico cuya dimensión es fraccionaria (o sea, con números decimales como 1.5 o 2.3346). Por ejemplo, el conjunto de Cantor tendrá una dimensión mayor que 0 (ya que es más que un único punto), pero menor que 1 (pues no alcanza a ser una línea). Esta es una definición sencilla pero no completamente correcta, porque algunos fractales tienen dimensión entera.

Una definición más acertada es¹¹:

Un fractal es un objeto geométrico cuya dimensión fractal no coincide con su dimensión topológica.

Recuadro 1: Definición de fractal

La dimensión topológica es la que ya conocemos: un punto tiene dimensión 0. Una línea, una circunferencia o el contorno de un polígono son de dimensión 1. Un círculo, un cuadrado, un plano son de dimensión 2. Un cubo, una pirámide y una esfera son de dimensión 3. Un tesseract es de dimensión 4. Y así sucesivamente.

¹¹ Hay otras definiciones, pero ninguna está completamente aceptada. Y esta es mi propuesta, pero sé que algunos fractales se quedan por fuera. Por ejemplo, se podría diseñar un fractal que parte de una línea (dimensión topológica = 1) a la que se le quitan puntos (como se hace con el fractal de Cantor) a la vez que se le añaden segmentos (como en el fractal de Koch) de modo que se compensen y al final la dimensión de Hausdorff también valga 1. En cierto modo, con los fractales pasa lo mismo que con los multifractales: todos los objetos lo son. La fractalidad o multifractalidad no es una definición de un tipo particular de objetos sino una medida, que se puede aplicar a todos. Por ejemplo, el segmento de la recta $[0,1]$ se puede construir recursivamente, dividiéndolo en tres partes (como el fractal de Cantor), pero sin quitar ninguna, lo cual da lugar a una dimensión de Hausdorff igual a 1. De la misma manera que los números reales incluyen a los enteros, los objetos fractales incluyen a los sólidos platónicos.

La dimensión fractal es una generalización del concepto de dimensión topológica, y puede medirse de varias formas. Las más usadas son:

- Dimensión de Hausdorff-Besicovitch.
- Dimensión de Minkowski-Bouligand, también llamada de conteo de cajas.

La dimensión de Hausdorff-Besicovitch es muy fácil de calcular cuando el fractal se construye de forma iterativa, como ocurre en todos los ejemplos anteriores. La fórmula es:

$$d_{HB} = \frac{\log(N)}{\log(Z)} \quad Ec. 8$$

siendo d_{HB} la dimensión de Hausdorff-Besicovitch, Z el *zoom* lineal necesario para que la nueva figura coincida con la original y N el número de copias de la nueva figura con las que se sustituye la original. En la figura 30 podemos ver que la dimensión de Hausdorff-Besicovitch coincide con la dimensión topológica en objetos tradicionales y también podemos apreciar que no importa el factor de *zoom* que usemos (en objetos regulares tenemos libertad para elegir cualquier valor). Y en la figura 31 podemos ver la dimensión de Hausdorff-Besicovitch de los fractales que acabamos de conocer.

¿Qué quiere decir este número? Fijémonos en los ejemplos de la tabla. Los fractales de Koch y Triángulo de Sierpinski tienen dimensión de Hausdorff-Besikovitch que es un número mayor que uno y menor que dos. Eso significa que ocupan más que una línea, pero no alcanzan a ocupar lo mismo que un plano, mientras que el conjunto de Cantor ocupa menos que una línea, pero más que un punto.

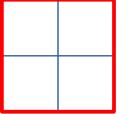
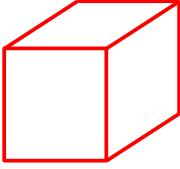
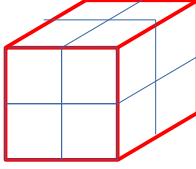
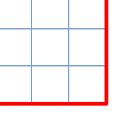
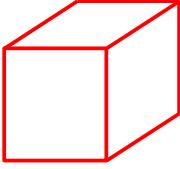
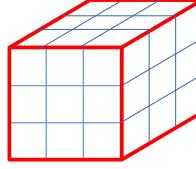
Nombre	Objeto original	d_T (dimensión topológica del objeto original)	Z (zoom lineal)	Nuevo objeto reducido por el zoom	Sustitución del objeto original por N copias del nuevo objeto	N (copias)	d_{HB} = $\frac{\log(N)}{\log(Z)}$
Punto	•	0	2	•	•	1	0
Segmento	—	1	2	—	—	2	1
Cuadrado		2	2			4	2
Cubo		3	2			8	3
Punto	•	0	3	•	•	1	0
Segmento	—	1	3	—	—	3	1
Cuadrado		2	3			9	2
Cubo		3	3			27	3

Figura 30: Dimensiones topológica y fractal de objetos ordinarios.

Nombre	Objeto original	d_T (dimensión topológica del objeto original)	Z (zoom lineal)	Nuevo objeto reducido por el zoom	Sustitución del objeto original por N copias del nuevo objeto	N (copias)	d_{HB} = $\frac{\log(N)}{\log(Z)}$
Arbóreo		1	2		Y	2	1
Koch y copo de nieve	—	1	3	—	—	4	1.26
Triángulo de Sierpinski	△	2	2	△	△	3	1.58
Conjunto de Cantor	—	1	3	—	—	2	0.63

Figura 31: Dimensión de Hausdorff-Besicovitch de algunos fractales.

Es importante señalar que el fractal arbóreo realmente es un objeto híbrido, pues de los tres segmentos que se forman en la primera iteración, solo dos se expanden. El otro segmento (el vertical) se mantiene sin cambios y existe únicamente para dar la sensación estética de que es el tronco del árbol. Y lo mismo ocurre en las siguientes iteraciones. Por eso, en este caso, N vale 2. Y por eso su d_{HB} vale 1, igual que su d_T , contradiciendo aparentemente la definición de fractal, pero ello ocurre porque no es completamente fractal. Si expandiéramos las tres ramas, entonces $N=3$ y $d_{HB}=1.584963$, y el resultado es el de la figura 32.

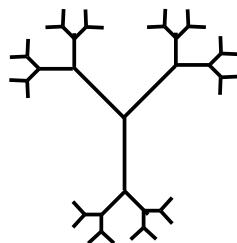


Figura 32: Fractal arbóreo.

Por otro lado, la **dimensión de Minkowski-Bouligand** d_{MB} se define con el número mínimo N_R de hiperesferas de radio R necesarias para recubrir el objeto fractal, cuando el radio tiende a cero. También se le llama conteo de esferas o

conteo de cajas¹².

$$d_{MB} = \lim_{R \rightarrow 0} \frac{\log(N_R)}{\log(1/R)} \quad Ec. 9$$

Y también se puede definir como el número N_L de hipercasillas de una cuadrícula de lado $1/2^L$ que se intersecan con el objeto fractal. Ambas definiciones son equivalentes.

$$d_{MB} = \lim_{L \rightarrow \infty} \frac{\log(N_L)}{\log(2^L)} \quad Ec. 10$$

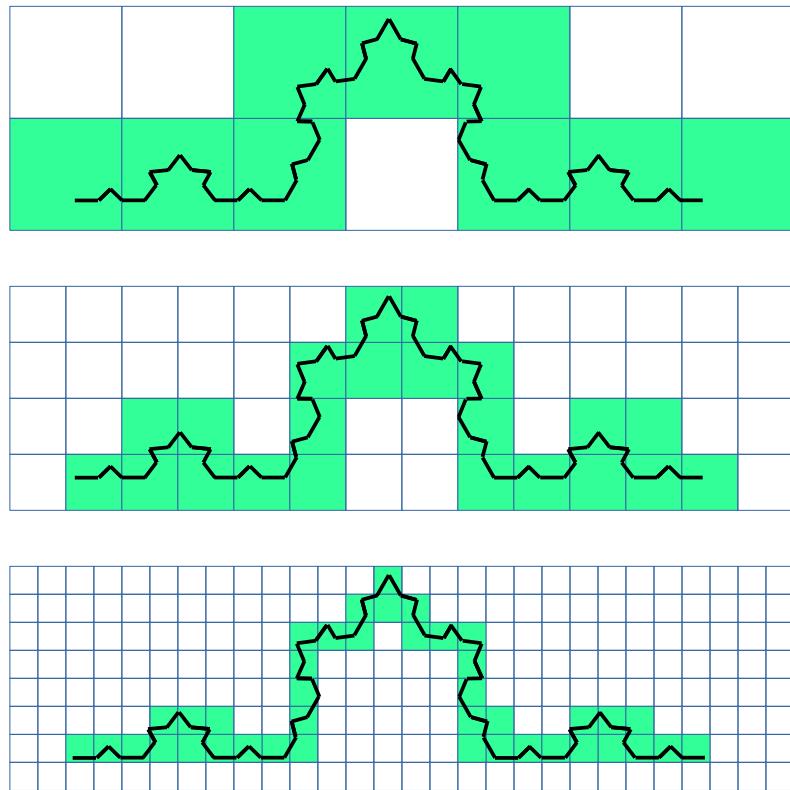


Figura 33: Cálculo de la dimensión d_{MB} del fractal de Koch por conteo de cajas.

La dimensión topológica de las hiperesferas y las hipercasillas debe ser estrictamente mayor que la dimensión fractal a medir. Por ejemplo, para el fractal de Koch habría que emplear círculos y casillas de 2 dimensiones. En la figura 33 podemos ver precisamente las iteraciones para $L=\{1,2,3\}$ para calcular la dimensión d_{MB} de este fractal usando conteo de casillas, y en la tabla 1 los cálculos correspondientes, donde N_L es el número de cajas que intersecan con el fractal, que están señaladas a color. Obviamente habría que iterar para sucesivos

12 Box counting.

valores de L , hasta que los resultados converjan.

La técnica de conteo de casillas¹³ es muy apropiada para cualquier tipo de fractal, incluidos los que se encuentran en la naturaleza.

L	N_L	log(N_L) / log(2^L)
1	9	3.17
2	22	2.23
3	41	1.79

Tabla 1: Cálculo de la dimensión d_{MB} del fractal de Koch por conteo de cajas.

Dado que para construir este tipo de fractales hay que iterar infinitas veces, estos objetos tienen infinita rugosidad, infinitos detalles y, habitualmente, un perímetro infinito. Por ejemplo, si calculamos el borde del copo de nieve, suponiendo que el lado del triángulo inicial mide 1 (tabla 2) vemos que tiende a infinito conforme lo hace el número N de iteraciones.

Iteración	Número de lados	Tamaño de un lado	Perímetro
0	3	1	3
1	3*4	1/3	4
2	3*(4*4)	(1/3)*(1/3)	16/3
3	3*(4*4*4)	(1/3)*(1/3)*(1/3)	64/9
...
N	3*4 ^N	(1/3) ^N	3*(4/3) ^N

Tabla 2: Cálculo del perímetro del fractal copo de nieve de forma iterativa.

¹³ *Grid counting*. Las cajas (o esferas) y las casillas son el mismo concepto, salvo que las casillas están alineadas según una retícula, mientras que las cajas (o esferas) no, y se impone la restricción de minimizar el número de cajas (o esferas) necesario para hacer el recubrimiento.

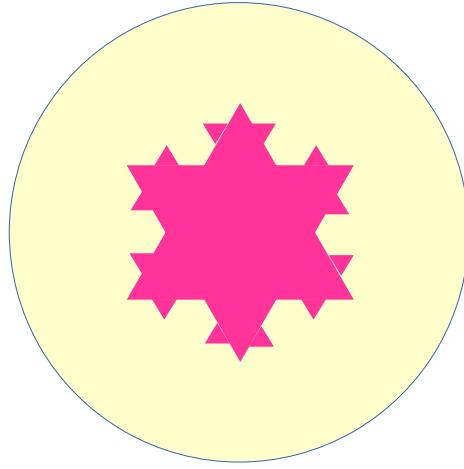


Figura 34: El área del fractal copo de nieve está acotada.

Y ello es particularmente extraño si consideramos que el área del copo de nieve es finita, dado que podemos dibujar un círculo de radio 1 centrado en el fractal, y el fractal no se sale de allí (figura 34). Por tanto, el área del fractal es menor a $\pi * 1^2 = \pi$.

¡Tenemos una superficie de perímetro infinito que encierra un área finita! Eso hace que estos objetos sean fascinantes y fueran tan difíciles de aceptar en su momento por la comunidad de matemáticos, que llegó a considerarlos monstruosidades.

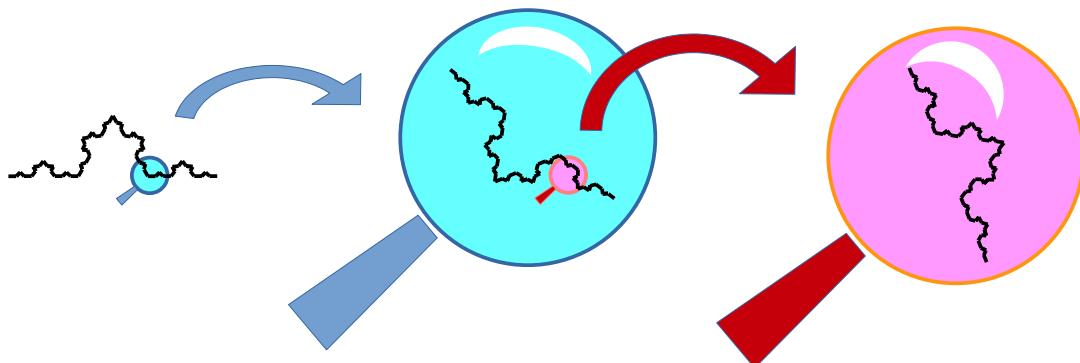


Figura 35: Invariancia frente a cambios de escala.

Los fractales son autosimilares, es decir, cualquier parte del objeto es idéntica a cualquier otra parte, salvo algún giro o desplazamiento que se deba hacer para que coincidan. Y también son invariantes frente a cambios de escala, es decir, si los miramos con un microscopio a diferentes aumentos, no podemos deducir en qué nivel de zoom estamos, basados en la imagen que se obtiene. Cualquier escala nos muestra lo mismo (figura 35).

Las funciones iterativas se pueden implementar usando gramáticas de Lindenmayer, también llamadas *sistemas-L*, que consisten en una gramática con variables, terminales, un símbolo inicial y reglas de producción. Y donde las variables y terminales se traducen finalmente a una acción en un dibujo. Por ejemplo:

Variables = {A, B}
Terminales = {}
Símbolo inicial = A
Reglas de producción:
 $A \rightarrow B$
 $B \rightarrow AB$

Figura 36: Una gramática de Lindenmayer.

Al iterar en esta gramática sale lo siguiente, donde podemos constatar que si contamos el número de símbolos de cada nivel obtenemos la secuencia de Fibonacci 1,1,2,3,5,8,13...:

A
B
AB
BAB
ABBAB
BABABBAB
ABBABBABABBAB

Figura 37: Secuencia resultante.

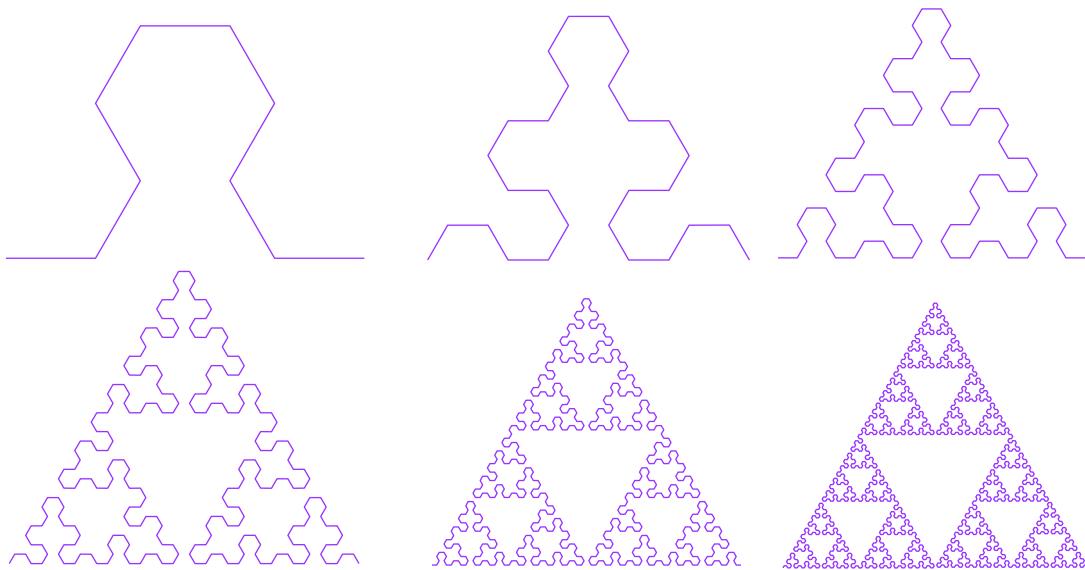


Figura 38: Sucesivas aproximaciones al triángulo de Sierpinski, usando una gramática de Lindenmayer.

El triángulo de Sierpinski (figura 38) se puede construir por medio de la gramática¹⁴ que se encuentra en la figura 39.

Variables = {A, B}
 Terminales = {D, I}
 Símbolo inicial = A
 Reglas de producción: $A \rightarrow DBIAIBD$
 $B \rightarrow IADBDAI$

donde aquí también se utilizan los “movimientos de la tortuga” del lenguaje LOGO, siendo:

- A: avanzar un paso
- B: también significa avanzar un paso
- I: girar 60 grados a la izquierda
- D: girar 60 grados a la derecha

Figura 39: Gramática para producir Sierpinski.

Y una planta más realista se puede construir con la gramática de la figura 40.

¹⁴ El código que genera el dibujo directamente en LibreOffice se puede encontrar en <https://github.com/angarciaiba/libroVA>.

Variables = {X, A}
 Terminales = {D, I, [,]}
 Símbolo inicial = X
 Reglas de producción: $X \rightarrow AI[[X]DX]DA[DAX]IX$
 $A \rightarrow AA$
 donde aquí también se utilizan los “movimientos de la tortuga” del lenguaje LOGO, siendo:
 A: avanzar un paso
 I: girar 25 grados a la izquierda
 D: girar 25 grados a la derecha
 X: una expresión de la gramática que no se traduce a ningún movimiento
 [: salvar en un *stack* el estado actual, es decir, la posición (x,y) y el ángulo de la tortuga
]: recuperar del *stack* la nueva posición actual

Figura 40: Gramática para construir una planta.

Cuyo resultado vemos en la figura 41.



Figura 41: Planta fractal generada con una gramática de Lindenmayer, después de 6 iteraciones.

Fuente: Sakurambo - Own work, CC BY-SA 3.0. Disponible en <https://commons.wikimedia.org/w/index.php?curid=2115456>

Cualquier algoritmo de tipo recursivo puede dibujarse gráficamente de modo que aparezca un fractal. Un ejemplo es el juego de las torres de Hanoi.

Problema 1: TORRES DE HANOI

El juego consiste en 3 torres, cada una con N anillas de tamaños crecientes hacia arriba. El objetivo es pasar todas las anillas de la torre de la izquierda a la torre de la derecha (usando la torre central como lugar auxiliar para hacer maniobras), siguiendo estas reglas:

- Solo se permite mover las anillas de una en una (no puedes tener en tus manos dos anillas a la vez).
- Las anillas pequeñas pueden situarse sobre las anillas grandes, pero no al revés.

Fractales de tiempo de escape

¿Una ecuación tan sencilla como la siguiente puede producir un objeto tan complejo como el de la figura 42?

$$\begin{aligned} z &\leftarrow 0 \\ z &\leftarrow z^2 + c \end{aligned}$$

Ec. 11

Resulta que sí. Los números z y c tienen parte real y parte imaginaria, y la figura está dibujada en el plano complejo. El algoritmo es el siguiente: cada punto c del plano complejo se pinta de un cierto color en función de lo que ocurra al inyectar ese número en la ecuación 11. Si después de iterar muchas veces el valor de z no diverge (oscila o se mantiene constante o tiende a un valor pequeño), entonces el punto c se pinta de color negro y se dice que pertenece al conjunto de Mandelbrot. Los puntos c que producen que z diverja se pintan de otro color, según algún patrón conforme más rápido diverjan.

Por ejemplo, para $c=0$, la sucesión de valores de z sale siempre 0, por lo que ese valor de c se pinta de negro.

Otro ejemplo: para $c=i$, entonces z toma sucesivamente los valores $\{0, i, -1+i, -i, -1+i, -i, \dots\}$, es decir, oscila, por lo que ese punto también se pinta de negro.

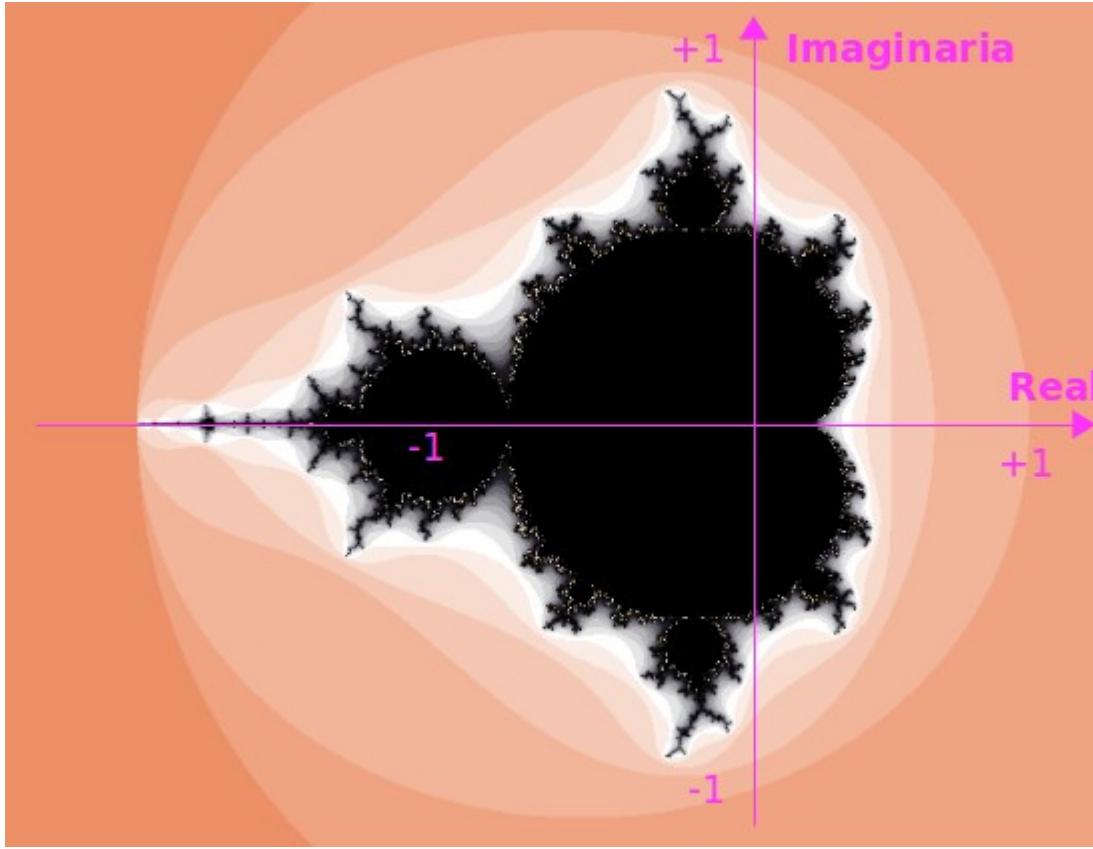


Figura 42: Fractal de Mandelbrot.

Otro ejemplo más: para $c=2$, entonces z toma sucesivamente los valores $\{0, 2, 6, 38, 1446, 2090918\dots\}$ que son rápidamente divergentes, por lo que se pintará de un color muy claro.

Y un último ejemplo: para $c=0.2+0.3i$ entonces z toma sucesivamente los valores $\{0, 0.2+0.3i, 0.15+0.42i, 0.0461+0.426i\dots\}$ y termina estabilizándose en $0.079+0.356i$ aproximadamente. Este punto c también hay que pintarlo de negro.

El conjunto de Mandelbrot es un fractal famoso al que se le han descubierto algunas propiedades:

- Es conexo, es decir, se puede trazar una línea continua desde cualquiera de sus puntos hasta cualquier otro sin salirse del conjunto. Y su complementario también es conexo.
- El borde (el límite entre los puntos que sí son del conjunto y los que no son) no es computable. Se puede uno aproximar tanto como quiera al borde, dejando que el computador corra más y más tiempo, pero nunca se obtendrá un punto con todas sus cifras decimales que esté exactamente en el borde. Este borde es lo que constituye el fractal en sí.

- La longitud de su borde es infinita, mientras que su área es finita.
- La dimensión fractal del borde es 2. Este resultado no es obvio y se requiere mucha matemática para demostrarlo.
- Es aproximadamente autosimilar pues muestra una invariancia frente a cambios de escala: al hacer *zoom* sobre cualquier parte del borde aparecen más figuras que se parecen mucho al conjunto original (figura 43).

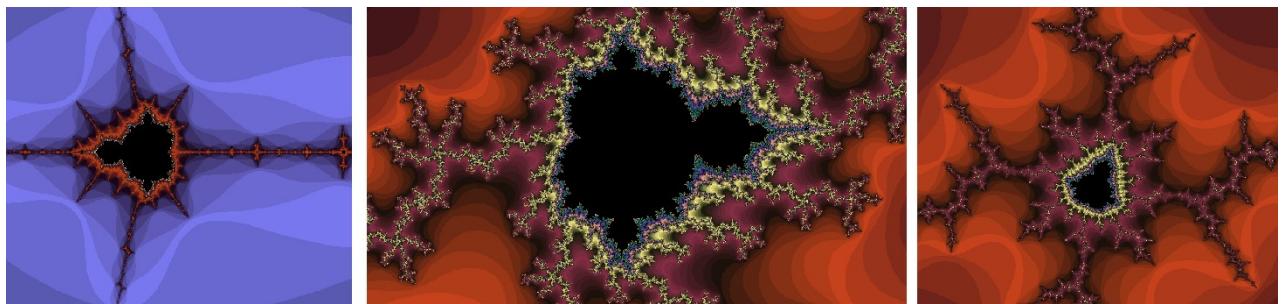


Figura 43: En los interiores de Mandelbrot.

Hay un software especialmente bueno para explorar este y otros fractales. Se llama *XaoS*, es gratuito, corre sobre cualquier plataforma y en el menú de ayuda trae unos tutoriales que nos enseñarán más cosas del mundo de los fractales.

A partir de cada punto del conjunto de Mandelbrot se puede construir otro fractal llamado conjunto de Julia, usando la misma fórmula iterativa: $z \leftarrow z^2 + c$, siendo c y z números complejos, pero ahora c es un punto elegido del conjunto de Mandelbrot. Los valores de z que hacen que la fórmula iterativa no diverja pertenecen al conjunto de Julia para ese valor c . Por eso se dice que Mandelbrot representa un catálogo de fractales (cada punto de Mandelbrot se transforma en un conjunto de Julia, como se ve en la figura 44).

Por ejemplo, para el valor de Mandelbrot $c=-1$ resulta que $z_{INICIAL}=0$ sí pertenece al correspondiente conjunto de Julia, ya que la fórmula $z \leftarrow z^2 - 1$ oscila, dando lugar a valores de $z=0, -1, 0, -1, 0, -1\dots$ Es decir, no diverge. Por tanto, pertenece al conjunto. Además, la secuencia es de periodo 2.

Este conjunto de Julia tiene dos puntos fijos. Salen de hacer $z^2 - 1 = z$ y de calcular las dos raíces. Son puntos fijos porque se transforman en ellos mismos, al iterar.

Como curiosidades, podemos decir que el conjunto de Mandelbrot consiste en todos los valores de c tales que $Julia(c)$ está conectado. O también que es la órbita de $z=0$ para la que z^2+c no tiende a infinito. La órbita de $z=0$ sabe cuál es la forma del conjunto de Julia para z^2+c , y la razón de ello es que $z=0$ es el punto

crítico de z^2+c , ya que la derivada de z^2+c es $2z$, que solo se anula para $z=0$.

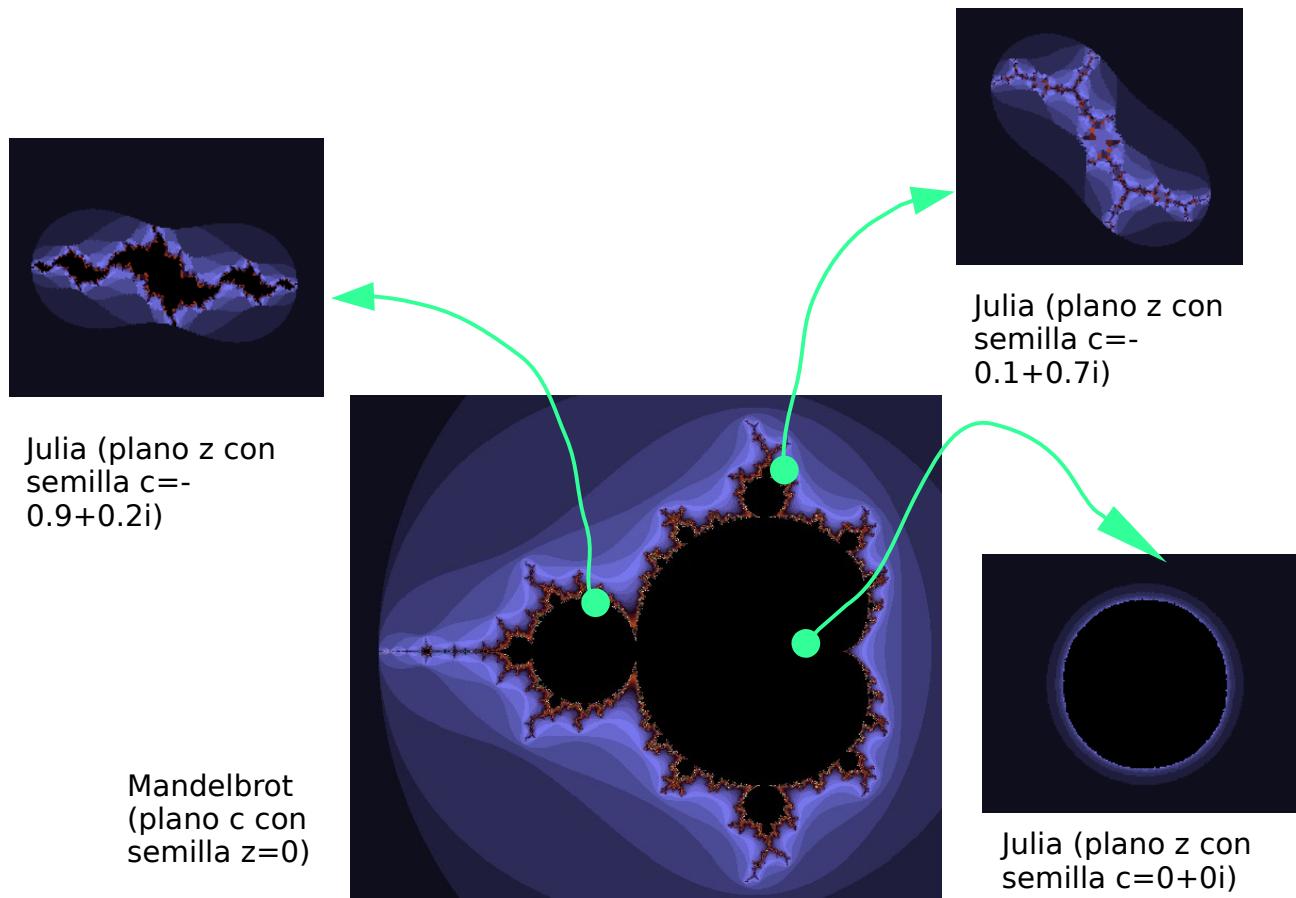


Figura 44: Mandelbrot como catálogo de Julias.

En Internet hay competencias para ver quién logra mayor profundidad de *zoom* en este bonito fractal (ver SethComposerGuy, 2013).

El conjunto de Julia con $c=-0.12+0.75i$ se llama “conejito de Douady” (figura 45) porque tiene un cuerpo y un par de orejas. Claro que cada oreja se bifurca en otro par de orejas, hasta el infinito. Este conjunto de Julia tiene infinitos puntos de unión en los cuales se intersecan tres regiones. Ello es debido a que este valor de c corresponde a un ciclo de periodo 3 en el conjunto de Mandelbrot. Hay un teorema de Julia y Fatou de 1919 que afirma que, dado un valor de c , el conjunto de Julia bien es conexo o bien es un conjunto de Cantor (de puntos aislados). Y existe una forma sencilla de saber en qué caso estamos: si el valor de c está dentro del conjunto de Mandelbrot, entonces el correspondiente conjunto de Julia es conexo. En caso contrario es un conjunto de Cantor.

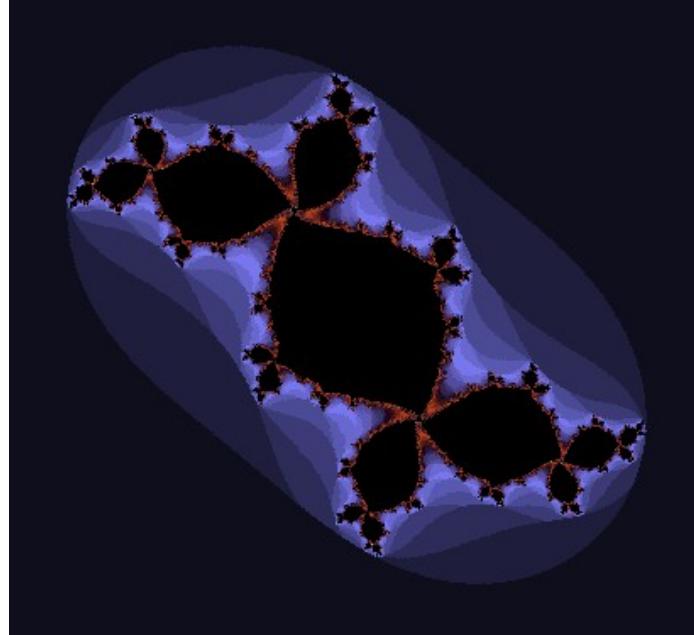


Figura 45: Conejito de Douady.

Es curioso ver que esta transformación pasa del mundo analógico al digital, con todo conectado a nada conectado, como se muestra en la figura 46.

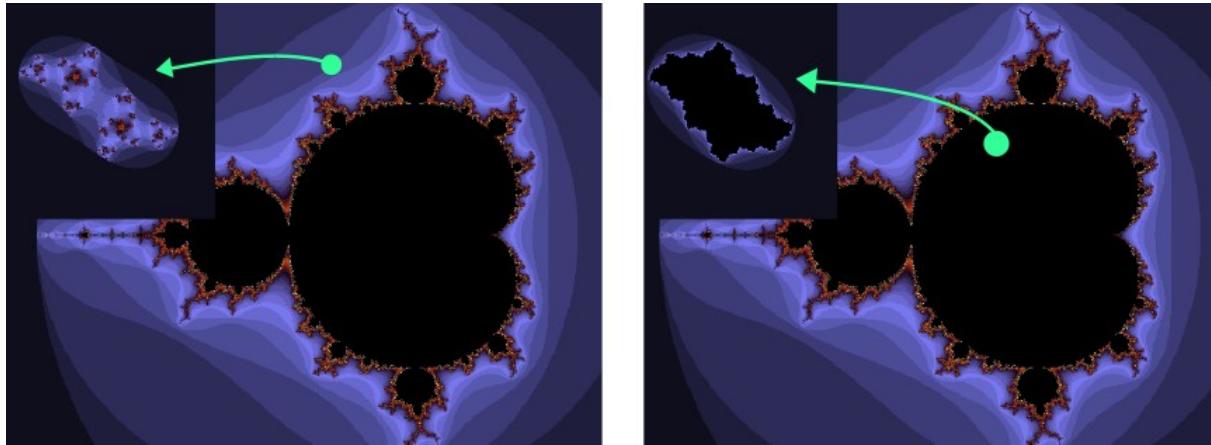


Figura 46: Julia inconexo y Julia conexo.

Los puntos del conjunto de Mandelbrot no divergen. Eso significa que o bien convergen o bien oscilan. En el caso de que oscilen, lo hacen con un cierto periodo que se puede hallar al observar la figura de Mandelbrot en los alrededores del punto en cuestión: si hay N ramificaciones, entonces el periodo es N .

En general, si se escoge un punto c dentro del conjunto de Mandelbrot cuyo periodo sea N , su conjunto de Julia correspondiente tendrá infinitos puntos de unión en los que se intersecan N regiones. Localmente Julia y Mandelbrot son similares (figura 47).

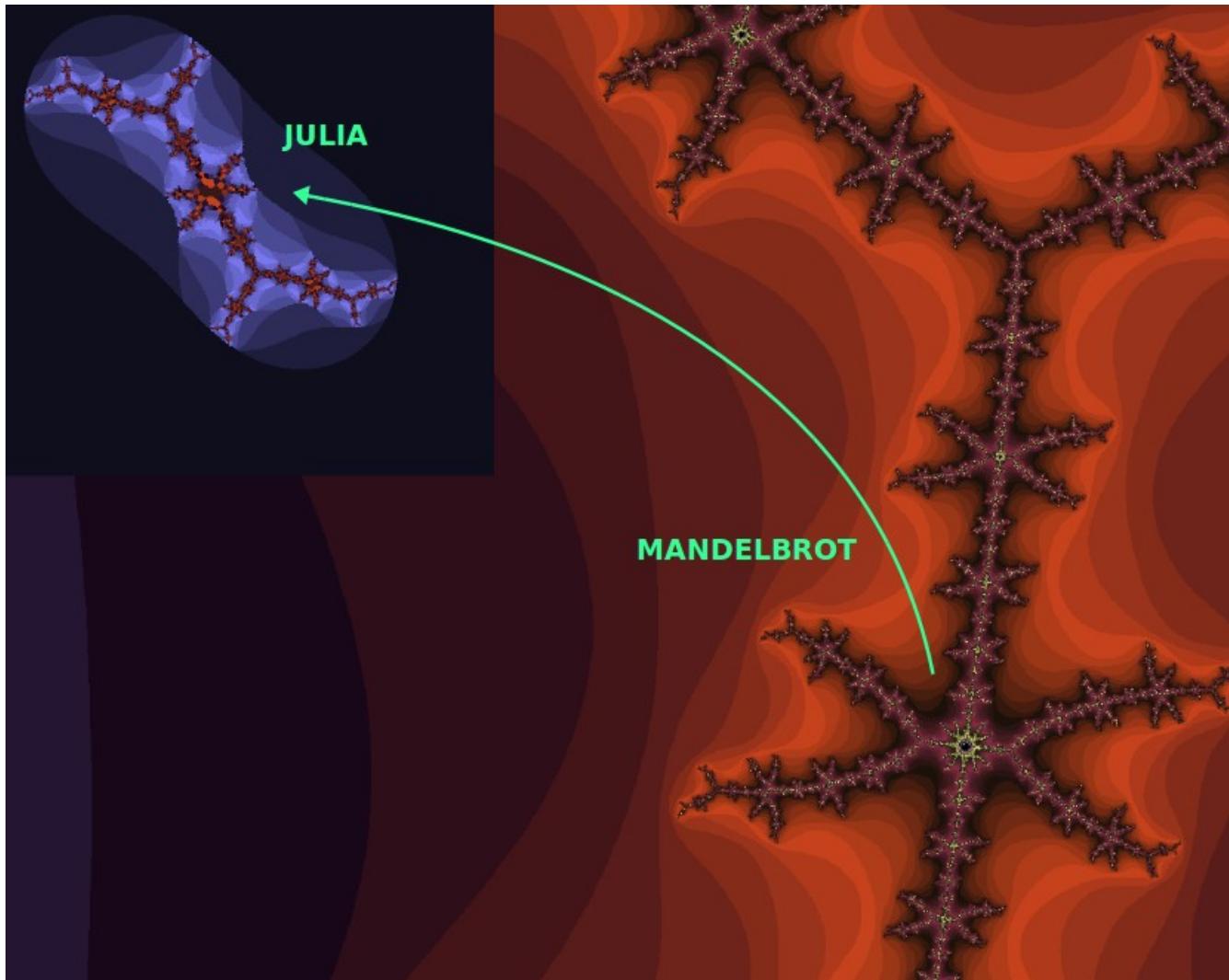


Figura 47: Localmente, Julia y Mandelbrot se parecen.

Hay muchas variantes que se pueden hacer sobre el fractal de Mandelbrot, que dan lugar a mundos fascinantes definidos en tres dimensiones, como los Mandelhub (Marczak, 2010) y los Mandelbox (Fractal universe, 2016). Y se hacen verdaderas obras de arte basándose en ellos, como Bib993 (2011).



Personaje 1

BENOÎT MANDELBROT (1924-2010)

Matemático polaco con nacionalidad francesa y estadounidense. Popularizó y les dio nombre a los fractales —probablemente inventados por Cantor—. La palabra “fractal” viene del latín *fractus* que significa roto, quebrado, fraccionado. Y esa es la idea que él quiso transmitir para estos objetos. A Mandelbrot se le debe el fractal que lleva su nombre.

Fuente: CC BY-SA 2.0 fr, Rama (2007).
Disponible en:
<https://commons.wikimedia.org/w/index.php?curid=4831595>

Mandelbrot escribió varios libros sobre fractales, siendo el primero y más popular *La geometría fractal de la naturaleza*. Pero uno más interesante llamado *Fractales y finanzas* que, si lo logramos develar, nos permitirá

entender las series de datos económicos como fractales autosimilares. Y así ¡nos podemos hacer ricos tratando de predecir la bolsa o el cambio entre monedas! Al parecer, ya ha habido investigadores que han hecho fortuna de esta manera. Sin embargo, el problema es que conforme la información se comparte entre más personas, el factor sorpresa desaparece. Por ejemplo, si usando estas técnicas todos nos damos cuenta que una acción va a subir en la bolsa, todos compraremos esa acción antes de que ello ocurra, y la alta demanda hará, por las leyes de mercado, que el precio de la acción ya no suba. “Hay que correr mucho para mantenerse en el mismo sitio” dijo la Reina Roja a Alicia en el famoso libro “A través del espejo”, de Lewis Carroll.

Fractales estocásticos

En cualquiera de los fractales anteriores se pueden añadir perturbaciones estocásticas para eliminar las simetrías y dar un aspecto, a estos objetos matemáticos, más parecido a lo que encontramos en la naturaleza. Por ejemplo, si se añade ruido a las longitudes y los ángulos del fractal arbóreo, se obtiene el diseño de la figura 48.

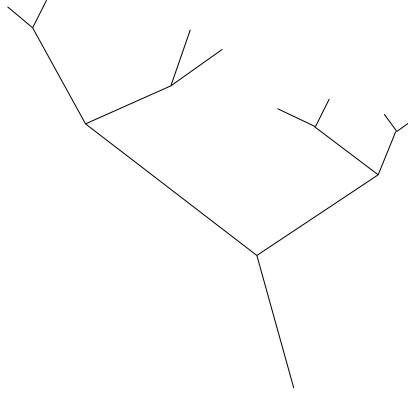


Figura 48: Fractal arbóreo estocástico.

Mandelbrot también propone el método del desplazamiento del punto medio como generador de curvas fractales que se asemejan las que encontramos en finanzas. Para ello se toma un segmento de recta y se le suma ruido *gaussiano* al punto de la mitad, con lo que se moverá hacia arriba o hacia abajo. Luego se continúa iterando recursivamente sobre los nuevos segmentos (figura 49).

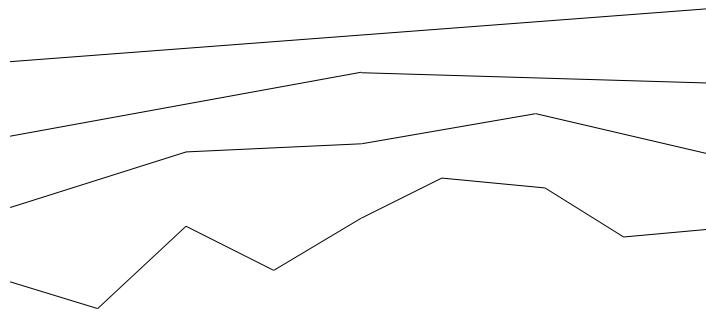


Figura 49: Fractal por desplazamiento del punto medio (se muestran 4 iteraciones, en sentido hacia abajo).

Sin embargo, también hay procedimientos aleatorios para generar fractales deterministas. Por ejemplo, partiendo de un triángulo equilátero y un punto elegido al azar dentro del triángulo. A continuación se pinta el punto intermedio entre el punto al azar y el vértice más cercano. Y luego se itera así: se elige un vértice al azar y se pinta el punto intermedio entre ese vértice y el último punto pintado. Conforme los puntos pintados se acumulan puede distinguirse el resultado, que es el triángulo de Sierpinski, al que se llega asintóticamente.

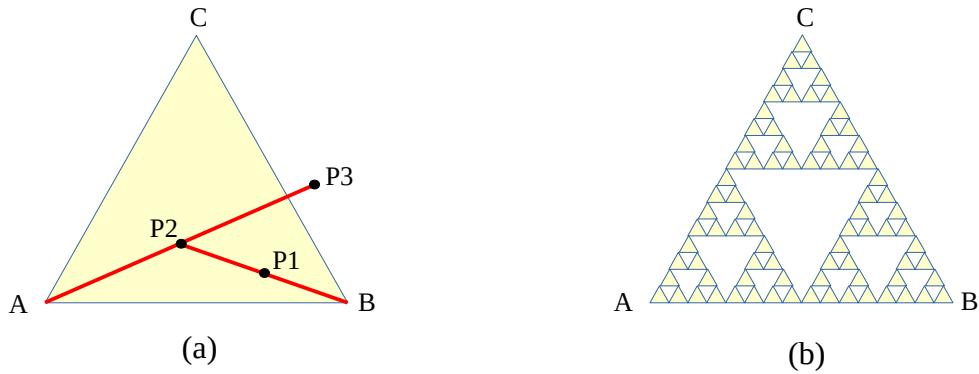


Figura 50: (a) Juego del caos; (b) estrategia ganadora.

Este método está basado en el “juego del caos”, que consiste en justo lo contrario: cada jugador elige un punto dentro del triángulo (figura 50-a), selecciona el vértice más cercano, y dibuja otro punto al doble de distancia de ese vértice, sobre la misma línea recta. Los jugadores se turnan en calcular los nuevos puntos y el primero que se ve obligado a poner uno de esos puntos por fuera del triángulo, pierde. El truco para no perder nunca es elegir un punto que pertenezca al fractal de Sierpinski (figura 50-b) tomando ese triángulo como base.

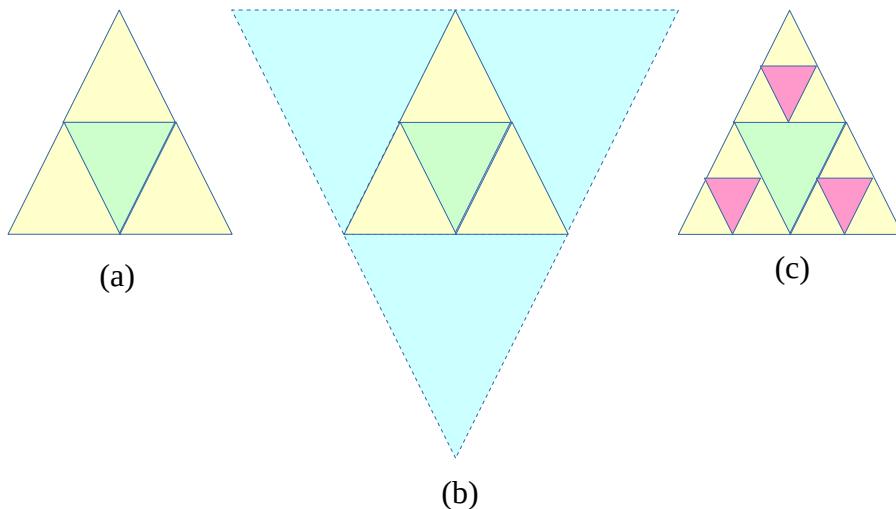


Figura 51: Demostración geométrica.

Para entender por qué esa es la estrategia ganadora, suponga que elige el punto inicial en la zona verde de la figura 51-b. Inevitablemente, el siguiente punto estará en una de las tres zonas azules de la figura 51-b y usted habrá perdido. Para evitarlo, nunca se debe elegir el punto inicial en la zona verde. Pero muchos puntos de las zonas amarillas tampoco son una buena elección, porque conducen en la siguiente jugada a la zona verde. Es el mismo argumento pero con triángulos de mitad de tamaño. Y por ello hay que descartar los puntos de las zonas rosadas de la figura 51-c. Iterando esta argumentación, iremos

construyendo el fractal de Sierpinski.

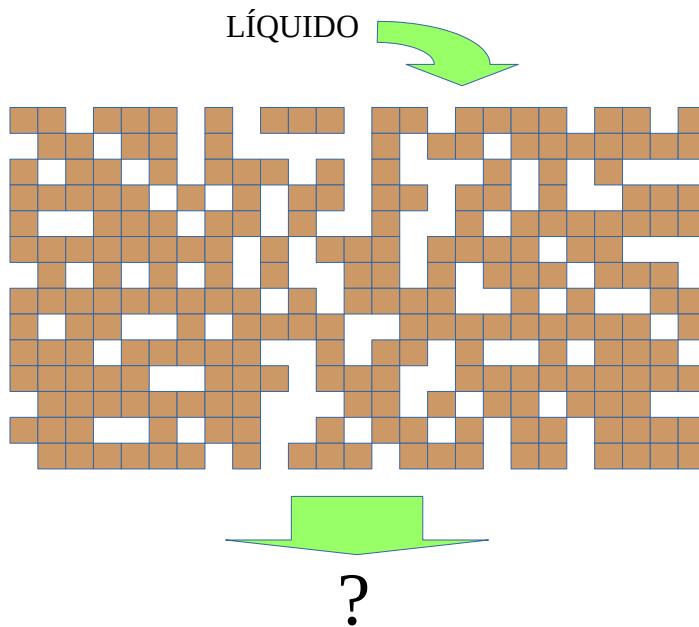


Figura 52: Retícula como modelo para estudiar la percolación.

Otro tipo de fractal aleatorio no determinista es el de percolación. Aquí tratamos de contestar la siguiente pregunta: si tenemos un material poroso (modelado por la retícula de la figura 52, donde las casillas coloreadas son el material y las blancas los poros), y echamos un líquido encima, ¿llegará hasta abajo?

Ocurre el mismo problema si tenemos un bosque y queremos saber si un incendio podría propagarse de un lado a otro, suponiendo que un árbol arde si está contiguo a otro árbol que está ardiendo. O si queremos estudiar cómo se propagan los rumores o las epidemias. O si queremos saber cuántos nodos de Internet deben fallar para que queden grandes partes de la red desconectadas. En estos últimos casos es mejor usar un grafo en vez de una retícula, pero la idea es la misma.

Si modelamos cada casilla de la retícula con una variable estocástica independiente de las demás casillas, digamos que hay material con probabilidad p y no hay nada con probabilidad $(1-p)$, entonces se sabe que hay una transición de fase en $p=1/2$. La probabilidad de que haya percolación disminuye exponencialmente para probabilidades menores a este valor, y aumenta para probabilidades mayores. Si en vez de la retícula se emplea un grafo, hay que averiguar dónde está ese umbral de forma experimental.

Pues bien, cuando borramos de manera estocástica casillas de una cuadrícula o arcos en un grafo, las estructuras resultantes son fractales, y se puede calcular su

dimensión con alguno de los métodos vistos, como el conteo de cajas.

Hay otro modelo fractal no determinista, muy relacionado con el anterior. Es el de difusión limitada (otros lo llaman de agregación-difusión), que también se puede generar sobre una retícula. Esta vez tenemos una casilla central fija y una casilla elegida al azar. De esta última sale una caminata aleatoria¹⁵ hasta que alcanza, por azar, a tocar la casilla central (recordemos que en una cuadrícula, cada casilla tiene 8 vecinas que la tocan). Entonces esa casilla queda también fija. Iniciamos entonces otro paseo aleatorio desde otra casilla elegida al azar. Aclaremos que los paseos no se guardan, sino solo la última casilla donde terminó el paseo, al tocar alguna de las casillas fijas. A las casillas fijas se le pegan (agregan) otras partículas que vienen de una caminata aleatoria. El resultado es un fractal con forma dendrítica, similar a los rayos eléctricos, al crecimiento bacteriano, a algunas plantas o al crecimiento de algunos cristales.

En la naturaleza se dan mucho estos tipos de fractales. El petróleo, por ejemplo, se encuentra en bolsas subterráneas que no son esféricas, sino fractales. Si taladramos el terreno encontramos el petróleo. Pero si la estructura no tiene percolación, entonces no podremos extraer mucho. Veamos otros ejemplos.

Fractales en la naturaleza

Como son objetos matemáticos, los fractales tienen infinitas rugosidades. Se puede hacer *zoom* sobre ellos de forma indefinida y siguen presentando el mismo aspecto. Por el contrario, con los objetos de la naturaleza siempre se presenta un límite al llegar a la escala atómica o antes. Sin embargo, muchos objetos naturales se pueden considerar fractales, haciendo la salvedad de que hay un límite de escala donde no tiene sentido seguir ampliando.

15 *Random walk* o movimiento browniano.



Figura 53: Brócoli.

La mayoría de los objetos naturales son fractales: los ríos, las nubes, las montañas, la red de arterias y venas en los animales, las conexiones de las neuronas, los vegetales como el brócoli (figura 53) —especialmente el *Romanescu*— y los helechos, los rayos y hasta las galaxias. Para entender por qué es así, primero consideremos objetos regulares como la esfera, el tetraedro o el cubo. La esfera es el objeto que minimiza su área para un volumen determinado y, por ello, es la forma geométrica que adopta el agua cuando nada la perturba. En ausencia de gravedad la gota se hace esférica porque, para un volumen de agua fijo, esa es la forma que minimiza su tensión superficial. Pues bien, los fractales hacen justamente lo opuesto: aumentan todo lo que pueden su área para un volumen determinado. Por ejemplo, podría pensarse que un pulmón es parecido a un balón esférico, que se llena de aire. Sin embargo, el área de una esfera es bastante pequeña y no habría manera de introducir rápidamente a través de ella ese oxígeno del aire en el cuerpo. Para aumentar el flujo de aire, el área debe hacerse mucho mayor y eso se consigue arrugándola sistemáticamente.

La evolución, que busca optimizar parámetros con el objetivo de hacer sobrevivir y reproducir los genes, encontró la forma de lograrlo: los bronquios se dividen en bronquiolos, que a su vez se vuelven a dividir, y así en unas 30 iteraciones, alcanzando al final a proporcionar 300 millones de alvéolos. La dimensión fractal de los pulmones es aproximadamente 2.9, o sea, bastante más que una superficie, casi alcanza a ser un volumen. Y el área total de los pulmones está entre 60 y 150 metros cuadrados¹⁶ ocupando apenas 6 litros.

¹⁶ Hay mucha dispersión en las medidas precisamente porque los fractales son divergentes, no se pueden medir con facilidad y no tiene sentido teórico su medida.



Figura 54: Hoja de helecho.

Otro ejemplo muy conocido es el helecho (figura 54), donde podemos observar autosemejanza en tres niveles (hojas pequeñas que forman hojas medianas que forman la hoja grande). Vamos a aprovechar para remarcar que los fractales son modelos, fórmulas matemáticas, donde se puede y se debe iterar infinitas veces. Mientras que los objetos del mundo real tienen limitaciones dimensionales, tanto en las escalas grandes como en las escalas pequeñas. Desde luego, no hay helechos más pequeños que un átomo, e incluso la autosemejanza se pierde aquí con apenas tres iteraciones. Hay quien los llama seudofractales.

Es muy razonable que la evolución seleccione formas fractales por dos motivos: por un lado se genera complejidad con bastante facilidad y a partir de muy poco cómputo, a base de iterar. La complejidad produce falta de predictibilidad, lo cual es necesario para sobrevivir, como veremos en el capítulo que habla de la libertad en el siguiente libro. Por otro lado, un fractal es una forma de comprimir información. De este modo, los genes tienen una forma económica de generar grandes estructuras usando poca información.

Otras razones son arquitecturales. La sangre debe llevar oxígeno y nutrientes a cada célula del cuerpo. Pero la sangre se mueve por tuberías unidimensionales, mientras que las células se reparten por un espacio tridimensional. ¿Cómo lograrlo? Pues con un fractal, dividiendo las arterias y venas en ramas sucesivamente, cada vez de un tamaño menor, hasta llegar a los capilares. La capa superficial del cerebro se llama el córtex y es la encargada de las funciones

más complejas de la mente. Para lograr aumentar mucho esta área, manteniéndola dentro de un volumen craneal de aproximadamente 1.5 litros, y poder interconectar fuertemente sus neuronas, la evolución también encontró que lo mejor era arrugar el córtex de forma sistemática, creando un fractal de dimensión 2.8 aproximadamente (Valerij, 2003).

Tanto es así que pueden detectarse enfermedades en una persona si la dimensión fractal de uno de estos órganos no coincide con los estándares.



Figura 55: Ramificaciones fractales de las begonias.

Las plantas tienen habitualmente un punto de apoyo en el suelo, de donde sale el tallo principal. No obstante, dado que para sobrevivir necesitan captar la mayor cantidad de luz posible, se ramifican tratando de minimizar que unas hojas tapen a otras, que las flores sean vistas desde muchos ángulos por los insectos y que faciliten el mayor intercambio de O_2 y CO_2 posible. Y todo ello sin afectar su solidez estructural. A su vez, las raíces desean capturar la mayor cantidad de agua y minerales en el suelo. Por ello es común ver estructuras fractales que se dividen en ramas cada vez más delgadas, como las de la figura 55.

Las costas (figura 56) y los ríos son otros ejemplos que incluía Mandelbrot en su primer libro. Si usamos una vara de 10 km para medir la longitud de un accidente geográfico, nos saldrá un valor. Si acortamos la vara a 1 km nos saldrá un valor mucho mayor, y conforme sigamos acortando el patrón de referencia, la longitud será cada vez mayor. Lo que es peor, no converge, sino que diverge hacia el infinito.

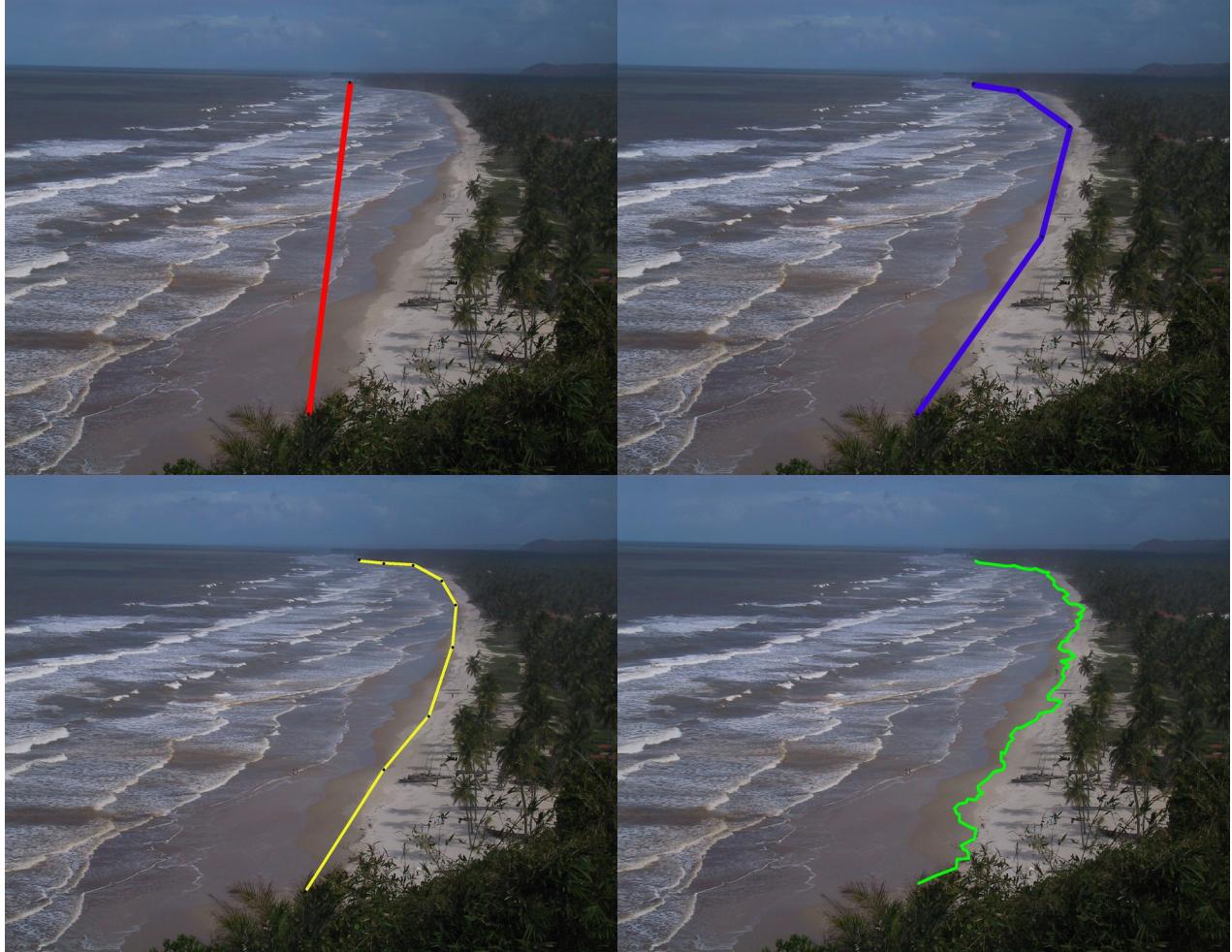


Figura 56: Medición de la línea de la costa en Itacaré, Brasil, con reglas progresivamente más pequeñas.

También vemos fractales en muchos objetos artificiales, como la gráfica a lo largo del tiempo del cambio de una moneda respecto a otra o la cotización de una acción en la bolsa. Esto no es una mera anécdota: algo importante habrá en los fractales cuando los sistemas típicos de generar complejidad finalmente terminan fabricando este tipo de objetos.

Espectro multifractal

Los objetos fractales que acabamos de ver tienen una única dimensión invariante a lo largo de todas sus escalas. Si ese no fuera el caso, si para cada escala hay una dimensión fractal distinta, entonces podemos calcularlas y generar así su espectro multifractal.

El espectro multifractal es a los objetos geométricos lo que el análisis de Fourier

es a las ondas de sonido. Si el espectro multifractal de un objeto es un único valor en todas las escalas, entonces el objeto es fractal. Si tiene un pequeño conjunto de valores, entonces se trata de un fractal dentro de otro, dentro de otro... y también tiene interés en cuanto a que comprime la información de cómo fabricarlo. Por ejemplo, tanto el ADN como las proteínas no son moléculas lineales, sino que en realidad cada una está plegada sobre sí misma. Y después de ello se vuelve a plegar sobre sí misma, y luego otra vez. Se han detectado varios plegamientos sucesivos, cada uno de los cuales es fractal, pero con dimensiones distintas. Pero si el espectro es continuo y complejo, entonces no nos dice mucho del objeto. Por ejemplo, si dos objetos tienen el mismo espectro multifractal, entonces podemos decir de ellos que... que... que tienen el mismo espectro multifractal. No hay mucho más que se pueda decir.

A veces se oye hablar de objetos multifractales, pero eso no es correcto, pues todos los objetos lo serían, ya que a todos se les puede calcular su espectro multifractal.

Desde el punto de vista matemático, un fractal comprime más información que un multifractal, es decir, con menos información genera más complejidad aparente. Pero los fractales son objetos abstractos pues en la realidad no existen estructuras con infinitas escalas. Llega un momento, tanto en las escalas más grandes (quizás el universo) como en las más pequeñas (quizás las partículas atómicas), donde la fórmula recursiva deja de funcionar. En ese aspecto el espectro multifractal modela mejor los objetos reales, pues nos dice hasta qué rango de escalas existe ese fractal, y lo que ocurre por fuera de ese rango.

En Obregón (2007) se hace una propuesta muy interesante para generar fractales, caos y series estocásticas a partir de un espectro multifractal dado, utilizando algoritmos genéticos y *simulated annealing*.

Resumen

Los fractales son objetos complejos que se caracterizan por su autosemejanza en muchas escalas, por tener una dimensión que puede no ser un número entero y por su infinita rugosidad.

Muchos sólidos regulares se producen cuando queremos minimizar el área ocupada por un volumen, a veces con el proceso sujeto a alguna otra restricción. Por ejemplo, si no hay ninguna otra restricción, se producirá una esfera a partir de

una cantidad dada de agua debido a que la tensión superficial intenta minimizar el área. Pero en otros casos lo que la naturaleza necesita es maximizar el área para un volumen dado. Y esto es lo que empuja a generar infinitas rugosidades en todas las escalas. Ejemplos de ello son los pulmones o el córtex cerebral.

En general, cuando la naturaleza (física, química, biología) se enfrenta a un problema irresoluble, la respuesta suele ser un fractal. Por ejemplo:

- Un tubo “desea” llevar sangre desde el corazón a todas las células de un cuerpo. Es decir, se necesita llenar un volumen con una línea, lo cual en principio es imposible. La naturaleza descubre que la forma de hacerlo es por medio de un fractal: el sistema circulatorio de arterias y venas.
- Un río o un rayo necesitan llevar su carga material o eléctrica lo más rápido posible a su destino. Como ello depende de las particularidades del medio que atraviesan —que puede cambiar de forma impredecible—, la respuesta es una trayectoria con forma fractal.
- Hay que aplanar una bola de maíz para formar una arepa. Si el proceso se hace simplemente aplastándola, sin amasarla continua y cuidadosamente, el resultado es un círculo plano con múltiples grietas de formas fractales, pues es imposible mapear sin distorsiones ni estiramientos una esfera sobre un círculo, como bien saben los cartógrafos.

Los fractales son importantes respecto al tema de la complejidad por dos razones:

- A partir de un algoritmo muy simple se generan objetos aparentemente muy complejos. Por ejemplo, a partir de un objeto que se sustituye por copias de él, con tamaños menores y quizás transformaciones simples (giros y desplazamientos). Definitivamente los fractales son una forma sencilla de generar complejidad. También, visto de otra manera, los fractales comprimen mucho la información geométrica de un objeto.
- Los fenómenos caóticos tienen siempre un fractal como órbita límite.

Para saber más

- **Heinz-Otto Peitgen y Dietmar Saupe (eds.). (1988). *The Science of Fractal Images*. New York: Springer-Verlag.**

Es un libro viejo y bonito, con contribuciones de científicos bien conocidos como Mandelbrot, Barnsley, Devaney. En este libro se explican las ecuaciones de muchos fractales, se presentan implementaciones de software y las imágenes resultantes, casi todas de alta calidad. Las aplicaciones que muestran son en síntesis de imágenes. Con ellas se pueden diseñar plantas, montañas, costas y planetas enteros. En las páginas centrales aparecen las mejores a color. Explica también las distribuciones $1/f$ y $1/f^2$ que fueron muy populares para crear música sintética. Y es de los pocos libros que habla sobre el controvertido parámetro de Hurst, que sirve para decidir si una serie de datos temporales es caótica, determinista o estocástica. La portada se hizo famosa pues aparece una parte del fractal de Mandelbrot pero artísticamente diseñado en 3D para simular montañas y acantilados.

- **Przemyslaw Prusinkiewicz y Aristid Lindenmayer (2004). *The Algorithmic Beauty of Plants*. New York: Springer-Verlag.**

Explica cómo programar la construcción de fractales usando la gramática de Lindenmayer, también llamada *L-system*. Comienza con sistemas 2D y continuos, como Sierpinski, la curva de Hilbert, Koch y otros similares, y continúa en 3D usando árboles y variables estocásticas para dar mayor realismo. Muestra muchas figuras de fractales generados así, en los cuales se aprecia la similitud con plantas reales. Introduce dos gramáticas de Lindenmayer, las libres de contexto y las sensibles al contexto, con diversas aplicaciones (estas últimas sirven para simular interacciones entre partes de la planta). Describe también una estructura fractal en árbol H que, aunque en el libro no lo dice, se emplea también en el diseño de *chips* en microelectrónica (yo la empleé en mi tesis doctoral), cuando se necesitan vías rápidas de comunicación para un gran conjunto de módulos de forma escalable, es decir, que se puedan añadir más módulos de manera indefinida. Más adelante propone dos modelos para explicar el crecimiento de algunas plantas: el planar (para girasoles, margaritas y similares) donde aparece las espirales de Fibonacci y el ángulo áureo, aunque no da para ello ninguna explicación biológica); y el cilíndrico (piñas, palmeras y similares), donde también aparecen las espirales. Después propone algoritmos para realizar animaciones del crecimiento de las plantas. Todo ello acompañado de las imágenes resultantes, muchas a color y bastante realistas.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

- Carroll, L. (2004). *A través del espejo*. Córdoba, Argentina: Ediciones del Sur.
- Goodwin, B. (1998). *Las manchas del leopardo*. Barcelona: Tusquets Editores.
- Gribbin, J. (2006). *Así de simple*. Barcelona: Crítica.
- Kiselev, V. G., Hahn, K. R. y Auer, D. P. (2003). Is the brain cortex a fractal? Academic Press. *NeuroImage*, 20, pp. 1765-1774.
- Mandelbrot, B. B. (1997). *La geometría fractal de la naturaleza*. Barcelona: Tusquets Editores.
- _____. y Hudson, R. L. (2006). *Fractales y finanzas*. Barcelona: Tusquets Editores.
- Monroy, C. (2002). *Curvas fractales*. México: Alfaomega.
- Obregón, N. (2007). Sistemas complejos, geofísica e ingeniería. En C. E. Maldonado (ed.), *Complejidad: ciencia, pensamiento y aplicación*, pp. 37-61. Bogotá: Universidad Externado de Colombia.
- Pagels, H. R. (1991). *Los sueños de la razón*. Barcelona: Gedisa.
- Schroeder, M. R. (1991). *Fractals, Chaos, Power Laws - Minutes from an Infinite Paradise*. New York: W. H. Freeman and Company.
- Stewart, I. (1997). *El laberinto mágico*. Barcelona: Crítica.
- XaoS (2017). *Software GNU XaoS*. Recuperado el 17 de julio de 2017. Disponible en: <http://matek.hu/xaos/doku.php>

PELÍCULAS Y VIDEOS

- Bib993 (2011). *Like in a dream II*. Recuperado el 26 de agosto de 2017. Disponible en: <https://www.youtube.com/watch?v=bKknJvZIn24>
- Fractal universe (2016). *Flying into the depth of Mandelbox*. Recuperado el 26 de agosto de 2017. Disponible en: <https://www.youtube.com/watch?v=QhMdL4kSnsq>

Langdon, P. (2009). *Fibonacci's Fractals*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=bE2EiI-UfsE>

Marczak, K. (2010). *Trip to center of hybrid fractal*. Recuperado el 26 de agosto de 2017. Disponible en: https://www.youtube.com/watch?v=E91yxk_pT_A

Poincarep (2007). *Conjuntos de Julia*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=CcqGJcnvJ-g>

SethComposerGuy (2013). *Mandelbrot fractal deep zoom 21 2^1116 HD*. Recuperado el 26 de agosto de 2017. Disponible en: <https://www.youtube.com/watch?v=PbwaFQ2r2c4>

TESIS Y TRABAJOS DE GRADO EN EVALAB

Torres, A. R. (2010). *Objeto virtual de aprendizaje para la teoría del caos y su relación con los fractales*. Cali: Universidad del Valle.

CAOS

“The future ain't what is used to be”
Yogi Berra

Cuando en conversaciones cotidianas se habla de caos, generalmente se hace alusión a algo desordenado o estocástico¹⁷. No obstante, aquí “caos” es un término técnico con una definición formal que vamos a ver enseguida. Para hacer énfasis en la diferencia, cabe señalar también que hay quienes lo llaman “caos determinista” o “caos matemático”, queriendo así decir que no tiene que ver con lo estocástico. Es una buena idea aunque, para no hacer el texto muy pesado de leer, lo llamaremos simplemente “caos”.

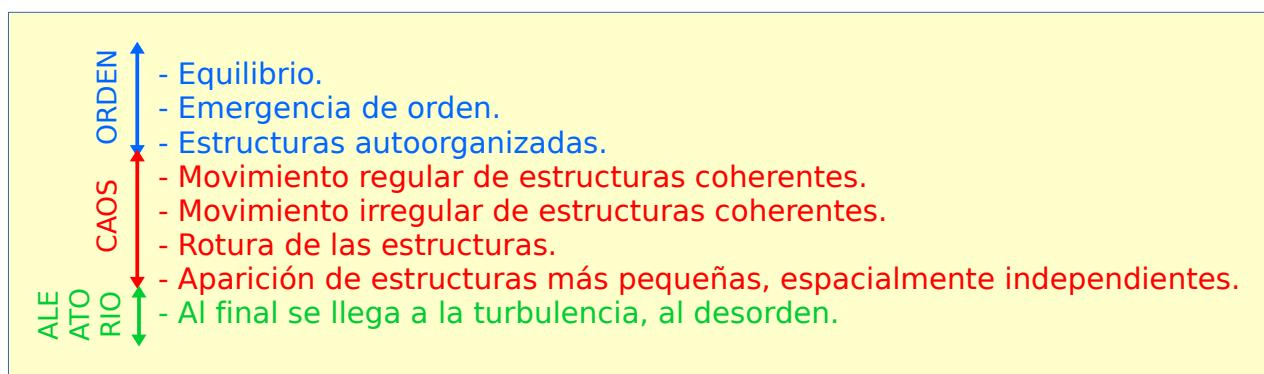


Figura 57: Transiciones del orden a la aleatoriedad pasando por el caos.

En el mundo real vemos muchos fenómenos que exhiben comportamientos a veces ordenados y predecibles, a veces desordenados e impredecibles. Hay una amplia gama de estos comportamientos. Podemos ver cosas que crecen y decrecen, que se estancan, que repiten lo mismo una y otra vez, o que fluctúan de manera errática. Los fenómenos caóticos están entre medias de los

completamente ordenados y los completamente estocásticos (figura 57). Pero caos no es sinónimo de desorden, porque bajo la apariencia de una gran complejidad se esconde típicamente una ecuación o un modelo de comportamiento muy simple, cuya evolución en el tiempo es compleja. El caos es una forma barata de generar complejidad a partir de casi nada.

Es importante esclarecer esa diferencia entre procesos caóticos y estocásticos, especialmente porque en muchos textos se les confunde. En la tabla 3 vemos algunos ejemplos que nos ayudarán a entender las diferencias.

	Orden	Caos	Estocástico
Ejemplo	Reloj	Nubes, clima	“Nieve” en un televisor sintonizado donde no hay canales
Predecibilidad	Muy alta	Finita, a corto plazo	Ninguna
Efecto de perturbarlo	Pequeño	Explosivo	Ninguno
Espectro	Armónico	Finito continuo	Muy ancho
Dimensión	Finita	Baja	Infinita
Control	Fácil	Difícil, pero efectivo	Imposible
Atractor	Punto, línea, toro...	Fractal	Ninguno

Tabla 3: Comparación entre sistemas ordenado, caótico y estocástico.

Y hay otro concepto adicional a considerar: la seudoaleatoriedad, que es muy parecida al caos. Entonces los eventos del mundo pueden ser (figura 58):

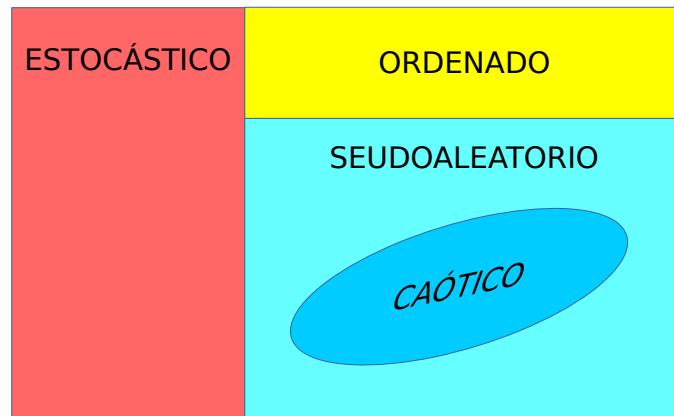


Figura 58: El mundo y sus causas.

- Estocásticos: si no tienen ninguna causa. No hay ningún algoritmo que genere los resultados, que aparecen al azar.
- Ordenados: si la causa es fácilmente entendible y el resultado es predecible y repetible. El algoritmo que convierte causas en resultados se conoce o puede llegar a conocerse con facilidad. Y es determinista, es decir, no

interviene el azar.

- Seudoaleatorios: si la causa no es fácilmente entendible, y el resultado es repetible solo si se conocen exactamente las causas (condiciones iniciales). El algoritmo que convierte causas en resultados existe y es determinista, pero probablemente no se conozca.
- Caóticos: si son seudoaleatorios y además cumplen otras tres condiciones que veremos enseguida: sensibilidad, densidad y mezcla.

Para complicar las cosas, puede haber eventos compuestos de varios de los anteriores.

Los fenómenos estocásticos son no deterministas, lo cual significa que ocurren sin que haya otro fenómeno desencadenante. Ocurren sin causa, al azar, de forma impredecible.

Los fenómenos ordenados, caóticos y seudoaleatorios son deterministas, es decir, hay una causa desencadenante fija tal que si repetimos las mismas condiciones iniciales se volverá a repetir el mismo fenómeno. La diferencia entre ellos es que en los ordenados la causa es conocida, fácil de entender y de predecir, mientras que en los otros dos no es así. Lo cierto es que si desconocemos el algoritmo que genera una secuencia seudoaleatoria, ella se comporta prácticamente igual que una secuencia estocástica.

Una pregunta filosófica importante: ¿existe realmente algún fenómeno estocástico? ¿O solo es nuestro desconocimiento el que los clasifica así? Después de todo, hace cientos de miles de años la salida del sol o los eclipses eran acontecimientos mágicos que, con el tiempo, fueron convirtiéndose en fenómenos ordenados (primero la salida del sol y mucho más tarde los eclipses), pues ese es el objetivo de la ciencia: convertir lo que parece estocástico en algo entendible y predecible.

Hoy en día se piensa que los únicos fenómenos verdaderamente estocásticos son los de la física cuántica. Sin embargo, también hay quienes creen lo contrario, es decir, que detrás de ellos hay fenómenos deterministas, según la teoría de las variables ocultas de David Bohm y otros. Los experimentos realizados en años recientes sobre la desigualdad de Bell han demostrado que no existen esas variables ocultas, pero incluso así aún queda otra posibilidad: que las variables ocultas sean caóticas¹⁸ y no exista nada verdaderamente estocástico. De modo que la pregunta sigue abierta.

18 Ver, por ejemplo, Thompson (2004).

Por otro lado, esta discusión es muy similar a la que damos en el capítulo sobre la libertad, del siguiente libro: ¿somos libres? No lo sabemos y no importa. Nos basta una definición relativa de libertad. Y aquí, en el tema del orden y del desorden, nos basta una definición relativa de predictibilidad.

De hecho, en ciencia deberíamos prescindir del concepto “estocástico” pues tiene algo de mágico. Veamos por qué: en un computador no hay forma de crear secuencias de números realmente estocásticas, a no ser que las traigamos desde fuera del sistema¹⁹. Si dentro del computador creamos un mundo virtual, pasará lo mismo: la única manera de que tenga secuencias estocásticas es si las inyectamos desde fuera. ¿Será que en nuestro mundo real pasa lo mismo y lo estocástico, o bien no existe, o bien es una primitiva (no derivable del resto de cosas), inyectada desde fuera?

Piense que no son solo unas condiciones arbitrarias iniciales dadas en el momento del *Big Bang*. Se requiere un flujo continuo de información procedente de fuera de este mundo cada vez que un átomo decide o no desintegrarse, cada vez que un gato de Schrödinger decide o no morir, cada vez que un electrón salta por efecto túnel en nuestro computador. Es una cantidad enorme de información que tendría que venir desde fuera del universo. No parece razonable.

De todos modos, nadie lo sabe, y en el capítulo “La gradualidad de la emergencia” del siguiente libro propongo otra alternativa, donde lo que creemos que es estocástico es la diferencia entre las percepciones de nuestro nivel y las reglas que realmente están trabajando en el nivel inferior.



Figura 59: Clases de fenómenos.

Por todo ello, es más sensato pensar que no hay nada realmente estocástico, que todo es determinista (ordenado, caótico o seudoaleatorio) y que los sucesos del mundo tienen un horizonte de predicción que será mayor o menor en función de la sensibilidad intrínseca del fenómeno a las condiciones iniciales y del grado de inteligencia que tenga el ente que está realizando la predicción (ver figura 59 y tabla 4).

¹⁹ En Linux existe el flujo /dev/random que se puede abrir y leer como si fuera un archivo y que contiene sucesos externos como la distancia temporal entre pulsaciones de teclas, los eventos de *mouse* y las llegadas de paquetes de red, que han sido filtrados para eliminar las partes redundantes. Este flujo de datos se puede considerar realmente estocástico.

Conforme pasa el tiempo, los fenómenos que eran inicialmente percibidos como estocásticos, la ciencia los convierte en deterministas.

Nuestro CONOCIMIENTO		HORIZONTE DE PREDICCIÓN de fenómenos		
FÓRMULA O MODELO DE COMPORTAMIENTO	ESTADO INICIAL (con ruido)	SEUDOALEATORIOS	CAÓTICOS	ORDENADOS
Desconocido	Desconocido	nulo	nulo	nulo
Desconocido	Conocido	nulo	pequeño	grande
Conocido	Desconocido	nulo	pequeño	grande
Conocido	Conocido	nulo	grande	perfecto

Tabla 4: Epistemología del determinismo.

Lo que el físico Eugene Wigner llama “la increíble efectividad de las matemáticas para entender el mundo”, es solo una ilusión. Las matemáticas nos dan un conocimiento perfecto cuando sabemos la fórmula y el estado inicial de un fenómeno ordenado. En los demás casos tenemos apenas aproximaciones.

El estado inicial hay que considerarlo siempre con ruido, es decir, errores en la medición. Pero si no tuviera absolutamente nada de ruido (por ejemplo, dentro de un computador), entonces con conocimiento perfecto (última fila) tendríamos predicciones perfectas para todos los fenómenos.

En este sentido, los fenómenos seudoaleatorios también podríamos llamarlos “fenómenos criptográficos”, pues solo el conocimiento perfecto de todo nos permite entenderlos. Por ejemplo, cuando estamos tratando de averiguar una contraseña, si no la conocemos perfectamente —esto es, si conocemos todas las letras menos una—, no hay nada que hacer; no obtenemos acceso al sistema.

A continuación veremos cómo pueden ocurrir fenómenos deterministas en los que, a pesar de conocerse muy bien todos sus detalles, es imposible hacer predicciones a largo plazo en ellos. Son los fenómenos caóticos.

Un sistema caótico muy sencillo: la curva logística

Para que haya caos en un sistema es condición necesaria que exista al menos una realimentación positiva y una negativa. Y la forma más sencilla de obtener caos es usar ecuaciones que se aplican iterativamente. Es decir, dada $f(x)$ y un valor inicial x_0 , aplicar:

$$x_{n+1} = f(x_n)$$

Ec. 12

O, dicho de otra manera, observar la sucesión

$$x_0, f(x_0), f(f(x_0)), f(f(f(x_0))), f(f(f(f(x_0))))\dots$$

Ec. 13

con el objetivo de ver si surge algún patrón interesante. Por ejemplo, si $f(x) = x^3$, el resultado de aplicar iterativamente $f(x)$ converge a cero o a infinito, dependiendo de la semilla, pues solo hay una realimentación positiva. Ello no es especialmente interesante. Pero podemos hacernos ciertas preguntas que sí lo son:

- ¿Existe una $p(x)$ tal que $p(p(x)) = -x$ para todo x ?
- ¿Existe una $q(x)$ tal que $q(q(x)) = 1/x$ para todo x ?

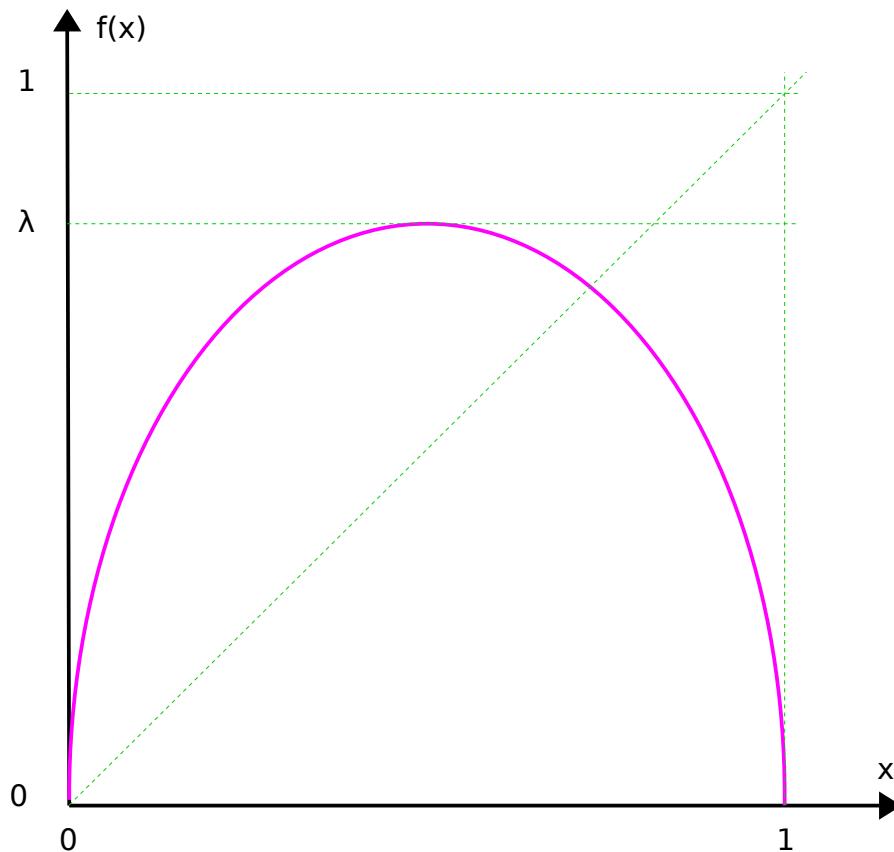


Figura 60: Curva logística.

Si existieran, ambas tendrían periodo 4. Y nos podemos preguntar en general, ¿qué tipo de funciones exhibirían comportamientos cíclicos? Desde luego, no las que son monótonas, como x^3 . Para que aparezcan comportamientos cíclicos se necesita una función no lineal, que crezca en algún tramo y que decrezca en otro. Por ejemplo, investiguemos la siguiente parábola (figura 60), conocida también como ecuación logística:

$$f(x) = 4\lambda x(1-x)$$

Ec. 14

Esta ecuación tiene un parámetro lambda (λ) ajustable entre 0 y 1. El origen de la ecuación viene de modelar el crecimiento de una población de animales que se reproducen a una tasa constante, pero que necesitan unos recursos para sobrevivir, y esos recursos son limitados.

Podemos considerar que hay dos bucles de realimentación (figura 61) actuando sobre la variable x (el tamaño de la población). Por un lado, hay una realimentación positiva cuando la población es pequeña: el crecimiento es exponencial debido a la tasa constante de reproducción. Inicialmente habrá una pareja; cuando se reproduzcan y tengan 2 crías, entonces habrá 2 parejas; cuando se reproduzcan esas 2 parejas habrá 4 parejas; luego 8, y así sucesivamente. Esto se modela con el término x . Por otro lado, hay una realimentación negativa cuando la población es grande: el término $(1-x)$ expresa que hay límites al crecimiento, esto es, que cuanto más se acerque la población a 1, el factor $(1-x)$ tenderá a 0, haciendo que la población disminuya por falta de recursos.

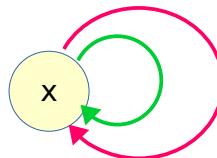


Figura 61: Bucle de realimentación en la ecuación logística (verde, positivo; rojo, negativo).

La ganancia de cada bucle no es constante, pues depende del tamaño de la población. Esto significa que habrá valores para los que predomine la realimentación positiva o la negativa o que ambos bucles se cancelen entre sí. Ello sugiere que el comportamiento de esta ecuación va a ser bastante sofisticado. Veámoslo. La ecuación discretizada queda:

$$x_{n+1} = 4\lambda x_n(1-x_n)$$

Ec. 15

Vamos a explorar esta ecuación con valores iniciales (o sea, la semilla) de x entre $0 \leq x \leq 1$ y valores de lambda entre $0 \leq \lambda \leq 1$. Esto es muy fácil de implementar usando una hoja de cálculo²⁰ o también gráficamente. Para cada valor de lambda vamos a ver unos patrones distintos que pueden exhibir un comportamiento periódico o caótico. Hay quien suele imaginar ese lambda como si fuera el dial de un sintonizador de radio con el que se capturan distintas emisoras, en este caso, distintos comportamientos del sistema. Aunque la verdad es que lambda está

20 Ver <https://github.com/angarciaiba/libroVA>

relacionado con la ganancia de ambos bucles de realimentación.

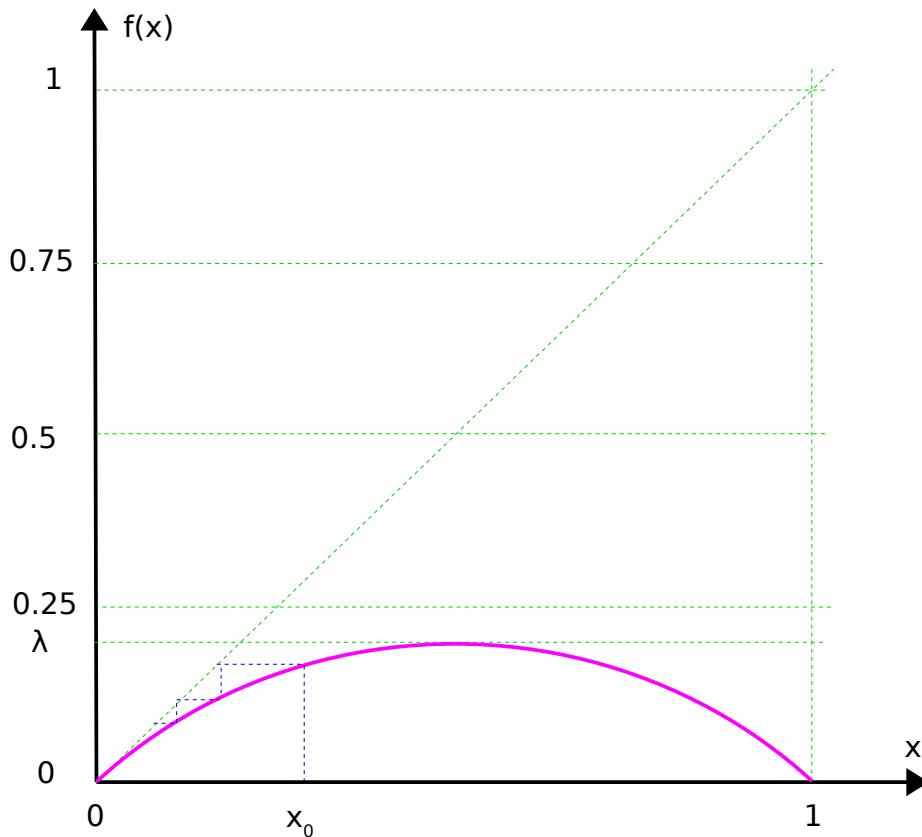


Figura 62: Curva logística con $\lambda < 0.25$.

Para valores de lambda pequeños (figura 62), concretamente con $\lambda < 0.25$, el resultado final tiende a 0, independientemente del valor semilla inicial de x , como se ve en la figura 63. Gráficamente, lo que se hace en la figura 62 es subir verticalmente desde x_0 hasta la curva $f(x)$, de allí se proyecta al nuevo valor de x , horizontalmente hasta la diagonal, y desde allí verticalmente hasta la curva, repitiéndose todas las veces que se quiera.

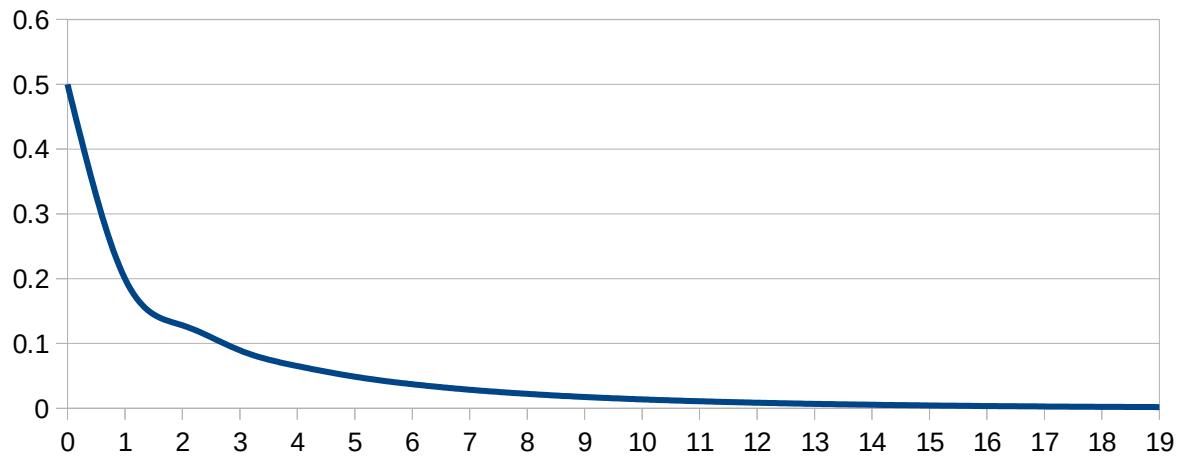


Figura 63: Serie temporal de la curva logística con $x_0=0.5$ y $\lambda=0.2$.

Pero si aumentamos λ un poco por encima de 0.25, el valor final (A_1) es distinto a 0 como puede verse en las figuras 64, 65 y 66.

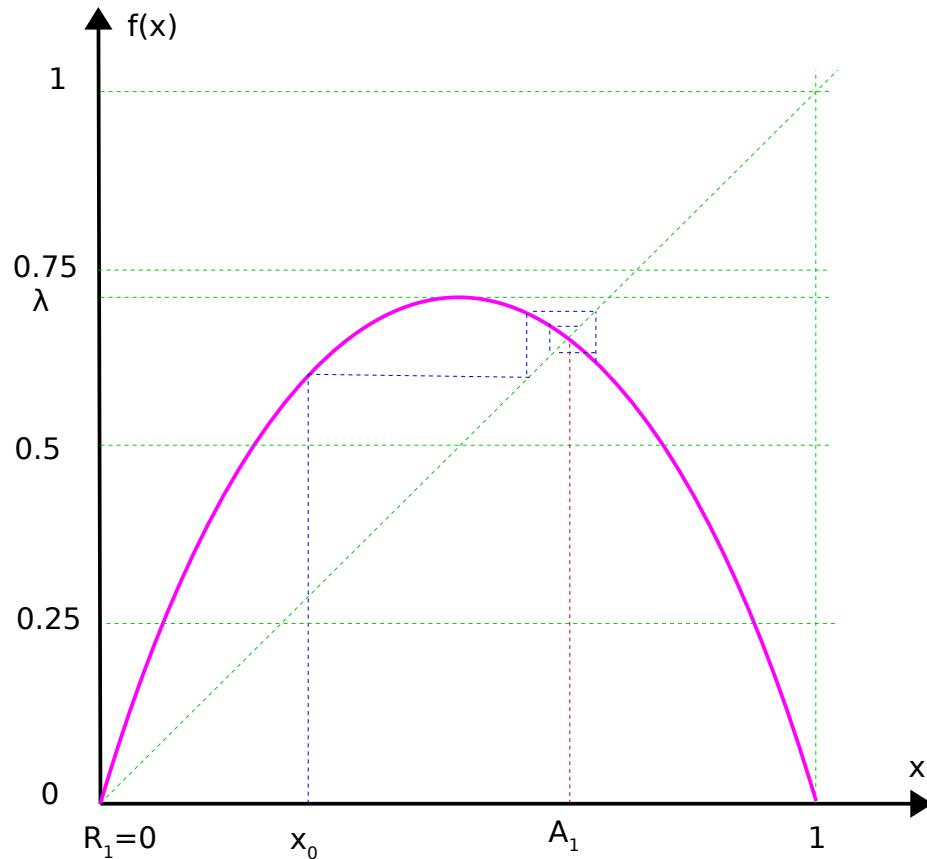


Figura 64: Curva logística para $0.25 < \lambda < 0.75$.

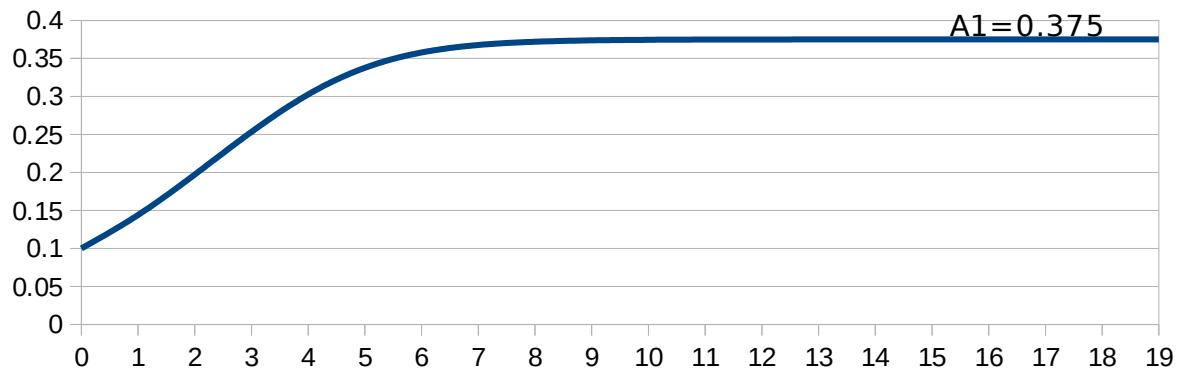


Figura 65: Serie temporal de la curva logística con $x_0=0.1$ y $\lambda=0.4$.

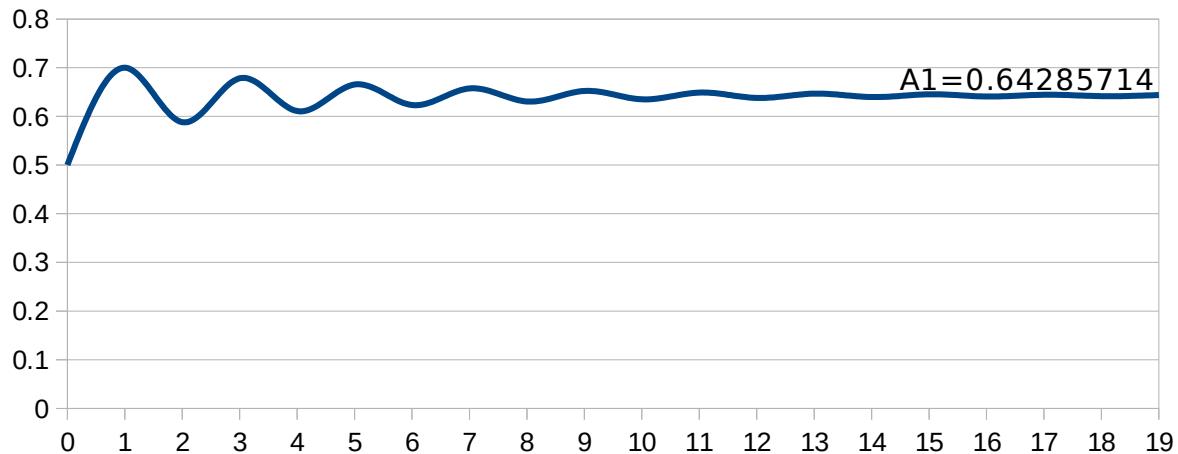


Figura 66: Serie temporal de la curva logística con $x_0=0.5$ y $\lambda=0.7$.

El valor final A_1 se llama atractor porque es a donde se tiende para un conjunto amplio de valores iniciales de x_0 . Y se llama cuenca de atracción de A_1 precisamente a todos esos valores iniciales x_0 . Esta curva tiene otro atractor $A_2=0$ cuya cuenca de atracción es únicamente el punto $x_0=0$. De hecho, si la semilla inicial se aleja ligeramente de 0, se salta al otro atractor. Por ello, a $x_0=0$ no se le llama atractor sino repulsor (cualquier punto ligeramente alejado de él, se irá alejando más con cada iteración).

Resumiendo, para $\lambda < 0.25$ hay un único atractor $A_1=0$ cuya cuenca de atracción son todos los puntos $0 \leq x_0 \leq 1$. Mientras que para $0.25 < \lambda < 0.75$ hay dos atractores, uno A_1 cuyo valor exacto depende de λ y está situado en la intersección de la diagonal con la curva, es decir, se puede calcular resolviendo $f(x)=x$, y cuya cuenca de atracción es $0 < x_0 \leq 1$; y otro que es repulsor $R_1=0$ cuya cuenca es

únicamente el punto $x_0=0$.

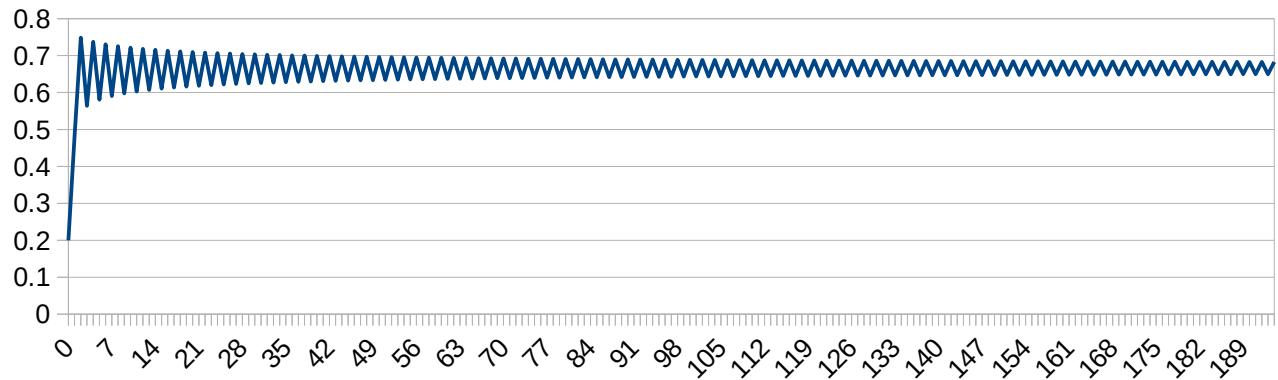


Figura 67: Serie temporal de la curva logística con $x_0=0.2$ y $\lambda=0.75$.

Subimos lambda un poco más hasta llegar exactamente a 0.75. En ese momento las oscilaciones se demoran muchísimo en amortiguarse para estabilizarse en un único punto (figura 67). De hecho, aunque disminuyen su amplitud, lo hacen asintóticamente hacia el atractor 0.666 conforme el tiempo tiende a infinito.

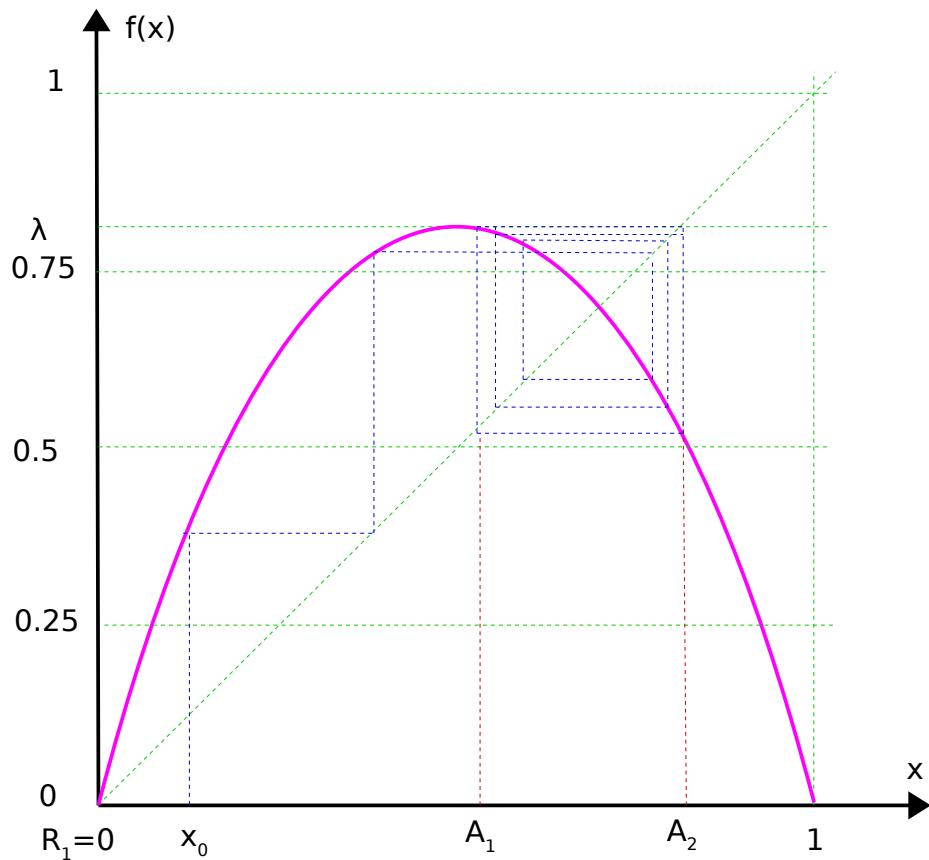


Figura 68: Curva logística para $x_0=0.2$ y $\lambda=0.8$.

Si seguimos subiendo el valor de lambda, cuando está un poco por encima de

0.75 el atractor A_1 se desdobra en dos: A_1 y A_2 . Es decir, la serie temporal oscila entre esos dos valores, como se observa en las figuras 68 y 69. Se dice que el atractor no es ya un punto, sino una órbita de dos puntos, o también, una órbita de periodo 2.

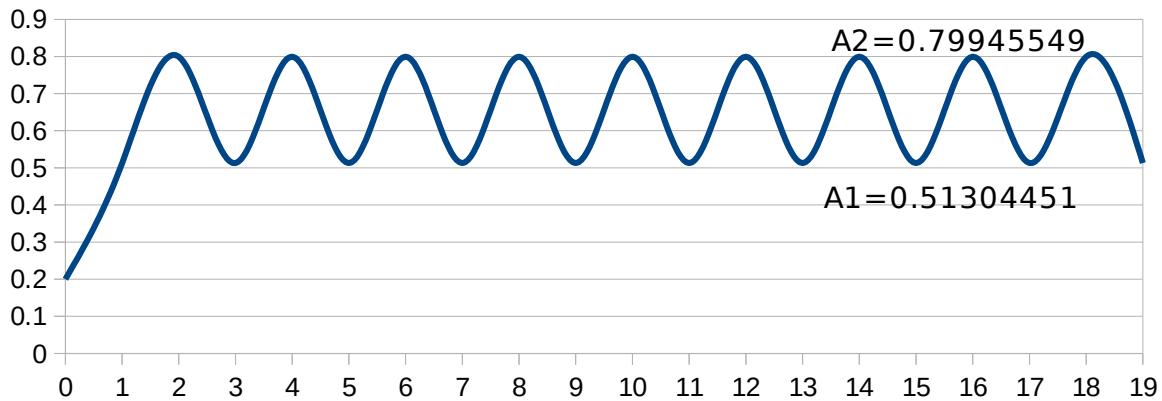


Figura 69: Serie temporal de la curva logística con $x_0=0.2$ y $\lambda=0.8$, de periodo 2.

Si seguimos aumentando el valor de lambda, el periodo pasa a ser 4 (figura 70), luego 8, y así cada vez más rápido, hasta que llega un momento en que aparece el caos, donde ya no es posible identificar un periodo de repetición de la onda temporal (figura 71). Allí se pueden observar cuatro periodos que se parecen, aunque no son exactamente iguales. En la figura 72 se ve una secuencia más larga de la misma onda, donde se aprecia con más claridad que no se repite.

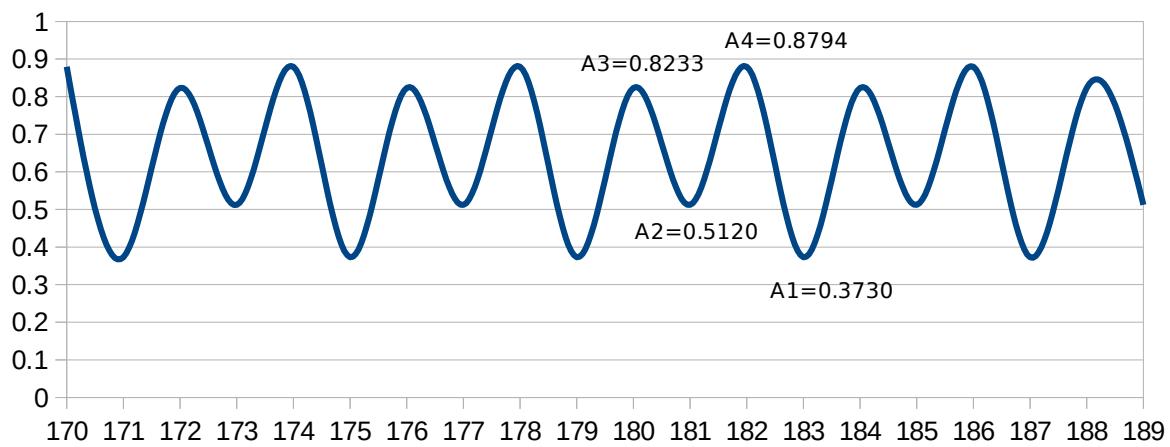


Figura 70: Serie temporal de la curva logística con $x_0=0.2$ y $\lambda=0.88$, de periodo 4.

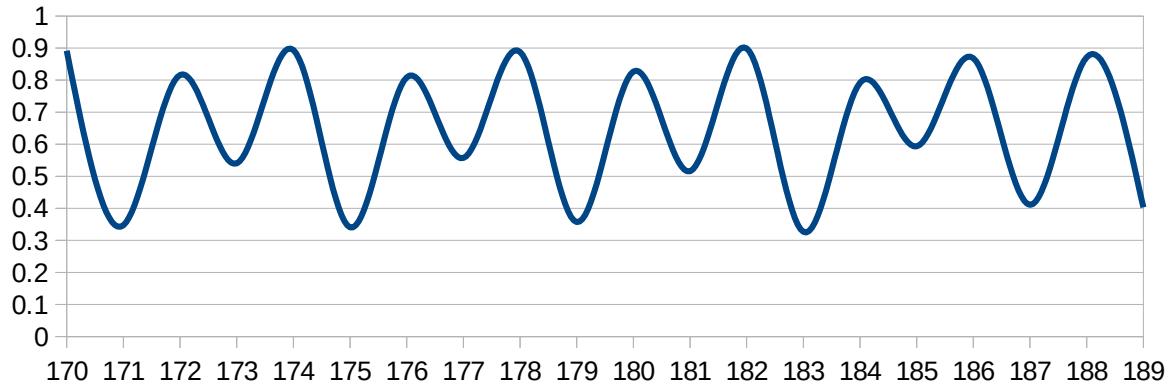


Figura 71: Curva logística en zona caótica para $\lambda=0.95$.

Esta es una de las características de los fenómenos caóticos: parece que hay una estructura periódica, pero cuando te fijas bien, te das cuenta que no es así. La onda es aperiódica o, si se quiere, de periodo infinito. Este es un primer obstáculo que impide predecir futuros valores de la secuencia a partir de los valores pasados.

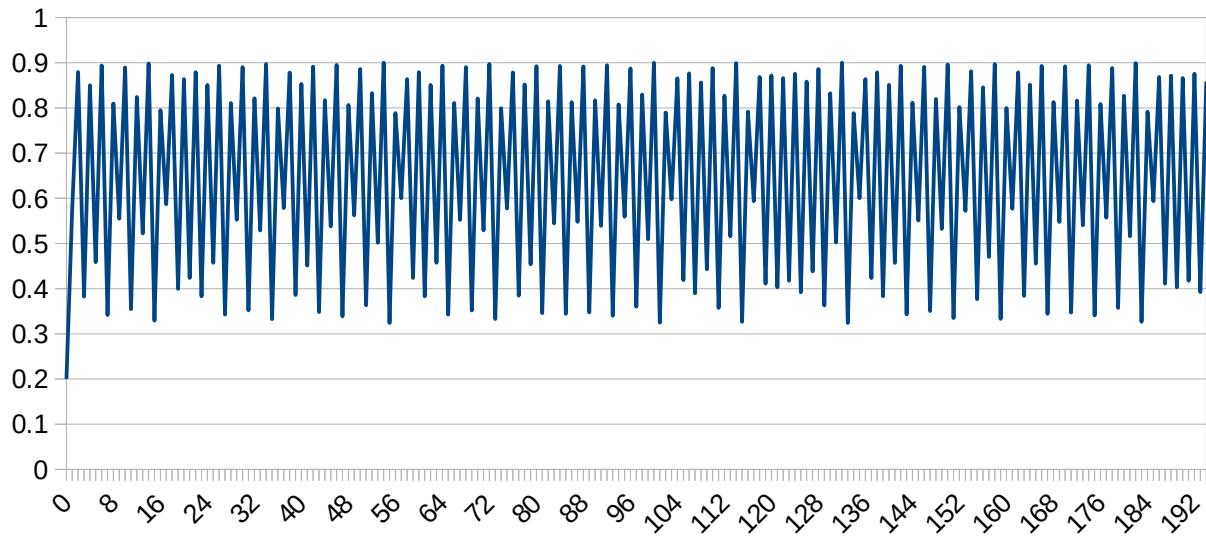


Figura 72: Una serie más larga de la misma ecuación anterior.

Una segunda característica de los fenómenos caóticos es su alta sensibilidad a los valores iniciales, lo que no ocurre con los fenómenos periódicos, como por ejemplo, un reloj. Si mi reloj lo pongo en hora hoy con un segundo de retraso, cuando mire la hora dentro de un año seguiré teniendo más o menos un segundo de retraso. Por el contrario, en los fenómenos caóticos una pequeña desviación en el valor inicial se traduce en un error que se amplifica exponencialmente conforme pasa el tiempo. En la figura 73 podemos ver la ecuación logística de

nuevo con $\lambda=0.95$ pero para dos valores iniciales casi idénticos: en azul con una semilla inicial de $x_0=0.2$; y en rojo observamos la misma ecuación, pero ahora la semilla es $x_0=0.2001$, es decir, un error de una milésima.

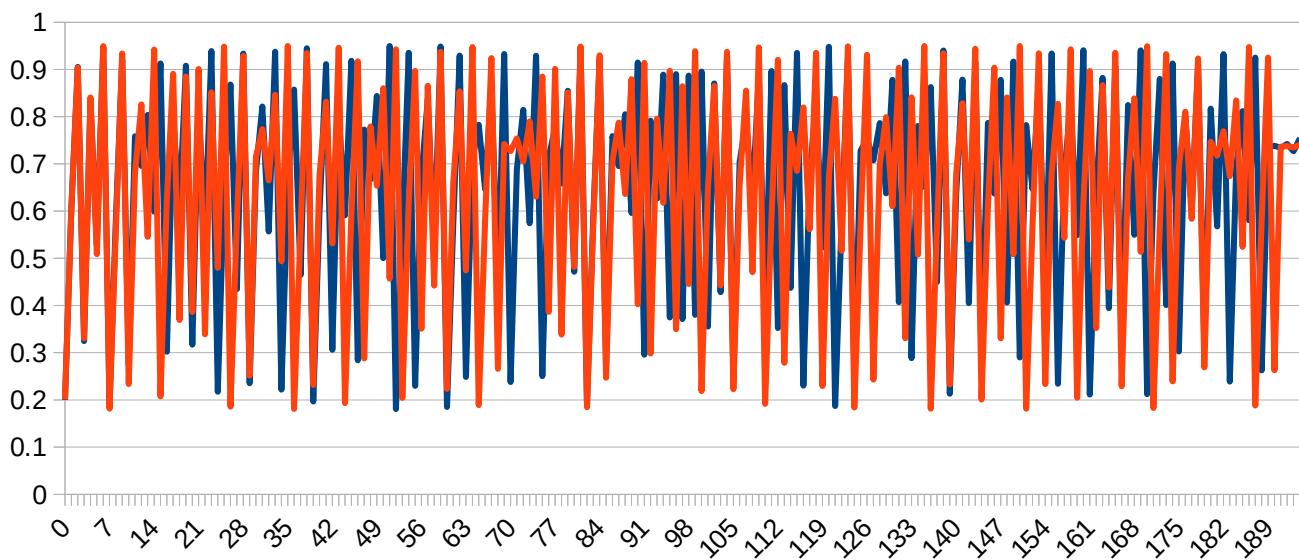


Figura 73: Curva logística en caos, con $\lambda=0.95$ para $x_0=0.2$ (en azul) y $x_0=0.2001$ (en rojo).

Y en la figura 74 vemos un detalle ampliado de la misma onda. Como vemos, son bastante distintas. El error inicial se amplificó apreciablemente a partir de la muestra 11 (ver el inicio de la onda en la figura 75). Esto es lo que se llama el “horizonte de predicción”: hasta la muestra 11 es fácil predecir lo que va a pasar, pero a partir de ahí no, porque cualquier pequeño error en la medida del punto inicial se amplifica exponencialmente. En este ejemplo, el horizonte de predicción está en la muestra 11, pero con otras ecuaciones, otras semillas u otros valores de lambda, puede ser mayor o menor. Cuando hay caos, se puede predecir el resultado del sistema a corto plazo, pero no a largo plazo.

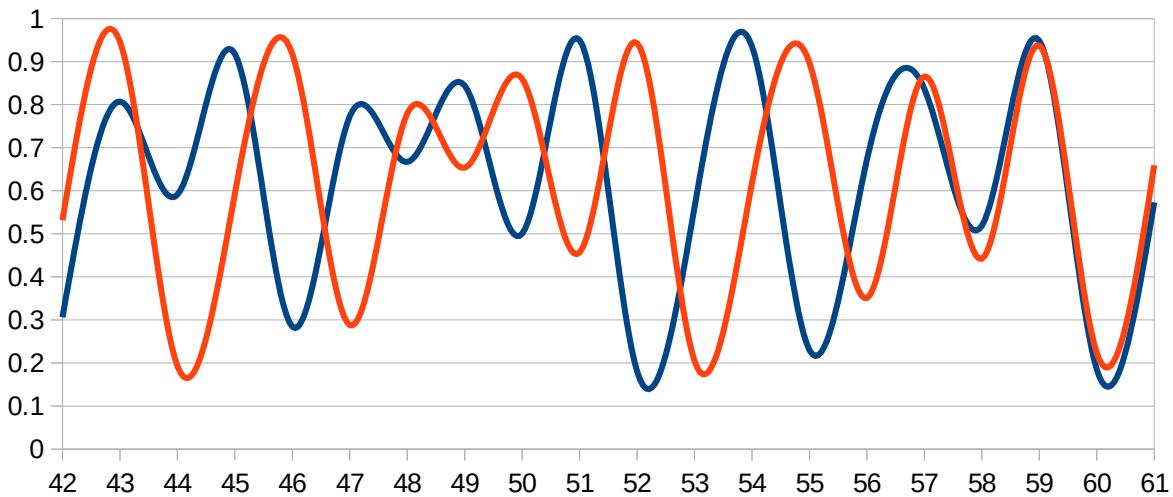


Figura 74: Detalle ampliado de la onda anterior.

El concepto importante a recordar aquí es que el caos es un fenómeno determinista, o sea, la ecuación no incluye ningún parámetro desconocido o estocástico. A pesar de ello, la salida del sistema a largo plazo no se puede predecir, salvo que se tenga un sistema aislado, sin ruidos ni errores en las medidas, lo cual es imposible en la práctica.

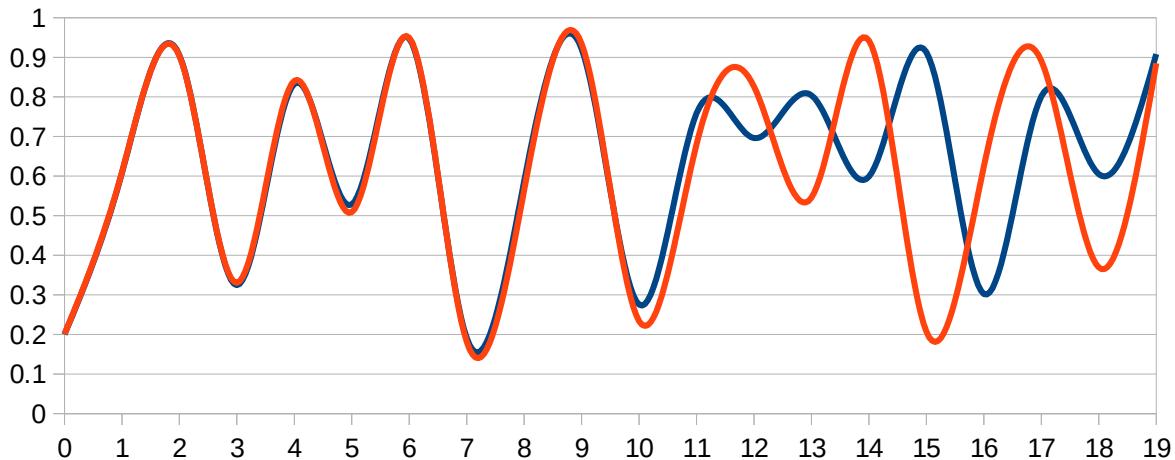


Figura 75: Inicio de la onda de la figura anterior, donde se aprecia el momento en que se produce la divergencia.

Volviendo al parámetro lambda, la primera duplicación de periodo ocurre cuando $\lambda=0.75$ y la segunda para $\lambda=0.86237$. Podemos hacer gráficos con los valores de los atractores en función de lambda y obtendremos algo similar a la figura 76.

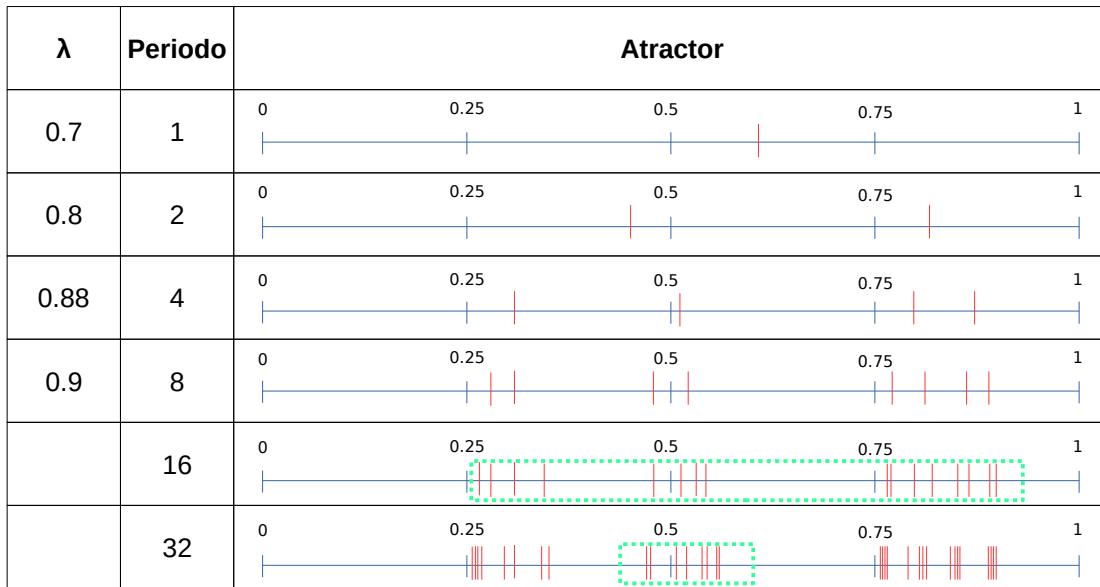


Figura 76: Duplicación de periodo en la curva logística.

Observemos que hay una cierta autosemejanza al pasar, por ejemplo, de periodo 16 a 32, si comparamos los atractores encerrados en los rectángulos verdes. De hecho, el conjunto de todos los atractores cuando variamos λ de manera continua es un fractal, como se ve en la figura 77, donde es posible averiguar para cada valor de λ cuál es el periodo de la órbita que se genera. Por ejemplo, para $\lambda=0.4$ solo hay un punto de la curva, por lo que el periodo es 1.

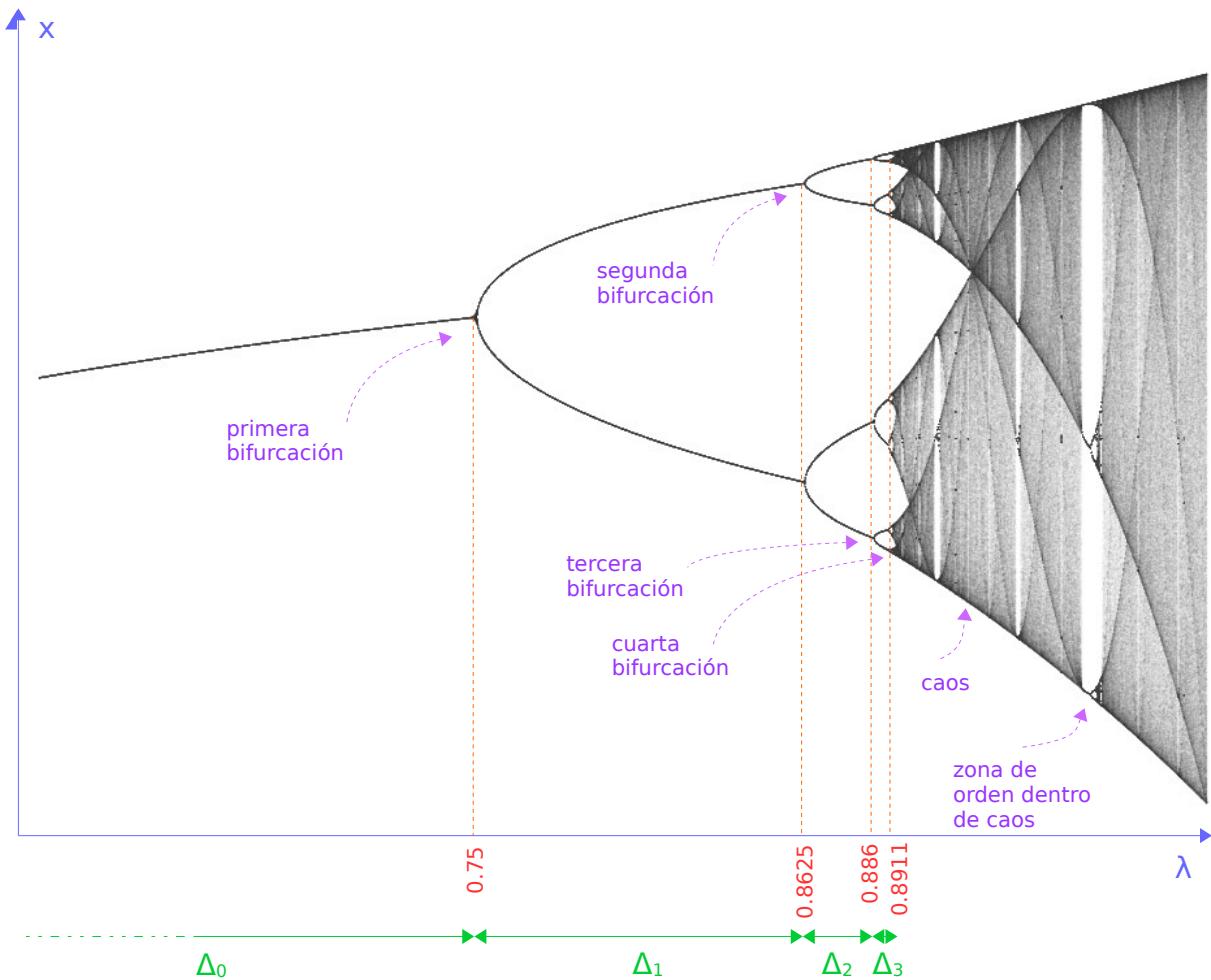


Figura 77: Diagrama de bifurcación de la curva logística.

Fuente: original por Ríos (2013), modificada por el autor.

Y vemos cómo ocurre la primera duplicación de periodo (la primera bifurcación) en $\lambda=0.75$. Un poco después aparece la segunda duplicación de periodo, y así cada vez más rápidamente, produciéndose una cascada de duplicaciones que termina en la zona de caos, donde se confunden unas bifurcaciones con otras. A ello se le llama “crisis”, según las definiciones presentadas más adelante. También puede observarse cómo, dentro de las zonas de confusión (donde hay caos), hay estrechas franjas verticales donde el sistema vuelve a ser periódico (es decir, ordenado). Y si aumentásemos más esas franjas veríamos que hay subfranjas mucho más estrechas donde vuelve a aparecer el caos. Es decir, hay zonas ordenadas dentro de las caóticas y zonas caóticas dentro de las ordenadas. Y la más mínima perturbación puede llevar el sistema de una a otra. El sistema es autosimilar cuando se hace zoom sobre él.

Además, la cascada de bifurcaciones ocurre cada vez más rápido y si miramos cómo se acelera obtenemos lo siguiente:

$$\delta_0 = \frac{\Delta_0}{\Delta_1} = \frac{(0.75-0)}{(0.8625-0.75)} = 6.666 \quad Ec. 16$$

$$\delta_1 = \frac{\Delta_1}{\Delta_2} = \frac{(0.8625-0.75)}{(0.886-0.8625)} = 4.787 \quad Ec. 17$$

$$\delta_2 = \frac{\Delta_2}{\Delta_3} = \frac{(0.886-0.8625)}{(0.8911-0.886)} = 4.608 \quad Ec. 18$$

Y en el límite:

$$\lim_{n \rightarrow \infty} \delta_n = \lim_{n \rightarrow \infty} \frac{\Delta_n}{\Delta_n} = 4.6692016 \dots \quad Ec. 19$$

que es irracional y se le llama número de Feingebaum, en honor a su descubridor. Es interesante hacer notar que estas ecuaciones se aplican a muchas funciones $f(x)$ no necesariamente parábolas, pues basta con que tengan un único valor máximo en el intervalo $[0,1]$. A esto se le denomina universalidad.

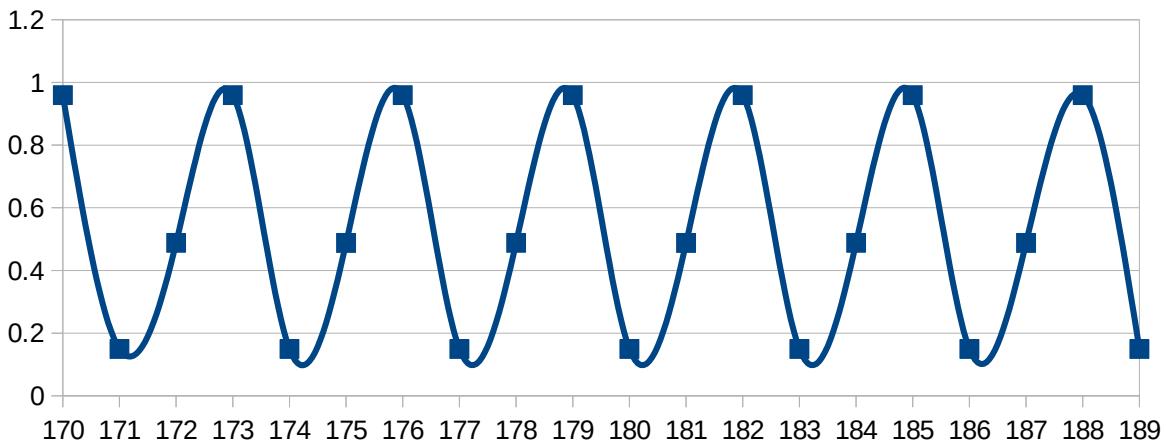


Figura 78: Curva logística con periodo 3 para $\lambda=0.96$.

También es interesante ver que en las zonas donde hay mezcla de órbitas (en la figura 77) pueden aparecer períodos que no son múltiplos de 2. Por ejemplo, para $\lambda=0.96$ el período es 3 (ver la correspondiente secuencia en el tiempo en la figura 78). Y a ello le siguen sucesivas duplicaciones de período (6, 12...).

En 1975, Li y Yorke ya habían demostrado que una función iterativa de período 3

implicaba caos. Efectivamente, así vemos que ocurre con la función logística.

Si realizamos un histograma de los valores temporales que va tomando la curva logística, vemos que no son equiprobables. Eso significa que aunque no podemos predecir el futuro puntualmente, si lo podemos hacer estadísticamente. En la figura 79 vemos un histograma para $\lambda=1$, donde la ecuación logística también es caótica: los valores extremos (muy alto y muy bajo) son muy probables, mientras que los valores medios no lo son. Es por eso que el clima se puede predecir en detalle (cuántos litros lloverá, la nubosidad, la fuerza del viento) hasta solo unos pocos días por adelantado, pero nada impide hacer predicciones estadísticas generales de más largo plazo (por ejemplo, si la próxima primavera va a ser muy lluviosa o no).

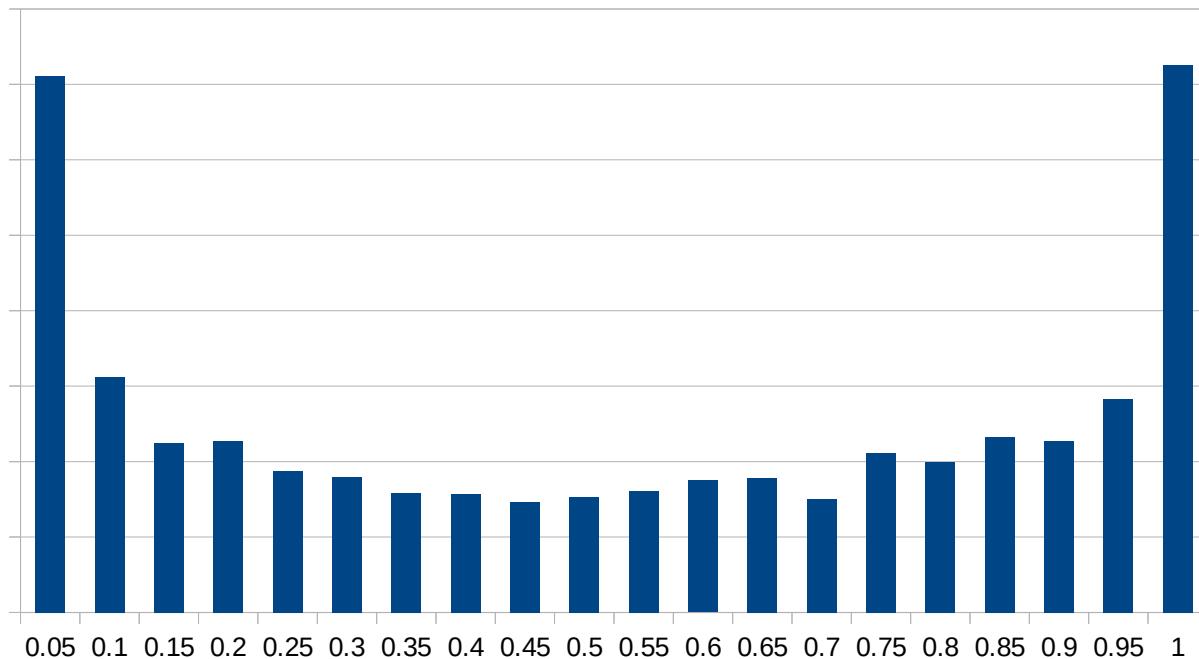


Figura 79: Histograma de valores temporales de la curva logística, de 0 a 1 con anchura 0.05.

Otros sistemas que exhiben caos

Se pueden proponer más ejemplos de ecuaciones deterministas que llevan al caos al dar ciertos valores a sus parámetros:

Sistema de Lorenz

Son ecuaciones diferenciales que modelan el clima:

$$\frac{\partial x}{\partial t} = \delta(y - x)$$

Ec. 20

$$\frac{\partial y}{\partial t} = \alpha x - y - xz$$

$$\frac{\partial z}{\partial t} = xy - \beta z$$

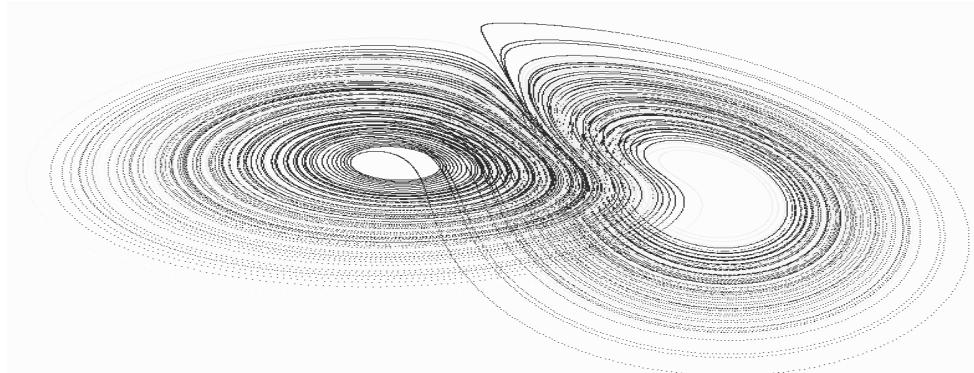


Figura 80: Atractor de Lorenz (ocupa 3 dimensiones) para $\delta=10$, $\alpha=28$, $\beta=8/3$.

Fuente: Ríos (2013).

Son tres variables de estado $\{x, y, z\}$ y tres parámetros de control $\{\delta, \alpha, \beta\}$. En la figura 81 puede verse el diagrama de bifurcaciones y en la figura 80 uno de los atractores (para $\delta=10$, $\alpha=28$, $\beta=8/3$), muy conocido por su parecido con las alas de una mariposa. Es importante hacer notar que en este atractor no se cruzan las órbitas (tal y como se comenta más adelante, en la figura 92), como podría aparentar, pues la figura tiene tres dimensiones.

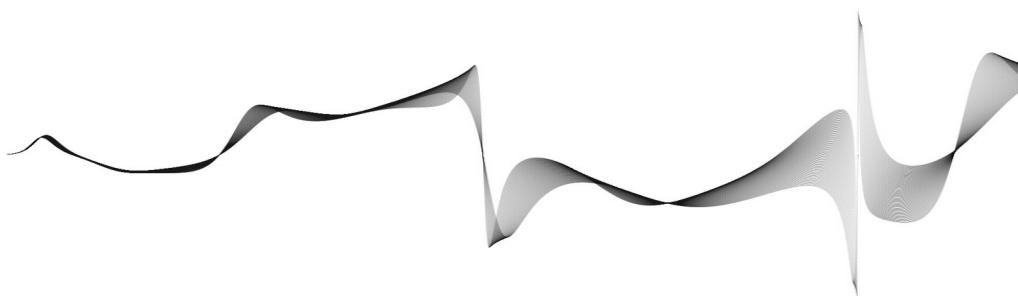


Figura 81: Diagrama de bifurcaciones de Lorenz (en abscisas, el parámetro α ; en ordenadas, la variable de estado x).

Fuente: Ríos (2013).

Personaje 2

EDWARD LORENZ (1917-2008)

Edward Norton Lorenz fue un meteorólogo y matemático estadounidense que descubrió por casualidad el fenómeno de la alta sensibilidad a las condiciones iniciales presente en los procesos caóticos. La historia es más o menos así: estaba haciendo una simulación con su modelo de tres ecuaciones para el clima. Los computadores de aquella época eran lentos por lo que iba imprimiendo resultados parciales conforme avanzaba la simulación. En un momento dado, Lorenz detuvo la simulación. Cuando la reanudó lo hizo introduciendo como datos de entrada unos resultados parciales que obtuvo, pero no se molestó en teclear todas las cifras decimales. Y lo que obtuvo le asombró: los cálculos divergían rápidamente de los valores que estaba obteniendo antes.

Mapa de Hénon

Es una sección de Poincaré de las ecuaciones de Lorenz, que diseñó Michael Hénon con el objetivo de obtener caos en un sistema más sencillo (figura 82). El sistema de ecuaciones es:

$$\begin{aligned}x_{n+1} &= y_n - ax_n^2 + 1 \\y_{n+1} &= bx_n\end{aligned}\quad \text{Ec. 21}$$

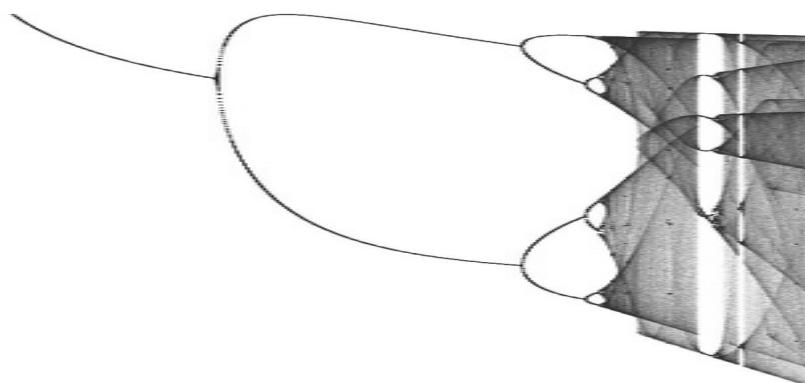


Figura 82: Diagrama de bifurcaciones de Hénon (en abscisas, el parámetro a ; en ordenadas, la variable de estado x).

Fuente: Ríos (2013).

Obsérvese que solo tiene dos variables de estado $\{x,y\}$ y dos parámetros $\{a,b\}$. Para algunos valores de $\{a,b\}$ se obtiene caos y para otros órbitas periódicas. Por ejemplo, se obtiene caos para $a=1.4$ y $b=0.3$ (figura 83) que se llama el sistema clásico de Hénon. Es complicado dibujar un gráfico con tantas variables y

parámetros, por lo que se opta por mantener un parámetro constante y variar el otro, dibujando solo una de las variables de estado (figura 82).

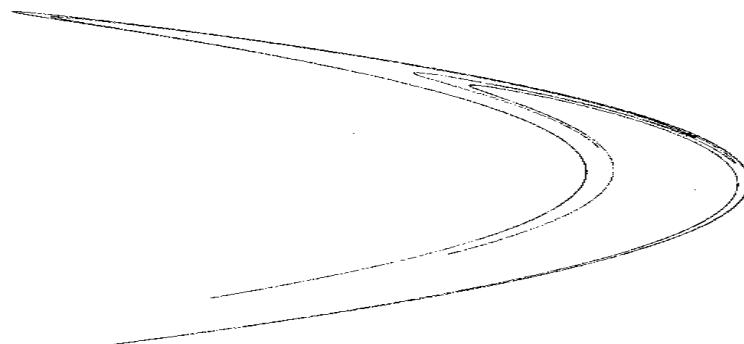


Figura 83: Uno de los atractores de Hénon para $a=1.4$, $b=0.3$.

Fuente: Ríos (2013).

Sistema de Ikeda

Procede de un modelo de física de resonadores ópticos y consiste en las ecuaciones:

$$\begin{aligned}x_{n+1} &= a + b(\cos(u)x_n - \sin(u)y_n) \\y_{n+1} &= b(\sin(u)x_n + \cos(u)y_n)\end{aligned}\quad \text{Ec. 22}$$

siendo $u = c - \frac{d}{1 + x_n^2 + y_n^2}$

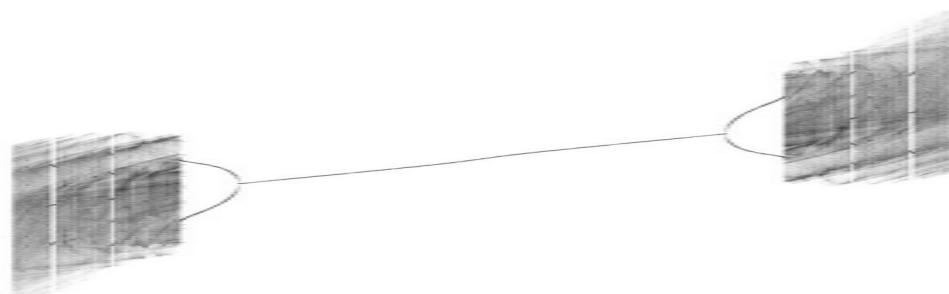


Figura 84: Diagrama de bifurcaciones de Ikeda (en abscisas el parámetro a ; en ordenadas la variable de estado x).

Fuente: Ríos (2013).

Aquí hay dos variables de estado $\{x, y\}$ y cuatro parámetros que controlan el comportamiento del sistema $\{a, b, c, d\}$. En la figura 84 puede verse el diagrama de bifurcaciones y en la figura 85 un atractor.



Figura 85: Un atractor de Ikeda para $a=1$, $b=0.9$, $c=0.4$, $d=6$

Fuente: Ríos (2013).

Se puede observar claramente la transición abrupta del caos al orden y luego del orden al caos, así como la mezcla de varios atractores en las zonas caóticas, que es un concepto llamado “crisis” (ver definiciones más adelante).

Sistema de Tinkerbell

Sus ecuaciones tienen dos variables de estado $\{x, y\}$ y cuatro parámetros $\{a, b, c, d\}$:

$$\begin{aligned} x_{n+1} &= x_n^2 - y_n^2 + ax_n + by_n \\ y_{n+1} &= 2x_n y_n + cx_n + dy_n \end{aligned} \quad \text{Ec. 23}$$

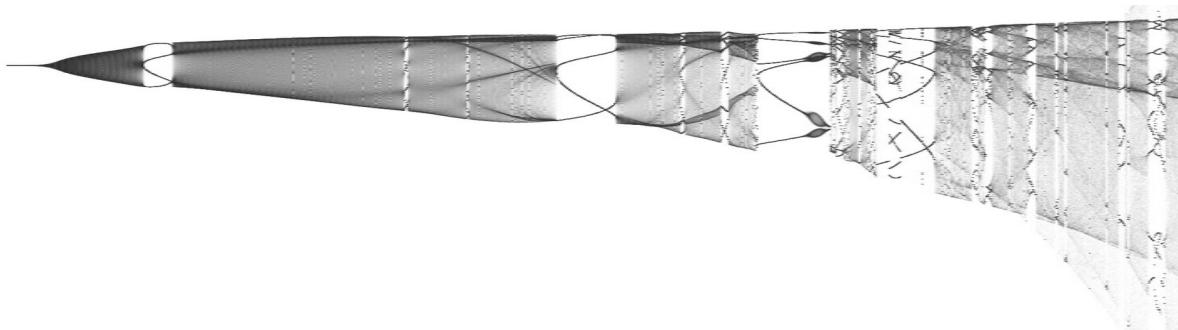


Figura 86: Diagrama de bifurcaciones de Tinkerbell.

Fuente: Ríos (2013).

En la figura 86 vemos su diagrama de bifurcaciones y en la figura 87, un atractor.

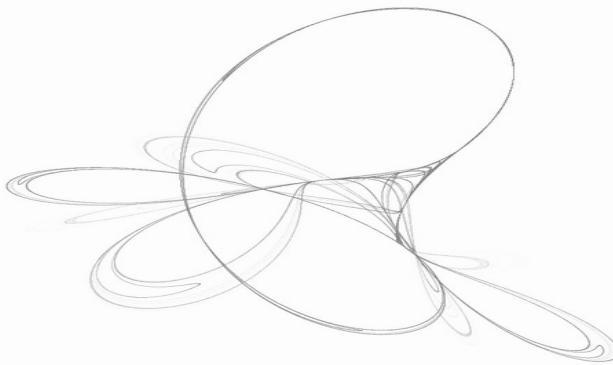


Figura 87: Atractor de Tinkerbell para $a=0.5$, $b=-0.6$, $c=2.2$, $d=0.5$.

Fuente: Ríos (2013).

Los ejemplos anteriores son modelos matemáticos, pero también puede verse el caos en fenómenos físicos, como los siguientes.

Llave de agua goteando

Cuando se abre o se cierra muy despacio una llave de agua, hay un momento en que se pasa de un goteo periódico a uno caótico. La figura 88 muestra fotografías del lavaplatos de mi casa en las que las líneas de color verde indican dónde están las gotas de agua. En la foto (a) se observa el fenómeno de duplicación de periodo, mientras que en la foto (b) puede verse el caos: se intuye que las gotas no son periódicas²¹, y si el experimento se prolonga durante bastante tiempo y se toman datos precisos se confirma que, efectivamente, no lo son.

Unos años más tarde cambié de casa y repetí el experimento, pero no logré obtener el régimen caótico. Para tratar de entender el asunto, hice pruebas con una manguera en el jardín, y allí sí se obtenía caos con facilidad. La razón de ello era que la nueva casa tenía mejores materiales y la tubería de la cocina que termina en la llave no vibra cuando pasa el agua por ella. Así logré confirmar un concepto importante sobre el caos: no se trata de un fenómeno mágico, ni de la “nueva era”. Tampoco ocurre obligatoriamente. Para que ocurra debe haber algún bucle de realimentación. En el caso de la casa vieja, cuando salía una gota, la tubería se desplazaba levemente hacia arriba, por la segunda ley de Newton, de acción y reacción. La tubería era flexible y podía hacerlo. Y esa vibración de la tubería era lo que volvía impredecible cuando iba a desprenderse la siguiente gota (figura 89).

²¹ Hay que tener en cuenta que no caen a velocidad constante sino a aceleración constante, por lo que es más preciso registrar el sonido que hacen al golpear el fondo del lavaplatos, y obtener así una secuencia temporal de eventos donde analizar la periodicidad. Pero, incluso así, las fotos dan una idea de lo que está ocurriendo.

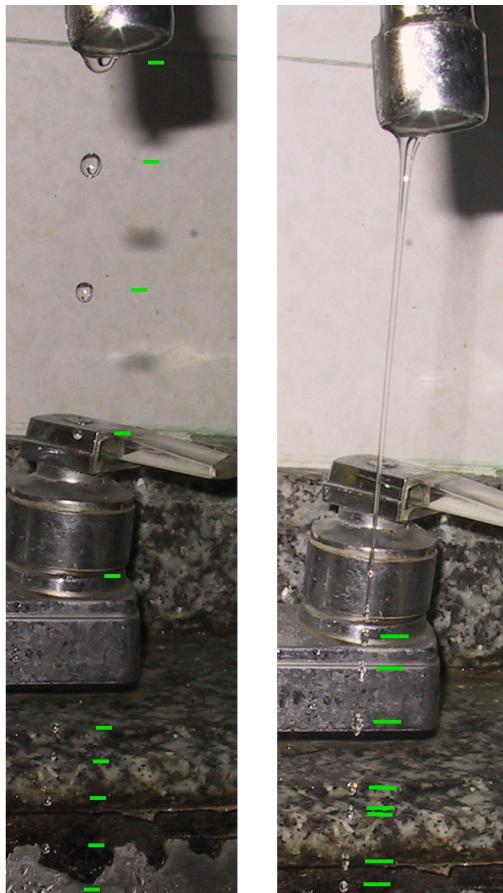


Figura 88: Caos en la cocina.

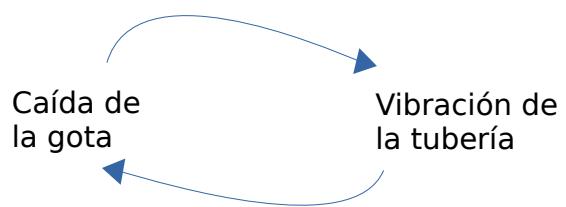


Figura 89: Bucle en llave de agua.

Seguramente hay muchas otras formas de obtener caos en una llave que gotea agua, pero en todas debe existir algún tipo de bucle de realimentación. De hecho, se requieren al menos dos bucles: uno de realimentación positiva y otro de realimentación negativa.

Péndulo doble

Si a la masa oscilante de un péndulo se le sujeta otro péndulo, tenemos un péndulo doble (figura 90). Jugando con las longitudes y las masas de cada péndulo obtenemos una variedad de fenómenos, incluyendo el caos.

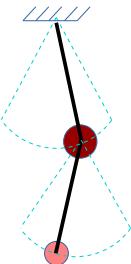


Figura 90: Péndulo doble.

Péndulo con imanes

Hay varias formas de hacerlo, pero la idea es que en la masa de un péndulo se coloque un imán, y justo debajo en su vertical se coloque otro u otros (repeliendo o atrayendo al primero). De este modo, no hay una única fuerza actuando sobre el péndulo sino varias. De nuevo, jugando con los parámetros (número, situación y fuerza de los imanes), se puede obtener caos con facilidad. Se puede simular el fenómeno en el computador, introduciendo las ecuaciones de gravitación del péndulo y las de los imanes. Después de oscilar de forma errática, el péndulo terminará fijo, atraído hacia un único imán. Partiendo de posiciones iniciales ligeramente distintas, el péndulo termina en sitios muy distintos. Por ejemplo, en el video de Peitgen (2016) se ha realizado el experimento con tres imanes, marcados con colores rojo amarillo y azul, y se exploran muchos puntos donde se suelta el péndulo, marcándolos luego con el color del imán donde aterriza. Conforme se va haciendo más fina la ubicación del punto inicial, la figura dibujada se va haciendo más enrevesada.

Sistemas fisiológicos

Muchos sistemas fisiológicos exhiben caos. Por ejemplo, el electrocardiograma de una persona sana puede parecer periódico, pero no lo es. Es caótico. Curiosamente, si exhibe una periodicidad exacta, entonces denota alguna enfermedad. Lo mismo ocurre con el electroencefalograma, que se vuelve periódico cuando el sujeto experimenta un ataque de epilepsia, cuando tiene Parkinson u sufre otras afecciones.

Esto debe ser motivo de reflexión: un sistema periódico y predecible no puede

reaccionar rápidamente a alteraciones de su entorno. No se puede adaptar. Un sistema completamente aleatorio no puede tampoco reaccionar adecuadamente. Está continuamente reaccionando de forma desadaptativa ante cosas que no han ocurrido. Mientras que un sistema caótico conserva una libertad de elección que le permite reaccionar ante cambios repentinos, tomar decisiones y adaptarse a problemas que se le presenten. Rodolfo Llinás decía que el pequeño temblor que vemos en la punta de los dedos cuando extendemos la mano es el efecto residual del caos que reina en el cerebro y se transmite por los nervios a los músculos. Y también decía que ese es el movimiento más rápido que el cuerpo puede realizar. El caos nos da libertad, como argumentábamos al analizar los sistemas complejos. Por ello, es muy frecuente encontrar que los sistemas vivos se encuentren en caos, al borde entre el orden y el desorden²².

Definiciones

A continuación vamos a dar unas cuantas definiciones que, aunque su origen es la teoría de control, a nosotros nos van a servir para entender mejor el caos.

Sistema dinámico: que cambia en el tiempo.

Estado: mínimo conjunto de variables que determinan el presente de un sistema. Si el sistema es determinista, en teoría también quedan determinados el pasado y el futuro. Por ejemplo, en un péndulo simple lo habitual es tomar como variables de estados su posición y su velocidad vectorial. En la práctica, dado que el péndulo es rígido, basta con la coordenada respecto al eje x (que lo definimos como paralelo al movimiento) y la componente de la velocidad en ese eje x (con signo).

Dimensión: número de estas variables.

²² En muchos libros se dice "*at the edge of chaos*" y se traduce por "estar al filo del caos", o sea casi en caos. Pero eso no es correcto. La traducción correcta sería "estar dentro del caos, que es una zona muy angosta, afilada, al borde entre el orden y el desorden".

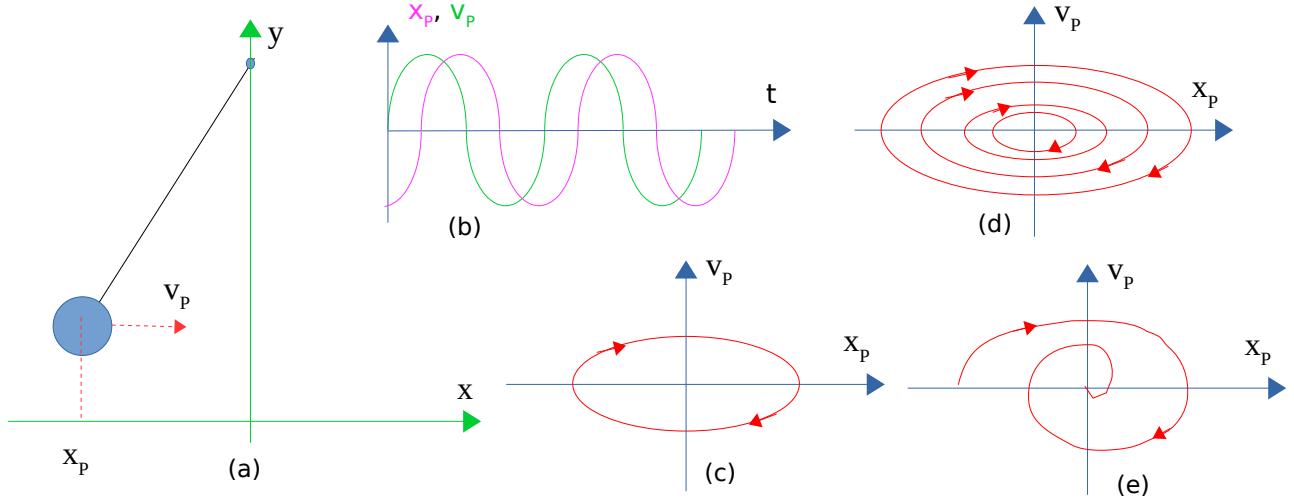


Figura 91: a) Péndulo con dos variables de estado; b) serie temporal de las dos variables; c) una órbita sin rozamiento; d) diagrama de estados, sin rozamiento; e) una órbita, con rozamiento.

Diagrama de estados o espacio de fases: conjunto de órbitas de un sistema, para diversas condiciones iniciales. El diagrama de estados tiene un eje para cada variable de estado. Y el tiempo no figura explícitamente (figura 91-d).

Órbita: trayectoria de estados por los que pasa un sistema conforme transcurre el tiempo (figura 91-c y d). Si el volumen donde transcurren las órbitas se conserva (aunque cambie de forma), se trata de un sistema *hamiltoniano* (no-disipativo) como el de la figura 91-c. Si se contrae, se trata de un sistema dissipativo, es decir, con rozamiento u otro tipo de pérdida de energía (figura 91-e).

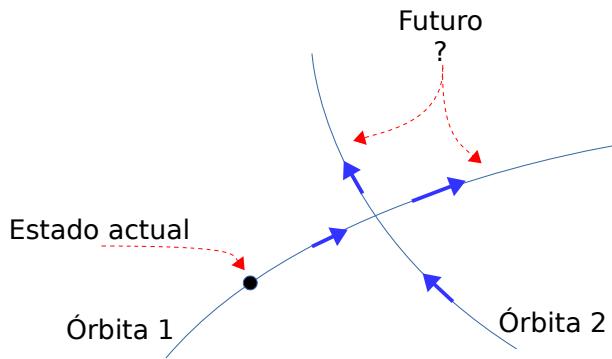


Figura 92: Explicación de por qué dos órbitas no pueden intersecarse.

En un sistema determinista las órbitas no se pueden interseccar debido a la definición de estado. Entendamos bien la razón: si dos órbitas se intersecasen, a partir de ese punto tendrían dos futuros (figura 92), lo cual es lógicamente imposible.

En sistemas continuos se requieren al menos 3 dimensiones para que pueda darse el caos (en 2 dimensiones las órbitas no pueden ser lo suficientemente complejas ya que, por definición, no pueden intersecarse con ellas mismas). Un ejemplo de ello es el atractor de Lorenz, que tiene 3 dimensiones.

En sistemas discretos puede darse el caos desde la primera dimensión, como hemos visto en la ecuación logística.

Atractores: órbitas límite a donde tienden todas las demás órbitas, conforme pasa el tiempo. En sistemas continuos de dimensión igual a dos, los atractores pueden ser:

- Punto fijo como, por ejemplo, un péndulo con rozamiento, que acabamos de ver en la figura 91-e.
- Periódico, como el péndulo sin rozamiento de la figura 91-c.
- Ciclo límite, al que las órbitas se van acercando más a medida que pasa el tiempo como, por ejemplo, para ciertos parámetros de las ecuaciones de Lotka-Volterra en modelos de predador-presa (figura 93-a).

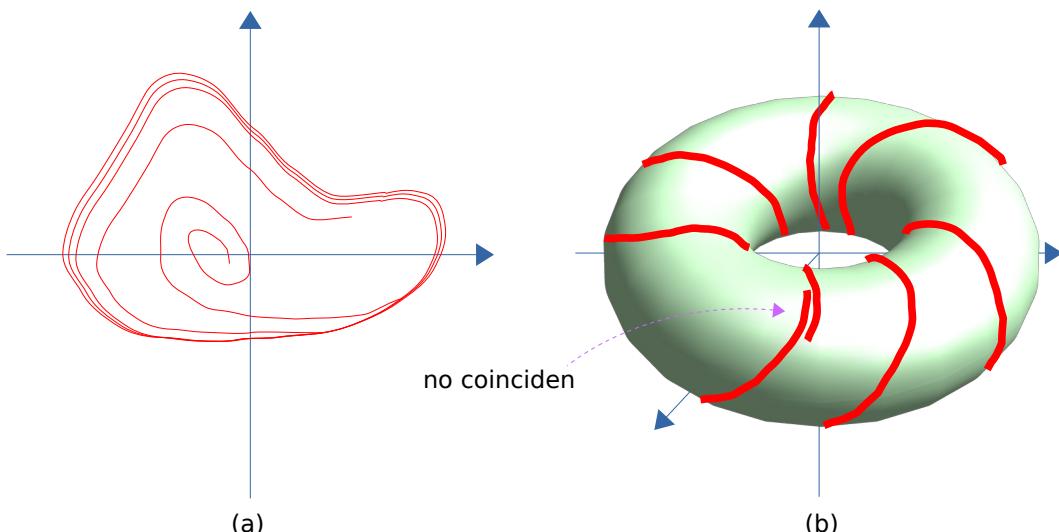


Figura 93: a) Ciclo límite; b) casi-periódico.

- En sistemas continuos de dimensión mayor que dos también pueden darse “ciclos límite casi-periódicos”, donde cada variable de estado evoluciona de forma periódica, pero donde las relaciones entre los periodos son un número irracional, por lo que la órbita nunca se repite (figura 93-b). Aun así, las trayectorias que parten de puntos cercanos no divergen, y ello indica que no se trata de caos. Por ejemplo, podemos pensar en una órbita que

vaya enrollada sobre un toro: si la relación entre periodos es $\sqrt{2}$ o cualquier otro número irracional, el estado jamás volverá a pasar por el mismo punto, aunque se aproximarán tanto como se desee en las sucesivas vueltas.

- Y también pueden aparecer los “atractores caóticos” que son no-periódicos y que muestran sensibilidad exponencial a las condiciones iniciales. Estos atractores fueron llamados históricamente “atractores extraños”. Pero para nosotros ya no son tan extraños: son fractales²³, como los vistos en el capítulo anterior. Y es condición necesaria²⁴ para que haya caos, que su atractor sea un fractal.

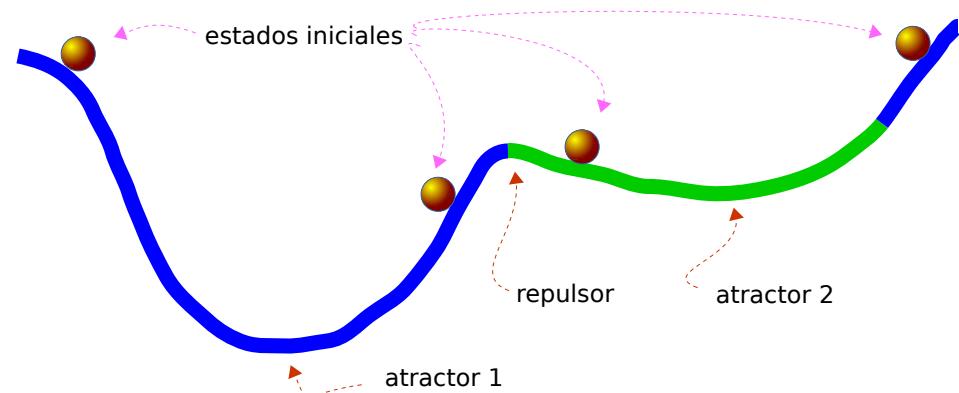


Figura 94: Atractor 1 y su cuenca de atracción en color azul; atractor 2 y su cuenca de atracción en color verde.

Cuencas de atracción: es la misma idea que los atractores, pero se hace énfasis en que hay unas condiciones iniciales que llevan hacia un atractor. A ese conjunto de condiciones iniciales se les llama cuenca de tal atractor. Cada atractor tiene su cuenca de atracción. Los repulsores, por el contrario, no la tienen. Si imaginamos que el estado es como una bolita que puede rodar por la montaña, las cuencas de atracción son los valles, y el atractor es el punto final de reposo (figura 94).

Por último, la definición principal que nos ocupa:

²³ Aunque conviene advertir que hay otros investigadores que definen los atractores extraños como curvas no diferenciables.

²⁴ No es una condición suficiente porque se han descubierto algunos casos de atractores fractales en sistemas dinámicos con coeficiente de Lyapunov negativo o cero, es decir, que no son exponencialmente sensibles a las condiciones iniciales (Grebogi 1984).

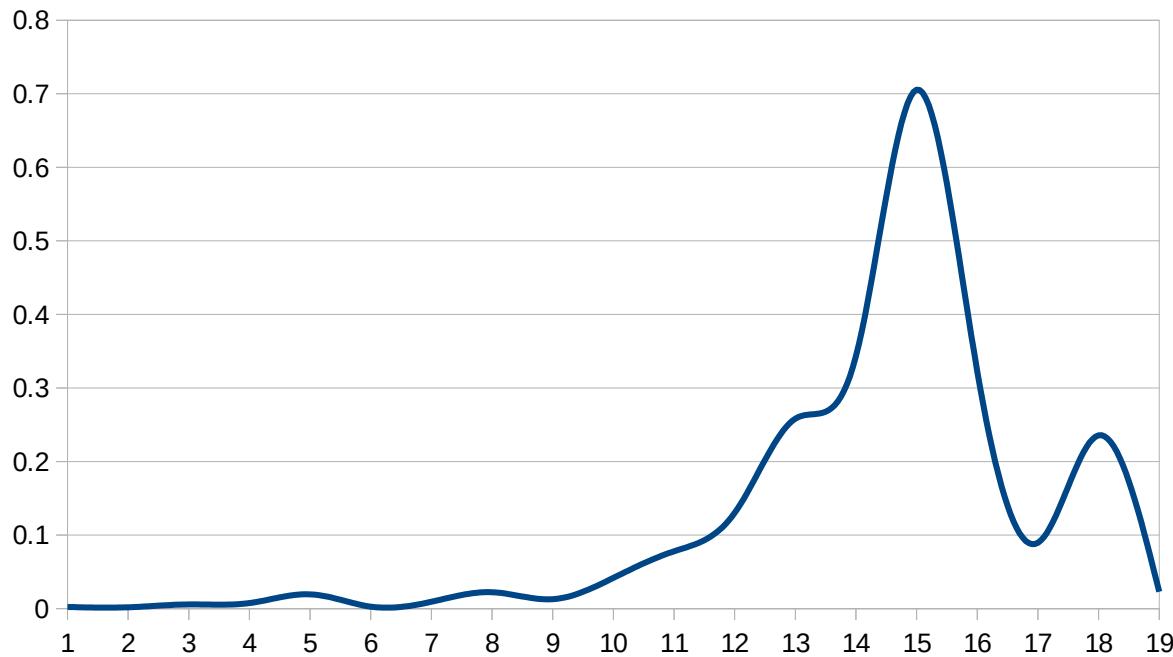


Figura 95: El valor absoluto del error crece exponencialmente.

Caos: para que un sistema esté en régimen caótico debe cumplir las tres condiciones siguientes (Martín, Morán y Reyes, 1998):

- Debe ser exponencialmente sensible a las condiciones iniciales. Esto es, si tenemos dos órbitas con condiciones iniciales casi idénticas, esa pequeña diferencia debe amplificarse proporcionalmente a $e^{\lambda \Delta t}$, donde Δt es un pequeño incremento de tiempo y λ es el llamado coeficiente de Lyapunov, que debe ser positivo para que haya caos. Esto solo interesa para Δt pequeños, ya que la variable de salida está acotada y no va a crecer exponencialmente. En la figura 95 podemos ver la diferencia en valor absoluto de las dos curvas de la figura 75: inicialmente el crecimiento de la diferencia entre dos órbitas es exponencial, hasta que se satura al valor máximo o mínimo posible que, en este caso, ocurre para $t=15$.
- Debe haber mezcla topológica de las órbitas, es decir, cualquier subconjunto de su órbita se puede transformar en cualquier otro subconjunto, si transcurre suficiente tiempo.
- Sus órbitas periódicas deben ser densas. También se formula como que su espacio de fases debe ser compacto. Y ello significa que cada punto del atractor caótico está infinitamente próximo a algún punto de una órbita periódica, de modo que la más mínima perturbación cambia el sistema de periódico a caótico y viceversa.

En muchos libros antiguos solo se menciona la primera de ellas, que es además la más popular gracias a un comentario de Lorenz en un congreso. Algo así como: “el batir de alas de una mariposa en Cali puede desencadenar un tornado en Malpelo”. Esta frase, que es muy evocativa y que ha servido hasta para dar título a películas de cine, hoy se usa exactamente al revés. Dado que cualquier fenómeno que estudiemos puede ser caótico (como el clima), para lograr predicciones más robustas se usan varios modelos y con condiciones iniciales ligeramente distintas a las medidas. Si todos los modelos predicen más o menos lo mismo, tenemos una buena probabilidad de acertar, mientras que si divergen fuertemente quiere decir que hemos entrado en zona caótica y es mejor no arriesgarse en apuestas. Por ejemplo, para predecir si un asteroide impactará sobre la Tierra, se usa el modelo estándar de gravedad *newtoniana* con las coordenadas iniciales y velocidades medidas de todos los cuerpos involucrados. Pero se corre el modelo muchísimas veces, y cada vez se le suma ruido a esas coordenadas y velocidades, tratando de modelar la imprecisión de las medidas. El conteo de las ejecuciones en las cuales el asteroide impacta frente al número total de ejecuciones nos da la probabilidad de que ese evento realmente ocurra.

O si se va a realizar una inversión de dinero en la bolsa (que ya se sabe que su comportamiento es caótico), primero se analizan varios modelos con ruido en las condiciones iniciales, y se espera el momento propicio para hacerla cuando la sensibilidad al ruido sea mínima.

Por supuesto, también se puede hacer al revés: si uno desea incertidumbre, creatividad y sorpresa, debe realizar cambios en su sistema caótico cuando la sensibilidad al ruido sea máxima.

Otras definiciones interesantes:

Crisis: cuando un atractor fractal aparece o desaparece bruscamente (y se convierte a un atractor no fractal), o cuando se mezcla bruscamente con otro atractor fractal.

Intermitencia: episodios de crisis continua, es decir, alternancia brusca entre dos atractores, al menos uno de ellos fractal, donde la conmutación entre un atractor y el otro ocurre muchas veces y de manera impredecible.

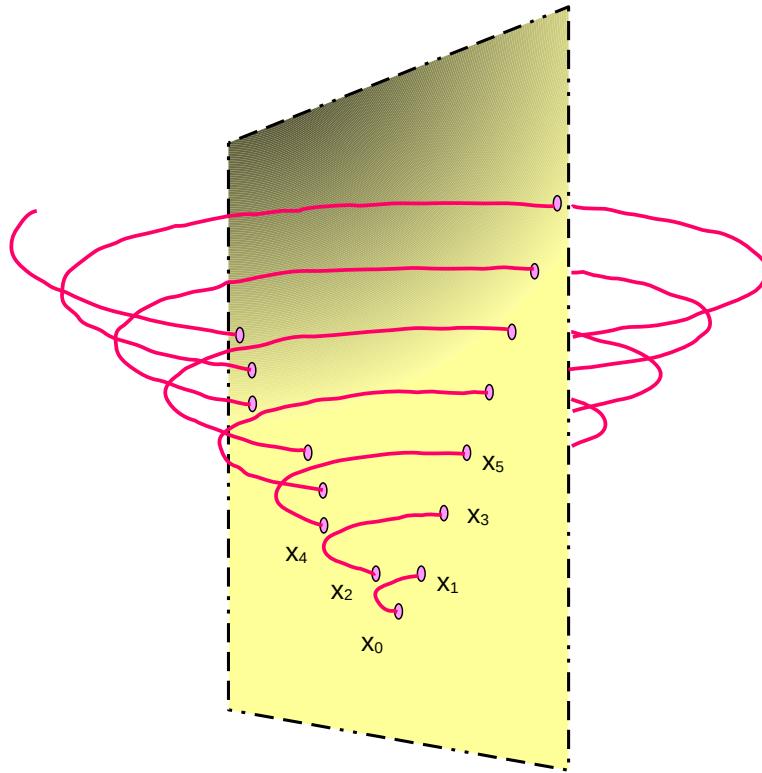
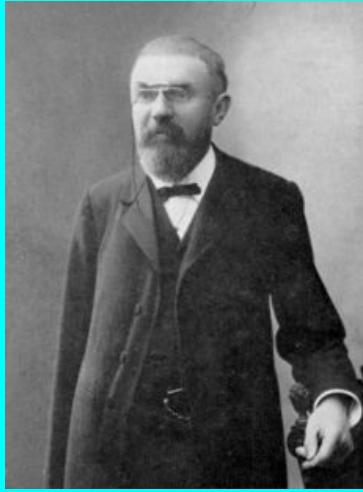


Figura 96: Mapa de Poincaré: $\{x_0, x_1, x_2, \dots\}$ de una órbita atravesando una sección.

Mapa de Poincaré: intersección de una órbita con un subespacio transversal a esa órbita. El subespacio debe tener una dimensión menor que el espacio de fases de la órbita. El resultado es una órbita discreta más sencilla que la original pero que conserva aproximadamente las mismas propiedades (figura 96).

Teorema de Whitney y Takens: permite reconstruir un atractor usando una secuencia temporal discreta de los estados por los que pasa. Para ello se usan varias series temporales construidas con la secuencia temporal de estados retrasada sucesivamente en el tiempo. El número de secuencias retrasadas que se requieren depende de la dimensión del atractor y, como tampoco suele conocerse previamente, se obtiene por prueba y error. Este teorema es muy útil porque en sistemas reales solo se dispone de secuencias temporales de sus salidas.



Fuente: Fotografía de dominio público. Disponible en:
<https://commons.wikimedia.org/w/index.php?curid=3662169>

Personaje 3

HENRI POINCARÉ (1854-1912)

Jules Henri Poincaré fue un matemático y físico francés, posiblemente el primero en descubrir el fenómeno del caos, aunque también trabajó en otros temas, como la teoría de la relatividad especial.

En una época histórica donde se pensaba que las matemáticas podían resolver cualquier problema, aún no se había encontrado una solución al llamado “problema de los tres cuerpos”: las leyes de Newton se pueden aplicar fácilmente a dos cuerpos, pero no a tres. Por ello, Poincaré participó en un concurso relacionado con ese tema propuesto por el rey de Suecia, quien quería saber si las órbitas de los planetas eran estables. Ya era conocido el cálculo

diferencial de Leibnitz y Newton, que usó para obtener aproximaciones al problema. Sus conclusiones fueron que, en general, no era posible calcular la trayectoria de tres o más cuerpos influidos mutuamente por la gravedad, pues cualquier pequeño error de medida inicial influía grandemente en los resultados. Pero también logró establecer que las órbitas conocidas no iban a cambiar sustancialmente en los próximos miles de años, lo cual fue suficiente para satisfacer al rey de Suecia y que le diera el premio. Aunque no le puso nombre, Poincaré fue el primero en entender y enfrentarse a problemas caóticos.

Hoy en día sabemos más sobre el problema de los 3 o más cuerpos. Por ejemplo, cuando dos objetos pequeños orbitan alrededor de uno grande en sincronía, esto es, los períodos de ambos tienen una relación entera, entonces el sistema es inestable. Esto se debe a que se encontrarán cerca (ejerciendo entre ellos la máxima influencia gravitatoria) una y otra vez en el mismo punto de sus órbitas. Es análogo a empujar un columpio siempre en sincronía: cada vez tomará más velocidad, pudiendo despedir al niño del columpio o hacerle dar vueltas de campana. Mientras que si se empuja el columpio en momentos elegidos al azar no se le transferirá energía apreciablemente.

Esa es la razón para que en los sistemas planetarios haya zonas orbitales vacías. En los sistemas de anillos de Saturno y otros planetas también existen zonas vacías —por los mismos motivos— que corresponden a órbitas sincrónicas con alguno de los satélites, por lo que el polvo fue expulsado de allí hace rato. De hecho, esa es la nueva definición de planeta: aquel cuerpo que ha barrido su órbita. Y Plutón todavía no lo ha hecho, por lo que ahora desciende a la categoría de los planetoides.

Resumen

Caos y fractales son dos aspectos del mismo objeto complejo. En la serie temporal vemos caos, y en su atractor vemos un fractal.

Volviendo a la argumentación principal del libro, el nivel de complejidad que se requiere para lograr caos es muy bajo, dado que se logra con una fórmula como la ecuación 15 (donde apenas aparecen sumas y multiplicaciones). Es decir, ni siquiera hace falta computación completa. Y para que aparezca solo se requiere una realimentación positiva y una negativa. El caos es creativo, pues permite generar complejidad donde antes no la había.

Pero, por otro lado, es difícil distinguir el caos de lo meramente estocástico. Como indicábamos en el capítulo dedicado a la complejidad, no hay un algoritmo que garantice realizar esta separación, pues para lograrlo se requiere inteligencia.

Si pensamos que un fenómeno es estocástico entonces no tiene ningún sentido tratar de controlarlo para sacarle provecho y mejorar nuestra supervivencia, dicho en términos evolutivos. Lo más que podemos hacer es obtener experimentalmente las distribuciones de probabilidad y apostar por la opción que tenga mejor esperanza matemática. Pero si nos damos cuenta de que el fenómeno realmente es caótico las cosas mejoran sustancialmente, pues eso significa que existe un modelo simple que puede ayudarnos a predecir el futuro, aunque permanezcamos limitados por un horizonte de predicción. Y podemos calcular los momentos donde el modelo es más o menos sensible al ruido. E incluso podemos controlarlo de forma muy efectiva. En EVALAB hemos hecho aportes en este sentido, controlando fácilmente sistemas caóticos usando algoritmos evolutivos (ver el trabajo de grado de Cristian Ríos, 2013). Los algoritmos evolutivos son especialmente indicados para estos casos donde se sabe que hay un orden estructural pero se desconocen los detalles, y dejamos a la evolución que se encargue de encontrarlos.

El caos tiene problemas muy serios desde el punto de vista ontológico. Decimos que hay caos en las ecuaciones de Lorenz que modelan el clima, pero sabemos que cualquier modelo es solo una aproximación a la realidad. Entonces, ¿también habrá caos en el clima? Generalizando, si tenemos una serie de datos registrados conforme transcurre el tiempo, de algún fenómeno físico, biológico, económico o cualquier otro, no se puede saber si la serie es caótica, porque para lograrlo tenemos que probar tres cosas: sensibilidad a condiciones iniciales, mezcla de órbitas y órbitas periódicas densas. Solo la primera condición se puede analizar

en un fenómeno real. Las otras dos se pueden analizar en modelos. Lo malo de los modelos es que suelen ser programas que corren en computadores digitales, por lo que nunca se llegan a manejar infinitas cifras decimales en los números. A consecuencia de ello, toda órbita termina por repetirse. Toda órbita es periódica. Es imposible tener caos en un sistema digital. En estos casos se habla de seudocaos. Es similar a lo que ocurre con los fractales, que es imposible que sean autosimilares en infinitas escalas, y nos conformamos con un margen y a los objetos resultantes los llamamos seudofractales. Y es lo mismo que ocurre con las variables aleatorias dentro de un computador, que resulta también que son periódicas —aunque con un periodo muy largo— y las llamamos entonces seudoaleatorias. Aunque no sepamos si realmente existe el caos, la metáfora que conlleva es muy útil y nos sirve para realizar predicciones más precisas que si no la usásemos.

Para saber más

- **Brian Goodwin (1998). *Las manchas del leopardo*. Barcelona: Tusquets Editores.**

Es un libro muy bueno y entretenido. Brinda un fundamento computacional a las coloraciones de la piel de los animales. Explica que no solo es la evolución la que guía el desarrollo de los seres vivos (criticando así a Dawkins), sino también las matemáticas, concretamente la teoría del caos. Muestra ejemplos de caos especialmente en reacciones químicas (Beloúsov-Zhabotinsky) y las compara con las ondas producidas por ciertos tipos de amebas para agregarse en un cuerpo multicelular. Habla, además, sobre hormigas naturales y artificiales, y tamaños críticos de hormigueros.

- **John Gribbin (2006). *Así de simple*. Barcelona: Crítica.**

Es un libro sencillo y tiene un prefacio algo pretencioso, al contrario de otros textos escritos por el mismo autor. Sin embargo, sirve de introducción a casi todos los temas de vida artificial (fractales, caos, teoría de juegos, autoduplicación, computabilidad, etc.). Explica varios sistemas caóticos como los asteroides, la herradura de Smale, la ecuación de Lorenz, la curva logística y el número de Feigenbaum. Muestra los fractales más conocidos. Da una buena introducción al caos y pone como ejemplos las reacciones químicas de Beloúsov-Zhabotinsky y las ecuaciones de Lotka-Volterra de predador-presa. Detalla cómo se produce la coloración de la piel de los

animales, con algunas conclusiones curiosas: la teoría predice que en los animales muy pequeños (ratón) y muy grandes (elefante), la piel debe ser de color uniforme, mientras que los de tamaño intermedio pueden tener franjas, rayas, pequeñas manchas, grandes manchas, conforme aumenta el tamaño (pero no del animal sino del embrión). Es imposible que las colas de esos animales terminen en manchas, sino que deben de ser a rayas. Muestra ejemplos de leyes de potencias en terremotos, tamaño de los meteoritos, ráfagas de luz de los quásares, temperatura de la Tierra, música, distribución de ciudades por cantidad de habitantes, atascos en autopistas, evolución de la bolsa, extinciones masivas, derrumbes en pilas de arena, entre otros. Habla también de los trabajos de Stuart Kauffman y John Maynard Smith. La competencia evolutiva es contra miembros de la misma especie: cuando un conejo brinca de una manera errática huyendo de un zorro, no está compitiendo evolutivamente contra el zorro sino contra los otros conejos. Ilustra además muchos sistemas de vida artificial, en especial DaisyWorld, que apoya la teoría GAIA de James Lovelock.

- **Nassim Nicholas Taleb (2008). *El cisne negro*. Barcelona: Paidós.**

Escrito por un corredor de bolsa que ha sufrido varios *cracks* y que demuestra lo falsas que son las asunciones estadísticas habituales (especialmente la campana de Gauss), este texto muestra que hay dos tipos de fenómenos aleatorios: “domesticados” (los juegos de casino, por ejemplo) que sí siguen distribuciones *gaussianas* de probabilidad; y los “salvajes” (casi todos los del mundo real), que siguen leyes potenciales, escalables, no integrables. Es decir, fenómenos caóticos. Los *cracks* de la bolsa, los terremotos, las guerras, la subida espectacular de las acciones de empresas tecnológicas y muchos otros fenómenos (adversos y también beneficiosos) siguen leyes potenciales. Analizarlos usando las campanas de Gauss es una equivocación y si lo hacemos suele acrecentar las pérdidas en el caso de los adversos, o impide que aprovechamos oportunidades en el caso de los benéficos.

- **Ian Stewart (1997). *El laberinto mágico*. Barcelona: Crítica.**

Libro sencillo sobre caos, con algunas cosas interesantes, aunque le falta profundidad. Explica la relación del juego “Torres de Hanoi” con el fractal de Sierpinski, el problema de Monty Hall, la inestabilidad del sistema solar y apenas menciona de pasada el control de caos para generar órbitas distintas a las tradicionales.

- **Ian Stewart (1991). *Juega Dios a los dados? La nueva matemática del caos*. Barcelona: Editorial Grijalbo-Mondadori.**

Habla de cómo diferenciar lo aleatorio de lo caótico, y de la complejidad. Cita los trabajos de Chaitin.

- **Miroslav M. Novak (ed.) (2006). *Complexus mundi: emergent patterns in nature*. New Jersey: World Scientific.**

Es una recopilación de artículos, usualmente bastante teóricos y de cierta dificultad. Recomiendo los siguientes:

Páginas 1-8: L. S. Liebovitch, V. K. Jirsa, L. A. Shehadeh, "Structure of genetic regulatory networks: evidence for scale free networks".

Páginas 9-31: Bruce J. West, "Modeling fractal dynamics". Plantea que las mediciones fisiológicas en humanos (flujo de velocidad arterial) son fractales. Y que cuando comienza una enfermedad su espectro multifractal se vuelve menos rico.

Páginas 113-133: Klaus Mainzer, "Complexity in nature and society". Excelente presentación de las ciencias de la complejidad y sus aplicaciones en economía. Después de dar definiciones (espacio de estados, dimensionalidad, atractores), cuenta que el sistema solar es no-computable (caótico). Muestra cómo reconstruir el espacio de estados de un sistema dinámico por medio de medidas de series temporales, retrasándolas en el tiempo varias veces. Describe además varios sistemas complejos físicos, químicos, ecológicos, las neuronas en el cerebro, así como fenómenos de autoorganización colectiva (tráfico de automóviles).

Páginas 155-170: Hermann Haken, "Synergetics on its way to life sciences".

Páginas 171-180: Nicoletta Sala, "Complexity, fractals, nature and industrial design: some connections".

Páginas 287-296: A. Nari, G. Ayad, S. Padulosi, T. Hodgkin, A. Martin, J. L. Gonzalez-Andujar, A. H. D. Brown, "Analysis of geographical distribution patterns in plants using fractals".

Páginas 297-304: Mario Markus, Malte Schmick, Eric Goles, "Hierarchy of cellular automata in relation to control of chaos or anticontrol".

Páginas 323-332: Dane R. Camp, "A cornucopia of connections: finding four familiar fractals in the Tower of Hanoi".

Páginas 333-342: W. Klonowski, E. Olejarczyk, R. Stepien, P. Jalowiecki, R. Rudner, "Monitoring the depth of anaesthesia using fractal complexity method".

- **David G. Green y David Newth (2005). *Towards a theory of everything? Grand challenges in complexity and informatics*. Complexity International, 8, pp. 1-12. Australia.** Disponible en: <https://goo.gl/bbFRVR>

El comienzo del artículo es muy interesante, introduciendo temas de complejidad y computabilidad (autoorganización, cambios de fase, la importancia de los grafos dirigidos, Prigogine, redes booleanas estocásticas...). Sin embargo, las conclusiones son de poca calidad.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Briggs, J. y Peat, F. D. (1994). *Espejo y Reflejo*. Barcelona: Editorial Gedisa.

Gleick, J. (1988). *Chaos. Making a New Science*. New York: Penguin Book.

Grebogi, C., Ott, E., Pelikan, S. y Yorke, J. A. (1984). Strange Attractors that are not Chaotic. *Physica D: Nonlinear Phenomena*, 13(1-2), pp. 261-268.

Li, T-Y. y Yorke, J. A. (1975). Period Three Implies Chaos. *The American Mathematical Monthly*, 82(10), pp. 985-992. DOI: <https://doi.org/10.2307/2318254>

Mandelbrot, B. B. (1997). *La geometría fractal de la naturaleza*. Barcelona: Tusquets Editores.

Martín, M. Á., Morán, M. y Reyes, M. (1998). *Iniciación al caos*. Madrid: Editorial Síntesis.

Novak, M. M. (2006). *Complexus Mundi. Emergent Patterns in Nature*. Singapore: World Scientific Publishing.

Schroeder, M. R. (1991). *Fractals, Chaos, Power Laws. Minutes from an Infinite*

Paradise. New York: W. H. Freeman and Company.

Spitznagel, C. R. (2000). *Vignette 9: The Chaos Game*. Recuperado el 2 de septiembre de 2017. Disponible en: <http://webserv.jcu.edu/math//vignettes/chaosgame.htm>

Thompson, C. H. (2004). *The Chaotic Ball: An Intuitive Analogy for EPR Experiments*. Recuperado el 8 de agosto de 2004. Disponible en: <http://arXiv:quant-ph/9611037v3>

PELÍCULAS Y VIDEOS

Aldoaldoz (2007). *Dynamic Geomag: Chaos Theory Explained*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=Qe5Enm96MFQ>

Peitgen, H. O. (2016). *Pendulum Interacting With 3 Magnets*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=xHux2AlM0a4>

Starrett, J. (2006). *Chaotic 1,2 pendulum*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=2JzMJNMYbRw>

TESIS Y TRABAJOS DE GRADO EN EVALAB

Ríos, C. L. (2013). *Control de sistemas dinámicos caóticos usando algoritmos evolutivos*. [Tesis Meritoria]. Cali: Universidad del Valle.

Romero, V. A. (2012). *Análisis del flujo de datos en redes de comunicaciones mediante teoría de caos*. Cali: Universidad del Valle.

Torres, A. R. (2011). *Objeto virtual de aprendizaje para la teoría del caos y su relación con los fractales*. Cali: Universidad del Valle.

LEYES DE POTENCIAS

El 20% de las personas más ricas de Italia poseen el 80% de los recursos. El 80% de las ventas viene del 20% de los clientes. Pero también un 20% de clientes son los que producen el 80% de las reclamaciones. Frases como esta se conocen como la regla 80/20 de Pareto, quien la enunció en 1896. Y son un caso particular de las leyes de potencias donde lo que se quiere recalcar es que la mayor parte de un fenómeno es debida a unos pocos sucesos. Se trata de una regla frecuente en ámbitos tan variados como la física, la biología, la economía o las ciencias sociales. Saber que un fenómeno sigue una ley de potencias puede tener consecuencias a la hora de afrontarlo. Por ejemplo, si averiguamos que el 80% de los delitos son causados por un 20% de delincuentes, la policía debería centrarse en buscar y detener a ese 20% —con lo cual la seguridad en la ciudad se mejoraría en un 80%— y no invertir tanto tiempo y recursos en el otro 80% de delincuentes, que solo aportarían una mejora del 20% a la seguridad de la ciudad.

Otro ejemplo es el de los terremotos. Hay una cantidad desmesuradamente alta de terremotos de magnitud pequeña, frente a una cantidad pequeñísima de terremotos de magnitud grande (figura 97). Otro ejemplo es que ha habido en nuestro planeta unas pocas grandes extinciones de seres vivos, y muchísimas pequeñas. Ambos fenómenos siguen leyes de potencias.

Las leyes de potencias se pueden formular matemáticamente modelando el fenómeno con una variable estocástica X , y caracterizando la probabilidad de que ocurra un evento de magnitud mayor o igual a un cierto valor x :

$$p(X \geq x) = K x^{-\alpha}$$

Ec. 24

donde α es una constante dependiente del fenómeno.

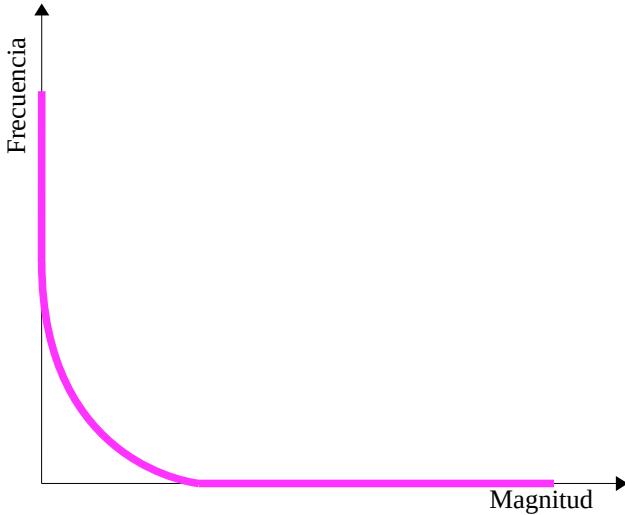


Figura 97: Frecuencia de los terremotos versus su magnitud.

Una propiedad interesante de las leyes de potencias es que no tienen una escala característica. Son libres de escala. Eso significa que si ampliamos o reducimos una de estas curvas, obtenemos la misma curva (ver ecuación 25, donde al multiplicar x por una constante c , al final sale la misma ley de potencias). Son autosemejantes, como los fractales.

$$p(X \geq cx) = K(cx)^{-\alpha} = Kc^{-\alpha}x^{-\alpha} = K'x^{-\alpha} = K'' p(X \geq x)$$

Ec. 25

Si tomamos logaritmos a ambos lados tenemos

$$\log(p(X \geq x)) = \log(K) - \alpha \log(x)$$

Ec. 26

Y si dibujamos esta ecuación usando ejes *log-log*, sale una línea recta de pendiente $-\alpha$ (figura 98).

Esta es una forma rápida de identificar fenómenos que siguen leyes de potencias, aunque como veremos enseguida, no es nada precisa.

Para considerar la frecuencia de un fenómeno x estamos usando su densidad de probabilidad $p(x)$, pero también se puede usar la densidad de probabilidad acumulada, en cuyo caso tendremos que hablar de la distribución de Pareto — como en los ejemplos que mencionamos al principio —, donde el número de eventos mayores que x es proporcional al inverso de una potencia de x . Pero el fenómeno es el mismo, lo que cambia es la forma de presentarlo. La principal ventaja de usar funciones de densidad de probabilidad acumulativas es que no requieren hacer histogramas, cuando partimos de una tabla de datos experimentales. Y, por tanto, no hay que hacer histogramas acumulando valores

en rangos, cuya anchura puede ser bastante discutible.

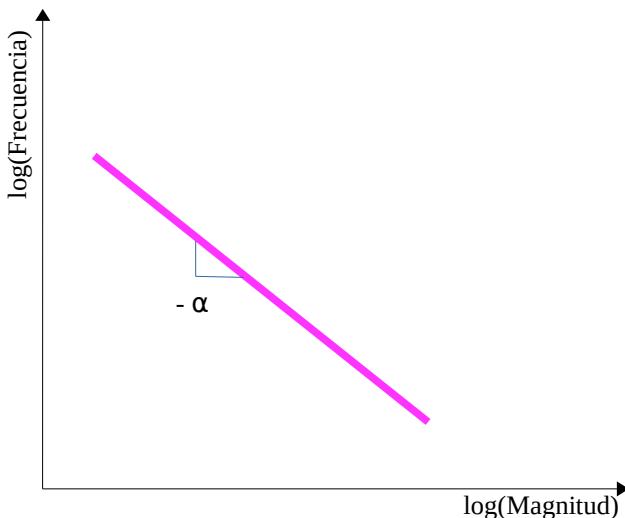


Figura 98: Ley de potencia en escala log-log.

Además de la regla de Pareto 80/20, las leyes de potencias tienen otras formulaciones. Están las leyes de Zipf que usan un gráfico de frecuencias (ordenadas) *versus ranking* (abcisas), es decir, se considera la frecuencia con que ocurre un fenómeno respecto a su orden en la tabla de frecuencias. Zipf ordenó las palabras que se encontraban en un libro escrito en inglés —primero la más frecuente y de última la menos frecuente— y encontró que para cada palabra su número de orden era inversamente proporcional a su frecuencia. Este tipo de distribuciones solo considera un tipo de variables, por ejemplo, la frecuencia de una palabra *versus su ranking*. Mientras que las distribuciones de Pareto consideran la relación entre dos variables distintas, como tamaño de una ciudad *versus el número de ciudades que tienen ese tamaño*.

Pero nada impide considerar en Pareto las dos siguientes variables: frecuencia de ocurrencia y *ranking* en una ordenación. Y entonces hay una relación obvia entre distribuciones de Pareto y de Zipf. Cuando decimos que el terremoto que ocupó el puesto N en el *ranking* tuvo una magnitud M (formulación de Zipf), ello equivale a decir que ha habido N terremotos de magnitud M o mayor (formulación de Pareto) (Adamic, 2017).

Y hay otras formulaciones más particulares, que se encuentran en artículos más antiguos. Por ejemplo, los primeros generadores algorítmicos de música usaban la distribución $1/f$, que significa que había muchas notas que diferían poco entre sí, y algunos pocos cambios grandes en las notas. Y la verdad, sonaban muy naturales, con melodías que tenían partes repetitivas y de vez en cuando algunas sorpresas, como es habitual en la música creada por humanos.

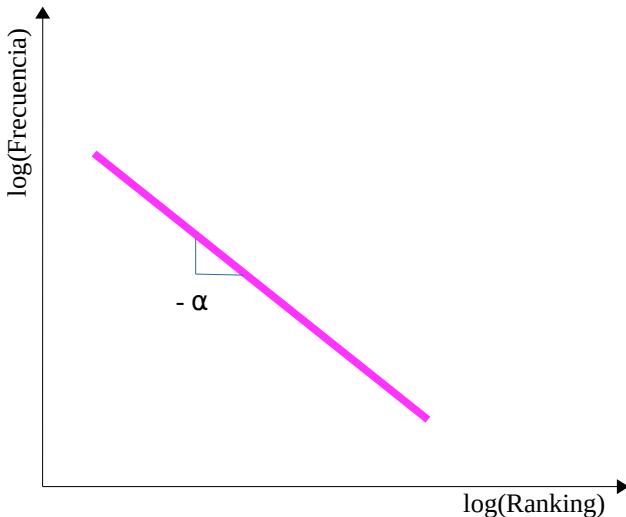


Figura 99: Distribución de Zipf (univariada).

Cuando no hay realimentaciones, las salidas de un sistema son fácilmente analizables pues podemos sacar promedios. La razón de ello es el teorema del límite central: cuando hay muchas variables estocásticas independientes, el promedio de todas esas variables sigue una distribución normal, también llamada campana de Gauss, donde el valor promedio está bien establecido, y la varianza (relacionada con la anchura de la campana) disminuye conforme aumenta el número de variables consideradas (figura 100). Eso es una buena cosa para los estadísticos pues a medida que se toman más muestras el promedio converge a un valor bien definido, lo que permite hacer predicciones sobre futuros valores de esas variables estocásticas.

Pero cuando hay al menos una realimentación positiva, la situación deja de ser así. Aparecen dependencias: la salida depende de las salidas anteriores. Y lo que le ocurre a una parte del sistema depende de lo que les ocurre a otras partes del sistema. Las variables estocásticas ya no son independientes, por lo que no aplica el teorema del límite central. En este caso no es sencillo sacar promedios, e incluso pueden no existir, en el sentido de que la suma no converja conforme se añadan más términos de la serie de datos, sino que presente fluctuaciones erráticas. Por otra parte, es difícil hacer predicciones acerca de lo que le pasará a ese sistema en el futuro. En la ecuación 24, cuando $0 < \alpha \leq 2$ entonces la distribución tiene una varianza infinita. Y cuando $\alpha \leq 1$ entonces la distribución también tiene una media infinita. En cualquiera de los dos casos, las correspondientes distribuciones son difíciles de manejar y no permiten hacer predicciones. En este sentido, los sistemas que responden a leyes de potencia, aumentan su libertad.

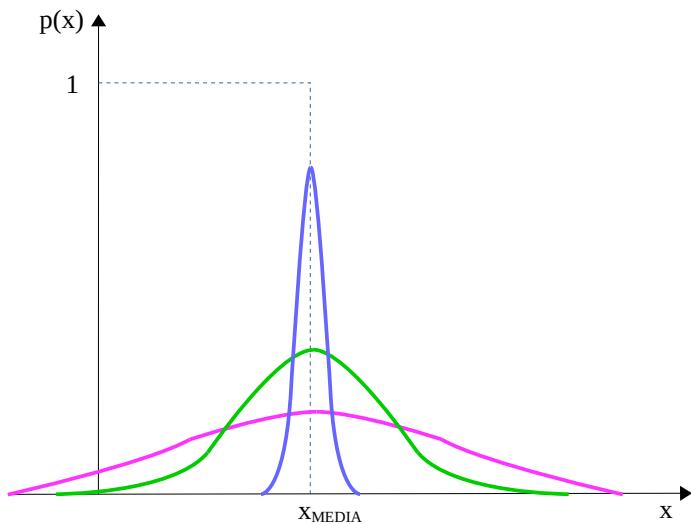


Figura 100: Distribuciones normales con varianza pequeña (azul), unitaria (verde) y grande (violeta).

Características de las leyes de potencias:

- Son libres de escala. Igual ocurre con las figuras fractales en el espacio, y con las series caóticas en el tiempo.
- No tienen promedio, por lo que no se pueden abordar con las herramientas tradicionales de la estadística. Tienen una “cola gorda”²⁵, o sea, que no disminuye rápidamente hacia cero. Y eso significa que puede haber eventos grandes e impredecibles.
- Al dibujarlas en un gráfico *log-log* sale una línea recta cuya pendiente es el exponente de la ley de potencias. Pero no toda línea recta implica una ley de potencias, pues otras distribuciones también dan líneas aproximadamente rectas. Además, las verdaderas leyes de potencia, de forma similar a lo que ocurre con los fractales y el caos, solo existen en los modelos matemáticos. Las que aparecen en la vida real no son perfectas, y hay escalas a las que dejan de cumplirse (lo que se llaman puntos de corte), por lo que pueden confundirse con otro tipo de distribuciones.
- Se generan de varias formas (Hilbert, 2013b):
 - Criticalidad autoorganizada. En las dunas, al superarse una pendiente crítica de alrededor del 35%, aumenta la probabilidad de que se produzca una avalancha que lleve el sistema por debajo de ese umbral. El tamaño de la avalancha respecto a su frecuencia sigue una ley de potencias. El fenómeno de avalancha es una realimentación positiva, pues un grano

²⁵ Se llama así en inglés, “*fat tail*”, pero no busquen el término en Internet, pues salen otras cosas.

de arena que rueda puede golpear y hacer rodar a muchos otros. Mientras que la pendiente umbral juega un papel de realimentación negativa hacia donde tiende a estabilizarse la duna.

- Conexión preferencial²⁶. Se modela muy bien con un grafo al que van añadiéndose nuevos arcos al azar, pero donde es más probable que llegue un nuevo arco a un nodo, cuantos más arcos ya tenga. A este fenómeno se le llama también *rich gets richer*. Hay muchísimos ejemplos en cualquier tema. Quien tiene muchos amigos conocerá a más gente a través de ellos y conseguirá así muchos más amigos. Los artículos más citados son más conocidos y, por ello, recibirán todavía más citas.
- Procesos de optimización (Sornette, 2000). Aunque no cualquier proceso, sino solo aquellos donde las variables a optimizar están relacionadas por medio de restricciones que siguen alguna ley de potencias.
- Crecimiento exponencial con difusión exponencial (de tecnologías, de animales o cualquier otro tipo). Son dos exponenciales que, al dibujarlas en una gráfica *log-log*, generan también una recta indicadora de ley de potencias (Hilbert, 2013).
- Relaciones geométricas área/volumen, que tienen que ver con el concepto de dimensión en los fractales, aunque realmente aquí no hay en absoluto fractales ni ningún tipo de complejidad. Cuando cualquier objeto cotidiano aumenta su volumen, su área crece más lentamente. Y la relación entre ambos es una ley de potencias.

$$\frac{\text{Área(esfera)}}{\text{Volumen(esfera)}} = \frac{4\pi r^2}{4\pi r^3/3} \propto r^{-1} \quad \text{Ec. 27}$$

Si el objeto es un fractal, por ejemplo los vasos sanguíneos tratando de llevar sangre a cada célula de un volumen, el exponente de la ley de potencias será distinto al de la ecuación anterior.

- Escalamiento alométrico. Relacionan algún fenómeno metabólico, como el consumo energético, las horas de sueño o la longevidad de un animal respecto a su peso. Suelen resultar exponentes de tipo $-n/4$ siendo n un entero pequeño (Hilbert, 2013b). Uno de los investigadores más destacados en esto es Geoffrey West (2011), quien ha encontrado una relación interesante entre bosques, ciudades y empresas. La ley de potencias más exacta y abarcante que conocemos es la que relaciona la

26 Preferential attachment.

tasa metabólica de un ser vivo con su peso, pues se aplica desde las amebas hasta las ballenas²⁷ y tiene un exponente de $\frac{3}{4}$. Eso significa que al duplicar el peso de un animal, solo requiere un 75% más de consumo diario de energía, para poder sobrevivir. Esto es, hay una economía de escala. Cuanto más grande es un ser vivo, más barato es de mantener. West encontró que en las ciudades pasa lo mismo: la infraestructura eléctrica, de calles y de comunicaciones crece más lentamente que el tamaño de la ciudad. Hay una economía de escala por medio de una ley de potencias similar a la biológica (aunque con un exponente de 0.85). A la vez que hay otras leyes de potencia, esta vez con exponentes mayores que 1 (concretamente 1.15), en los aspectos culturales de las ciudades, como las ofertas de trabajo, instituciones educativas, hospitalares, trabajos creativos, patentes, aunque también en crímenes y enfermedades. Esto es lo que atrae a la gente del campo a las ciudades y genera una distribución de tamaños de ciudades *versus* frecuencias que también sigue una ley de potencias, aunque ahora por el mecanismo de conexión preferencial. Lo mismo ocurre con las empresas, cuyo crecimiento va guiado por una ley de potencias con exponente menor que uno, es decir, hay economías de escala. Y la consecuencia que saca West es que como los seres biológicos nacen, crecen inicialmente muy rápido, luego más lento, hasta que finalmente se estancan y mueren, lo mismo ocurre con las empresas.

Un buen lugar donde encontrar las leyes de potencias que modelan diversos fenómenos junto a sus exponentes es KONECT (2017). Otro donde encontrar cómo generar estas distribuciones es el texto de Mitzenmacher (2004). Y una bonita e interesante introducción al tema sin ninguna complicación matemática la encontramos en Vsauce (2015).

Las leyes de potencia pueden ser crecientes, decrecientes o con colas hacia ambos lados. Lo más fácil de entender de una ley de potencias es que las colas no decrecen rápidamente. No hay un valor a partir del cual se pueda despreciar lo que queda de la cola. O sea que si hacemos una integral de la función de probabilidad desde menos infinito hasta más infinito para averiguar el valor promedio, no se puede despreciar ninguna parte de la función, no se puede truncar, todos los valores contribuyen al promedio y, precisamente por eso, el promedio diverge. Lo vamos a entender mejor con un par de ejemplos, sacando la estadística de la altura de las personas por un lado, y de sus salarios, por otro.

27 Aunque ya hay trabajos (Kolokotrones, 2010) que matizan que esta relación no es completamente potencial.

	Altura (cm)	Salario (USD)
Persona 1	170	450
Persona 2	168	350
Persona 3	184	600
Persona 4	181	0
Persona 5	184	480
Persona 6	179	520
Persona 7	157	390
Persona 8	177	410

Figura 101: Datos de altura y salario de 8 personas.

En una calle de Cali preguntamos la altura y el salario mensual que ganan todos los transeúntes. El resultado puede ser como lo que muestra la figura 101: son 8 personas y su promedio es 175 cm de altura y USD 400 mensuales de salario. Dejamos pasar más tiempo y ampliamos la cantidad de calles para hacer la encuesta. Van 500 personas y, por casualidad, pasó por una de las calles un jugador de baloncesto que media 210 cm. Al añadirlo a la lista, el promedio de alturas subió algo, pero realmente muy poco. Casi no afectó el promedio. Podemos estimar lo que afectó como $(210-175)/500 = 0.07$ cm, es decir, menos de un milímetro. El promedio de altura es ahora 175.07 cm.

Pero también encuestamos a un empresario de mucho éxito, cuyas ganancias mensuales son de USD 1.000.000. Al añadir este valor, el promedio salarial nos cambió por completo, y ahora es USD 2.399. Con un solo nuevo dato, el promedio salarial se multiplicó por 6. Si seguimos añadiendo muestras a la encuesta, por ejemplo, ampliéndola a todo el país e incluso a todo el planeta, nos iremos encontrando con algunas personas muy altas (pero nunca mayores a 272 cm que es el actual *Guiness Record*). De modo que el promedio de alturas converge rápidamente y no le afectan pequeñas desviaciones como esta. La altura de una persona está limitada por razones físicas y biológicas. No ocurre lo mismo para los salarios, que no tienen límite. El *record* mundial está en USD 333.000.000 mensuales, que movería sustancialmente cualquier promedio que tuviéramos. Y si dejamos pasar unos años, ese *record* seguirá subiendo.

Al hecho de que puedan aparecer datos extremadamente grandes e imprevistos se le llaman “cisnes negros” (figura 102), término popularizado por Nassim Taleb (2008) en el libro del mismo nombre, en el que cuenta anécdotas personales

sobre su vida de corredor financiero en Estados Unidos. Nassim nos explica que se pensaba que todos los cisnes eran blancos y que no se tenía ningún indicio de que pudiera ser de otra manera hasta que se encontró uno negro en las primeras expediciones a Australia. Los cisnes negros ocurren en economía, en la bolsa — con cierta frecuencia para nada previsible— y también en los terremotos que producen muchos daños pero nunca se sabe cuándo van a suceder. En definitiva, Nassim está hablando de leyes de potencia en su libro aunque nunca lo menciona por este nombre. Él piensa que el futuro no se puede predecir desde el pasado, y que lo único que hacen los mal llamados expertos es crear una narrativa causal convincente cuando sucede algún cisne negro, pero *a posteriori*. Nassim tiene un segundo libro continuación de este, donde explica que los cisnes negros también pueden ser eventos improbables pero de grandes ganancias, y que hay que saber exponerse a ellos.



Figura 102: Cisne negro, São Paulo.

Lo que más me impactó de este libro es su definición de “héroe anónimo”. Dice Nassim: supongamos que en fechas anteriores al ataque a las torres gemelas del 11-S, un funcionario regulador de seguridad en las compañías aéreas se da cuenta de que existe la posibilidad de un ataque con aviones, debido a que la puerta de la cabina de los pilotos permite entrar a cualquier pasajero. Podría sacar una ley obligando a las compañías aéreas a poner allí una puerta de alta seguridad. Y es natural pensar que las compañías aéreas se van a negar a hacer ese gasto extra, alegando que nunca ha pasado nada que pueda indicar que usar puertas endebles supone un grave peligro. Seguimos imaginando que, después de

un gran tira y afloja, el funcionario visionario logra promulgar la ley, con lo que todas las compañías aéreas se verán forzadas a poner esas puertas de seguridad. ¿Qué pasará después? Nada. No habrá atentados, y al cabo de los años lo despedirán por obligar a hacer gastos desmesurados en medidas que no sirvieron para nada. Habrá salvado muchas vidas, pero precisamente por eso nadie se dará cuenta de ello. Es un héroe anónimo que morirá de hambre.

Aquí radica una de las causas de la impredecibilidad. El futuro, ontológicamente, no existe. De modo que si alguien trata de modificarlo no sabrá nunca si lo ha logrado. Por supuesto, esto es de aplicación a fenómenos modelados por leyes de potencias. Cuando tenemos otras leyes de probabilidad con promedios y varianzas bien determinados sí se pueden hacer predicciones razonables.

Por desgracia, las leyes de potencias nos generan mucha confusión. Por un lado, que en una gráfica *log-log* aparezca una recta no quiere decir que se trate de una ley de potencias. Hay exponenciales lentas y curvas *log-normal* que también podrían confundirse con rectas. Se debe usar algún método para determinar la confiabilidad del ajuste de datos a un modelo potencial. Lo que es peor, se han hecho experimentos con simuladores sintéticos de leyes de potencias y a veces los algoritmos que emplean los estadísticos fallan en reconocerlas.

Por otro lado, aunque la mayoría de las veces en las que hay una ley de potencias esto es un indicador de un fenómeno complejo subyacente, en ocasiones no es así. Por ejemplo, al generar palabras al azar en un idioma inventado (poniendo simios a escribir en un teclado) también aparecen las mismas leyes de potencias que encontró Zipf en el inglés, español y demás idiomas humanos.

Por último, también hay muchas publicaciones con equivocaciones o que se contradicen unas a otras. Por ejemplo, no es cierto que partir un palo en trozos al azar genere una distribución potencial, como incorrectamente indica Gell-Mann (1996) que, por lo demás es un excelente libro y de los primeros en tratar el tema de la complejidad. Al igual que en los fractales y en el caos, existen varios mitos alrededor de este tema, tanto por exceso (creer que casi todo genera leyes de potencias) como por defecto, que ahora está muy de moda, como una especie de efecto rebote. Recordemos que estamos trabajando con modelos, que son únicamente una aproximación a la realidad. En este sentido no existe ningún fractal en la naturaleza, ni tampoco el caos, ni las leyes de potencia. No nos descubren nada nuevo quienes critican estos modelos. Pero los modelos son simplificaciones útiles, que capturan algo importante de los objetos naturales, aun cuando no sean perfectos. En este sentido es válido decir que un helecho es un fractal porque podemos ver la misma estructura geométrica en 4 niveles. Desde

luego que hay límites, tanto en lo grande (es imposible tener helechos que sobrepasen el tamaño del universo) como en lo pequeño (de tamaño menor a un átomo). ¿Es el clima caótico? ¿O solo lo es su modelo simplificado, de las ecuaciones de Lorenz? Esas reflexiones tan punitivas no nos dejan ver el bosque: estamos estudiando modelos de donde se pueden deducir propiedades aproximadas de los objetos que representan.

De todos modos se puede hacer una reflexión interesante: si resulta ser que el mundo en que vivimos es solo computación, quizás algún día podamos llegar a tener modelos que coincidan exactamente con la realidad, si logramos caracterizar bien cada uno de los fenómenos complejos que aparecen (caos, fractales y leyes de potencia).

Resumen

Cuanto más rico eres, más dinero puedes conseguir. Cuanto más poder tienes, más poder conseguirás. La gente prefiere vivir en las ciudades más grandes, con lo que se agrandarán aún más. Los artículos de investigación más referenciados tienen mayor visibilidad, por lo que es más probable que vuelvan a ser citados en trabajos posteriores. Los investigadores quieren publicar en las revistas más grandes y con mayor prestigio, con lo cual esas revistas crecerán aún más. Podríamos seguir enumerando casos. Pero a donde quiero llegar es que si no te gusta esta situación, si no te gustan las concentraciones de poder, de dinero o prestigio, no busques soluciones en la economía ni en la sociología. Búscalas en las matemáticas. Porque este proceso ocurre debido a realimentaciones que habitualmente dan lugar a distribuciones estadísticas potenciales, *log-normales* o exponenciales. Y también al revés. Cuando en nuestros datos aparezcan alguna de estas distribuciones, es un buen indicador de que estamos lidiando con un fenómeno de alta complejidad.

Para saber más

- **Per Bak y Kan Chen (1991). Self-Organized Criticality. Rev. Scientific American, 264(1), pp. 46-53.**

Las pilas de arena se derrumban al alcanzar cierta pendiente. Suelen ocurrir muchos derrumbes pequeños y pocos grandes. A ello se le llama *power*

laws (leyes de potencias), que son otra forma de decir “caos”. Las dunas son todas similares debido a este fenómeno de autoorganización de la pendiente.

- **Henrik Jeldtoft Jensen (1998). *Self-Organized Criticality*. New York: Cambridge University Press.**

Crítica al artículo anterior. De todos modos, lo que dice Bak es correcto si se sustituye los pequeños granos de arena por bolas con cierta fricción. Propone otros ejemplos donde sí se dan las leyes de potencia.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Adamic, L. A. (2017). Zipf, Power-laws, and Pareto: a ranking tutorial. *Information Dynamics Lab, HP Labs*. Recuperado el 10 de septiembre de 2017. Disponible en: <http://www.labs.hp.com/research/idl/papers/ranking/ranking.html>

Gell-Mann, M. (1996). *El quark y el jaguar*. Barcelona: Tusquets.

Hilbert, M. (2013). Scale-free power-laws as interaction between progress and diffusion: a critical evaluation of fat-tail distributions. *Complexity*. Wiley. DOI: <https://doi.org/10.1002/cplx.21485>

Kolokotrones, T., Savage, V., Deeds, E. J. y Fontana, W. (2010). Curvature in metabolic scaling. *Nature* 464, pp. 753-756. DOI: <https://doi.org/10.1038/nature08920>

KONECT (2017). Power law exponent. Recuperado el 10 de septiembre de 2017. Disponible en: <http://konect.uni-koblenz.de/statistics/power>

Mitzenmacher, M. (2004). A Brief History of Generative Models for Power Law and Lognormal Distributions. *Internet Mathematics*, 1(2), pp. 226-251.

Sornette, D. (2000). *Critical Phenomena in Natural Sciences. Chaos, Fractals, Selforganization and Disorder: Concepts and Tools*. Springer.

Taleb, N. N. (2008). *El cisne negro*. Barcelona: Paidós.

PELÍCULAS Y VIDEOS

Hilbert, M. (2013b). 9 CCSSCS: Leyes de potencias. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=4uDSEs86xCI>

Vsauce (2015). The Zipf Mystery. Recuperado el 20 de octubre de 2016. Disponible en: <https://www.youtube.com/watch?v=fCn8zs912OE>

West, G. (2011). The surprising math of cities and corporations. TED. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=XyCY6mjWOPc>

ALGORITMOS EVOLUTIVOS

La historia de los algoritmos que tratan de simular un proceso evolutivo es paralela a la historia de los primeros computadores. Uno de los éxitos de esa época fue un *software* para jugar a las damas, desarrollado por Arthur Samuel en 1959, donde ya hay algunos indicios de comportamiento adaptativo. Sin embargo, es a otros investigadores a los que se les reconocen los méritos de las mejores propuestas, seguramente porque son las más generales, y también por su más amplia divulgación. En Estados Unidos el primero fue John Holland, a quien se le considera el padre de los algoritmos genéticos y su aplicación a animales artificiales que se adaptaban a un entorno artificial (*Animats*). Uno de sus estudiantes, John Koza, diseñó una variante que permitía fabricar programas automáticamente. Luego, de forma independiente, se hicieron otros trabajos también en Estados Unidos, entre ellos los de Lawrence Fogel (programación evolutiva para predecir series de datos temporales), y en Europa destacan los de Rechenberg, Schwefel y Bienert, que desarrollaron las estrategias evolutivas como un método para diseñar perfiles de alas de aviones. De ellos se derivan todos los trabajos posteriores.

También vamos a ver *Simulated Annealing*, que es un algoritmo inspirado en la física pero, curiosamente, muy similar a los algoritmos evolutivos basados en la biología, por lo que se explicará también aquí. Este hecho es otro indicador de que la evolución no solo es una cuestión de animalitos, sino que es un algoritmo general que el universo usa prácticamente para todo.

Todos estos algoritmos cumplen con las cuatro características que debe haber para que haya evolución: población de entes, que se reproducen, con errores en las copias, y sometidos a una presión selectiva. Sin embargo, algunos de ellos nacieron con restricciones producto de la poca capacidad de cómputo en aquella época, como tener poblaciones muy reducidas (a veces de apenas dos entes); o tener poblaciones de tamaño fijo (debido a la imposibilidad de crear matrices dinámicas en los lenguajes de programación de entonces); o usar codificaciones binarias (definitivamente un error); o usar operadores de reproducción equivocados (el estándar de hoy es tener al menos dos: la mutación y el cruce); o

aplicar una presión selectiva determinista (elegir siempre a los mejores, lo cual es, definitivamente, un error).

Las cuatro condiciones para que haya evolución son muy precisas, aunque se pueden resumir un poco más debido a que si hay un proceso de copia es inevitable tener una población. Por otro lado, en el universo físico en que vivimos siempre podemos contar con que haya errores, aunque conviene advertir que en universos digitales ello no tiene que ser así: el ruido es prácticamente inexistente, por lo que debe ser introducido a propósito. La presión selectiva sí es importante en cualquier caso, porque si se elimina nos quedamos con un proceso de crecimiento exponencial nada más, donde todo es posible.

En un mundo con recursos finitos, como en el que vivimos, la presión selectiva es fuerte: solo podrán reproducirse los que logren adquirir y usar esos recursos de forma más rápida y eficiente que los demás. Pero en universos digitales no tiene por qué ser así. En un mundo con infinitos recursos, la evolución produce cualquier tipo de entes, pues la presión selectiva es suave (nada está prohibido y se acepta el derroche). Todo es posible, en mayor o menor medida. En un mundo donde todo es posible, nada se puede entender, clasificar ni predecir. La inteligencia no puede existir. No valdría para nada. No tendría ninguna importancia ni utilidad.

Por ello y como resumen, se puede decir que los motores de la evolución son tres: la reproducción —que produce más de lo mismo—, las mutaciones —que producen variedad—, y la presión selectiva —que aumenta la complejidad del sistema—. Y teniendo en cuenta los matices que acabo de señalar, podemos decir que la evolución es un algoritmo que emerge cuando hay reproducción, que a su vez emerge cuando hay suficiente complejidad.

La evolución existe en biología y Darwin lo descubrió, pero no está limitada a ella. Puede ocurrir y ocurre en sistemas mecánicos, hidráulicos, electrónicos, económicos, sociales o artísticos. En particular donde es más fácil implementar el algoritmo de la evolución es con herramientas informáticas. Allí aparecen dos grandes grupos de algoritmos evolutivos, la evolución de objetos y la evolución de programas, que veremos en detalle a continuación.

Cuando los objetos son estructuras de datos:

- **Algoritmos genéticos.** Lo que evoluciona son vectores de números enteros, booleanos, símbolos o cadenas de caracteres.
- **Evolución diferencial.** Una variante del anterior, con operadores de

reproducción distintos.

- **Algoritmos genéticos híbridos de Taguchi.** Otra variante del anterior, con un operador de cruce más sofisticado.
- **Estrategias evolutivas.** Lo que evoluciona son vectores de números flotantes.
- **Enfriamiento simulado.** Lo que evoluciona son cualquier tipo de dato o estructura de datos.

Cuando los objetos son programas de computador:

- **Programación genética.** Lo que evoluciona son programas de computador.
- **Gramáticas evolutivas.** Lo que evoluciona son programas de computador.
- **Programación por expresión genética.** Lo que evoluciona son programas de computador.

Y cuando tenemos realizaciones mixtas, donde se pueden interpretar los objetos como datos o como programas:

- **Sistemas clasificadores.** Lo que evoluciona es un conjunto de reglas *IF – THEN – ELSE*.
- **Programación evolutiva.** Lo que evoluciona son típicamente máquinas de estado, que se pueden considerar estructuras de datos o también reconocedores de lenguajes regulares.

En la práctica no hay por qué mantener estas distinciones históricas. Nada impide mezclar distintos tipos de datos e incluso datos con programas y hacerlos evolucionar a ambos.

Algoritmos genéticos

Los algoritmos genéticos²⁸ fueron ideados por John Holland en los años 70 y popularizados por uno de sus estudiantes, David Goldberg. Los libros y artículos de Holland son muy simples, y se recomienda mejor leer los de Goldberg, que

²⁸ *Genetic Algorithms*.

contienen además varias aplicaciones interesantes. Los algoritmos genéticos se usan principalmente en optimización paramétrica: queremos encontrar la mejor solución para un problema que se modela con un conjunto de parámetros. Los algoritmos de optimización se pueden clasificar de varias formas:

Clasificación según el tipo de solución:

- Numérico: la solución son N parámetros.
- Combinatorio: la solución son N parámetros ordenados.

Clasificación según el grado de uso del azar:

- Determinista.
- Estocástico.
- Orientado.

Clasificación según la dirección de búsqueda:

- Primero en profundidad (explotador).
- Primero en anchura (explorador).

Clasificación según los candidatos a solución:

- Simple.
- Múltiple.

Clasificación según use información específica:

- General.
- Heurística.

En este sentido, los algoritmos genéticos se emplean en problemas numéricos. Son orientados, es decir, ni completamente deterministas ni completamente estocásticos. La búsqueda la hacen simultáneamente en profundidad y en anchura, o sea, son exploradores-explotadores. Manejan múltiples candidatos a solución simultáneamente. Y son generales.

Adicionalmente, pueden modificarse ligeramente para resolver problemas combinatorios y también permiten fácilmente incorporar heurísticas, pero entonces se vuelven menos generales. Hay que tener en cuenta (figura 103) que los algoritmos más generales son menos eficientes y viceversa.

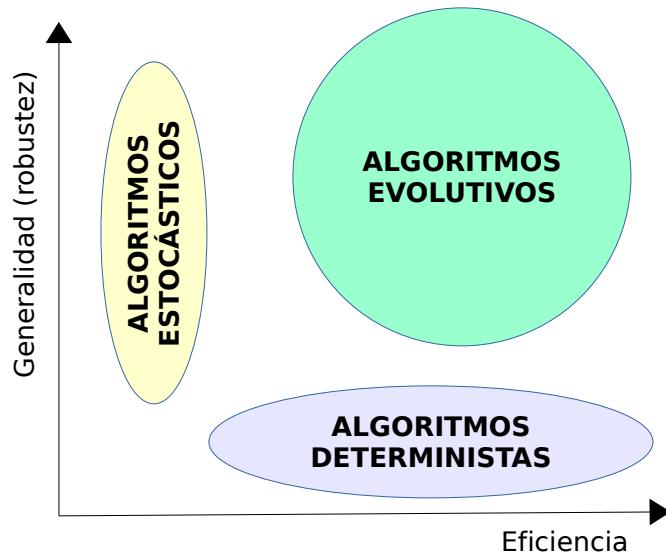


Figura 103: Generalidad versus eficiencia en cualquier tipo de algoritmo de optimización.

Los algoritmos deterministas no son buenos cuando se enfrentan a problemas multimodales (con varios óptimos, ver figura 104), pues se van a quedar atrapados en el primer óptimo local que encuentren.

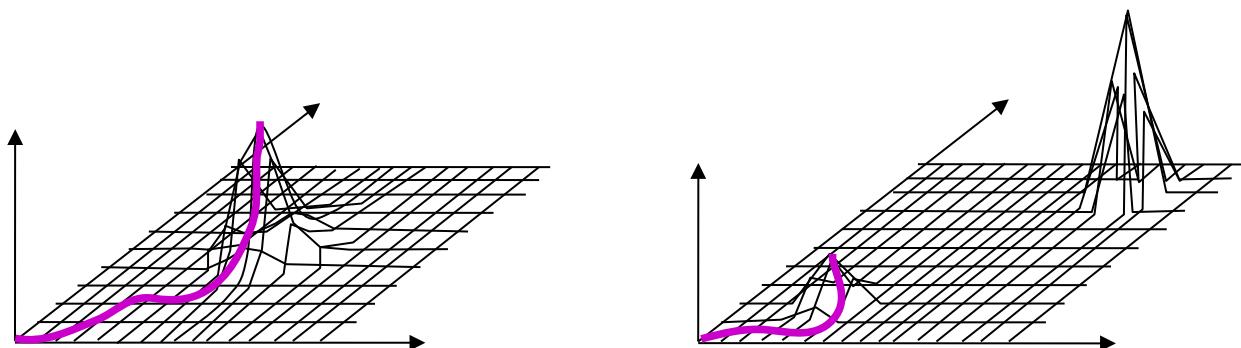


Figura 104: Problema con un solo máximo y con varios máximos.

Los algoritmos genéticos exploran varias soluciones simultáneamente y lo hacen de forma no determinista, por lo que suelen ser mejores que otros algoritmos cuando el problema tiene muchos óptimos locales.

Diagrama de flujo de datos

Un algoritmo genético está conformado por una población de cromosomas (típicamente 100, 1000 o más). Cada cromosoma representa una posible solución (buena o mala) a mi problema. Al enfrentar cada cromosoma al problema se obtiene una aptitud que indica lo bueno o malo que es. Después se hace una selección de unos cuantos cromosomas que son los que pasarán al *mating pool* para reproducirse. Los cromosomas hijos resultantes se inyectan a la población. Y todo ello se vuelve a repetir. Cada repetición constituye una generación. El diagrama de flujo de datos puede verse en la figura 105.

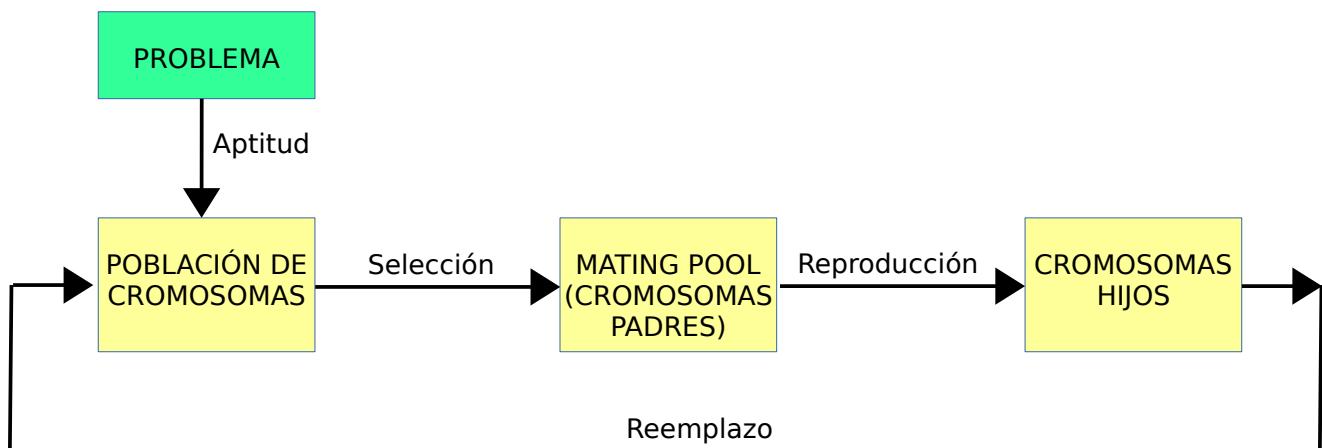


Figura 105: Flujo de datos de todo algoritmo evolutivo.

El número de generaciones puede ir desde unas pocas decenas hasta miles de millones o más, dependiendo del tiempo de cómputo disponible y de la calidad de las soluciones encontradas.

A continuación, en los siguientes apartados se explican los detalles del algoritmo.

Población de cromosomas

Se requiere una población de objetos que, en la jerga computacional, se llaman cromosomas (robando términos alegremente a la biología). Cada cromosoma está formado por un conjunto de parámetros (llamados genes). Todos los cromosomas tienen los mismos genes y en el mismo orden. Cada gen puede tomar distintos valores llamados alelos. En general, los alelos que toman los genes de un cromosoma diferirán de los alelos de otros cromosomas en el mismo gen. Y los genes tienen un significado posicional, es decir, si los cromosomas son vectores de 35 elementos, existirá el gen 0, el gen 1, y así hasta el gen 34. Por otro lado, cada gen puede tener un conjunto de alelos distinto al de otro gen.

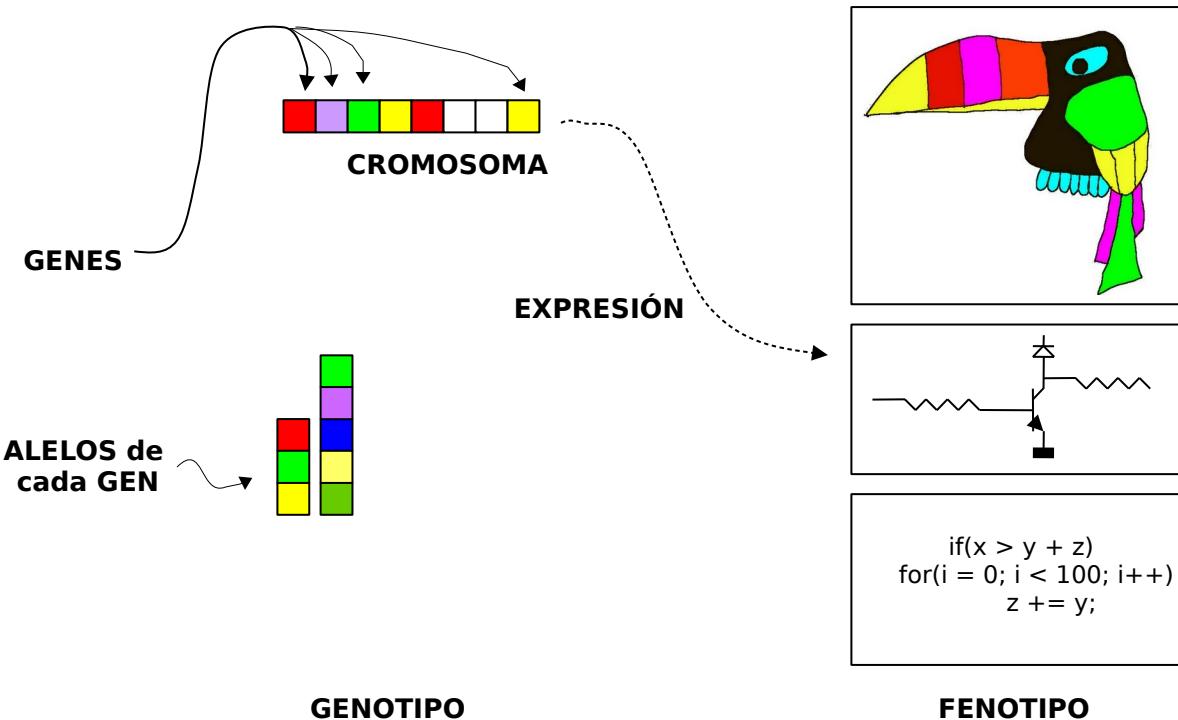


Figura 106: El fenotipo se expresa, creando el genotipo.

Se suele diferenciar el cromosoma del objeto que representa: el cromosoma es un conjunto de parámetros que está en el espacio de los genotipos, mientras que el objeto es físico, tiene forma, está en el espacio de los fenotipos. En la figura 106 destacamos como ejemplos un tucán, un circuito electrónico y un programa de computador. La traducción de un cromosoma a su objeto físico se llama expresión.

A cada cromosoma se le calcula su aptitud para saber lo bueno o lo malo que es resolviendo un problema. Cada cromosoma codifica, de alguna manera, una solución posible al problema (puede ser una solución buena, mala o malísima, pero solución a fin de cuentas).

Esto es lo importante: cada cromosoma contiene información de un objeto que representa la solución a mi problema. Por ejemplo, si quiero diseñar un automóvil que sea veloz, que consuma poco, que sea seguro y que sea fácil de aparcar, cada cromosoma debe describir un auto de entre todos los autos posibles. La forma de codificar el objeto es muy variada. En el caso de los algoritmos genéticos se logra con un vector de parámetros (cilindrada del motor, radio de las ruedas, tipo de combustible), pero nada impide usar una estructura de datos más elaborada (listas, matrices, árboles).

Habitualmente se emplean algoritmos genéticos cuando el espacio de búsqueda es enorme (por ejemplo, hay trillones de formas de fabricar un automóvil),

mientras que la población de cromosomas suele tener un tamaño limitado (dependiendo de la capacidad de cómputo disponible), pero es razonable tener apenas 100, 1000 o 10 000 cromosomas. Los cromosomas se generan inicialmente al azar, de modo que es de esperar que al principio implementen soluciones muy malas (un auto con ruedas gigantes pero con motor muy pequeño, o cosas más disparatadas).

Es muy importante que la codificación de objetos en cromosomas cumpla con dos propiedades:

- **Completitud:** todo objeto factible debe tener un cromosoma que lo represente.
- **Cerradura:** todo cromosoma debe representar un objeto factible.

En la figura 107 se muestran dos contraejemplos.

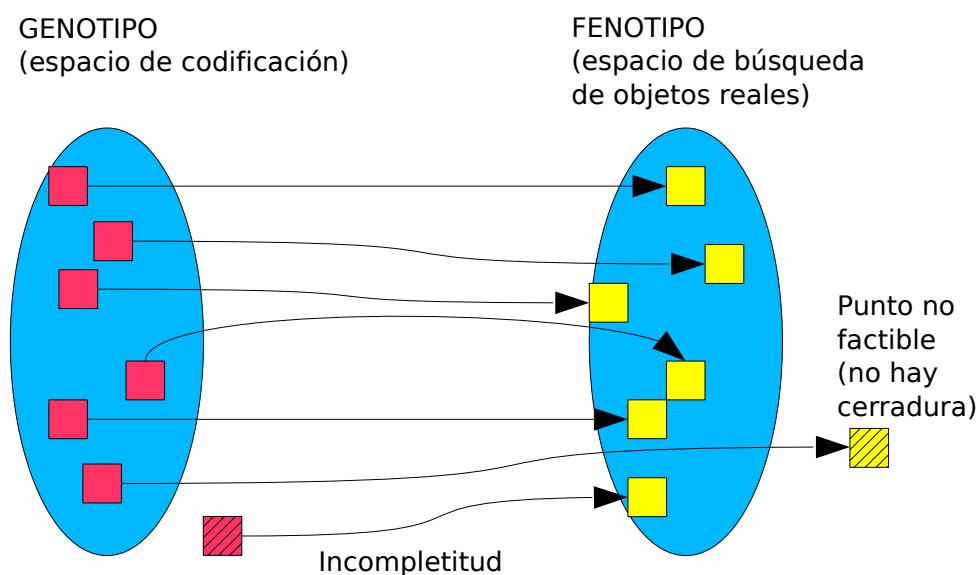


Figura 107: Ejemplo de incompletitud y de no cerradura.

Entonces como decíamos, hay que disponer de una población de estos cromosomas. ¿Cuántos? Cuantos más, mejor. Sin embargo, dependemos de las limitaciones de memoria de nuestro equipo de cómputo de modo que podemos pensar en un número entre 100 y 10 000. Inicialmente rellenaremos cada gen de estos cromosomas con cualquier alelo válido, elegido al azar (figura 108).

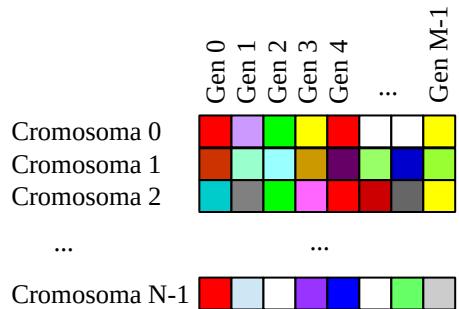


Figura 108: Población de N cromosomas con M genes cada uno, inicializada al azar.

Cálculo de la aptitud

Cada cromosoma se expresa en el objeto que representa y luego se le enfrenta con el problema que tratamos de resolver. En el ejemplo anterior, debemos tomar cada cromosoma y fabricar el automóvil que representa o, al menos, construir una simulación computacional de este. Y el problema a resolver puede ser un circuito de pruebas, es decir, una carretera con rectas, curvas y obstáculos, bien sea real o simulada. Esto último es más barato, en materiales y en tiempo. La única desventaja de las simulaciones es que pueden perderse detalles menores que sean importantes. Por ello es común comenzar con simulaciones y después expresar los mejores cromosomas en objetos reales, para continuar la evolución allí.

Al enfrentar el objeto así construido al problema, debemos calcular su aptitud, que es un número que indica lo bien o mal que se desempeñó. Las aptitudes se suelen mantener normalizadas (por ejemplo, entre 0% y 100%) de modo que si el auto circula por la pista de pruebas pero se choca o se sale en las curvas o no llega al destino le asignaré 0% y conforme vea que avanza por la carretera, le iré asignando una aptitud cada vez mayor, hasta llegar al 100% si lo hace muy bien y cumple con todos mis requerimientos de velocidad, consumo y seguridad.

No se requiere que la aptitud sea una función, matemáticamente hablando, y mucho menos que sea continua, derivable o repetible. Es simplemente un indicador de lo buena o mala que es una solución. Solo se le exige que sea monótona, en el sentido de que soluciones mejores tengan aptitudes mejores. Pero incluso si esto no se logra del todo, el algoritmo genético funcionará también, aunque irá degradando la calidad de las soluciones.

Muchas veces se llama función de evaluación a la función de aptitud cuando está sin normalizar. Pero ello no tiene mayor importancia. Dependiendo del método de selección que utilicemos, que vamos a ver a continuación, se requerirá que la

aptitud esté normalizada. Pero no siempre es necesario hacerlo. El proceso de normalización se lleva a cabo siguiendo el sentido común, es decir, buscando (teórica o prácticamente) el valor máximo y dividiendo cada aptitud de cada cromosoma por este valor. A veces no hay un valor máximo teórico, pero siempre podremos obtener el mayor valor de todos los cromosomas existentes en un momento dado. La normalización en ocasiones se hace de forma no lineal, como exponencial o logarítmica, aunque ello rara vez aportar alguna ventaja. Más adelante veremos que sí hay una forma de normalizar, llamada *ranking*, que tiene muchas ventajas prácticas.

En la figura 109 observamos un ejemplo sencillo de espacio de búsqueda unidimensional con una población de 3 cromosomas generados al azar. Aunque hemos dibujado la aptitud como una función continua, para darnos una idea de cómo es el espacio de búsqueda, lo cierto es que solo conocemos los 3 puntos calculados para los cromosomas.

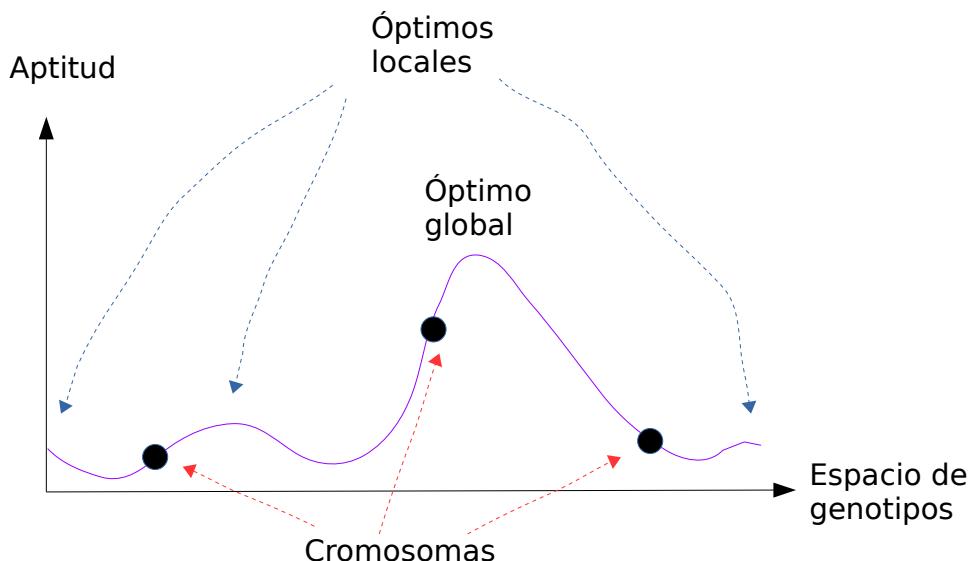


Figura 109: Ejemplo de espacio de búsqueda, con 3 cromosomas.

Selección

A continuación se hace un proceso de selección que consiste en elegir a los K mejores cromosomas con mayor probabilidad. Eso significa que los peores tienen menos probabilidad, pero no se les descarta de antemano. Es importante que la selección no sea determinista (como los algoritmos de escalada, que eligen siempre las mejores soluciones) porque en caso contrario se corre más riesgo de quedar atrapado en óptimos locales, como veíamos en figura 104.

Hay varios algoritmos para hacer la selección. Los más usados son:

Sorteo. La probabilidad de ser seleccionado es directamente proporcional a la aptitud. Es como si en un sorteo de lotería se repartieran 100 billetes entre los cromosomas concursantes. Hay un cromosoma de aptitud 3 al que se le asignan 3 billetes, otro de aptitud 25 al que se le asignan 25 billetes y otro de aptitud 72 al que se le asignan 72 billetes. Después se juega la lotería, seleccionando al azar uno de los 100 billetes. Está claro que con este esquema los cromosomas de mayor aptitud tienen ventaja, pero los cromosomas de menor aptitud también pueden ser seleccionados eventualmente.

Entonces, para cada uno de los N cromosomas con aptitud u_i se calcula su probabilidad de ser seleccionado, así:

$$p_i = \frac{u_i}{\sum_{i=1}^N u_i} \quad \forall i=1, \dots, N \quad Ec. 28$$

Se calculan luego las probabilidades acumuladas:

$$\begin{aligned} q_0 &= 0 \\ q_i &= p_1 + \dots + p_i \quad \forall i=1, \dots, N \end{aligned} \quad Ec. 29$$

Por último, se generan K números aleatorios r_j y se seleccionan los K cromosomas tales que:

$$q_{i-1} < r_j \leq q_i \quad \forall j=1, \dots, K \quad Ec. 30$$

Pongamos un ejemplo, con una población de $N=8$ cromosomas y un *mating pool* de $K=3$ cromosomas.

CROMOSOMA	Evaluación e_i	Aptitud normalizada (%) u_i	Probabilidad de ser seleccionado p_i	Probabilidad acumulada q_i
C1	40.5	15	0.15	0.15
C2	13.5	5	0.05	0.2
C3	27	10	0.1	0.3
C4	40.5	15	0.15	0.45
C5	81	30	0.3	0.75
C6	27	10	0.1	0.85
C7	13.5	5	0.05	0.9
C8	27	10	0.1	1

Tabla 5: Ejemplo de cromosomas.

Al evaluar los cromosomas, la suma de todas sus evaluaciones da 270. Entonces, dividiendo cada evaluación por 270 y multiplicando por 100, salen sus respectivas aptitudes en porcentajes. El cálculo de probabilidades de selección y probabilidades acumuladas es directo siguiendo las fórmulas anteriores y sus resultados pueden verse en la figura 110.

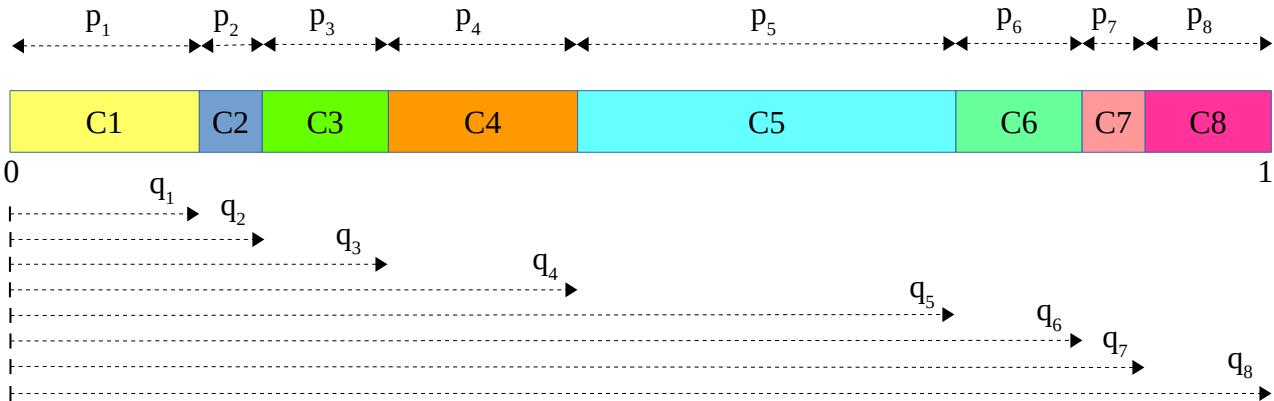


Figura 110: Ejemplo de cálculo de probabilidades de selección y probabilidades acumuladas.

Si, por ejemplo, al generar los 3 números aleatorios salen 0.58, 0.14 y 0.69, entonces los cromosomas seleccionados serán C5, C1 y C5 (figura 111).

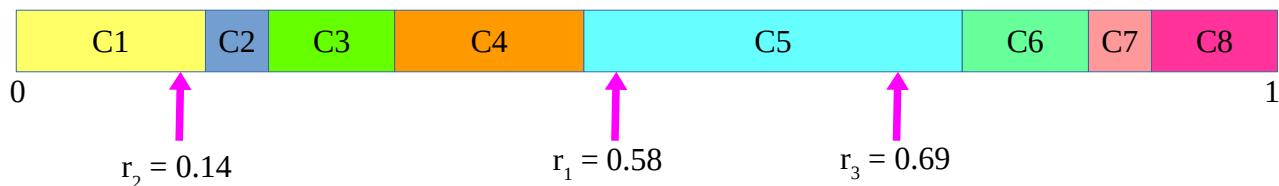


Figura 111: Ejemplo de selección de 3 cromosomas por sorteo.

Ruleta. Es igual al anterior, con la única diferencia de que no hace falta generar K números aleatorios (recordemos que ello es costoso computacionalmente), sino que basta generar uno solo (r). Si pensamos en la regla anterior como si fuera una ruleta circular, los demás números se sitúan de forma equiespaciada, a partir del número aleatorio generado.

$$r_j = \frac{r + j - 1}{k} \quad \forall j = 1, \dots, k \quad \text{Ec. 31}$$

Continuando con el mismo ejemplo, si el número aleatorio sale $r=108$ grados, como son 3 cromosomas a seleccionar entonces hay que sumar $360/3=120$ grados para ir obteniendo los demás, con lo cual saldrán $r_1=108$, $r_2=228$ y $r_3=348$ grados (figura 112). Esto implicaría que se seleccionan los cromosomas C4, C5 y C8. En la práctica no hace falta usar un círculo y grados sino que

podemos continuar en el segmento real $[0,1]$ y hacer las operaciones de suma módulo 1.

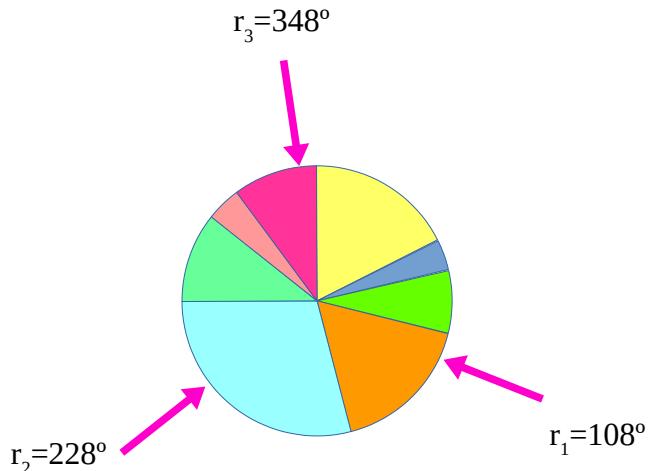


Figura 112: Ejemplo de selección de 3 cromosomas por ruleta.

Muestreo universal por restos. Cada cromosoma i se selecciona directamente $\text{floor}(K*p_i)$ veces. Las vacantes que queden hasta completar las K muestras se reparten por el método del sorteo o de la ruleta. Este sistema garantiza no perder a los mejores cromosomas. De hecho, garantiza una distribución de los nuevos cromosomas proporcional a su aptitud, sin intervención del azar.

Volviendo al ejemplo anterior, con $K=3$ no se seleccionaría ningún cromosoma directamente y los tres habría que elegirlos por sorteo o ruleta. Para hacer el ejemplo más didáctico, supongamos que hay que seleccionar $K=7$ cromosomas. Entonces, en la tabla 6 podemos ver que saldrían directamente seleccionados un cromosoma C1, un C4 y dos C5, que suman cuatro cromosomas. Los otros tres, para completar $K=7$, se seleccionarán usando cualquiera de los otros métodos.

CROMOSOMA	Evaluación e_i	Aptitud normalizada (%) u_i	Probabilidad de ser seleccionado p_i	$K*p_i$	$\text{floor}(K*p_i)$
C1	40.5	15	0.15	1.05	1
C2	13.5	5	0.05	0.35	0
C3	27	10	0.1	0.7	0
C4	40.5	15	0.15	1.05	1
C5	81	30	0.3	2.1	2
C6	27	10	0.1	0.7	0
C7	13.5	5	0.05	0.35	0
C8	27	10	0.1	0.7	0

Tabla 6: Ejemplo de muestreo universal por restos con $K=7$.

Torneo. Es el sistema más rápido. Se toman al azar 2 o 3 cromosomas de la

población, y se selecciona el que tenga mayor función de aptitud. Se repite este procedimiento K veces.

A pesar de que este método tiene malas propiedades matemáticas (por ejemplo, no garantiza que los mejores sean seleccionados nunca), en la práctica ello no suele ser un problema. Como el número de generaciones suele ser muy alto, la probabilidad de que nunca seleccione a los mejores es muy baja.

Otra ventaja es que no se requiere normalizar la aptitud.

Y una tercera ventaja es que ni siquiera se requiere calcular la aptitud de toda la población. Basta con ir calculando la de los cromosomas que salen elegidos para hacer el torneo.

Todo ello hace que sea el sistema de selección más simple y efectivo para implementarse en *software*.

Reproducción

Ahora hay que reproducir los cromosomas seleccionados que se encuentran en el *mating pool*, para generar los cromosomas hijos. Existen muchos operadores de reproducción y podemos inventar muchos más. Nos detendremos en los más básicos y comunes.

Mutación. Se elige al azar un cromosoma del *mating pool*, se elige al azar uno de sus genes y se cambia su alelo por otro generado al azar (figura 113).

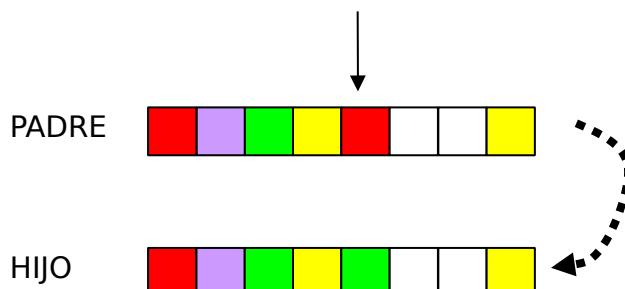


Figura 113: Mutación.

Existen otros operadores de mutación y podemos inventar más, pero lo importante es que cumplan con una propiedad: al aplicar este operador infinitas veces sobre cualquier cromosoma deben generarse todos los cromosomas posibles. Con ello nos aseguramos de que, en el caso peor, esté realizando una búsqueda exhaustiva probabilista. Y, desde luego, que genere variedad en la población, recuperando alelos que podrían haberse perdido con los procesos de

selección y de cruce. Habitualmente la mutación produce un cambio pequeño y ello significa que explota las soluciones encontradas hasta el momento en el espacio de búsqueda, es decir, hace una búsqueda en profundidad.

Cruce uniforme. También se llama recombinación uniforme. Se necesitan dos padres, que generarán un hijo. Cada gen del hijo se selecciona al azar entre el correspondiente gen de los padres (figura 114).

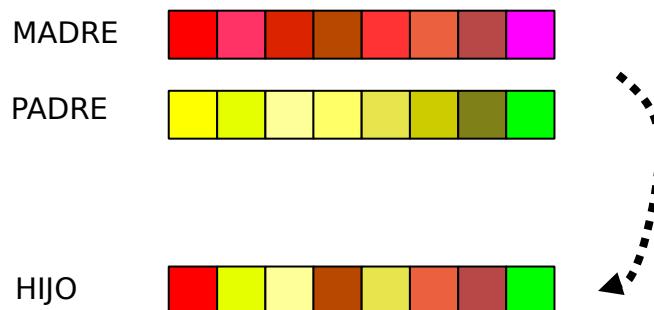


Figura 114: Cruce uniforme.

Nada impide obtener de una vez otro hijo, con los genes complementarios, no usados, de los padres.

La misión del cruce es menos crítica que la mutación, pero el objetivo es realizar cambios grandes para explorar el espacio de búsqueda, usando a los padres (que si han sido seleccionados es probable que sean buenos) como base para buscar combinaciones mejores, es decir, hace una búsqueda en anchura.

En la figura 115 vemos algunos ejemplos de generación de hijos (puntos azules): el cromosoma *A* lo hizo por mutación, al igual que el cromosoma *B*. Sus hijos son muy parecidos a los respectivos padres, pues los cambios son pequeños, por lo que están haciendo búsqueda local, en profundidad (explotación de soluciones conocidas). A su vez, *B* y *C* se cruzaron generando un hijo que se parece algo a *B* y algo a *C*, porque está entremedias de los dos. Los cambios son mayores y hace búsqueda en anchura (exploración de soluciones desconocidas).

La mutación de *A* produce un hijo con mejor aptitud, lo cual implica que tiene una gran probabilidad de ser seleccionado para futuras generaciones. La mutación de *B* empeoró su aptitud y, por tanto, es menos probable que salga seleccionado. Este es el sustento de los algoritmos genéticos. Los operadores de reproducción generan variedad mientras que la selección presiona en la búsqueda de los mejores, de los más aptos para solucionar mi problema.

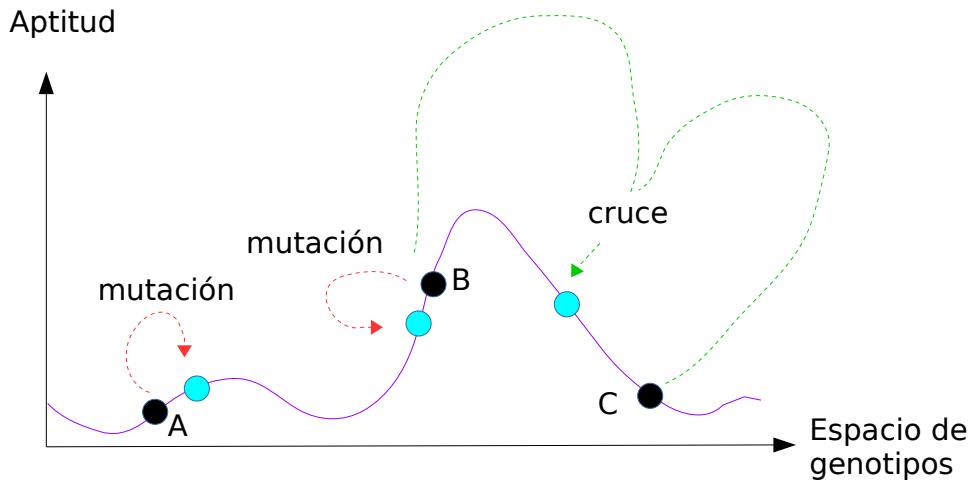


Figura 115: Ejemplos de mutación y cruce.

Los operadores de reproducción se pueden aplicar en paralelo sobre el *mating pool* de forma porcentual (un porcentaje del *mating pool* se muta y otro porcentaje se cruza) o probabilista (para cada cromosoma del *mating pool* hay una cierta probabilidad de que participe en una mutación o en un cruce). Pero también se pueden aplicar secuencialmente, o sea, primero se cruzan dos cromosomas y el resultado se muta. En general, los detalles de implementación son irrelevantes, salvo uno (el mencionado en la mutación): siempre debe haber algún operador que produzca pequeños cambios, de modo que se generen todos los cromosomas posibles si se deja pasar suficiente tiempo.

En algunos libros los porcentajes o probabilidades se expresan relativos al número de cromosomas o de genes de la población o del *mating pool*. Todo ello es irrelevante, aunque suele confundir a quien se inicia programando en estos algoritmos. Lo que sí suele ser sano es correr el algoritmo genético dos veces (o más): una con mucha mutación y poco cruce, y otra con poca mutación y mucho cruce. Obviamente, al final se elige la mejor solución de entre todas las generadas.

Hay trabajos donde se ha tratado de caracterizar los porcentajes de mutación y cruce más adecuados, pero ello en general es muy dependiente del problema, por lo que más vale hacer las dos pruebas mencionadas. Hay otros trabajos donde se incorporan los porcentajes de mutación y de cruce como genes adicionales en los cromosomas, tratando de que la misma evolución encuentre los valores más adecuados. Esta es una buena idea, aunque complica un tanto el algoritmo.

Reemplazo

Finalmente hay que inyectar los hijos en la población. Aunque es una operación sencilla, hay varias formas de hacerlo.

En los inicios del uso de estas técnicas se trataba de mantener la población constante, de modo que si se inyectaban digamos 15 hijos, había que eliminar 15 cromosomas en la población original. Imagino que se hacía así porque las matrices de los lenguajes de programación de la época eran estáticas, de tamaño fijo. Hoy día se puede simplemente añadir los hijos a la población.

De todos modos, si se desea eliminar cromosomas de la población (para evitar consumir mucha memoria, que conlleva a la vez a mayores tiempos de cómputo) se pueden seguir estas estrategias:

- **Eliminar cromosomas al azar.** Es lo más rápido y recomendable.
- **Eliminar a los peores.** Es razonable, pero se gasta tiempo ordenando los cromosomas por aptitud.
- **Eliminar a los peores con mayor probabilidad.** Es muy razonable. También se llama proceso de selección inversa. Esto es lo que ocurre en biología. Hay un proceso de selección para reproducción y hay otro (selección inversa) para sobrevivir. Si se emplea selección inversa por torneo, es un proceso rápido.
- **Asignar a cada cromosoma una edad.** Debe ser igual a cero en el momento de ser creado, y debe incrementarse en uno con cada generación que pase. Después hay que eliminar los cromosomas según vayan superando un cierto valor, por ejemplo 100.

En cualquiera de los casos anteriores, conviene no eliminar al mejor de los cromosomas. A esto se le llama elitismo, y es conveniente usarlo porque en cualquier caso el mejor no conviene perderlo. Tampoco es bueno que haya mucho elitismo (mantener más de uno o dos de los mejores cromosomas) porque eso perjudica la exploración de nuevas zonas en el espacio de búsqueda.

Hay también otra forma de simplificar el algoritmo genético que consiste en que el *mating pool* tiene un tamaño $K=1$. En cada generación solo se produce un hijo y se elimina un parente. A esta variante se la llama de “estado estacionario”, porque casi no cambia la población de una generación a la siguiente. Es la más rápida y cómoda de implementar en software porque nos quitamos de encima la

duda de cuál es el tamaño óptimo del *mating pool*.

Al respecto, téngase en cuenta que, *grossó modo*, da igual tener una población de 200 cromosomas que se ejecuten por 3000 generaciones a usar 300 cromosomas por 2000 generaciones. En ambos casos, se efectuarán aproximadamente 600 000 evaluaciones de aptitud, o sea, se van a explorar 600 000 puntos del espacio de búsqueda. De modo que con la variante de estado estacionario debemos aumentar el número de generaciones para compensar que en cada generación solo se evalúan dos o tres de cromosomas (los que producen el hijo que va a llegar al *mating pool*). Por ejemplo, 150 000 generaciones y una población de cualquier tamaño de donde se seleccionen al azar 4 cromosomas que compitan entre sí por torneo para generar dos cromosomas padres en el *mating pool*. El resultado es aproximadamente el mismo y se evitan ineficiencias (por ejemplo, con el algoritmo genético básico podrían llegar al *mating pool* bastantes cromosomas que luego no produzcan ningún hijo, de modo que se perdió tiempo evaluándolos y seleccionándolos). Este será el algoritmo genético que implementaremos al final de este capítulo.

Ejemplo 1: salas de cine

Heredé de un tío una cadena de 4 salas de cine, pero no entiendo nada del negocio. Mi objetivo es maximizar los ingresos. Y tengo varias preguntas cuya respuesta desconozco:

- ¿El precio de la entrada debería ser 2000, 3000, 5000, 8000 o 12 000?, ya que eso es lo que he visto que cobran otros cines.
- ¿Debería haber películas comerciales, subtituladas o en blanco y negro?
- ¿El horario debería ser en la mañana, en la tarde o en la noche?

En este caso, cada sala es un cromosoma, y cada una de mis preguntas es un gen. Con ello me aseguro que puedo implementar salas con cualquiera de las características enunciadas. Mi población es muy reducida, pues solo tiene 4 cromosomas. Esto no es lo habitual, pero no tengo elección pues es una limitación física del problema. Como hemos dicho, cada cromosoma tiene 3 genes con los alelos que se muestran en la tabla 7.

GEN	ALELOS
Gen 0: Precio	2000, 3000, 5000, 8000, 12 000
Gen 1: Calidad	Comercial, Subtitulada, Blanco y Negro

Tabla 7: Genes y alelos del ejemplo de las salas de cine.

Una vez definido el cromosoma, debo generar al azar la población inicial, programar de esa manera las salas y esperar un día a ver cuánto recaudo en cada una. Imaginemos, por ejemplo, que es como dice la tabla 8.

Sala	CROMOSOMA	GENES			Dinero recaudado	Aptitud normalizada
		Precio	Calidad	Horario		
S1	C1	3000	Comercial	Mañana	70 000	26.3%
S2	C2	3000	Comercial	Tarde	150 000	56.4%
S3	C3	5000	Subtitulada	Noche	40 000	15.0%
S4	C4	2000	Blanco y negro	Noche	6000	2.3%

Tabla 8: Población inicial del ejemplo de las salas de cine y aptitudes resultantes.

Vistos los resultados, he obtenido las mayores ganancias en la sala 2. De este modo, podría limitarme a explotar esa combinación, programando todas las salas con entradas a 3000, películas comerciales y horario en la tarde. Aparentemente, explotando esta solución conocida obtendré ganancias muy altas, pero ¿no será que existen otras combinaciones que me pueden dar más dinero? Es decir, ¿me conviene explorar más? Para quien sepa de negocios, bajar la entrada a 2000 y poner horario nocturno podría darme todavía más ganancias, pero yo no sé nada de eso. Por lo tanto, me toca poner en marcha un algoritmo genético. El proceso que viene ahora es hacer una selección de los mejores con mayor probabilidad.

Supongamos entonces que hacemos selección por torneo de dos padres, con el algoritmo de estado estacionario, para generar un único hijo resultante de cruzar los dos padres y mutar el resultado, inyectando este hijo a la población y eliminando el cromosoma peor.

Entonces elegimos al azar dos cromosomas (supongamos que salen C2 y C3) y por torneo nos quedamos con el mejor (C2). Ahora elegimos al azar otros dos padres (por ejemplo, salieron C4 y C4) con lo cual el mejor es C4. Ya tenemos los dos padres (C2 y C4). Para generar el hijo (llámémoslo C5) por cruce uniforme seleccionamos al azar los respectivos genes de la madre y del padre, por ejemplo {padre, padre, madre}, con lo cual el hijo sale con genes {3000, Comercial, Noche}. Ahora vamos a mutarlo para lo cual elegimos al azar un gen (supongamos que salió el de "precio") y elegimos al azar uno de los alelos (supongamos que salió 8000). Por último, inyectamos este cromosoma C5 a la población, eliminando el peor que hay hasta el momento, que es C4. Con ello nos

queda la población indicada en la tabla 9, después de transcurrir la primera generación. Además, dejamos pasar un día para ver cuánto recauda cada sala y calculamos las nuevas aptitudes.

Se observa que en este problema hay inevitablemente ruido en la evaluación: no tenemos garantías de recaudar lo mismo si programamos la misma película en idénticas condiciones. Lo que sí podemos observar es una serie de preferencias del público, y eso es lo importante para el cálculo de la aptitud.

	CROMOSOMA	GENES				
Sala		Precio	Calidad	Horario	Dinero recaudado	Aptitud normalizada
S1	C1	3000	Comercial	Mañana	60 000	14.3%
S2	C2	3000	Comercial	Tarde	180 000	42.9%
S3	C3	5000	Subtitulada	Noche	20 000	4.7%
S4	C5	8000	Comercial	Noche	160 000	38.1%

Tabla 9: Generación 1 del ejemplo de las salas de cine y nuevo cálculo de la aptitud.

Y así repetiríamos generación tras generación, lo cual nos lleva a una pregunta: ¿cuándo detenerse? La respuesta es muy obvia: cuando tengamos una solución aceptable, cuando veamos que no aparecen mejores soluciones (es decir, el algoritmo ha convergido al óptimo global, o se ha quedado atascado en un óptimo local) o cuando ya no nos quede más tiempo disponible.

Ejemplo 2: invertir una matriz

Supongamos que no sabemos nada de álgebra y queremos invertir una matriz A , es decir, calcular una matriz A^{-1} tal que $A * A^{-1} = I$ siendo I la matriz identidad (todas los elementos de la diagonal principal igual a 1, y 0 en el resto). Por ejemplo:

$$A = \begin{pmatrix} 4 & -1 & 2 \\ 3 & 5 & -1 \\ 2 & -2 & 1 \end{pmatrix} \quad Ec. 32$$

Queremos calcular:

$$A^{-1} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \quad Ec. 33$$

Entonces el cromosoma será un vector de 9 genes con todos sus alelos números flotantes (figura 116).

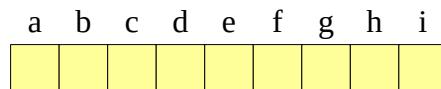


Figura 116: Cromosoma para invertir una matriz de 3x3.

Y la función de aptitud a minimizar será:

$$\text{error} = |A \times A^{-1} - I| \quad \text{Ec. 34}$$

Se puede, entonces, tomar como aptitud el error cambiado de signo. De este modo, al maximizar la aptitud se minimizará el error.

$$\text{aptitud} = -\text{error} = -|A \times A^{-1} - I| \quad \text{Ec. 35}$$

El resto es ya conocido.

Problemas que pueden aparecer

Los algoritmos genéticos son fantásticos, muy sencillos, potentes y generales. Pero no están exentos de problemas, que veremos a continuación junto a las posibles soluciones.

Como hemos visto en los ejemplos, solo hay dos cosas difíciles cuando se quiere aplicar un algoritmo genético a la solución de un problema de optimización:

- Diseñar el cromosoma.
- Definir cómo se va a calcular la aptitud.

El resto es bastante automático y general. Solo estas dos partes del algoritmo dependen del problema concreto que se quiera solucionar y a ellas debe gastarse suficiente tiempo pensando en cuáles van a ser las mejores alternativas.

Diseñar el cromosoma. Recordemos que es importante lograr la completitud y la cerradura en el diseño del cromosoma (figura 107). Sin completitud quizás no alcancemos buenas soluciones porque no existe la forma de codificarlas en los cromosomas. Sin cerradura quizás el algoritmo genético gaste mucho tiempo generando soluciones inválidas (también llamadas puntos no factibles) cuya

aptitud es cero por lo que, si son muchas, no producirán ninguna presión evolutiva en busca de mejores soluciones.

La completitud suele ser más fácil de lograr (y de detectar si no se cumple), y se soluciona añadiendo más genes que codifiquen adecuadamente todo el espacio de soluciones posibles. Por contra, la cerradura suele ser difícil de lograr porque habitualmente lo que significa es que hay restricciones duras en el enunciado del problema. En caso de no lograrse, se suelen aplicar técnicas de penalización, que veremos enseguida.

Sin embargo, incluso aunque cumplamos con la completitud y la cerradura, puede haber codificaciones de cromosomas francamente malas. El algoritmo genético funcionará, pero requerirá mucho más tiempo para encontrar soluciones. Veamos un ejemplo usando el conocido problema computacional de las N-Damas: el objetivo es situar N damas de ajedrez en un tablero de $N \times N$ casillas de modo que ninguna ataque a otra. Se sabe que es un problema NP y que existen soluciones para todo $N \geq 4$. En la figura 117 podemos ver algunas soluciones para $N=4$, para $N=5$ y para $N=6$.

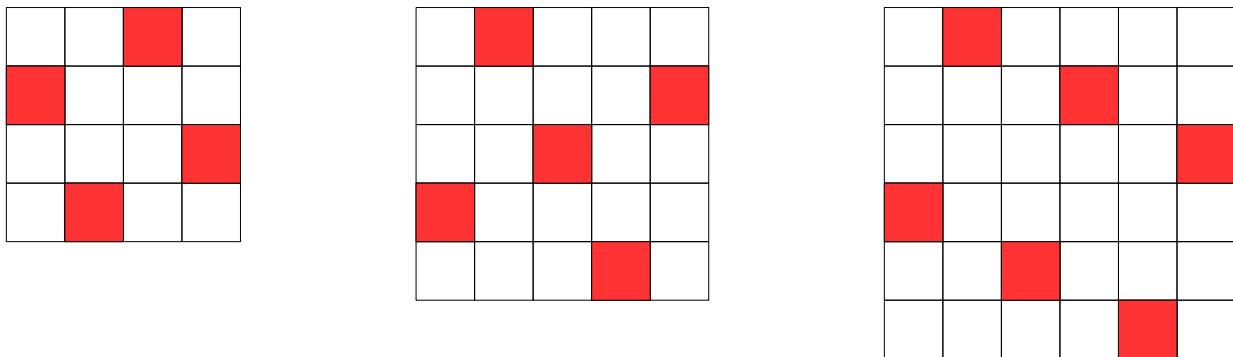


Figura 117: Solución a las N-Damas para $N=\{4, 5, 6\}$.

El cromosoma que rápidamente se nos ocurre lo podemos ver en la figura 118 para el problema de las 4-damas: consiste en estirar el tablero de 4x4 para formar un vector de 16 casillas, donde todos los alelos son binarios e indican si existe o no existe una dama en esa casilla.



Figura 118: Cromosoma ingenuo, para las 4-damas.

Dado que hay 16 genes binarios, el espacio de búsqueda es de 2^{16} posibilidades. No parece muy grande, pero para el caso general de N-Damas es de $2^{N \times N}$, lo cual es prohibitivo. Obviamente el problema es NP, de modo que sabemos que el

espacio de soluciones crecerá exponencialmente, pero ¿será que el exponente de la exponencial se puede reducir? La respuesta es afirmativa si pensamos en la codificación de la figura 119, donde ahora se necesita un cromosoma de N genes únicamente. El gen va a indicar la columna donde se encuentra la dama, y el alelo del gen va a indicar la fila. O sea, hay una dama en la columna 0, fila 1; otra en la columna 1, fila 3; otra en la columna 2, fila 0; y otra en la columna 3 fila 2.

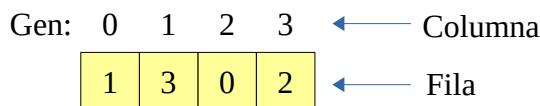


Figura 119: Un cromosoma mejor pensado, para el problema de las 4-damas.

Cada gen tiene N alelos, y hay N genes, por lo que el espacio de búsqueda se reduce a N^N (lo que es igual a $2^{N \times \log_2(N)}$ que es menor que $2^{N \times N}$ de la primera codificación).

Además, en la primera codificación, los operadores de mutación y cruce pueden añadir o quitar damas, de modo que no sean N, con lo cual estaremos explorando soluciones no factibles y es más probable que el algoritmo genético se atasque. Mientras que en la segunda codificación todas las soluciones son factibles y, como añadido, es imposible que tengan cruces en las columnas. Es decir, en cada columna solo hay una dama por lo que no hace falta verificar ataques en la misma columna. Lo único que resta por verificar son los ataques en las filas y en las diagonales.

Todavía se puede hacer una mejora, en este caso, al operador de mutación. Si nos damos cuenta, con este nuevo cromosoma el problema ya no es paramétrico sino que se ha convertido en combinatorio, pues la solución consiste en ofrecer todas las filas en los genes, pero en un orden específico. En el ejemplo de la figura 119 las filas aparecen en el orden 1,3,0,2. Pero otras soluciones posibles podrían ser 2,0,3,1, y otras más. Tienen que aparecer todos los alelos pues es obligatorio que haya una dama en cada fila. Y manejando adecuadamente el orden de esos alelos lograremos evitar ataques en las diagonales. El operador de mutación típico para problemas combinatorios como este lo veremos más adelante, en la figura 126, donde se eligen dos genes al azar y se intercambian sus alelos.

El cálculo de la aptitud se limitará entonces a contar el número de ataques en las diagonales porque, por diseño, este cromosoma no tiene ataques en las filas ni en las columnas. El problema se vuelve así más rápido de explorar para un algoritmo genético.

Diseñar la función de aptitud. Habitualmente es trivial hacerlo debido a los pocos requerimientos que tiene. La aptitud no tiene por qué ser una función matemática, ni continua, ni derivable. Puede obtenerse como resultado de la evaluación de un modelo, de la votación de personas, de un experimento físico, o cosas similares. Lo único que importa es que arroje un número que sea mayor cuanto mejor sea la solución.

Sin embargo, también puede haber problemas. El más importante ocurre para situaciones de tipo “aguja en un pajar” donde hay algunas soluciones aisladas, rodeadas de un vasto espacio sin ninguna solución. Un ejemplo es el problema computacional de satisfabilidad. En la figura 120 vemos un ejemplo para la función lógica

$$F(a,b,c,d) = \bar{a} \cdot (a + \bar{b}) \cdot (a + b + d)$$

Ec. 36

Esta función tiene 4 variables y, por tanto, su espacio es de $2^4 = 16$ posibilidades, muy pequeño para poderlo visualizar bien. Pero conforme aumenta el número de variables, el espacio de búsqueda lo hace exponencialmente. Los problemas de satisfabilidad son NP-completos. El cromosoma se implementaría con 4 genes binarios (uno para cada variable a,b,c,d) y la aptitud se calcularía aplicando la ecuación 36.

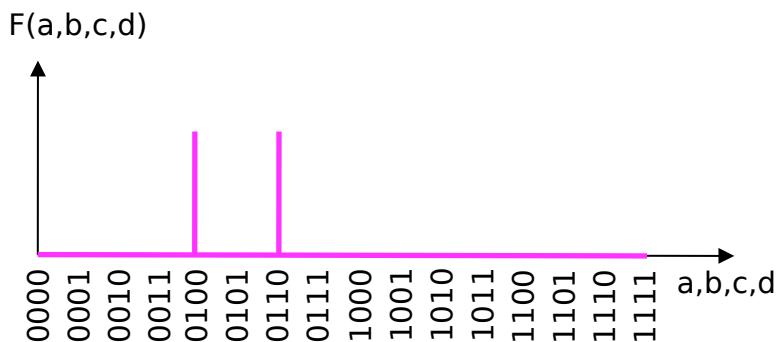


Figura 120: Problema de tipo “aguja en un pajar”.

En este ejemplo hay dos soluciones, en 0100 y 0110 . Los problemas de tipo “aguja en un pajar” se caracterizan porque no importa cuán cerca esté a una solución. La aptitud saldrá de todos modos 0%, excepto en el momento de llegar a esa solución donde saltará a 100%. De modo que no hay ninguna presión selectiva que ayude a orientar la búsqueda en una u otra dirección.

Conviene advertir que los problemas de tipo “aguja en un pajar” son los más difíciles para cualquier tipo de algoritmo de búsqueda y optimización, no solo para los algoritmos genéticos.

La única forma de abordarlos es tratando de buscar algún truco que me ayude a suavizar la función de aptitud. En este caso, una idea podría ser contar el número de términos que se activan en la ecuación 36. Son 3 términos multiplicativos, de modo que la función valdrá 1 cuando los 3 términos valgan 1. Esto es simplemente una heurística que funcionará en algunos casos, pero no en general, pues el problema, como dijimos, es NP.

Los problemas de tipo “aguja en un pajar” son muy frecuentes en criptografía. Digamos que se me ha olvidado la clave de entrada a mi computador. Pruebo con una, y el computador me responde “frío, frío”. Pruebo con otra y el computador me dice “vas mejorando, ya has acertado las tres primeras letras”. Pruebo con otra y me dice “caliente, caliente, ya solo tienes una letra equivocada”. Sería bueno que el computador respondiera así (para los *crackers*), pero la realidad es muy distinta. Al teclear una contraseña el computador solo da dos respuestas: “te equivocaste” o “acertaste”. De modo que no da ninguna información adicional para orientar la búsqueda.

Y entonces, todo depende del ingenio humano para lograr suavizar la función de aptitud. Con los primeros computadores, no se usaba criptografía realmente sino que se comparaba la clave almacenada con la que introducía el usuario, letra a letra. En cuanto una letra fallaba, de inmediato comenzaba a imprimir el mensaje de acceso denegado. Midiendo el tiempo de demora entre la introducción del *ENTER* de la contraseña y la salida del mensaje, se podía deducir cuántas letras del principio de la contraseña eran correctas. De modo que un problema NP se volvía $O(N)$. Solo había que probar con la primera letra, hasta que el tiempo de respuesta aumentaba bruscamente, lo cual significaba que habíamos acertado con ella. Luego, dejando esa primera letra fija, había que probar con la segunda letra, y así sucesivamente.

Esto fue corregido rápidamente, y la industria del *software* produjo un programa de verificación de contraseñas que antes de imprimir el mensaje de acceso denegado se quedase en un bucle sin hacer nada, solo para esperar un tiempo equivalente al del acierto de la contraseña completa. Con ello se lograba que el tiempo de respuesta fuera siempre el mismo. Este sistema también fue violado si se tenía acceso físico a la *CPU* que ejecutaba el programa, midiendo su corriente de consumo: en los computadores, cuando se hacen cálculos reales el consumo de corriente es alto, mientras que cuando simplemente se está esperando sin hacer nada el consumo es menor. Midiendo el consumo de corriente se podía averiguar de nuevo cuántas letras desde el principio de la contraseña eran correctas. Todo ello al final desembocó en el empleo de criptografía *DES* y *RSA*, y así este hueco de seguridad también fue eliminado.

Con esto no quiero animar a nadie a *hackear*. Lo que quiero contar es que, dado cualquier problema teóricamente irresoluble, puede haber una forma ingeniosa de plantearlo donde la solución sea trivial. Y esa forma ingeniosa no está especificada de ninguna manera en el planteamiento del problema. Requiere inteligencia genuina.

La aptitud puede tener también otras dificultades, como que se formen superindividuos, definidos como unos pocos cromosomas muy malos pero mucho mejores que el resto. En la figura 121 y tabla 10, la aptitud del superindividuo B es apenas 4, cuando el óptimo global está en 100. Pero 4 es mucho mayor que las aptitudes de los demás cromosomas.

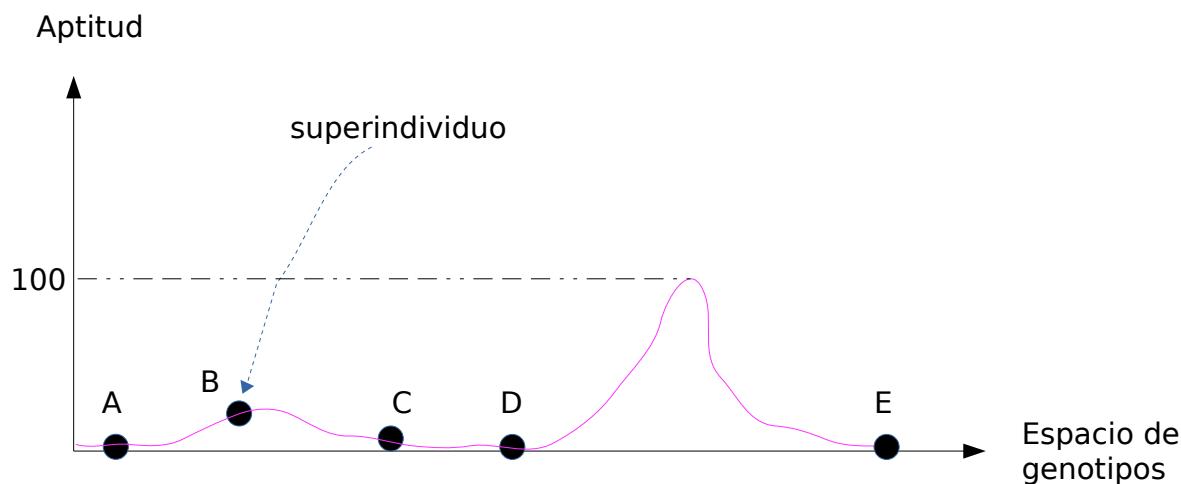


Figura 121: Ejemplo de superindividuo.

CROMOSOMA	APTITUD
A	0.10
B	4.00
C	0.20
D	0.08
E	0.05

Tabla 10: Aptitudes cuando hay un superindividuo.

Si utilizamos selección por sorteo, por ruleta (figura 122) o muestreo universal por restos, esos superindividuos en unas pocas generaciones van a extinguir a los demás cromosomas, haciendo que desaparezca la variedad en la población y provocando una convergencia prematura a una mala solución.

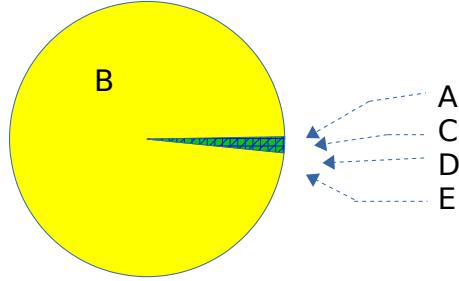


Figura 122: Ruleta con un superindividuo.

La selección por torneo no padece este problema, al menos no es tan acusado. Otra alternativa es hacer selección por *ranking*, donde primero se ordenan los cromosomas según su aptitud (primero el de peor aptitud) y el número de este *ranking* se toma como el valor sobre el que realizar la selección, es decir, el número de orden es el área en la ruleta (tabla 11 y figura 123).

CROMOSOMA	APTITUD	RANKING
A	0.10	3
B	4.00	5
C	0.20	4
D	0.08	2
E	0.05	1

Tabla 11: Ranking.

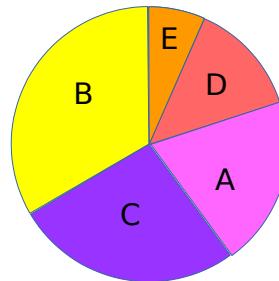


Figura 123: Ruleta usando ranking.

De esta forma los superindividuos no aplastan a los demás, y se preserva la variedad en la población.

Paralelización

Otra característica positiva de los algoritmos genéticos es que son fáciles y

efectivos de paralelizar, usando hilos, núcleos o computadores. Hay varias formas de hacerlo:

- Paralelizar la evaluación y la reproducción, manteniendo la población centralizada. Se puede hacer, pero no es muy útil pues se pierde demasiado tiempo en las comunicaciones.
- De grano fino. Se emplea *hardware* específico, típicamente usando *FPGA*, donde hay muchas *CPU* en forma de matriz 2D, y cada *CPU* está conectada únicamente a sus vecinas de arriba, abajo, a la derecha y a la izquierda. Cada *CPU* mantiene una población muy pequeña de cromosomas, quizás apenas 2, que cruza y muta e intercambia los hijos con las *CPU* vecinas.
- Modelo de islas con migración. Se implementa en una red de computadores, todos ellos corriendo el mismo algoritmo genético pero con distintas poblaciones de cromosomas. De vez en cuando los mejores cromosomas se envían a los otros computadores. En este caso el paralelismo es muy eficiente porque casi no se pierde tiempo en comunicaciones entre computadores. La única precaución a tener en cuenta es que las poblaciones deben ser distintas en cada computador, y eso no es obvio de lograr porque hoy día los computadores mantienen sus relojes sincronizados gracias a servicios de red, y resulta que el inicializador del generador de números aleatorios (el famoso *srand(time(0))* en *C++* o similar en otros lenguajes) usa la fecha y hora actuales para generar la semilla inicial, de modo que todos los computadores generarán la misma semilla y, de allí, las mismas secuencias, los mismos alelos y las mismas mutaciones. Para evitarlo, es recomendable ejecutar el *srand(time(0))* en un único computador, y desde allí generar unos cuantos números aleatorios que sirvan como semillas para los demás computadores. En EVALAB hemos realizado una implementación del modelo de islas (Pineda y Estacio, 2003) usando todos los computadores del laboratorio.
- Arquitectura de inyección. También se implementa en una red de computadores, pero las comunicaciones tienen forma de árbol binario. Se usa cuando la función de aptitud es muy compleja conteniendo varios objetivos. En todos los computadores se ejecuta el mismo algoritmo, pero la función de aptitud se va refinando: en los computadores-hoja es una función sencilla, y conforme se va avanzando por las ramas se van añadiendo más requerimientos a la función de aptitud, hasta llegar al computador-raíz, donde la función de aptitud es la completa. Inicialmente los computadores-hoja comienzan a ejecutar el algoritmo genético y cuando van encontrando

soluciones a su función de aptitud, van inyectándolas al siguiente nivel. Al final, en el computador-raíz habrá una población de cromosomas que han ido superando todos los niveles, por lo que será más fácil encontrar allí la solución al problema total. Un ejemplo de aplicación puede ser en el diseño de circuitos impresos, que tienen varios requerimientos: en el primer nivel únicamente se busca que las conexiones entre componentes sean correctas y que no haya cortocircuitos; en el segundo nivel se añade el grosor de las pistas; en el tercer nivel se añaden requerimientos de minimización de interferencias electromagnéticas; y en el cuarto y último nivel se añaden requerimientos de disipación de calor. En cada nivel se busca cumplir un nuevo objetivo manteniendo también los objetivos anteriores.

Aplicaciones de los algoritmos genéticos a problemas combinatorios

En los problemas combinatorios tenemos un conjunto de objetos y la solución a mi problema consiste en entregarlos en un orden concreto. El más conocido es el del vendedor viajero. Un vendedor tiene que pasar por un conjunto de ciudades para vender sus productos. ¿En qué orden debe visitar las ciudades para minimizar la distancia recorrida? (ver un ejemplo con 9 ciudades en la figura 124). Sin embargo, hay muchos otros problemas combinatorios. Como acabamos de ver, las N-Damas se puede convertir a combinatorio.

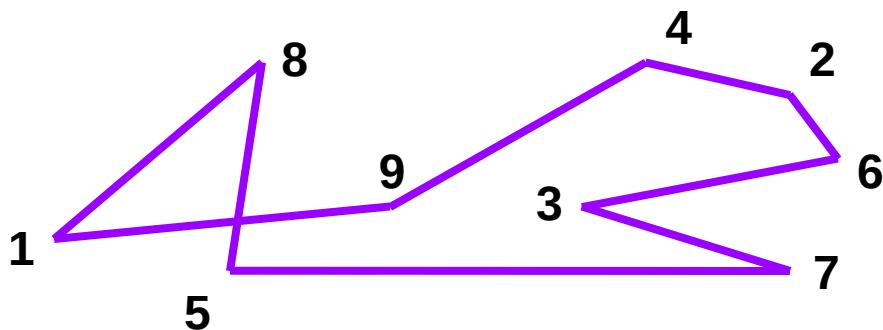


Figura 124: Problema del vendedor viajero.

Más formalmente, el problema del vendedor viajero se enuncia así: dadas N ciudades y los costes de viajar de una a otra $\{c_{ij}\}$ con $i,j \in \{1,...N\}$, se trata de calcular una trayectoria completa (que visite todas las ciudades), cerrada (que empiece y termine en la misma), conexa (conectada continuamente) y cuya

distancia recorrida total sea menor o igual a una constante D . Esta es la versión NP-completa del problema. Si se busca la trayectoria más corta posible, el problema es NP-hard. El espacio de búsqueda es el de las permutaciones de las N ciudades.

No existe una forma práctica de codificar una solución a este problema en un cromosoma, para ser usada con los operadores de cruce y mutación habituales, pues estas operaciones dejan de ser cerradas. Típicamente el cromosoma será como el de la figura 125, donde los genes ya no tienen ningún significado posicional, sino solo de orden. En los alelos están las ciudades a visitar.

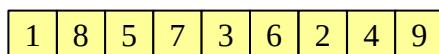


Figura 125: Cromosoma que implementa una posible solución al problema del vendedor viajero.

Y la representación no es única pues también valdrían 8,5,7,3,6,2,4,9,1 o 4,2,6,3,7,5,8,1,9, y muchas otras.

El operador de mutación debe cambiarse por el indicado en la figura 126 para evitar que desaparezcan ciudades.

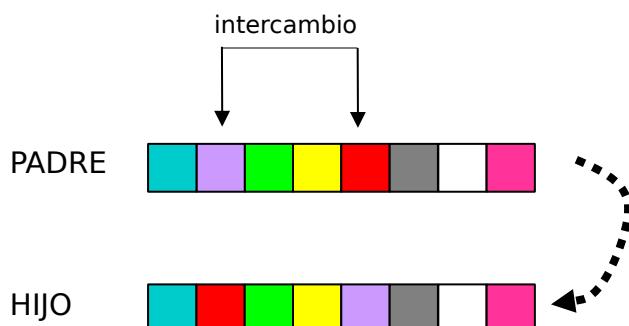


Figura 126: Mutación por intercambio.

Observemos que esta mutación produce el cambio más pequeño posible en un cromosoma para problemas combinatorios. Y que si iteramos infinitas veces se producirán todos los cromosomas posibles, de modo que cumplimos con la condición esencial de la mutación.

Desgraciadamente el cruce no es tan sencillo. El cruce uniforme también producirá puntos no factibles, donde se visitarán varias veces la misma ciudad y habrá ciudades que no se visiten nunca. Hay varias propuestas de nuevos operadores de cruce donde se eviten estos problemas (cruce por orden, cruce por emparejamiento parcial) pero todas son insatisfactorias pues los hijos dejan de

parecerse a los padres. La recomendación es no usar cruce en estos casos pues en últimas, no es tan importante. En su lugar, si se desea, se puede utilizar un operador de mutación que haga cambios más grandes.

Aplicaciones de los algoritmos genéticos a problemas multiobjetivo

Lo que se va a contar en este capítulo se aplica a cualquier algoritmo de optimización y no únicamente a los algoritmos genéticos. Lo que ocurre es que en los algoritmos genéticos es muy fácil resolver estos problemas.

Es raro tener un problema con un único objetivo. Lo habitual es que haya varios objetivos que pueden ser incluso contradictorios. Por ejemplo, quiero diseñar una bicicleta eléctrica que pese poco, pero que el motor sea muy potente, que la batería ofrezca mucha autonomía y que el precio sea asequible.

Cuando se desean optimizar varios objetivos simultáneamente, es intuitivo definir una función de aptitud como combinación lineal de los objetivos y luego optimizar esa función:

$$f(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \alpha_3 f_3(x) + \dots$$

Ec. 37

Sin embargo, esta forma de trabajar es ingenua e incorrecta, pues la mayoría de las veces el óptimo de la función linealizada proporciona soluciones mediocres, que no son buenas bajo ninguno de los objetivos individuales. Enseguida veremos por qué.

Otras posibilidades:

- Utilizar un solo objetivo (el más importante). Una vez encontrado el óptimo, se añade a la función de aptitud el siguiente objetivo. Y así sucesivamente. Esto es lo que se hace en la arquitectura de inyección ya comentada.
- Utilizar especiación (diversas especies de cromosomas). Cada una de ellas busca un objetivo. La selección se efectúa dentro de una especie, mientras que el cruce se efectúa entre especies.
- Usar óptimos de Pareto. Esta es la mejor y se explicará a continuación en detalle.

Primero daremos unas sencillas definiciones:

Dominancia de Pareto. Dado un conjunto de N funciones $f_k(x)$ con $x, x_1, x_2 \in X$, y $k = \{1, 2, 3, \dots, N\}$ se dice que x_1 domina a x_2 si y solo si:

$$f_k(x_1) \geq f_k(x_2) \quad \forall k \quad Ec. 38$$

Óptimos de Pareto. Dados $x_0, x \in X$ se dice que x_0 es óptimo de Pareto cuando no existe ninguna solución mejor en ningún objetivo. Es decir:

$$f_k(x_0) \geq f_k(x) \quad \forall k, \forall x \in X, x_0 \neq x \quad Ec. 39$$

Corolario. Si un punto domina a todos los demás puntos, se dice que es óptimo de Pareto.

Veamos unos ejemplos con solo dos objetivos (f_1, f_2) para que sean fáciles de visualizar. En la figura 127, el punto x_1 domina a x_2 . Y no hay ninguna relación de dominancia entre x_1 y x_3 ni entre x_2 y x_3 .

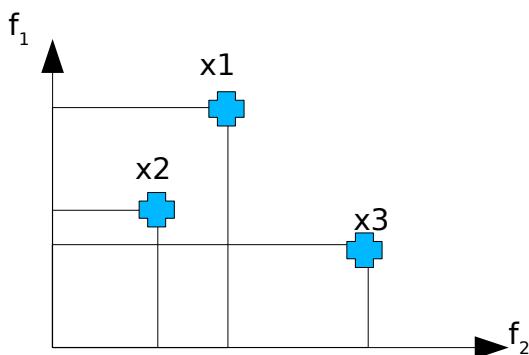


Figura 127: Ejemplo de dominancia de Pareto: x_1 domina a x_2 .

En la figura 128, x_1 domina a x_2 y a x_3 , por lo que x_1 es óptimo de Pareto. No hay ninguna relación de dominancia entre x_2 y x_3 .

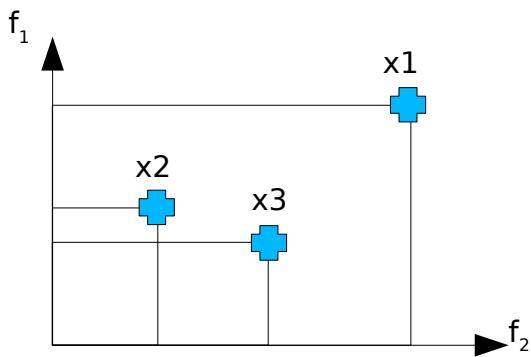


Figura 128: Ejemplo de dominancia: x_1 es óptimo de Pareto.

Y en la figura 129 puede verse un ejemplo donde x_1 domina a x_2 que a su vez domina a x_3 . Obviamente, x_1 también domina a x_3 , por lo que x_1 es óptimo de Pareto.

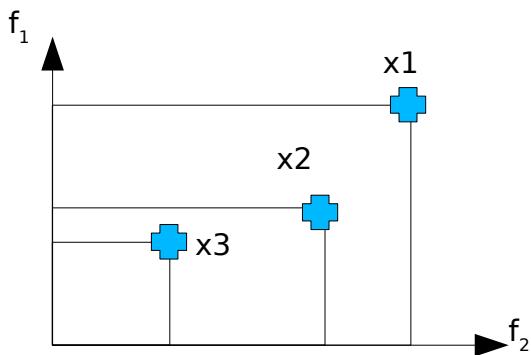


Figura 129: Ejemplo de dominancia: x_1 domina a x_2 que a su vez domina a x_3 .

¿Para qué nos sirve este concepto de dominancia? Veámoslo con un ejemplo: quiero comprar una bicicleta eléctrica y me ofrecen las que vemos en la tabla 12:

Modelo	Autonomía (km)	Peso (kg)	Precio
B1	40	33	1000
B2	50	20	1200
B3	50	20	1100

Tabla 12: Ejemplo con bicicletas eléctricas.

Este es un problema de 3 objetivos donde quiero maximizar la autonomía a la vez que minimizar el peso de la bicicleta y su precio. Por supuesto, donde quiero minimizar puedo cambiarlo de signo para convertirlo en maximizar. Está claro que la bicicleta B3 domina a B2 porque tienen la misma autonomía y peso, pero el

precio de B3 es mejor. Eso significa que, al ser B2 una solución dominada, no tiene ningún sentido seguir considerándola como una opción. Cualquier persona racional la tacharía de inmediato de la lista. Nos queda entonces considerar B1 y B3, y vemos que ninguna es óptimo de Pareto. B1 domina en un objetivo (precio) mientras que B3 domina en dos objetivos (autonomía y peso).

En principio cuando tenemos varias propuestas y ninguna domina a la otra (o sea, no hay óptimo de Pareto), todas ellas forman lo que se llama la “frontera de Pareto” (ver otro ejemplo en la figura 130), y son soluciones válidas al problema. La frontera de Pareto puede variar conforme visito más tiendas y obtengo más cotizaciones. Claro que al final, en un caso real, hay que comprar una bicicleta, por lo que tendremos que decantarnos bien sea por B1, bien sea por B3. Si no hay información adicional (preferencias, importancia de los objetivos, dinero disponible) lo más razonable es elegir B3 porque domina en más objetivos que B1.

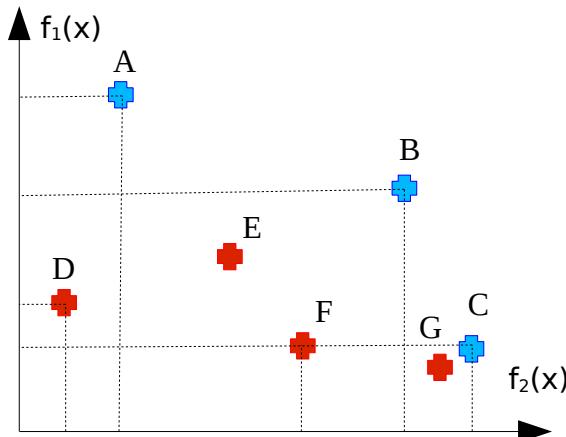


Figura 130: A, B y C forman parte de la frontera de Pareto, mientras que D, E, F y G están dominados.

Hagamos lo mismo con algoritmos genéticos. Cada cromosoma representa un punto en un espacio multidimensional de objetivos (figura 130). La idea es determinar cuál es la frontera de Pareto y darle algún tipo de privilegio, por ejemplo, eliminar los cromosomas dominados, reproducir con mayor probabilidad los que están en la frontera, o algo similar. De este modo, la frontera irá expandiéndose conforme pasan las generaciones (figura 131).

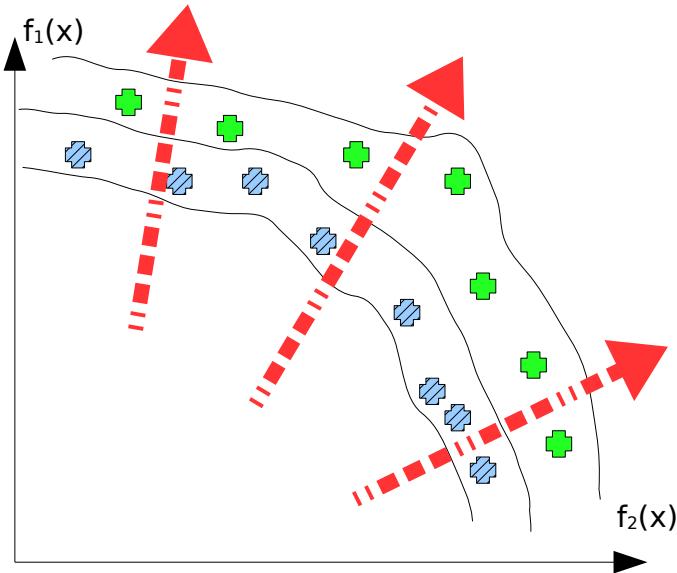


Figura 131: Evolución de la frontera de Pareto.

A veces se admite que la frontera de Pareto tenga un cierto grosor, es decir, se acepta allí algunos cromosomas dominados pero que están muy cerca de cromosomas dominantes, con el objetivo de aumentar la variedad. Si vamos a hacerlo, en el ejemplo de la figura 130 se admitiría al cromosoma G dentro de la frontera de Pareto.

Otra forma de aplicar Pareto a los algoritmos genéticos es, en el momento de hacer la selección, utilizar el número de veces que domina cada cromosoma. En la figura 132 se muestra un ejemplo con 3 funciones objetivo a maximizar. En las abscisas están 5 cromosomas que vamos a suponer son los que conforman la población en un momento dado.

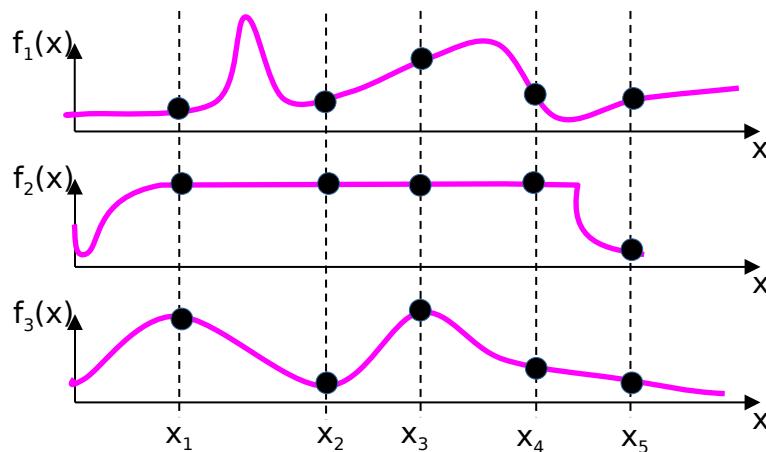


Figura 132: Selección por dominancia.

Contemos, para cada cromosoma, cuántos objetivos domina. El resultado puede verse en la tabla 13.

Cromosoma	Número de objetivos en los que domina
x1	2
x2	1
x3	3
x4	1
x5	0

Tabla 13: Dominancias de cada cromosoma.

Es importante entender que x_3 es dominante en $f_1(x)$ ya que es mejor o igual a todos los demás cromosomas. Aunque entre x_1 y x_2 hay un pico de la función $f_1(x)$ que sobrepasa a $f_1(x_3)$, ello no cuenta porque allí no hay ningún cromosoma. Es decir, hemos pintado las tres funciones en trazo continuo por razones pedagógicas, pero la verdad es que desconocemos el valor de esas funciones salvo en los sitios donde hay cromosomas (o sea, en los puntos negros).

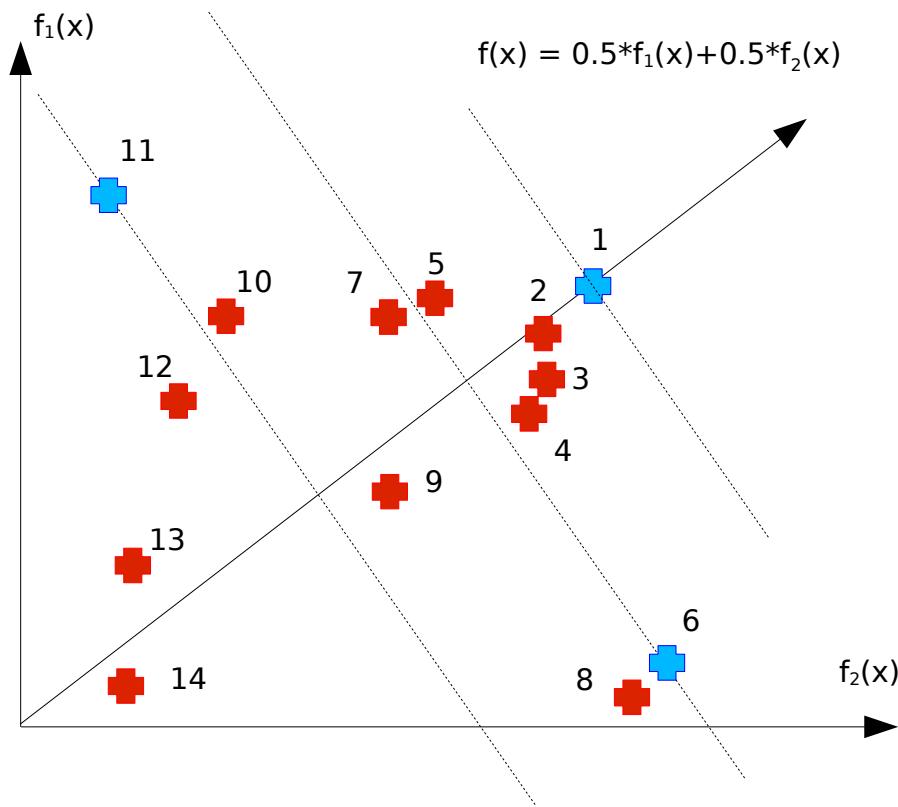


Figura 133: Linealización con 50% y 50%.

Para terminar, vamos a explicar por qué linealizar todos los objetivos en uno solo no es una buena idea. Supongamos que tenemos dos objetivos y la población de soluciones dada en la figura 133 donde los cromosomas azules son la frontera de Pareto, y los rojos son los dominados. Si hallamos el promedio de ambos objetivos

para resumirlos en uno solo, esto es, $f(x) = 0.5*f_1(x)+0.5*f_2(x)$, con ello creamos un nuevo eje (en diagonal) respecto al cual hay que maximizar. El mejor punto según este criterio lo hemos marcado con el número 1, el siguiente con el 2, siendo el último el 14.

Aquí ya se evidencian problemas: el punto 11, a pesar de estar en la frontera de Pareto, es superado por muchos puntos mediocres {2, 3, 4, 5, 7, 8, 9 y 10}. Lo mismo le ocurre al punto 6, que es superado por los mediocres {2, 3, 4 y 5}.

Pero la injusticia no termina allí. Supongamos ahora que las ponderaciones para linealizar los dos objetivos sean 20% y 80% respectivamente. El resultado lo tenemos en la figura 134 donde hemos vuelto a asignar un número de orden a cada solución (1 a la mejor, 14 a la peor). Allí podemos comprobar que la mejor solución ahora es la que antes quedó en sexto lugar, y la siguiente es la que quedó antes en primer lugar.

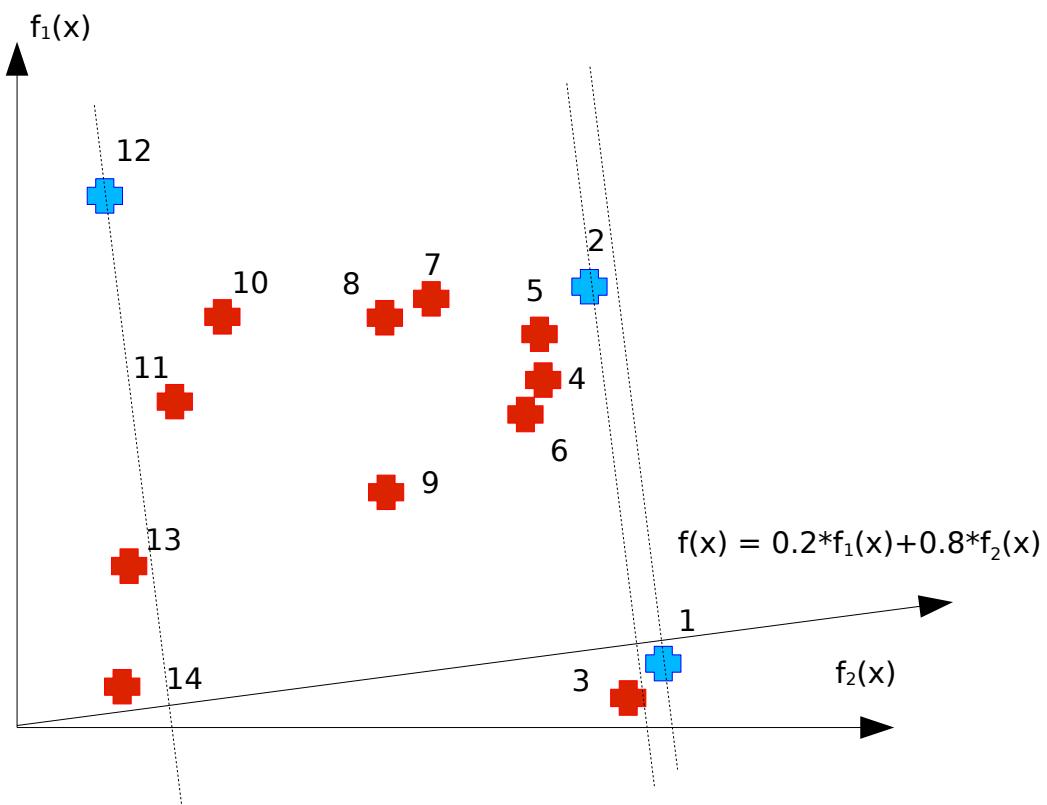


Figura 134: Linealización con 20% y 80%.

Al cambiar las ponderaciones cambia el orden radicalmente y, además, hay de nuevo una injusticia con el punto 12 que, estando en la frontera de Pareto, queda superado por los mediocres {3, 4, 5, 6, 7, 8, 9, 10 y 11}.

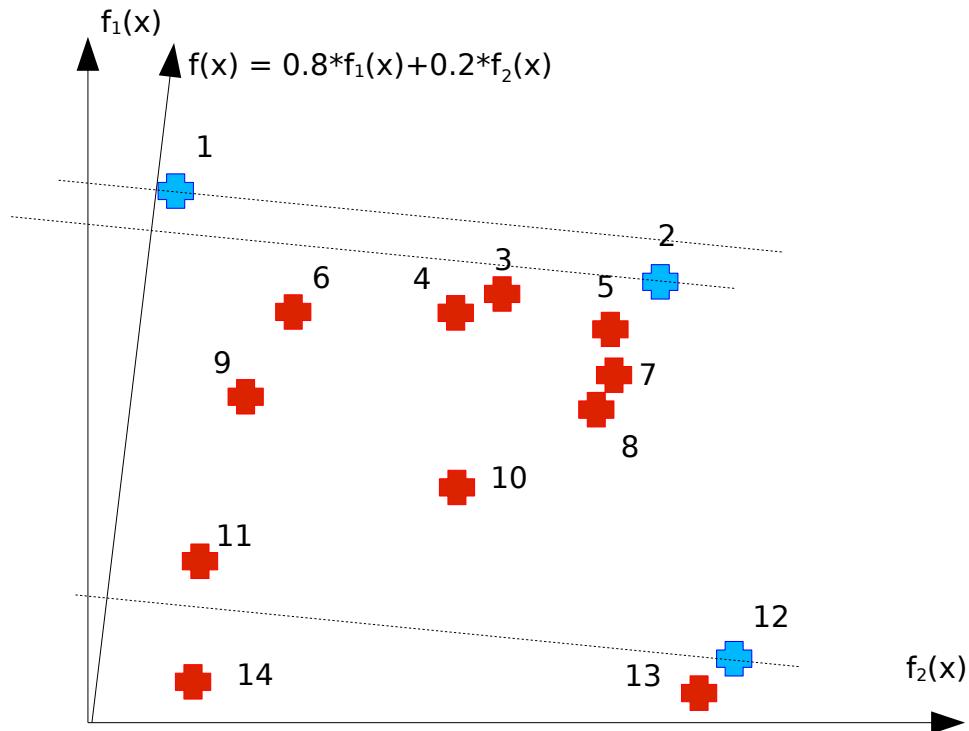


Figura 135: Linealización con ponderaciones 80% y 20%.

Repetiendo de nuevo el experimento con ponderaciones 80% y 20%, cuyo resultado podemos ver en la figura 135, observamos que ahora el mejor cromosoma es el que quedó en lugar 11 en la primera clasificación. Otro cambio de orden radical y nuevas injusticias.

Lo único que se puede demostrar es que al linealizar el primer punto siempre pertenece a la frontera de Pareto. Pero el orden cambia con los pesos asignados, y no de forma leve, sino radical. Además, se cometan injusticias con puntos mediocres que quedan mejor clasificados que puntos que pertenecen a la frontera de Pareto, es decir, los realmente mejores.

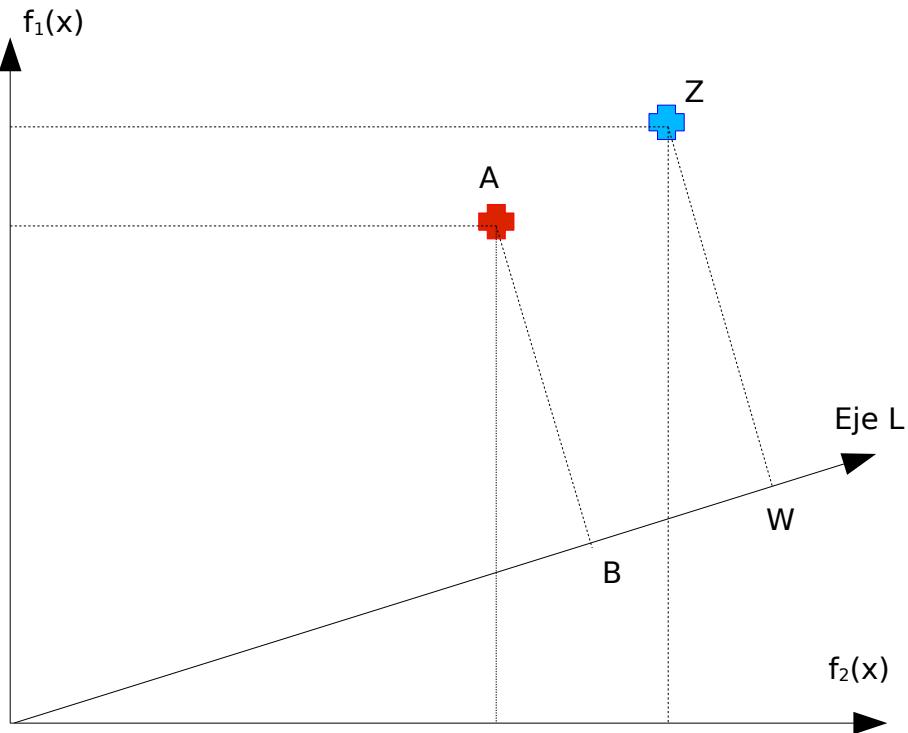


Figura 136: El mejor punto de cualquier ponderación siempre está en la frontera de Pareto.

Teorema. Cuando se hace un *ranking* con ponderaciones positivas, la mejor solución siempre está en la frontera de Pareto, independientemente de las ponderaciones que se asignen a cada eje para hacer el *ranking*.

Demostración. Supongamos que no es así, es decir, que existe un punto dominado A (o sea, que no está en la frontera de Pareto) que queda primero en un cierto *ranking*. Ese *ranking* se caracteriza por unos pesos multiplicativos positivos sobre los ejes, que producen una transformación lineal, o sea, un nuevo eje L , dentro del primer cuadrante (figura 136).

Entonces la perpendicular a ese eje L que pasa por ese punto A , interseca al eje L en un cierto punto B .

Sin embargo, al ser dominado existirá un punto Z que domine a A , es decir, que sea mejor o igual a él en todas las coordenadas.

Ello implica que la recta que pase por Z siendo perpendicular al eje L , intersecará a Z en un punto W tal que se cumplirá $W > B$ respecto al eje L , al ser todas las ponderaciones positivas (primer cuadrante en la figura 136).

De modo que ello contradice que A sea el primero en el *ranking* respecto a un eje L .

Por tanto la suposición es falsa, es decir, no existe ningún *ranking* (eje *L*) en donde la proyección *B* de un punto dominado *A* quede en primer lugar.

De ahí se deduce que en cualquier *ranking*, independientemente del peso que se dé a cada eje, la primera solución siempre pertenece a la frontera de Pareto.

La mejor solución siempre es una excelente solución, lo cual es bueno. Desgraciadamente no hay garantías para el resto de puntos que conformen la frontera de Pareto, pues pueden quedar en cualquier orden en un *ranking*, en función de los pesos que se den a cada eje.

Una última reflexión: realmente solo tiene sentido linealizar objetivos cuando hay un factor de conversión. Por ejemplo, quiero maximizar las ganancias de mis inversiones en pesos colombianos, dólares y euros. En realidad no son tres dimensiones sino solo una porque existen factores de conversión entre monedas. Es lo mismo que si quiero maximizar la carga en un camión de dos compartimentos y en un lado mido en kilos y en otro en gramos. Dado que $1000 \text{ g} = 1 \text{ kg}$ entonces no son dos dimensiones distintas, sino solo una. Pero, en el ejemplo de la bicicleta, no hay ningún factor que convierta kg a km de modo que son dimensiones distintas, cuyos objetivos no se pueden linealizar.

A pesar de que este tópico es bien conocido, sigue siendo muy común encontrar problemas multiobjetivo donde se linealiza para encontrar una solución. Lo vemos continuamente en las clasificaciones de las mejores universidades y lo hemos criticado (Delgado, 2017) pues este error suele tener repercusiones sociales importantes.

Recomendaciones

Si son tan buenos, ¿debo usar siempre un algoritmo genético para problemas de optimización? Definitivamente no, y en el segundo libro veremos las razones teóricas para ello (el teorema de *no-free-lunch*). De momento plantearemos los escenarios donde conviene o no conviene usar algoritmos genéticos.

Si se conoce un algoritmo determinista que solucione el problema en un tiempo razonable, utilice ese algoritmo. Los algoritmos genéticos van a ser más lentos y menos precisos. Ejemplo: si necesita calcular el máximo de una función continua, derivable y con uno o pocos picos, utilice el algoritmo del gradiente.

Si el problema tiene muchas restricciones duras (esto es, valores de los parámetros que producen soluciones inválidas), entonces no suele ser

conveniente utilizar algoritmos genéticos, pues no van a encontrar la solución o van a requerir demasiado tiempo. Un ejemplo con restricciones duras puede ser programar horarios y aulas para una universidad. Aquí hay varias restricciones duras: no se pueden dar dos clases en la misma aula a la misma hora, un profesor no puede dar dos clases en el mismo horario, el número de estudiantes en un aula debe ser menor o igual al número de sillas, y algunas otras. Los algoritmos genéticos no van a funcionar aquí debido a que los operadores de reproducción producirán muchas soluciones inviables. Conforme aumentan las restricciones, la mayor parte de los cromosomas (probablemente todos) van a representar soluciones inviables. La aptitud de esas soluciones es cero y la presión selectiva no operará con ellas. No hay forma de orientar el algoritmo para que, a partir de esas soluciones, encuentre otras mejores. El algoritmo genético degenerará en una búsqueda al azar. En este caso es mejor emplear otro tipo de algoritmo, como los basados en restricciones.

La única alternativa es convertir de alguna forma las restricciones duras en restricciones blandas, y allí sí pueden funcionar los algoritmos genéticos espléndidamente. En el ejemplo anterior podríamos aceptar soluciones con más estudiantes que sillas, penalizando progresivamente la función de aptitud (si nos pasamos poco, penalizamos poco, y si nos pasamos mucho, penalizamos mucho). Como consecuencia, el algoritmo sigue operando con la selección presionando para lograr mayores aptitudes, lo que implica menores penalizaciones, es decir, que tratará de buscar aulas donde quepan realmente los estudiantes. Y, por otro lado, si la solución final es definitivamente inviable (por ejemplo, un grupo de 45 alumnos programado en un aula de 40 sillas), el rector de la universidad puede decidir comprar las 5 sillas faltantes y acomodarlas de alguna manera para evitar dejar de dictar una asignatura. En este sentido, los algoritmos genéticos pueden ofrecer aproximaciones razonables a soluciones, aunque sean levemente inviables. Los algoritmos basados en restricciones no pueden hacerlo porque podan inmediatamente el árbol de búsqueda cuando no se cumple alguna restricción, impidiendo acercarse a soluciones levemente inviables.

Si el problema no tiene propiamente una función de aptitud, o la función no es derivable o no es continua, o el problema está mal definido, o tiene mucho ruido, o cambia con el tiempo, o el espacio de búsqueda es enorme o no existe ningún algoritmo que lo resuelva en tiempo razonable (típicamente en problemas NP o peores), entonces los algoritmos genéticos son una buena opción, probablemente la única. Un ejemplo es el diseño artístico. Es imposible definir matemáticamente si una pintura o una canción es más bonita que otra. No hay posibilidad de sacar la derivada de la belleza de una escultura. E incluso el concepto de originalidad, creatividad y belleza en el arte cambia con el tiempo y con la persona que

observa. Por otro lado, el espacio de diseño es enorme. Piense en cuantos cuadros distintos se pueden pintar. En estos casos, los algoritmos genéticos suelen funcionar muy bien. Habitualmente lo que se hace es tomar como función de aptitud el criterio subjetivo de muchos observadores humanos, exponiendo los objetos artísticos fabricados por los cromosomas a través de una aplicación web. Los objetos más votados por los humanos se reproducirán más. En EVALAB hemos realizado dos trabajos de grado en este sentido para diseño de logotipos (Camayo, 2009) y para creación de nuevos sonidos (Barona, 2013).

Otras aplicaciones

Una de las primeras aplicaciones que reavivó el interés por estos algoritmos, y que fundó el área de *hardware evolutivo*, la realizó Adrian Thompson en 1996 usando *FPGA*²⁹. El objetivo era distinguir entre dos tonos, uno de 1 kHz y otro de 10 kHz (usado para reconocer los dígitos a partir de las pulsaciones en los actuales teléfonos multitono).

Una *FPGA* es un chip que contiene millones de puertas lógicas *AND*, *OR*, *NOT* y bits de memoria, que son las puertas básicas con las que se puede construir cualquier circuito digital, incluso un computador, aunque la tecnología de aquella época solo integraba unos pocos cientos. Lo interesante de la *FPGA* es que las conexiones entre las puertas son programables por *software*. Hay millones de conmutadores que permiten conectar o desconectar las salidas de cualquier puerta con las entradas de otras. Cada conmutador contiene un bit de memoria que puede ser programado (0=desconexión; 1=conexión). En la figura 137 puede verse un esbozo de ello, aunque las arquitecturas reales de las *FPGA* suelen ser más complejas. Lo relevante es que en los chips convencionales esas conexiones las hace el fabricante y quedan para siempre, mientras que en las *FPGA* se pueden hacer y deshacer tantas veces como se quiera, enviando la correspondiente secuencia de ceros y unos desde el exterior de la *FPGA* (por ejemplo, desde un computador) hacia su matriz de conmutadores.

Thompson diseñó en su computador un algoritmo genético donde los cromosomas eran secuencias completas para programar los conmutadores de su *FPGA*. Por mutación y cruce se generaban nuevas secuencias. Para evaluar un cromosoma se enviaba esa secuencia de ceros y unos a la *FPGA*, se le injectaban tonos de 1 kHz y 10 kHz y se veía si los conseguía identificar, en función de lo cual se le asignaba una aptitud. Y ya tenemos la evolución en marcha. Al principio, todos los cromosomas eran inútiles, pero con el paso de las generaciones fueron

²⁹ Field Programmable Gate Array, o sea, matriz de puertas lógicas programable después de fabricar e incluso vender el circuito.

mejorando, hasta alcanzar el éxito completo en la generación 4000.

La primera conclusión es que los algoritmos genéticos pueden diseñar complejos circuitos electrónicos sin saber nada del tema.

La segunda conclusión fue más interesante aún. Cuando Thompson revisó el interior de la *FPGA* para averiguar qué clase de circuito había diseñado la evolución, no entendió nada. Había puertas completamente desconectadas del resto, pero que si se eliminaban, el circuito dejaba de funcionar. Había otras puertas con entradas al aire, lo cual está completamente prohibido a los ingenieros electrónicos, pues esas entradas captan todo tipo de ruidos electromagnéticos volviendo impredecibles sus salidas. Había cortocircuitos. En fin, si este circuito hubiera sido diseñado por un estudiante, habría recibido un cero como calificación. Y, sin embargo, funcionaba. La conclusión a la que se llegó es que el circuito operaba correctamente porque las puertas se comunicaban entre sí no solo por los cables internos y los commutadores, sino también por ondas electromagnéticas. Y de allí la importancia de las entradas sin conexión, que eran antenas receptoras, y los cortocircuitos, que eran emisores. La evolución empleó todo lo que encontró para lograr su objetivo. No se limitó a la electrónica digital sino que también usó la física electromagnética. Igual que ocurre en la evolución biológica que, como decía Jacobs, no hace ingeniería sino *bricolage*.

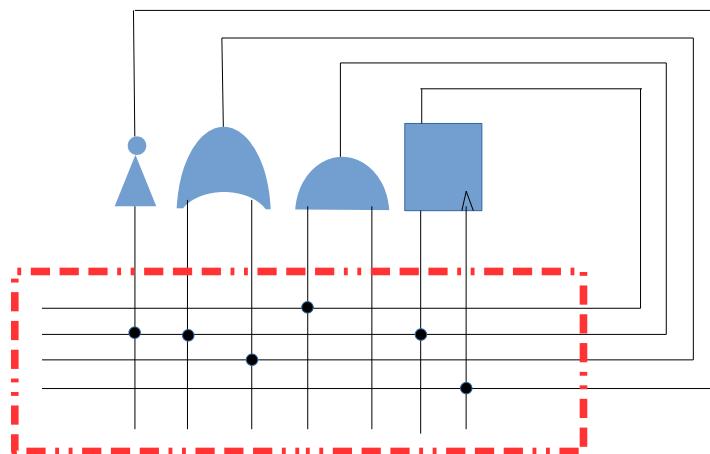


Figura 137: Lo básico de una *FPGA*: en azul, de izquierda a derecha hay una puerta NOT, una OR, una AND y un bit de memoria, con las entradas abajo y las salidas arriba. En el recuadro rojo está la matriz de commutadores, donde un punto negro significa que hay conexión.

En EVALAB hemos usado estos algoritmos para resolver varios problemas interesantes: en optimización paramétrica, para minimizar el desperdicio en el corte de telas por medio de maquinaria automática (Guzmán, 2008); en búsqueda

combinatoria para asignar espacios y horarios en consultorios médicos (Castrillón, 2015), en universidades (Villegas, 2001; Barón, 1999), esta última usando otra técnica evolutiva, llamada *simulated annealing*, que veremos enseguida; en la búsqueda de estrategias óptimas en juegos como el dilema del prisionero (Gómez, 2001), el juego del Otelo (Villate, 2012), el Tetris (Triana, 2013) o juegos de guerra (Posada, 2014); y también los hemos empleado para búsqueda de imágenes etiquetadas (Lourido y Solano, 2004).

En el dilema del prisionero, aparte de confirmar otros experimentos sobre el origen de la cooperación, obtuvimos uno de nuestros primeros resultados sorprendentes. La teoría de juegos (que se verá en el correspondiente capítulo) dice que no hay ninguna estrategia que tenga garantías de ganar siempre, aunque se pueden diseñar estrategias adaptativas que suelen ganar en muchos casos. Sin embargo, al correr nuestro algoritmo genético, obteníamos una estrategia que ganaba siempre. Al revisar el código nos dimos cuenta de un pequeño error: la variable *ultimaJugadaRealizada* debería ser un atributo de los objetos *Jugador*, pero erróneamente la habíamos declarado como atributo de la clase. Eso significa que era común a todos los jugadores. El jugador evolutivo se aprovechaba de ello para mirar cuál era la última jugada realizada por el jugador contrario y sacar ventaja de ello. Como dice la segunda regla de Orgel: “¡la evolución es más lista que tú!”. Ello también nos hizo reflexionar sobre las potencialidades de los algoritmos evolutivos para descubrir errores en *software*. Algo similar acaba de ocurrir con los *chatbots* de *Facebook*, que los pusieron a dialogar entre sí y descubrieron por si solos un *bug* en el *software* que les permitió crear su propio idioma, para lograr una comunicación más eficiente.

También aplicamos algoritmos genéticos para mejorar el filtro de entrada a la carrera de ingeniería de sistemas (*computing science*) de la Universidad del Valle (Torres, 2013). La idea era analizar la base de datos de estudiantes que entró a esta carrera y su nota promedio de salida (o sí había salido expulsado). Se empleaba un examen clasificatorio de entrada, que evaluaba diferentes áreas de conocimiento (matemáticas, español, inglés, física, química y ciencias sociales). El conjunto de parámetros a optimizar estaba constituido por las ponderaciones de cada una de estas áreas, lo cual indica su importancia en la carrera. La idea era buscar las características de los estudiantes que maximizaban la probabilidad de terminar la carrera y con una buena nota promedio. Con ello esperamos orientar mejor a los estudiantes al elegir su carrera, evitando la frustración de salir expulsados a la mitad. Como curiosidad, antes de realizar este estudio el componente de matemáticas era el que tenía mayor peso, pero al finalizar este trabajo nos dimos cuenta que era una equivocación, pues estudiantes con buenos conocimientos en matemáticas no necesariamente tenían buenas notas en

computación.

¿El final de las ingenierías?

Estas aplicaciones nos obligan a reflexionar sobre el futuro que nos espera. Porque, como es bien sabido, las ciencias tratan de descubrir cómo funciona el mundo, es decir, lo estudian para sacar sus ecuaciones de funcionamiento. Mientras que las ingenierías usan esas ecuaciones de funcionamiento de la materia sólida, de las ondas, de los fluidos y de la electricidad para diseñar vigas, circuitos electrónicos, tuberías y cables. La ciencia hace análisis mientras que la ingeniería hace síntesis, principalmente.

Pues bien, los algoritmos evolutivos hacen también síntesis a partir del análisis, como puede verse en la figura 138, de modo que tenderán a sustituir a los ingenieros de todas las áreas.

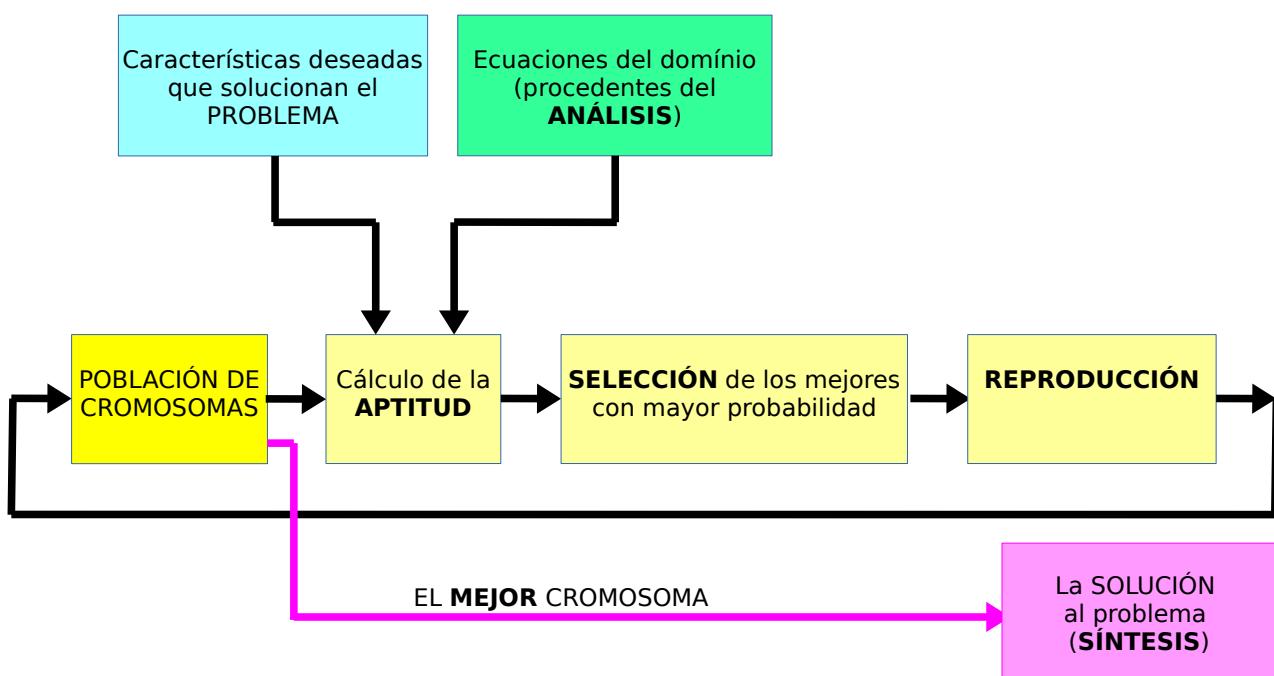


Figura 138: Síntesis, vía análisis.

Esta es otra forma de ver la figura 105, donde se hace más explícito como se calcula la aptitud de cada cromosoma a partir las características que deseo como solución a mi problema y las ecuaciones que intervienen allí (que nos las da el análisis que realiza la ciencia). De la automatización no se salvan ni la ingeniería, ni la educación, ni la ciencia, pero aquí estamos viendo lo que va a ocurrir con la primera.

Respecto a la ciencia, ya hay muchos éxitos en automatizar la tarea del descubrimiento científico usando series de datos procedentes de experimentos, de donde emergen las ecuaciones subyacentes a un fenómeno físico, aunque sean no lineales (Bongard y Lipson, 2007; Koza, 1992). Con ellas se pueden redescubrir las leyes de la física conocidas, como las leyes de Kepler, y descubrir otras nuevas.

Y no solo sustituirán a los ingenieros, sino también a cualquier proceso creativo, como el diseño industrial, la música, la poesía, la arquitectura, la pintura, por nombrar algunas. Hoy día muchos artistas reconocen que no pueden firmar sus obras, pues realmente fueron realizadas por algoritmos evolutivos. Quiero mencionar solo a dos de los más antiguos artistas que emplean esta tecnología con resultados excepcionales: Scott Draves (2017) y William Latham (2017). En los respectivos enlaces de la bibliografía de este capítulo, se puede conseguir *software* gratuito para desarrollar nuestros propios experimentos artísticos. El resultado suele ser sorprendente porque uno se da cuenta de que allí hay arte, a la vez que reconoce con asombro que no parece arte generado por un ser humano.

En el caso de las disciplinas mixtas entre ingeniería y arte, como el diseño industrial y la arquitectura, hay también resultados sorprendentes, algunos de los cuales pueden verse en Bentley (1999).

Respecto a la ingeniería de *software*, está claro que la programación evolutiva y sus variantes generan programas, así sean cortos debido a las limitaciones tecnológicas actuales. Aunque ya se comienza a vislumbrar cómo hacer programas más largos utilizando la misma técnica de la figura 138 y la metodología BDD. Básicamente, el programador se limitará a diseñar las pruebas que desea que pase el *software* (el bloque de color azul de la mencionada figura). En *software* no hay “leyes de la física”, de modo que el bloque de color verde desaparece. Cada cromosoma es un programa generado al azar. Y el cálculo de la aptitud consiste en contar cuantas pruebas pasa el cromosoma. A mayor número de pruebas correctas, mayor es su aptitud. Esta idea la planteamos como tesis doctoral en EVALAB, aunque desgraciadamente no se logró concretar, pero ya ha sido llevado a cabo por Arcuri y Yao (2014), de modo que la ingeniería de *software* también está *ad portas* de desaparecer.

Mitos ampliamente divulgados sobre los algoritmos genéticos

Hay muchos mitos y son difíciles de erradicar porque en algunos casos quienes los promueven son los mismos creadores de los algoritmos, quizás como una estrategia para diferenciarse unos de otros. Vamos a mencionar los más importantes:

Genes binarios. En la literatura, especialmente la más antigua, se menciona que los cromosomas deben usar genes binarios, es decir, los alelos deben ser {0,1}. Esta idea obliga a codificar en binario las soluciones a cualquier tipo de problema. Y hay argumentos a favor de esa idea. Quizás su mayor ventaja es que simplifica el análisis teórico de las mutaciones y cruces, pero al final, no produce ninguna mejora en la eficiencia de las implementaciones. Lo más sensato es emplear una codificación directa y adecuada al tipo de problema que vayamos a resolver. Si en un gen debemos codificar si un automóvil va a llevar frenos ABS o no, es perfecto que ese gen sea binario. Pero si vamos a codificar el diámetro de las ruedas en milímetros, es mejor usar un número entero.

El cruce debe ser de un punto de corte. En la literatura, especialmente en la más antigua, se menciona que el cruce debe hacerse de uno o dos puntos. Recordemos rápidamente como es el cruce de un punto: se seleccionan dos cromosomas (los padres) y un punto de corte para ambos que los divide en dos trozos cada uno. Se toma la parte derecha del padre y se junta con la izquierda de la madre. Se toma la parte izquierda de la madre y se junta con la derecha del padre. Con ello se obtienen dos hijos. Hay mucha literatura que reconoce sus ventajas, y mucha más que destaca sus defectos. Se menciona la “hipótesis de los bloques constructivos”, que hace tiempo se ha demostrado que es falsa, y que dice que gracias a ese cruce, cierto bloque de parámetros que dan buen resultado con el padre se podrán juntar con otro bloque de la madre, produciendo un hijo con lo mejor de ambos. Esto solo es cierto para un limitadísimo tipo de problemas. Lo que es peor, en el momento de diseñar el cromosoma es importante poner juntos los genes que van a codificar esos bloques, para que el cruce no los separe, y esa información no suele ser obvia. Se han diseñado esquemas alternativos para que los genes puedan moverse de sitio y agruparse por sí mismos, pero lo único que se logra es complicar el algoritmo y volverlo mucho más lento. Por otro lado, el cruce uniforme no sufre de esos problemas, y si efectivamente existen bloques constructivos los encontrará independientemente de la posición de los genes en el cromosoma.

El cruce es lo importante. En el mismo orden de ideas, en la literatura, especialmente la más antigua, se dice que el operador de cruce es el fundamental en los algoritmos genéticos, y que el operador de mutación es accesorio, que debe usarse con baja probabilidad y que incluso se puede suprimir. Nada más lejos de la realidad, pues el operador de mutación es el fundamental, ya que sirve para generar variabilidad, cosa que el de cruce no puede hacer. Una vez perdido un alelo en un gen de la población, solo la mutación puede volverlo a crear. Y sirve para explorar todos los casos posibles si se le da suficiente tiempo, cosa que el cruce tampoco puede hacer. La mutación es un operador de cambios pequeños, por lo que explota las mejores soluciones. Es importante que haya un operador explorador que realice cambios grandes, y efectivamente eso se puede lograr con el operador de cruce, pero también con muchos otros operadores incluyendo mutaciones grandes. Además, en biología puede apreciarse que las mutaciones están presentes en todos los seres vivos, mientras que el cruce es un operador de alto nivel que solo funciona en organismos superiores y que requiere de una considerable infraestructura previa, como la diferenciación sexual.

Reflexionemos un poco más sobre este aspecto. El cruce es un operador de muy alto nivel que, en biología, funciona solo entre organismos de la misma especie, esto es, los que tienen muchísimos genes idénticos. De modo que lo que realmente hace es una búsqueda local en un espacio muy pequeño. Genera variedad, pero sin salirse de una frontera que enmarca a la especie. Por el contrario, la mutación no tiene límites. En biología hace una búsqueda global, y es la única que realmente puede generar una nueva especie.

El cruce requiere datos fuertemente estructurados. Por ejemplo, sería útil en programación orientada a componentes, para sustituir un componente por otro funcionalmente equivalente. La mutación es de bajo nivel y, por ello, la mayoría de las veces es destructiva. Pensemos, por ejemplo, si hacemos un cambio al azar en un programa en Ruby, lo más probable es que produzca algún error de sintaxis (claro que si se hacen millones de mutaciones, de vez en cuando se producirá alguna mejora al código). Pero en muchos otros problemas no existen esos datos fuertemente estructurados, de modo que la mutación es el único operador razonable.

Tienen soporte teórico y garantía de funcionamiento. Nada de ello es cierto. Holland desarrolló la “teoría de los esquemas” para dar un soporte teórico a los algoritmos genéticos, pero aunque contiene ideas muy llamativas, también estaba incompleta y adolecía de varios errores. Los errores se repararon y se terminó de desarrollar, pero el resultado es una fórmula que no vamos a presentar aquí por su inutilidad práctica, donde la mayoría de los términos son desconocidos

y de donde no sale ninguna recomendación ingenieril a seguir. Otros investigadores han probado algunas propiedades límite, por ejemplo, cuando la población es infinita, lo cual no sirve de consuelo a quien vaya a usar estos algoritmos en la práctica. Y también se han utilizado otros métodos como los modelos ocultos de Markov, con un éxito limitado y poca aplicabilidad práctica. De la misma manera, no hay garantías de que los algoritmos genéticos converjan, ni tampoco de que si lo hacen, lo hagan al óptimo global. Y si los ejecutamos varias veces sobre el mismo problema, lo habitual es que ofrezcan cada vez una solución distinta. Esta falta de garantías y ausencia de determinismo puede sonar muy desanimadora, pero la verdad es que es una de las grandes fortalezas de los algoritmos genéticos: son algoritmos creativos gracias a su aleatoriedad. En el segundo libro veremos que el teorema de Gödel limita la capacidad de los sistemas deterministas.

Sistemas clasificadores evolutivos

Los sistemas clasificadores evolutivos³⁰ también fueron ideados por John Holland como una aplicación de los algoritmos genéticos para robots artificiales (*Animats* = *Animal+robot*), que debían moverse en un cierto entorno, interactuar con él y aprender para sobrevivir. Hoy día forman parte de lo que se ha dado en llamar Aprendizaje de Máquina³¹.

Se emplea cuando se desea controlar un entorno caracterizado por cambios constantes, y eventos importantes mezclados con ruido, donde es necesario responder a esos eventos en tiempo real y donde los objetivos son implícitos o están mal definidos o hay un objetivo general difícil de cumplir como, simplemente, sobrevivir.

Es similar a un sistema experto, pues está basado en reglas de tipo *IF-THEN*, pero el conocimiento lo va adquiriendo sobre la marcha. Esa es la gran ventaja, porque no necesita un experto humano. Los expertos humanos son caros y poco colaborativos (después de todo, el objetivo es sustituirles por un *software*). Y el producto final es frágil, tanto si el experto comete algún error con las reglas que enuncia, o posee conocimientos que no logra hacer explícitos o tiene lagunas en esos conocimientos. También es frágil en el sentido de que su ámbito de

30 *Learning Classifier Systems*.

31 *Machine Learning*.

aplicación es estrictamente aquel donde fue diseñado, no permitiéndose ningún cambio. Por el contrario, un sistema clasificador evolutivo siempre está aprendiendo, por lo que puede remediar sus propios errores y lagunas y se adapta a nuevos entornos con facilidad.

Los sistemas clasificadores evolutivos están basados en una población de reglas de aprendizaje sencillas, también llamadas clasificadoras:

IF <condición> THEN <mensaje>

A ello se le llama “sistema de producción”, que es computacionalmente completo, y se implementa por medio de un algoritmo genético, donde cada regla es un cromosoma.

Tanto las condiciones como los mensajes utilizan un código binario {0,1,#}, donde “#” significa “no importa”.

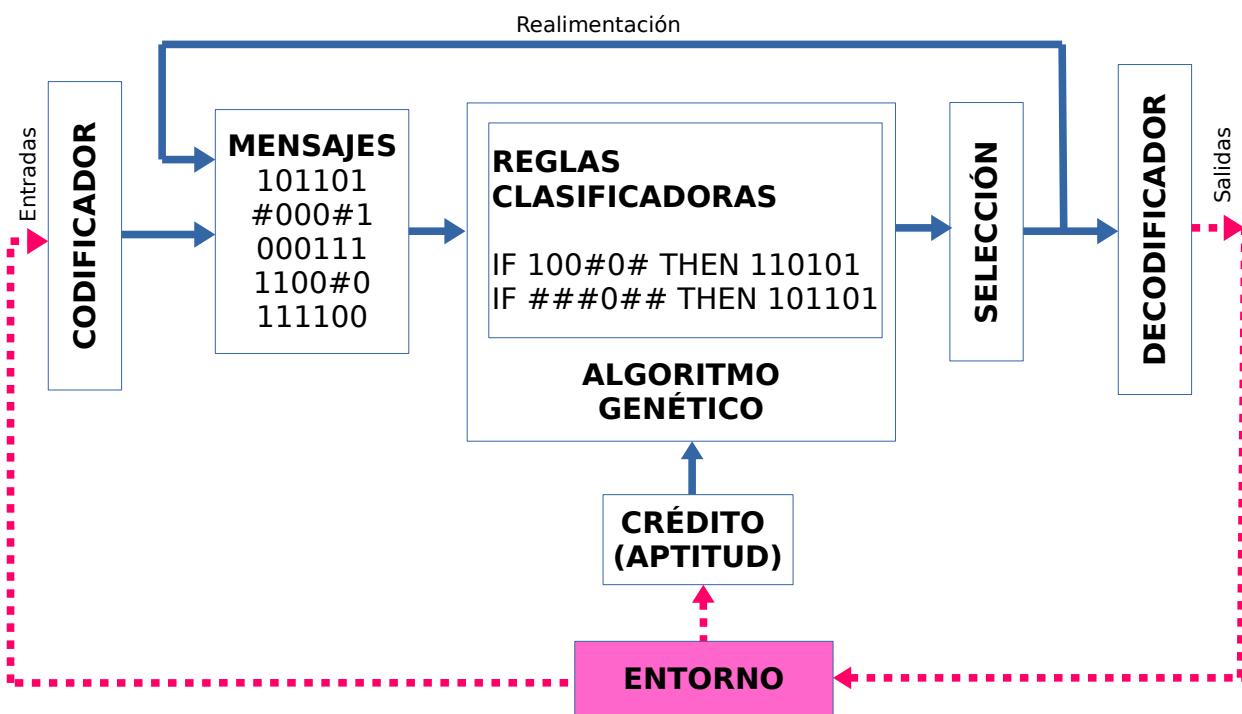


Figura 139: Diagrama de bloques de un sistema clasificador evolutivo.

Como puede verse en la figura 139, está constituido por los siguientes módulos en color azul (los de color rojo corresponden al entorno):

- Módulos de codificación/decodificación del entorno en mensajes internos.

- Módulo del algoritmo genético (donde los cromosomas son las reglas clasificadoras).
- Módulo de selección.
- Módulo de crédito (recompensa, fuerza o aptitud).

Las entradas procedentes del entorno se codifican en una o más cadenas binarias, que se inyectan a un tablón de mensajes.

Las reglas clasificadoras chequean el tablón de mensajes. Si hay alguna cadena que coincide con su respectiva condición IF, entonces se activa la respectiva regla. A continuación, el tablón de mensajes se vacía.

Cada regla clasificadora tiene asociada una aptitud. Las que estén activas compiten por ser seleccionadas (ruleta, torneo...), en función de su aptitud. Las que sean seleccionadas, generan su respectivo mensaje, que se inyecta al tablón de mensajes y que también puede ser decodificado para producir una salida hacia el entorno.

Todas las reglas activadas deben pagar un “impuesto”, que se les descuenta de la aptitud. Y el entorno debe ofrecer una recompensa a las reglas que generaron salidas buenas y que se suma a la aptitud. Esa recompensa se distribuye a todas las reglas que contribuyeron a producir esa salida.

Hay varios algoritmos de distribución, pero uno muy usado (y también muy criticado) es el de la “brigada de baldes” (figura 140), que funciona así: la regla que envió al entorno la salida, recibe una recompensa. Una parte proporcional de esa recompensa la cede a la regla que produjo el mensaje anterior que la activó, y así hacia atrás, siguiendo la trayectoria inversa que generó el mensaje de salida.

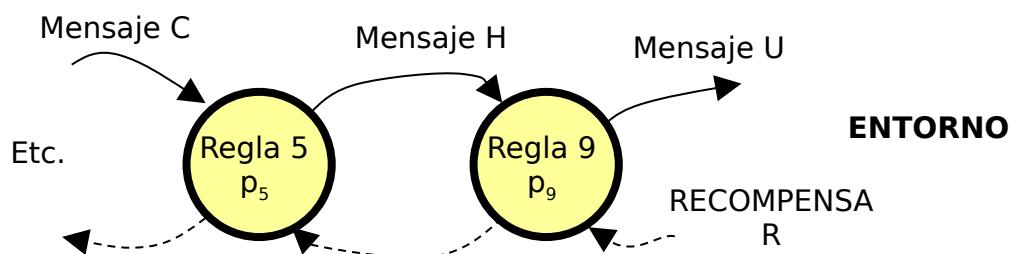


Figura 140: Reparto de recompensas con algoritmo brigada de baldes.

La recompensa se reparte usualmente con la siguiente fórmula:

$$p \leftarrow p + \beta * (R - p) \text{ con } 0 < \beta \leq 1$$

Ec. 40

siendo p la aptitud, R la recompensa del entorno y β la tasa de aprendizaje.

El principal problema de los sistemas clasificadores evolutivos es que cuando los mensajes de salida son producidos por cadenas muy largas de reglas clasificadoras, el reparto de la recompensa hace que a cada una le toque muy poco. Y entonces, el sistema aprende muy despacio o no aprende.

El objetivo del algoritmo genético es ir creando nuevas reglas que reciban mayores recompensas, o sea, que estén mejor adaptadas al entorno.

Hay dos variantes de estos algoritmos:

- **Variante Michigan** (de Holland, que es la que acabamos de ver). Cada una de las reglas clasificadoras es un cromosoma del algoritmo genético.
- **Variante Pittsburgh** (de S. F. Smith). Todo el conjunto de reglas clasificadoras es un cromosoma del algoritmo genético.

Estos algoritmos son un campo de investigación muy activo porque la verdad es que no funcionan bien, aunque deberían hacerlo. El problema está en el bucle de realimentación de la figura 139. Si no existiera, ambas variantes serían equivalentes y funcionarían muy bien. Sin embargo, su existencia implica, en la variante Michigan, la colaboración entre cromosomas dentro del algoritmo genético. Eso es difícil de lograr porque las reglas reproductivas hablan de competencia, no de colaboración (recordemos que se seleccionan los mejores con mayor probabilidad). Entonces, la colaboración no va a surgir, si surge es por casualidad y además es inestable.

La variante Pittsburgh lo soluciona haciendo que todas las reglas formen parte de un cromosoma. Los mejores cromosomas serán seleccionados con mayor probabilidad, por lo que ahora sí hay una presión para que emerja la cooperación dentro de cada cromosoma. Pittsburgh funciona bien pero, a cambio, no puede ser entrenado en tiempo real con el entorno real ya que cada conjunto de reglas querrá hacer una cosa distinta con el entorno, incompatible y potencialmente irreversible. Por ejemplo, si el robot es un coche automático y el entorno es la carretera, un conjunto de reglas puede querer girar a la derecha, mientras que el otro puede querer girar a la izquierda. Con uno de los cromosomas va a haber un accidente, y entonces el otro cromosoma tampoco va a poder seguir conduciendo. Lo ideal es que el entorno sea *resetable* a su estado inicial (lo que llamaremos “problemas clonables” en el capítulo “Inteligencia” del segundo libro), pero ello

rara vez se da.

La variante más interesante es la de Michigan, aunque no funcione bien todavía. Esta variante modela muy bien lo que ocurre en entornos humanos, donde cada cromosoma sería una persona. Pensemos en un equipo de fútbol formado por once personas. Simplificando diríamos que el equipo gana si marca muchos goles. En un momento dado hay un delantero que marca gol. La recompensa del entorno la recibe él directamente en forma de premios, el apoyo de los *fans*, elogios, y contratos de publicidad, pero el mérito no es solo de él. Seguramente un mediocampista le hizo un pase perfecto que le sirvió para marcar el gol. A su vez, el mediocampista recibió el balón de un defensa que cortó una jugada peligrosa del equipo contrario. El algoritmo de brigada de baldes debería funcionar aquí, asignando a cada jugador una parte de la recompensa. Pero este algoritmo es inestable: puede asignar recompensas a jugadores que no hayan hecho nada importante, ni bueno ni malo, simplemente estaban allí para poner el pie. Y es inequitativo: cuanto más lejos en la cadena, menos recompensa se recibe. Quizás el mayor mérito fue del defensa pues su acción tomó a todos por sorpresa, y el mediocampista y el delantero lo tuvieron fácil, pero el defensa recibirá siempre poca recompensa, mientras que el delantero recibirá mucha.

Por otro lado, también se puede pensar que la recompensa real se la lleva todo el equipo, que gana o pierde partidos y que quizás llegue a ser campeón en su torneo. En este caso, nos podemos preguntar por el reparto de recompensas interno (el salario, básicamente): ¿cómo se hace? Hay muchos factores que inciden, como el prestigio, el apoyo de los *fans*, los comentarios de la prensa, las jugadas clave donde participó, siendo todos ellos bastante difusos y subjetivos. ¿Cómo evitar que a un equipo muy bueno entren jugadores que no quieran trabajar? Dawkins demuestra que la colaboración de grupo no es evolutivamente estable en un ambiente genético. Quedaría por ver en ambientes distintos al genético. Un club de fútbol no funciona como los genes porque los hijos de los mejores jugadores no entran al equipo directamente, e incluso es más probable que se conviertan en ingenieros de *software*. Por ello tendríamos que considerar en qué ambientes y con qué estrategias se puede lograr que emerja la cooperación. Más adelante lo analizaremos en el capítulo “Teoría de Juegos”.

El problema de la falta de convergencia de la variante Michigan es muy parecido al problema de la falta de convergencia en las redes neuronales cuando tienen demasiadas capas: es difícil atribuir la recompensa por un buen resultado a alguna capa concreta. Dado que en redes neuronales ese problema ya está solucionado con los *autoencoders* conectados en cascada, es bastante posible que se pueda diseñar una técnica similar para la variante de Michigan.

Programación evolutiva

La programación evolutiva³² fue ideada por Lawrence J. Fogel en 1966. En aquella época él trabajaba con una población de solamente 2 cromosomas, posiblemente por las limitaciones de cómputo, y hacía la selección de forma determinista (entre el padre y el hijo, se quedaba con el mejor, borrando el otro y generando un nuevo hijo). Esto, como ya sabemos, conduce a óptimos locales, especialmente con poblaciones tan pequeñas, por lo que lo recomendable hoy día es usar poblaciones grandes y un método de selección probabilista como los presentados para los algoritmos genéticos.

Los cromosomas pueden ser cualquier cosa, y realmente él no hablaba de separar el espacio de genotipos y el de fenotipos, sino que trabajaba directamente en este último. Las aplicaciones típicas que muestra son para entrenamiento de redes neuronales y para diseño de máquinas de estados finitos.

No hay operador de cruce y, a cambio, hay varios de mutación. Lo importante de estos operadores es que produzcan pequeños cambios la mayoría de las veces y grandes cambios pocas veces. Esto es consistente con la propiedad que debe cumplir el operador de mutación. También produce simultáneamente búsqueda en profundidad y en anchura. Por último, suena muy bien ya que —como se ve en el capítulo de “Leyes de Potencias”— la mayoría de los objetos naturales y de sus problemas asociados tienen pocos grandes aspectos y muchos pequeños detalles.

A continuación veremos un ejemplo de diseño de una máquina de estados finitos para la predicción de una secuencia temporal. El proceso con una población de 2 máquinas es:

- Generar al azar una población de máquinas de estados finitos, donde tanto los símbolos de entrada como los de salida son el conjunto de símbolos de la secuencia temporal a predecir. Y los estados y sus transiciones son arbitrarios (al azar), así como el estado inicial.
- A cada máquina, hacerla recibir sucesivamente cada símbolo de entrada de la secuencia temporal para ver cómo predice el siguiente símbolo. Su salida en cada instante temporal discreto es la predicción del siguiente símbolo de la entrada. La aptitud de la máquina se calcula contando el número de aciertos.

32 *Evolutionary Programming*.

- Se hace una selección de las mejores con mayor probabilidad (como decíamos, Fogel seleccionaba únicamente la mejor).
- A partir de las seleccionadas se generan otras máquinas hijas aplicándoles varias mutaciones. Las mutaciones posibles son:
 - Cambiar un símbolo de entrada.
 - Cambiar una transición de estado.
 - Cambiar (añadir o quitar) el número de estados.
 - Cambiar el estado inicial.
- Estas máquinas hijas se vuelven a inyectar a la población, borrando el mismo número de originales si queremos mantener un tamaño fijo.

Veamos un ejemplo: queremos predecir la producción en toneladas de un cultivo de tomates orgánico. Tenemos los datos de las últimas temporadas, que fueron 2,2,1,0,1,3,3,0,3,0,1. Entonces creamos al azar una población de máquinas de estados finitos. Una de ellas la podemos ver en la figura 141, que tiene 3 estados $\{A, B, C\}$, siendo B el estado inicial, y cuyas entradas y salidas pueden valer $\{0, 1, 2, 3\}$.

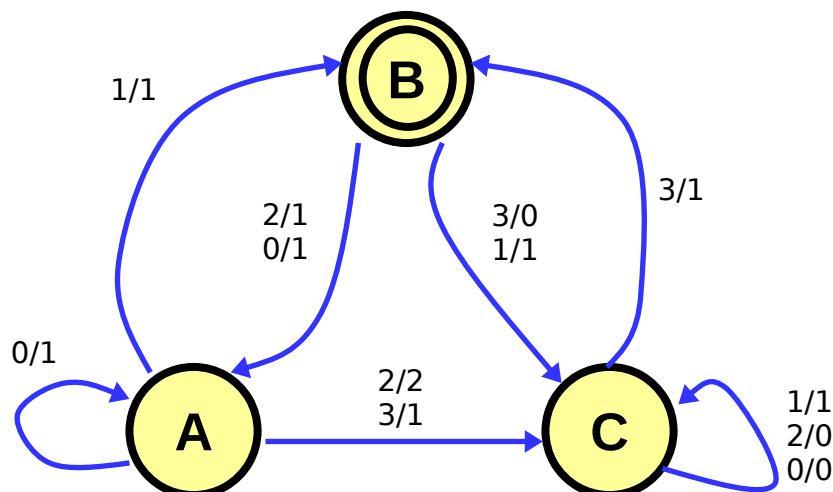


Figura 141: Máquina de estados finitos.

Estos diagramas se leen así: si estando en el estado A llega una entrada 0 , continúo en el estado A y emito una salida 1 . Si estando en el estado A llega una entrada 1 , paso al estado B y emito una salida 1 . Si estando en el estado B llega una entrada 3 , paso al estado C y emito una salida 0 . Etcétera.

Si a esta máquina le inyectamos la secuencia de entrada conocida, obtendremos los resultados indicados en la tabla 14.

Estado actual	B	A	C	C	C	C	B	C	C	B	A
Símbolo de entrada	2	2	1	0	1	3	3	0	3	0	1
Símbolo de salida			1	2	1	0	1	1	0	0	1
Aciertos			0	0	0	0	0	0	1	0	1

↓ EJE DE TIEMPO →

Tabla 14: Transiciones de la máquina de estados.

Esta tabla se lee así: estando en el estado *B* y recibiendo la entrada 2, pasa al estado *A* y genera la salida 1. Como la entrada siguiente es 2, entonces no ha acertado. Estando ahora en *A* y con entrada 2 se pasa a *C* generando la salida 2. Como la entrada siguiente es 1, tampoco ha acertado. Etcétera.

El total de aciertos de esta máquina es 2, por tanto esa es su aptitud, que sirve para realizar la selección.

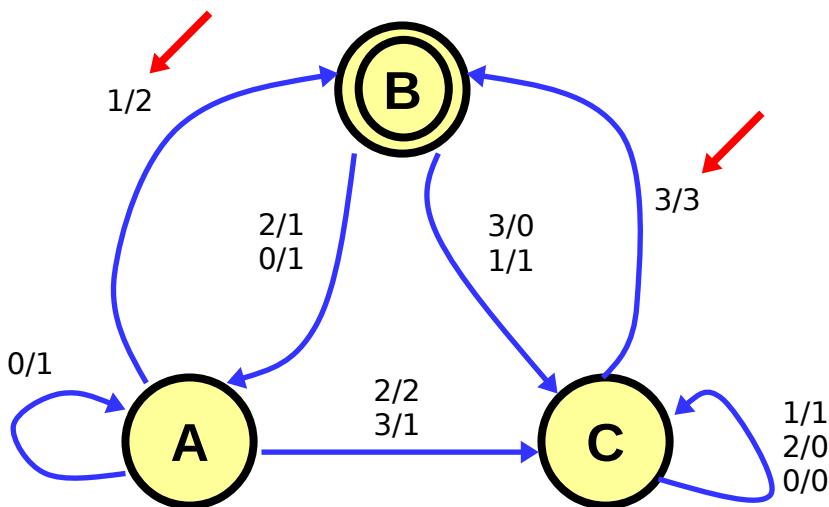


Figura 142: Máquina de estados finitos después de sufrir dos mutaciones.

Supongamos que esa misma máquina fue seleccionada para la reproducción y que el azar decide realizar con ella dos mutaciones, en los símbolos de salida indicados con flechas rojas en la figura 142.

Entonces, para evaluarla se vuelven a inyectar en secuencia todos los símbolos de entrada, obteniéndose las transiciones que vemos en la tabla 15.

Estado actual	B	A	C	C	C	C	B	C	C	B	A
Símbolo de entrada	2	2	1	0	1	3	3	0	3	0	1
Símbolo de salida		1	2	1	0	1	3	0	0	3	1
Aciertos		0	0	0	0	0	1	1	0	0	1

EJE DE TIEMPO →

Tabla 15: Transiciones en la máquina de estados hija.

Podemos observar que ahora acertó en 3 símbolos de salida, por lo que su aptitud es 3. El resto del proceso es igual al de los algoritmos genéticos, realizando sucesivas generaciones hasta lograr una cantidad de aciertos aceptable. En ese momento podremos usar la mejor máquina de estados finitos de la población para predecir el futuro.

Con el paso del tiempo dispondremos de más datos para la secuencia de entrada, que se deberán usar para seguir generando mejores soluciones.

Estrategias evolutivas

Las estrategias evolutivas³³ fueron ideadas por Rechenberg, Schwefel y Bienert en Berlín en 1963-1972, tratando de diseñar un ala de avión en un túnel de viento, introduciendo cambios al azar sobre un ala que tenía varias bisagras para permitir rotaciones y cambio de su perfil. Esto se pudo generalizar para llevarlo actualmente a aplicaciones *software*:

- La población es un vector de cromosomas y cada cromosoma es un vector de genes, exactamente igual que en algoritmos genéticos. Pero aquí, cada gen es un número flotante.
- La reproducción se logra con mutación y cruce, que son distintos a los de los algoritmos genéticos:
 - La mutación se hace sumando ruido *gaussiano* a cada uno de los genes. Esto es una gran idea, porque significa añadir cambios pequeños muy frecuentemente y cambios grandes con poca frecuencia.
 - El cruce se hace promediando los genes respectivos.
- La desviación típica del ruido *gaussiano* se puede cambiar dinámicamente de tres maneras para mejorar la búsqueda:

33 *Evolutionary Strategies*.

- Haciendo que decrezca en el tiempo. Esto tiene sentido porque al principio de la evolución es bueno explorar fuertemente el espacio de búsqueda (búsqueda en anchura), mientras que al final se requiere afinar las soluciones encontradas (búsqueda en profundidad).
- De forma adaptativa. Cuando la frecuencia de las mutaciones que tienen éxito después de varias generaciones es alta se disminuye la desviación típica (con el objetivo de hacer búsqueda local). Y cuando es baja, se aumenta (con el objetivo de generar variedad y salir del óptimo local en que se encuentre el algoritmo).
- Añadiendo al cromosoma más genes que representen la desviación típica de cada uno de los genes originales. Y dejar que la evolución se encargue de encontrar los valores más adecuados en cada momento.

En estas estrategias se introduce una nueva notación para explicar cómo es la población y la selección, de la forma siguiente:

- Se llama una estrategia (μ, λ) a aquella en la que μ padres generan λ hijos, los μ padres se descartan y los λ hijos compiten entre ellos.
- Se llama una estrategia $(\mu + \lambda)$ a aquella en la que μ padres generan λ hijos, y los μ padres y los λ hijos compiten entre ellos.

Los primeros trabajos fueron (1+1), seguramente por la baja potencia de cómputo de aquella época. Por ejemplo, si me dicen que hay una estrategia evolutiva de tipo (200+5) significa que tenemos una población de 200 individuos, que generan 5 hijos. Y de los 205 individuos se seleccionan los 200 mejores (de manera determinista) para formar la siguiente generación. Recordemos aquí de nuevo que, aunque históricamente algunos algoritmos hagan selección determinista, es más conveniente hacerla probabilista para escapar de óptimos locales.

Enfriamiento simulado

El enfriamiento simulado³⁴ fue ideado por Kirpatrick y Cerny en 1985. Está inspirado en la forma como los átomos van agregándose para pasar de líquido a sólido cristalino cuando se va bajando suavemente la temperatura. A pesar de su fundamento físico-químico, se le considera equivalente a una estrategia evolutiva de tipo (1+1).

³⁴ *Simulated Annealing*, también conocido como temple, cristalización, recocido o solidificación simulados.

La metáfora puede verse en la figura 143, donde los átomos “buscan” situarse en un lugar del retículo donde su energía potencial sea mínima. Pero pueden “equivocarse”, dado que se depositan al azar. Las equivocaciones se corrigen gracias a la energía cinética de los otros átomos, que arrancarán a aquellos que estén mal situados.

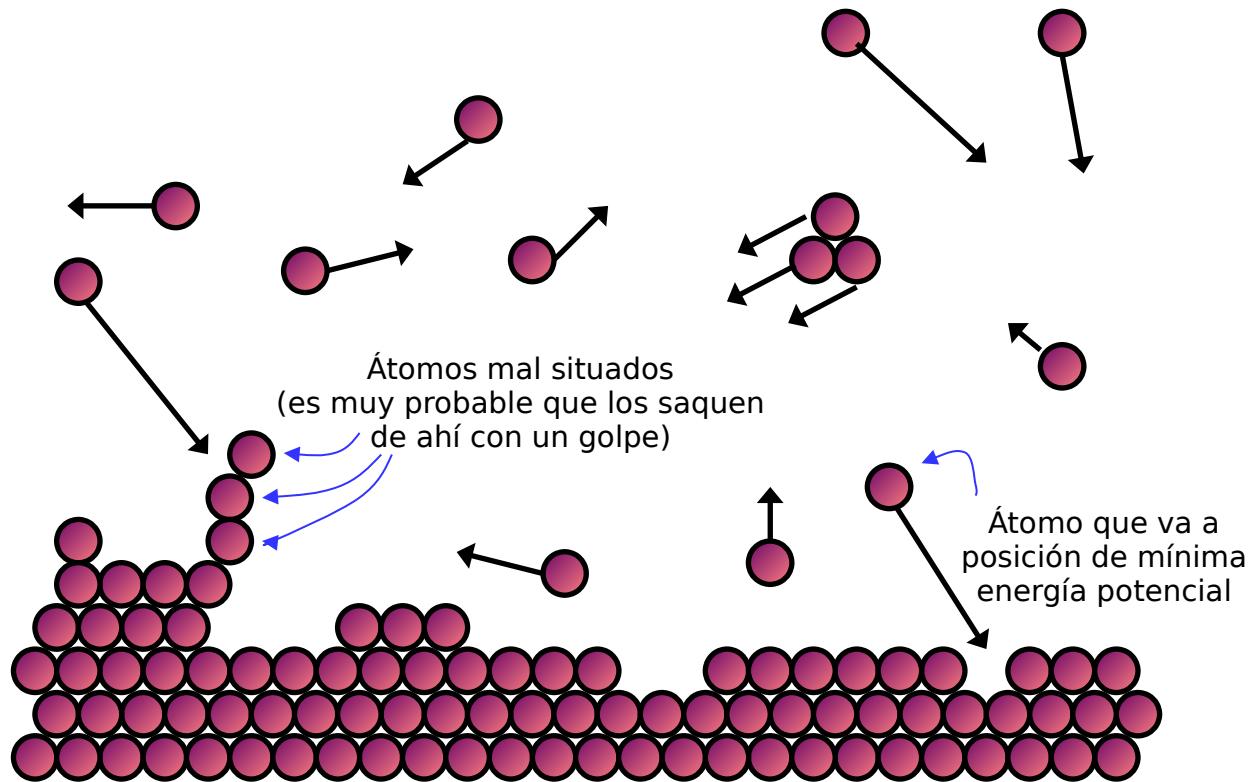


Figura 143: Fenómeno físico que usa como analogía el enfriamiento simulado.

Es importante ir bajando poco a poco la temperatura para disminuir la probabilidad de arrancar de su posición a los átomos que sí están bien situados, y forzar la solidificación de todos ellos en sus lugares de mínima energía potencial.

Entonces, el algoritmo trata de replicar este proceso en términos generales. La diferencia más importante respecto a otros algoritmos evolutivos es que ahora hay un parámetro llamado temperatura, que comienza en un valor alto y debe ir decrementándose hasta un valor mínimo. Y esa temperatura se usa en el proceso de selección: cuando la temperatura sea alta, se aceptarán con mayor probabilidad soluciones malas. Y conforme la temperatura disminuya, la probabilidad de aceptar soluciones malas también disminuye.

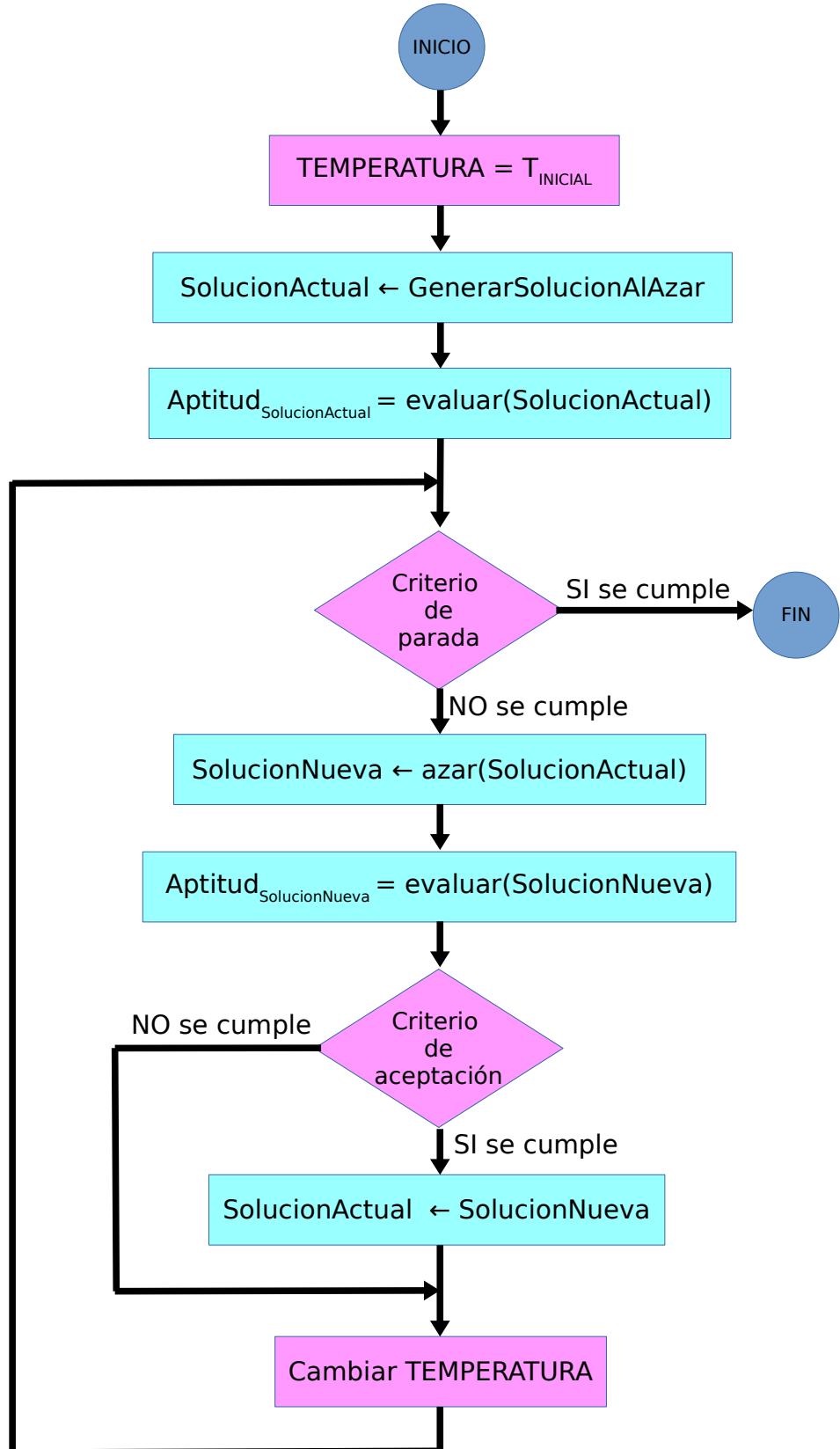


Figura 144: Diagrama de flujo de ejecución del enfriamiento simulado.

El algoritmo completo lo podemos ver en el diagrama de flujo de ejecución de la figura 144. Estos diagramas ya casi no se usan, pero para este caso resulta muy apropiado³⁵. En color rojo vemos los procesos relacionados con el nuevo parámetro de la temperatura, que hay que explicar en más detalle.

El criterio de parada es similar al de los algoritmos genéticos. Puede ser:

- Si la aptitud llegó al 100% o a un valor aceptable para el usuario.
- Si se ha alcanzado el valor final de la temperatura.
- Si se han ejecutado un cierto número prefijado de iteraciones.
- Si se ha agotado el tiempo de cómputo disponible.

Además, hay que decidir una temperatura inicial ($T_{INICIAL}$) y una temperatura final (T_{FINAL}), así como una fórmula de enfriamiento.

- La temperatura inicial debe ser lo suficientemente alta para que la probabilidad de aceptar cualquier solución sea 1. Valores mucho más altos no mejoran el algoritmo, mientras que valores muy bajos lo empeoran, siendo 100 el valor típico.
- La temperatura final debe ser lo suficientemente baja para que el sistema pueda converger, siendo el valor típico de 0.1.

A su vez, el cambio de temperatura puede hacerse de varias formas:

- Enfriamiento geométrico. No se suele recomendar por ser demasiado rápido:

$$T \leftarrow \alpha * T \text{ con } \alpha \in [0.9, 0.99] \quad Ec. 41$$

- Enfriamiento Lundy y Mess. Es bastante bueno.

$$T \leftarrow \frac{T}{1 + \beta * T_N} \text{ con } T_N = \frac{T}{T_{INICIAL}} \text{ y } \beta \in (0,1) \quad Ec. 42$$

- Enfriamiento Dowsland: es adaptativo, de modo que hace calentamiento en vez de enfriamiento, si la última solución fue rechazada. Con ello logra aumentar la diversidad de las soluciones cuando no parecen mejorar. A cambio, podría no terminar jamás:

³⁵ En muchos libros se habla de costo en vez de aptitud: mucha aptitud significa costo bajo.

Ec. 43

$$T \leftarrow \begin{cases} \frac{T}{1+\beta*T_N} & \text{si la nueva solución se aceptó} \\ \frac{T}{1-\alpha*T_N} & \text{si la nueva solución se rechazó} \end{cases}$$

con valores típicos $\alpha \approx 0.2$ y $\beta \approx 0.02$

Por otro lado, el criterio de aceptación de la nueva solución es:

- Si la nueva solución es mejor que la antigua ($\text{APTITUD}_{\text{NUEVA}} > \text{APTITUD}_{\text{ACTUAL}}$), se acepta la nueva.
- Si es peor, se genera un número aleatorio (rand) entre 0 y 1 y solo se acepta la nueva solución si se cumple que:

$$\text{rand} < e^{\frac{\text{APTITUD}_{\text{ANTIGUA}} - \text{APTITUD}_{\text{ACTUAL}}}{T}}$$

Ec. 44

Siendo e la base de los logaritmos naturales.

Si analizamos esta fórmula (que viene dada por analogía con el proceso físico de solidificación³⁶), vemos que si la temperatura es alta entonces hay una alta probabilidad de aceptar malas soluciones. Y si la temperatura es baja, la probabilidad también es baja. La probabilidad de aceptación de una solución disminuye cuanto peor sea la aptitud de esa solución. Todo es razonable y adecuado respecto a lo que hemos visto en los demás algoritmos evolutivos.

La nueva solución se genera a partir de la solución actual introduciendo algún pequeño cambio al azar y equivale a las mutaciones en los algoritmos evolutivos.

Tanto en enfriamiento simulado como en estrategias evolutivas, bajo ciertas condiciones está garantizada teóricamente la convergencia al óptimo, aunque las garantías son probabilistas, y se cumplen cuando el tiempo de búsqueda tiende a infinito, es decir, hay convergencia asintótica. De hecho, estos dos algoritmos son tan similares (a pesar de estar uno inspirado en la física y el otro en la biología), que se dice que el enfriamiento simulado es una estrategia evolutiva de tipo (1+1).

³⁶ En Física, $p = e^{-\Delta E / kT}$ siendo p la probabilidad de que un cambio de estados se consolide (se acepte), ΔE la diferencia de energías entre el estado nuevo y el actual (que, en nuestra fórmula, se cambia de signo para reflejar que las aptitudes son lo contrario que las energías potenciales de los estados físicos), k es la constante de Boltzman (que no tiene interés en un algoritmo como este) y T es la temperatura absoluta a la que se encuentra el material que se está enfriando.

Evolución diferencial

La evolución diferencial³⁷ fue creada por Kenneth Price y Rainer Storn en 1997. Como en cualquier algoritmo evolutivo, hay una población de NC cromosomas, y cada cromosoma es un vector de N números flotantes que modelan las posibles soluciones. En cada generación se intentan reproducir todos los cromosomas. La principal diferencia respecto a otros algoritmos evolutivos es que no hay mutación y el operador de cruce es de la siguiente manera: para cada cromosoma x_i con $i=\{1,2\dots NC\}$ de la población se eligen al azar otros tres cromosomas x_a , x_b , x_c distintos entre sí y distintos a x_i . Se crea un nuevo cromosoma $x_z=x_a+F*(x_b-x_c)$ siendo F una constante (las operaciones matemáticas se hacen como si cada cromosoma fuera un vector, usando sus genes como componentes). Y se hace un cruce uniforme entre x_i y x_z . Este cruce no es equiprobable, sino que la probabilidad de elegir un gen de x_z es p_{CR} y la de elegir un gen de x_i es $(1-p_{CR})$.

Por último, si la aptitud del nuevo cromosoma así fabricado es mejor que la del original x_i , se elimina x_i y se añade el nuevo a la población.

Este algoritmo depende de 3 constantes que hay que ajustar, NC , F y p_{CR} . Los valores típicos son:

- $NC = 10*n$
- $F=0.8$ Otra táctica es seleccionar en cada generación un valor de F al azar en el intervalo $[0.5, 1.0]$.
- $p_{CR}=0.9$

Al no haber mutación, no hay quien lo rescate si converge a un óptimo local, cuando todos los cromosomas sean similares. En cualquier caso, no cuesta nada añadir mutación, así no lo hayan planteado los autores del algoritmo. De la misma manera, en vez de hacer una selección determinista (elegir el mejor entre el padre y el hijo), se puede hacer probabilista.

La idea principal de la evolución diferencial es generar cromosomas variando los genes donde todavía no ha habido convergencia. Ello se logra eligiendo dos cromosomas al azar y restándolos como si fueran un vector. La resta dará cero en las componentes del vector (genes) que tengan valores muy parecidos, es decir, donde los alelos ya hayan convergido.

37 *Differential Evolution*.

Algoritmo genético híbrido de Taguchi

El algoritmo genético híbrido de Taguchi³⁸ fue creado por Jinn-Tsong Tsai, Tung-Kuan Liu, y Jyh-Horng Chou en 2004, aunque tuvo otros antecesores menos eficientes y a partir de él se han creado otras variantes, como el de Yang en 2013. Está basado en el método Taguchi que se emplea en control de calidad y optimización de procesos. Cuando el problema tiene pocas variables el algoritmo genético básico funciona bien, aunque con muchas variables es ineficiente. Para mejorarlo se puede usar este otro tipo de algoritmos cuya principal diferencia es la forma de hacer el cruce, usando dos nuevos conceptos: las matrices ortogonales de Taguchi y la relación señal-ruido (*SNR*).

La mayoría de los cruces al azar van a dar resultados muy pobres, especialmente cuando el cromosoma es muy largo. Entonces lo que se hace es una serie de experimentos con cruces y a partir de ellos, usando como criterio la *SNR*, se determina cuánto contribuye cada gen a la aptitud del cromosoma resultante. De este modo se elige el mejor de los cruces posibles.

$$SNR = (a_j - a_{MEJOR})^2$$

Ec. 45

Siendo a_j la aptitud del cromosoma en el j-ésimo experimento y a_{MEJOR} la mejor aptitud conocida hasta el momento. Hay otras fórmulas alternativas, pero esta es la más razonable, pues no presupone nada sobre la función a optimizar.

En función del número de genes del cromosoma (n) se fabrica una matriz llamada “latina” de tamaño $n+1$ filas por n columnas, que se identifica con la notación:

$$L_{n+1}(Q^n)$$

Ec. 46

Las matrices son ortogonales en el sentido de que sus columnas son linealmente independientes. Aquí no se va a mostrar cómo se construyen las matrices latinas, pero se puede encontrar la forma de hacerlo en el trabajo de grado de EVALAB de los estudiantes Cristhian Fuentes y Óscar Tigreros (2017), donde también incluyen una comparación de estos algoritmos respecto a los tradicionales. En el trabajo emplean BDD como metodología de desarrollo, usando *Cucumber* para las pruebas.

38 Hybrid Taguchi-Genetic Algorithm.

Por ejemplo, si tenemos 7 genes se requieren 7 columnas de factores, y la matriz latina es $L_8(2^7)$ que se muestra en la figura 145.

Experimento número (j)	Factores (i)						
	1	2	3	4	5	6	7
1	L	L	L	L	L	L	L
2	L	L	L	M	M	M	M
3	L	M	M	L	L	M	M
4	L	M	M	M	M	L	L
5	M	L	M	L	M	L	M
6	M	L	M	M	L	M	L
7	M	M	L	L	M	M	L
8	M	M	L	M	L	L	M

Figura 145: Matriz latina para $n=7$.

Donde:

- $Q=2$ es el número de cromosomas a cruzar. En algoritmos genéticos es siempre 2, y los vamos a llamar cromosoma L y cromosoma M .
- n es el número de genes del cromosoma (las variables que intervienen en la función a optimizar), y tiene que tener la forma $n=2^k-1$ siendo k un número natural. Si el número de genes no coincide con ningún valor 2^k-1 , entonces se elige el k inmediatamente mayor y se rellena el cromosoma con genes inútiles. O, lo que es equivalente, se eliminan columnas de la matriz latina (da igual las que se eliminan, pues todas son ortogonales entre sí). En la figura anterior $n=7$.
- $n+1$ es el número de experimentos de cruce a realizar.

Las matrices latinas que se usan son $L_8(2^7)$, $L_{16}(2^{15})$, $L_{32}(2^{31})$, $L_{64}(2^{63})$, $L_{128}(2^{127})$, $L_{256}(2^{255})$, $L_{512}(2^{511})$, $L_{1024}(2^{1023})$ y sucesivas, para problemas de hasta 7, 15, 31, 63, 127, 255, 511 y 1023 genes, respectivamente.

Supongamos que la función a minimizar sea la siguiente (es importante aclarar que nosotros sabemos que su mínimo está en $x_i=0 \forall i$, pero el algoritmo no lo sabe):

$$f(x) = \sum_{i=1}^7 x_i$$

Ec. 47

Y que los cromosomas seleccionados para cruzar son los de la figura 146, donde las variables (los genes) son números enteros.

Genes (i)							
	1	2	3	4	5	6	7
Cromosoma L	2	14	0	8	7	5	4
Cromosoma M	4	5	12	6	8	6	5

Figura 146: Cromosomas que van a cruzarse.

Entonces deben realizarse los experimentos de cruce indicados en la figura 145, cuyo resultado podemos ver en la figura 147.

Experimento número (j)	Genes (i)						
	1	2	3	4	5	6	7
1	2	14	0	8	7	5	4
2	2	14	0	6	8	6	5
3	2	5	12	8	7	6	5
4	2	5	12	6	8	5	4
5	4	14	12	8	8	5	5
6	4	14	12	6	7	6	4
7	4	5	0	8	8	6	4
8	4	5	0	6	7	5	5

Figura 147: Resultado de los 8 experimentos de cruce.

Después se calculan las aptitudes del cromosoma resultante en cada experimento (a_j) y los correspondientes SNR_j (figura 148). Vamos a suponer que la mejor aptitud que se tenía hasta este momento es $a_{MEJOR}=39$.

Experimento número (j)	Genes (i)							Aptitud	
	1	2	3	4	5	6	7	a_j	SNR_j
1	2	14	0	8	7	5	4	40	1
2	2	14	0	6	8	6	5	41	4
3	2	5	12	8	7	6	5	45	36
4	2	5	12	6	8	5	4	42	9
5	4	14	12	8	8	5	5	56	289
6	4	14	12	6	7	6	4	53	196
7	4	5	0	8	8	6	4	35	16
8	4	5	0	6	7	5	5	32	49

Figura 148: Cálculo de la aptitud y el SNR de cada experimento (suponiendo $a_{MEJOR}=39$).

Por último se calcula cuánto afecta cada cromosoma original (L y M) a la aptitud, por medio de los coeficientes de sensibilidad a cada cromosoma, definidos así:

$$E_{ik} = \sum_{\forall j : MatrizLatina_{ij}=k} SNR_j \quad Ec. 48$$

Es decir, para cada gen i se hallan dos coeficientes E_{jL} y E_{jM} . El primero con la suma de los SNR_j donde intervenga el cromosoma L , y el segundo donde intervenga el cromosoma M (figura Error: no se encontró el origen de la referencia).

Cromosoma (k)	E_{ik}						
	1	2	3	4	5	6	7
L	50	490	70	342	282	348	222
M	550	110	530	258	318	252	378

Figura 149: Coeficientes de sensibilidad de cada gen a cada cromosoma.

Por ejemplo, E_{5M} es el efecto que tiene el cromosoma M en los SNR del gen 5, y se calcula así:

$$E_{5M} = \sum_{\forall j : MatrizLatina_{5j}=M} SNR_j = SNR_2 + SNR_4 + SNR_5 + SNR_7 = 4 + 9 + 289 + 16 = 318 \quad Ec. 49$$

Por último, para cada gen se elige cuál es el coeficiente mayor (si se desea maximizar la función) o menor (si se desea minimizarla) como se ve en la figura 150. Y en función de ello se determina si el gen óptimo proviene del cromosoma L o del M (figura 151) para generar el cromosoma óptimo como resultado final del

cruce. Allí también se ha calculado su aptitud para que veamos que, efectivamente, se produce una mejora.

Cromosoma (k)	E_{ik}						
	1	2	3	4	5	6	7
L	50	490	70	342	282	348	222
M	550	110	530	258	318	252	378
Min { E_{jL}, E_{jM} }	50(L)	110(M)	70(L)	258(M)	282(L)	252(M)	222(L)
Máx { E_{jL}, E_{jM} }	550(M)	490(L)	530(M)	342(L)	318(M)	348(L)	378(M)

Figura 150: Minimizar o maximizar.

	Genes (i)							Aptitud
	1	2	3	4	5	6	7	
Cromosoma óptimo (minimizando)	2	5	0	6	7	6	4	30
Cromosoma óptimo (maximizando)	4	14	12	8	8	5	5	56

Figura 151: Cromosoma óptimo.

Como resultado de ello se tiene que para 7 genes se han realizado solamente 8 experimentos de cruce con los que se ha podido determinar el cruce óptimo. Teniendo en cuenta que existen $2^7=128$ cruces posibles, esto representa una mejoría sustancial.

Otra variante sobre este HTGA es el CCHTGA³⁹ propuesta por Poorjandaghi en el 2014, donde en cada generación los cromosomas se dividen al azar en trozos más o menos iguales (de un tamaño elegido al azar), y todos los subcromosomas homólogos forman parte de una subpoblación que se intenta maximizar (o minimizar) por separado usando un HTGA. Después se juntan los subcromosomas resultantes para formar de nuevo los cromosomas con la estructura original. Esto hace que el espacio de búsqueda disminuya, y con ello el tiempo de ejecución⁴⁰.

39 Cooperative Coevolutionary Hybrid Taguchi Genetic Algorithm.

40 En el siguiente enlace se puede encontrar una hoja de cálculo para hacer pruebas con este algoritmo:
<https://github.com/angarciaiba/libroVA>

Programación genética

La programación genética⁴¹ fue ideada por John Koza, alumno de Holland, en 1990. Koza publicó una serie de tres libros que se encuentran en la bibliografía, pero que considero muy repetitivos a excepción de las primeras páginas donde explica este algoritmo.

En la programación genética, el diagrama de flujo de datos es el mismo que el de los algoritmos genéticos, y la principal diferencia es que el cromosoma contiene un programa ejecutable, en forma de árbol sintáctico. La mutación y el cruce cambian un poco para adaptarse a ello, y la evaluación de la aptitud pasa por ejecutar el programa representado en el cromosoma.

De modo que usando esta técnica podemos dejar que la evolución escriba programas de manera automática, sin intervención humana. Realmente no son programas muy largos. Son más bien sentencias. Y se requiere mucha potencia de cómputo para poder evaluarlas, por ejemplo, un *cluster* de computadores.

Veamos los detalles:

Koza utilizó el lenguaje *LISP* para construir los cromosomas, aunque ciertamente se puede hacer con cualquier otro lenguaje. Tiene ventajas usar un lenguaje interpretado, pues así es más fácil evaluar el cromosoma, llamando al intérprete para que lo ejecute.

Como el objetivo es construir un programa que cumpla con ciertos requerimientos, lo primero que se debe hacer es definir cuáles van a ser los datos de entrada de ese programa (llamados terminales) y cuáles las operaciones permitidas sobre esos datos (llamadas funciones). En los terminales hay que incluir también las constantes que se puedan requerir (como π , e y 46) así como las funciones que entreguen salidas pero no reciban entradas como por ejemplo ocurre en un robot con los sensores de distancia a obstáculos.

Pongamos un ejemplo sencillo: queremos fabricar un sistema de alarma para la casa. Tenemos como entradas tres datos booleanos $\{D_0, D_1, D_2\}$ que representan si la puerta está abierta, si la ventana está abierta y si el propietario está en casa. Queremos que la salida sea una alarma, que indique sonoramente una condición de riesgo (típicamente que haya una ventana o puerta abierta cuando el propietario no está en casa). Como los datos son booleanos, es

⁴¹ *Genetic Programming*.

razonable asumir que las funciones también deben serlo. Definamos entonces:

- Conjunto de terminales = $\{D0, D1, D2\}$
- Conjunto de funciones = $\{\text{AND}, \text{OR}, \text{NOT}\}$

En este escenario, una posible expresión simbólica podría ser $(\text{AND} (\text{OR} D1 D0))$

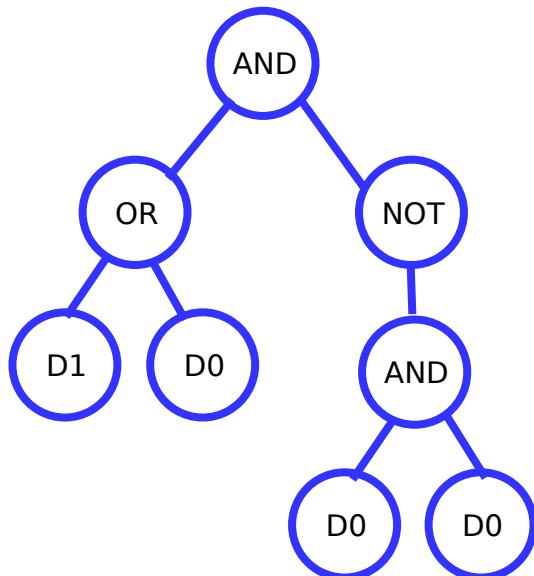


Figura 152: Ejemplo de cromosoma en un programa genético.

$(\text{NOT} (\text{AND} D0 D0))$) cuyo cromosoma vemos en la figura 152. Obsérvese que los terminales están en las hojas del árbol, y que las funciones son los nodos que tienen arcos hacia abajo (que son sus argumentos de entrada).

Al igual que con los algoritmos genéticos, aquí también deben cumplirse las condiciones de completitud y cerradura. Pero ahora, la cerradura tiene nuevas implicaciones, pues se requiere que cada una de las funciones sea capaz de recibir como argumento cualquier dato (tanto en valor como en tipo) que pueda retornar cualquier otra función. Por ejemplo, el álgebra de Boole es cerrada ya que cualquier función recibe como entrada valores booleanos y retorna como salida valores booleanos:

- Conjunto de funciones = $\{\text{AND}, \text{OR}, \text{NOT}\}$
- Conjunto de terminales = $\{\text{true}, \text{false}\}$

Lo que no ocurre con la aritmética de números flotantes, ya que la división por cero y la raíz cuadrada de números negativos generan resultados que en la mayoría de lenguajes de programación no son válidos y no se pueden usar como

entradas para otras funciones. Muchas veces lanzan excepciones y abortan la ejecución del programa. Para evitarlo se aconseja redefinir estas funciones retornando algún valor arbitrario que evite la excepción (ver un ejemplo en la figura 153).

```
def dividir(numerador, denominador)
    if denominador == 0
        return 1000000
    else
        return numerador / denominador
    end
end

def sqrt2(numero)
    return sqrt(abs(numero))
end
```

Figura 153: Operaciones aritméticas protegidas.

Ocurre algo similar cuando se necesita usar operaciones aritméticas y lógicas a la vez. En estos casos, en vez de utilizar `{false, true}` se pueden definir arbitrariamente como equivalentes a `{=0, !=0}`, por ejemplo. Así, los resultados booleanos pueden interoperar con funciones aritméticas y viceversa (figura 154).

De este modo, el cromosoma de la figura 155 es válido y ejecutable. Hay que tener en cuenta que esta mezcla de operaciones va a surgir inevitablemente, debido a la naturaleza aleatoria de los operadores de mutación y cruce.

```

def mayorQue(a, b)
    if a > b
        return 1
    else
        return 0
    end
end

def and(a, b)
    If a != 0 && b != 0
        return 1
    else
        return 0
    end
end

```

Figura 154: Algunas operaciones aritmético-logicas interoperables.

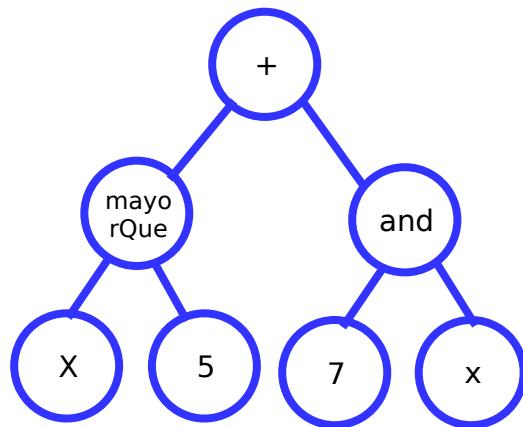


Figura 155: Cromosoma válido, con mezcla de operaciones aritméticas y lógicas.

La población inicial de cromosomas se crea al azar. Para cada cromosoma, se elige al azar entre funciones y terminales y se van llenando los arcos hacia abajo con más funciones o terminales elegidos también al azar (figura 156).

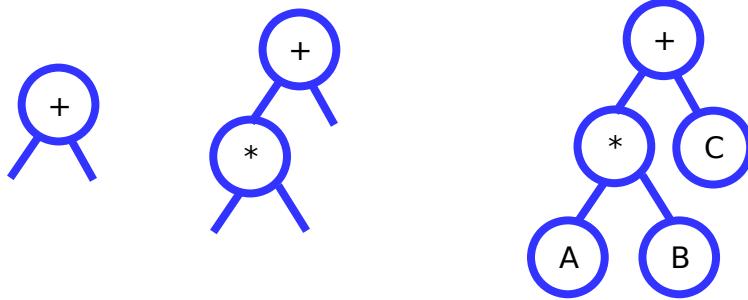


Figura 156: Creación de un cromosoma al azar.

Hay tres estrategias que nos permiten completar un cromosoma:

- **Total⁴²**. Crear los cromosomas con todos los caminos de longitud L . Para ello, en los nodos intermedios se seleccionan aleatoriamente solo funciones, y cuando se llega a la longitud L deseada, se seleccionan aleatoriamente solo terminales.
- **Creciente⁴³**. Crear los cromosomas con caminos de longitud variable, menor que L . Para ello, en los nodos intermedios se seleccionan aleatoriamente funciones y terminales, y cuando se llega a la longitud L deseada, se seleccionan aleatoriamente solo terminales.
- **Progresivo⁴⁴**. Crear un número igual de cromosomas con una profundidad especificada entre 2 y el máximo L . Por ejemplo, si se desea $L=6$, el 20% de la población tendrá profundidad 2, el 20% 3, el 20% 4, el 20% 5 y el 20% tendrá profundidad 6. Este método es el que genera más variedad.

La aptitud de un programa se calcula ejecutándolo y midiendo lo mal o bien que cumple con sus objetivos. Ello indica el punto débil de la programación genética: requiere mucha potencia de cómputo.

Los operadores de reproducción son:

- **Cruce**. Intercambiar dos subárboles al azar de dos individuos (ver ejemplo en la figura 157).
- **Mutación**. Eliminar al azar un subárbol y generar allí otro al azar (ver ejemplo en la figura 158).
- **Permutación**. Similar al cruce, pero dentro del mismo individuo.

⁴² Full.

⁴³ Grow.

⁴⁴ Ramped half and half.

- **Edición.** Eliminar bloques inútiles tales como una expresión multiplicada por cero (sustituirla por cero) y las operaciones con constantes, sustituirlas por el resultado. Koza propone hacer esta edición de los cromosomas a mano, pero eso es laborioso. Por otro lado, hacerla automáticamente es imposible en el caso general, y solo puede aspirarse a hacer en casos sencillos como los mencionados. La única utilidad de la edición es la reducción del tiempo de cómputo al calcular la aptitud ejecutando el cromosoma, por lo que no es realmente algo muy importante hoy día.
- **Encapsulación.** Seleccionar un trozo de código al azar (un subárbol del cromosoma) y encapsularlo en una función, que habrá que definir.

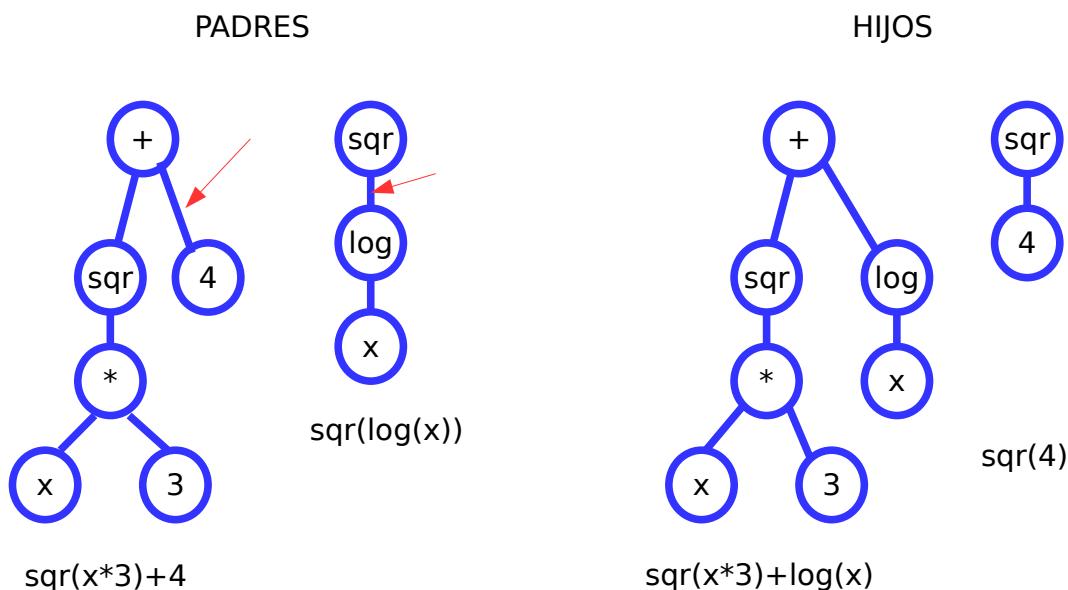


Figura 157: Cruce. Las flechas indican los puntos de corte elegidos al azar.

El usuario debe saber cuáles funciones son más apropiadas para un determinado problema, y lo mismo con los terminales. Si hay funciones o terminales superfluos aumenta mucho el tiempo para encontrar la solución. Y si faltan, no encontrará la solución. Las constantes (como π , 1, 2 y 46) se pueden dar como terminales o también se pueden generar al azar.

Algo interesante es que durante la evolución suelen aparecer muchos intrones⁴⁵, es decir, fragmentos de código que no hacen nada, por ejemplo $x-x$, $0*(\log(x))$

⁴⁵ Término tomado de la biología: un intrón es una región del ADN que no se expresa, que no sirve para nada. Lo opuesto es un exón, una región del ADN que se expresa, típicamente para participar en la fabricación de proteínas. Hay varias teorías que tratan de explicar por qué existen los intrones si aparentemente no realizan ninguna función. Una de ellas dice que antes eran exones que dejaron de tener importancia. Otra, que ayudan a realizar cambios evolutivos a largo plazo. La más reciente explica que ayudan indirectamente en la expresión de los exones. Pero quizás lo más espectacular para nosotros es que también aparecen cuando hacemos evolución de programas.

$+x*3$). Es interesante porque en la evolución biológica también aparecen intrones, aunque en la programación genética son simplemente un desperdicio de memoria y de tiempo de ejecución.

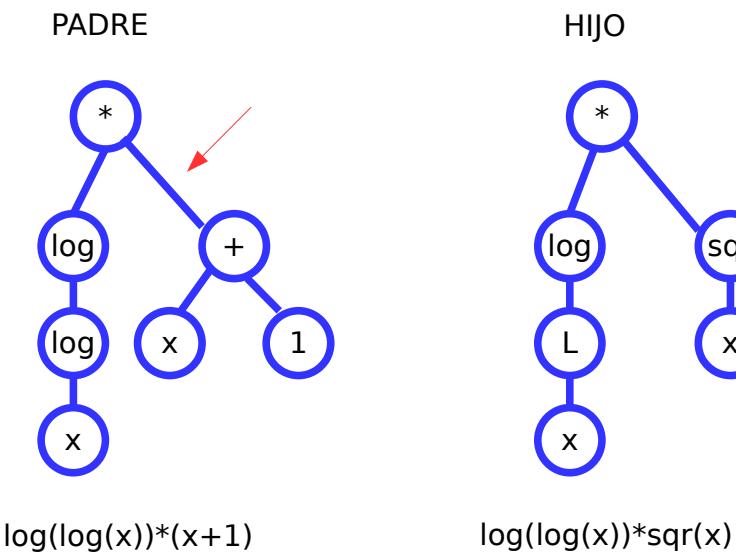


Figura 158: Mutación. La flecha indica el punto de corte elegido al azar.

Las aplicaciones de la programación genética son muy numerosas:

- Integración, derivación e inversión simbólicas. Predicción de secuencias. Regresión simbólica. Todas son similares. Si tenemos una serie temporal (por ejemplo, los litros de lluvia por metro cuadrado en Cali, día a día) y queremos predecir qué ocurrirá mañana, lo que muchas veces se hace es tratar de interpolar linealmente los puntos (por “mínimos cuadrados”) para ver por dónde va a pasar esa recta en días futuros. Pero el fenómeno físico puede no corresponderse con una ecuación lineal, de modo que la predicción puede fallar bastante. Es mejor plantear un cromosoma donde puedan salir otro tipo de ecuaciones (polinómicas, con funciones trigonométricas o logaritmos) y dejar que la evolución encuentre la más adecuada. La aptitud de cada cromosoma se calcula como la sumatoria de los errores de ajuste (cambiado de signo) de la ecuación del cromosoma respecto a todos los puntos conocidos. En la figura 159 podemos ver que el cromosoma de color púrpura implementa la ecuación $f(t)=t*\log(t+1)+10*\cos(100*t)+5$, que es la que tiene el menor error cuadrático. La interpolación lineal (en verde) es bastante mala en este ejemplo.
- Descubrimiento de leyes a partir de datos empíricos. Por ejemplo, las tres leyes de Kepler han sido redescubiertas por estos algoritmos a partir del

conjunto de datos astronómicos obtenidos por Tycho Brahe.

- Compresión de datos con pérdidas (imágenes).
- Diseño en ingeniería civil, arquitectura, muebles, arte.
- Estrategias en robots (hormigas artificiales).
- Diseño de circuitos digitales.
- Comportamiento emergente (hormigas artificiales, agentes, robots).
- Diseño de controladores y automatismos (péndulo invertido, aparcar un camión largo).
- Diseño de estrategias óptimas en juegos (*Pac-Man*, *Otelo*, damas, y muchos otros).
- Generación de secuencias seudoaleatorias.
- Clasificación de datos (*Clustering*, *Data Mining*).

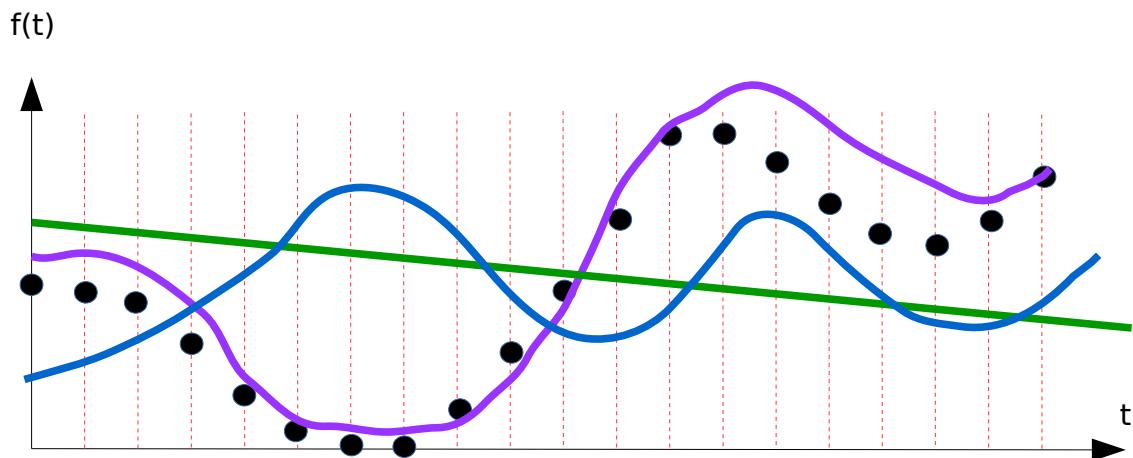


Figura 159: Interpolación simbólica de una secuencia de puntos (en negro). Se muestran dos cromosomas: en púrpura el que se aproxima más a la solución, y en azul otro que no se acerca tanto. En verde está el resultado de interpolar linealmente.

Para concluir, es bueno mostrar los problemas que se presentan en programación genética con el fin de entender las dos variantes expuestas en los siguientes capítulos.

- Manejar árboles es engoroso.

- Se requiere una gran capacidad de cómputo debido a que hay que ejecutar los programas que producen los cromosomas, y puede haber miles de cromosomas y millones de generaciones.
- La ejecución de código arbitrario en un cromosoma puede producir problemas de seguridad. Para evitarlo, hay que crear un ambiente cerrado adecuado (un *sandbox*) donde el malfuncionamiento de un programa procedente de un cromosoma no afecte a nadie más. Además, es recomendable lanzar a la vez un hilo con una temporización, para que cuando expire se mate el programa, pues probablemente se haya quedado atascado en algún bucle. Si eso ocurre se le asigna la aptitud más baja.
- La mutación casi nunca produce cambios pequeños, lo cual no es bueno. Viendo el ejemplo de la figura 158, si el arco seleccionado al azar está muy arriba del árbol (muy cerca del nodo raíz) la mutación hará un cambio muy grande. Y si el arco seleccionado está cerca de las hojas, la mutación realizará un cambio muy pequeño. Aunque, de todos modos, sí cumple la propiedad esencial de que produciendo infinitas mutaciones a cualquier cromosoma se generarán todos los cromosomas posibles.

Evolución gramatical

La evolución gramatical⁴⁶ fue ideada por Michael O'Neill, J. J. Collins y Conor Ryan en 1997. Sirve para generar automáticamente programas, pero los cromosomas ya no son árboles sino vectores de enteros, como en los algoritmos genéticos. Su expresión en programas se hace por medio de una gramática BNF (*Backus-Naur Form*), en la que hay que definir:

- Un conjunto de símbolos terminales (+, -, 1,2,3...) y un conjunto de símbolos no-terminales.
- Unas reglas de producción que permiten generar los símbolos no-terminales en función de ellos mismos y de los terminales.

Una gramática puede expresarse por medio de una tupla $\{N, T, P, S\}$ donde N es el conjunto de símbolos no-terminales, T el conjunto de símbolos terminales, P el conjunto de reglas de producción que mapean los elementos de N en T , y S es el símbolo inicial, que debe ser miembro de N .

⁴⁶ *Grammatical Evolution*.

Es habitual no usar toda una especificación BNF de un lenguaje (pues suele ser extremadamente larga), sino solo un subconjunto de interés.

La traducción del cromosoma al programa se consigue leyendo secuencialmente los genes (números enteros) y usándolos para tomar decisiones sobre la regla de producción a elegir en la gramática BNF. En la figura 160 hay un ejemplo de regla que tiene 3 opciones numeradas a la derecha como 0, 1 y 2.

```
<sentencia> := if (<expresion>) { <sentencia>; } else { <sentencia>; } [0]
          | <sentencia> ; <sentencia> [1]
          | <variable> = <expresion>; [2]
```

Figura 160: Ejemplo de regla de producción en BNF.

La regla 0 nos dice que una sentencia puede ser un *if-else* con sus tres correspondientes sentencias (la condición, lo que se hace si es verdad, lo que se hace si es falso). La regla 1 nos dice que una sentencia puede expandirse a dos sentencias consecutivas, una detrás de otra. La regla 2 nos dice que una sentencia puede estar conformada por una expresión cuyo resultado se asigna a una variable. En las tres reglas, todos los *tags* encerrados entre <> son no-terminales, que deben expandirse subsecuentemente.

Para elegir cuál de las 3 reglas se selecciona en el ejemplo, debemos extraer un gen del cromosoma (que va a ser un número entre 0 y 255) y, ya que hay 3 posibilidades, calculamos su módulo 3. El resultado puede ser 0, 1 o 2 y, según salga, se elegirá una de las tres reglas de producción.

Y así sucesivamente hasta lograr un programa completo (con todas las reglas convertidas a nodos terminales) o hasta que se acaben los genes.

Si se acaban los genes y todavía quedan símbolos no-terminales sin resolver, hay tres opciones:

- Se puede volver a comenzar, extrayendo genes desde el principio del cromosoma, reusándolos. Esta no suele ser una idea recomendable porque se crean dependencias artificiales entre distintas partes del programa.
- Se dice que el programa resultante no es ejecutable, y se le asigna el menor valor de adaptación posible.
- Se alarga el cromosoma, para poder disponer de más genes. Enseguida veremos cómo se hace, pero solo se logra una solución parcial pues como

las reglas de producción se eligen al azar, potencialmente podría ocurrir que sigan saliendo reglas con no-terminales y, como todo tiene un límite, al final habrá que aplicar la opción anterior.

En estas gramáticas evolutivas se han introducido dos nuevos operadores, aparte de la mutación y el cruce que son iguales que en los algoritmos genéticos:

- **Duplicación.** Se seleccionan al azar unos cuantos genes consecutivos y se copian al final del cromosoma. Sirve para resolver parcialmente el problema mencionado.
- **Poda.** Si un cromosoma no necesitó usar todos sus genes para generar un programa, se aplica este operador con una cierta probabilidad. La poda consiste en quitar los genes no usados. De esta manera se eliminan los intrones (genes que no se expresan). La única ventaja de este operador es ahorrar memoria.

Debido a estos operadores, cada cromosoma puede tener una longitud distinta. Y los genes dejan de tener significado posicional. Esto hay que tenerlo en cuenta para hacer correctamente la mutación y el cruce. Por ejemplo, si se cruza un cromosoma de 100 genes (el padre) con otro de 120 (la madre), el resultado será un cromosoma de 120 genes donde los primeros 100 proceden al azar del padre o de la madre, y los últimos 20 proceden directamente de la madre.

A continuación podemos ver un ejemplo más realista donde la gramática se muestra en la figura 161, un posible cromosoma en la figura 162 y la expresión del cromosoma en un programa en la figura 163.

En las gramáticas evolutivas hay problemas de dependencias similares a los de la programación genética: cuánto más lejos de la raíz está un gen, más probable es que su expresión se vea alterada por una mutación, y potencialmente pueden salir cromosomas incompletos a los que hay que dar aptitud nula. Como ventajas se puede mencionar que los cromosomas son más sencillos de almacenar y manejar —pues son vectores de enteros en vez de árboles—, y que la mutación y el cruce son sencillos.

N = {expresion, op, pre_op, var}		
T = {sin, cos, tan, log, +, -, /, *, X, () }		
S = <expresion>		
P =		
(1) <expresion> ::=	<expresion> <op> <expresion>	[0]
	(<expresion> <op> <expresion>)	[1]
	<pre_op> (<expresion>)	[2]
	<var>	[3]
(2) <op> ::=	+ [0]	
	- [1]	
	* [2]	
	/ [3]	
	% [4]	
(3) <pre_op> ::=	sin [0]	
	cos [1]	
	log [2]	
(4) <var> ::=	X [0]	
	Y [1]	

Figura 161: Ejemplo de una gramática BNF.

6	34	12	15	7	7	0	36
---	----	----	----	---	---	---	----

Figura 162: Ejemplo de un cromosoma para la gramática BNF.

Iniciamos con S=	<expresion>
6%4=2 => regla [2]:	<pre_op> (<expresion>)
34%3=1 => regla [1]:	cos(<expresion>)
12%4=0 => regla [0]:	cos(<expresion> <op> <expresion>)
15%4=3 => regla [3]:	cos(<var> <op> <expresion>)
7%5=2 => regla [2]:	cos(<var> * <expresion>)
7%4=3 => regla [3]:	cos(<var> * <var>)
0%2=0 => regla [0]:	cos(X * <var>)
36%2=0 => regla [0]:	cos(X * X)

Figura 163: Expresión del cromosoma.

Programación por expresión genética

La programación por expresión genética⁴⁷ fue creada por Cándida Ferreira en el año 2000. Resuelve muchos de los problemas anteriores y es, posiblemente, el algoritmo evolutivo más general.

Una advertencia: los cromosomas propuestos por Ferreira son mucho más complejos que antes, y la notación de gen es un poco distinta a lo que hemos visto en el resto del libro. El nuevo gen tiene estructura interna y el nuevo cromosoma también. Veremos todo esto en detalle a continuación. El resto (población, evaluación de aptitud, selección, reproducción y reemplazo) es similar a los algoritmos genéticos.

El cromosoma sigue siendo un vector lineal de genes y cada gen codifica una sentencia de un programa usando símbolos terminales y no-terminales. El último gen es ligeramente distinto pues sirve para agrupar los anteriores en un único programa. Para ello usa símbolos no-terminales y los genes anteriores como terminales.

Todos los genes tienen la misma longitud y cada gen está formado por tres dominios: cabeza, cola e índices de constantes. En la cabeza puede haber terminales y no-terminales mientras que en la cola solo puede haber no-terminales. Los índices a constantes se explican después. El número de símbolos en la cabeza (h) se decide en función de lo complejo que se sospeche sea el problema, mientras que el número de símbolos en la cola (t) se calcula con la ecuación 50:

$$t = h * (n - 1) + 1$$

Ec. 50

Donde n es el número máximo de argumentos que tengan las funciones no-terminales.

Usando este tamaño de cola se garantiza que cualquier cromosoma pueda ser completado, independientemente de los no-terminales que tenga (lo cual es una ventaja respecto a las gramáticas evolutivas).

La forma de transformar un gen en una sentencia de un programa es a través de

⁴⁷ *Gene Expression Programming*.

las llamadas expresiones-k. En ellas se rellena el árbol sintáctico con los símbolos del gen, de arriba a abajo y de izquierda a derecha. Por ejemplo, si los no-terminales son {sqr, log, +, -, *, /} y los terminales son las variables {a, b, c} entonces n=2, ya que el número de argumentos de entrada de la suma, resta, multiplicación y división es 2, mientras que la raíz cuadrada y el logaritmo solo necesitan 1.

Supongamos que decidimos h=4. Eso hace —aplicando la ecuación 50— que t=5. En la figura 164 podemos ver un ejemplo de un gen con cabeza (genes 0, 1, 2 y 3) y cola (genes 4, 5, 6, 7, y 8) y en la figura 165 su correspondiente árbol sintáctico llenado, como dijimos, de arriba a abajo y de izquierda a derecha.



Figura 164: Gen con cabeza (en color verde) y cola (en color azul).

Es importante remarcar que:

- En la cola no hay funciones, es decir, solo hay terminales.
- No es obligatorio usar toda la cola. En este caso solo se emplea el primer símbolo.

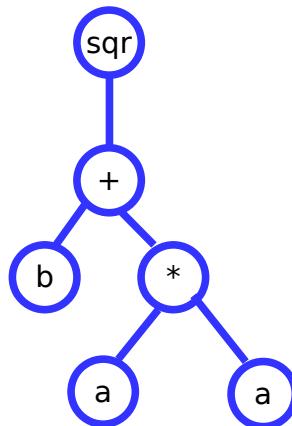


Figura 165: Árbol sintáctico.

La sentencia correspondiente es $\text{sqrt}(b+a*a)$.

A parte de la población de cromosomas, existe un vector lineal contenido solo constantes, que pueden ser útiles para la ejecución de programas. Este vector de

constantes sufre de vez en cuando mutaciones, independientemente de la población general. En los genes existe un tercer dominio que hace referencia a esas constantes por medio de índices a ese vector. Y hay un nuevo símbolo terminal que es la "?" y que puede aparecer, como cualquier otro terminal, en los otros dos dominios (cabeza y cola). En la figura 166 (la zona de color magenta son los índices que se refieren al vector de constantes) podemos ver un ejemplo, y en la figura 167 su conversión a árbol sintáctico en dos pasos: primero se genera la expresión-k y luego se sustituyen los símbolos "?" por constantes: la primera "?" se cambia por el índice 7 (el primerr gen magenta) que nos lleva a la constante -8 en el vector de constantes; y la segunda "?" se cambia por el índice 6 que nos lleva a la constante 4.4 en el vector de constantes. Si hubiera una tercera constante se cambiaría por el índice 2 que nos lleva a la constante 0.9.

	0	1	2	3	4	5	6	7				
Vector de constantes	3.14	41.5	0.9	0.02	7	-1	4.4	-8				
	0	1	2	3	4	5	6	7				
Gen	/	+	?	*	c	?	b	a	b	7	6	2

Figura 166: Vector de constantes, común a todos los genes de todos los cromosomas. Y un ejemplo de un gen con cabeza (color verde) cola (color azul) e índices a constantes (color magenta).

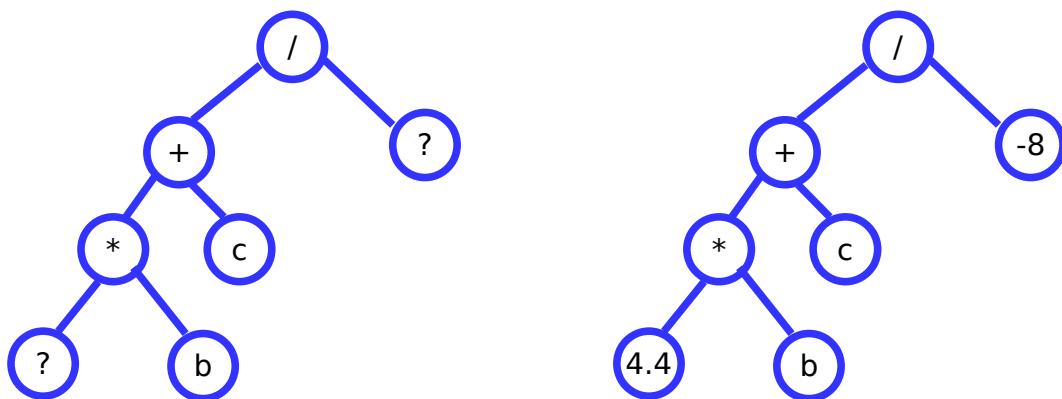


Figura 167: Conversión a árbol sintáctico en dos pasos.

La sentencia que resulta es $(4.4*b+c)/(-8)$.

Esta es una muy buena idea, aunque la implementación es un poco deficiente

porque no está claro cómo actúa la presión selectiva sobre estas constantes, ya que solo existe un vector de constantes y no hay una función de selección sobre ellas.

La forma del cromosoma en programación por expresión genética es un vector de genes, como se ve en la figura 168.

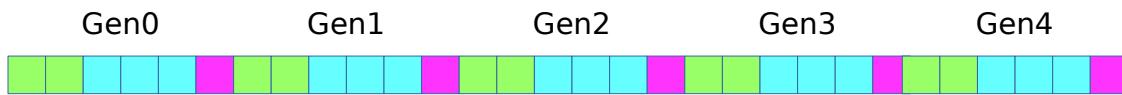


Figura 168: Cromosoma como vector de genes con cabeza, cola e índices a constantes.

Otra novedad muy interesante es que uno o más de los últimos genes pueden incluir referencias a los genes anteriores dentro de sus terminales. Con ello se logra la construcción de programas complejos, y no simples sentencias (figura 169).

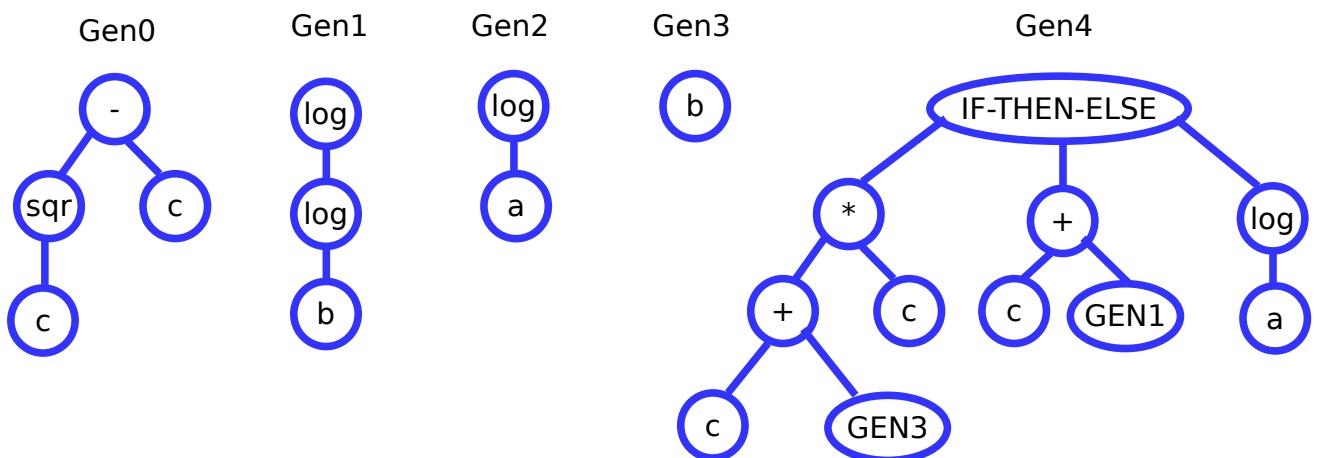


Figura 169: Ejemplo más complejo, donde el último gen puede llamar a otros genes, como si fueran funciones.

El programa correspondiente es el de la figura 170.

```
if(c+(b)*c)
    c+(log(log(b)))
else
    log(a)
```

Figura 170: Programa.

Coevolución

Cualquiera de los algoritmos evolutivos anteriores se puede usar en un escenario coevolutivo, que significa que el problema no es estático sino que también evoluciona.

En biología prácticamente todo es coevolutivo, porque cuando un ser vivo evoluciona lo hace adaptándose al comportamiento de sus predadores y de sus presas. Incluso en el caso de las plantas, donde su comida es inerte (luz, agua, aire, minerales del suelo), también coevolucionan conforme todos estos factores cambian (desaparecen minerales, cambia la composición del aire, aumenta la humedad, por mencionar algunos). Y dado que los seres vivos también producen cambios en la atmósfera, en la humedad de su entorno, en la disponibilidad de minerales y nutrientes, hay que considerar que todo el planeta Tierra es una red compleja de sistemas que se realimentan y coevolucionan unos con otros. Esto es lo que James Lovelock denominó la teoría de Gaia.

Esto se conoce también como las carreras de armamentos entre predador y presa (que veremos en el capítulo “Teoría de Juegos”) donde, a lo largo de generaciones, el predador desarrolla nuevas estrategias para capturar a la presa, mientras la presa desarrolla nuevas estrategias para huir del depredador. También se le conoce como la carrera de la Reina Roja: en *Alicia a través del espejo*, Lewis Carroll nos relata el encuentro de Alicia con un personaje de ajedrez, la Reina Roja, que vive en un extraño país donde al caminar, cada vez que da un paso hacia adelante el suelo se mueve la misma distancia hacia atrás. La Reina Roja comenta que hay que correr mucho para mantenerse en el mismo lugar. La coevolución es igual.

La coevolución puede usarse de dos formas principales:

- En juegos, en sentido amplio, podemos tener un jugador evolutivo tratando de superar a otros jugadores con estrategias bien conocidas. Una vez que el jugador evolutivo ha logrado superarlos, el nuevo algoritmo resultante se añade a los jugadores con estrategias conocidas, y se rearranca el proceso de evolución. Como resultado se irán obteniendo estrategias de juego cada vez más sofisticadas. Esto es lo que emplea rutinariamente Google (por ejemplo, con Alpha Go Zero) y otras empresas para mejorar sus algoritmos de inteligencia artificial. Aquí hacemos coevolución de soluciones.
- En problemas muy complejos, se puede partir de una versión muy

simplificada del problema y, conforme el algoritmo evolutivo la vaya solucionando, se le puede añadir más complejidad hasta convertirlo en la versión real y completa. Aquí hacemos coevolución de problemas y soluciones.

Algoritmo evolutivo general básico

Ya que existen tantas variantes de algoritmos evolutivos, se propone aquí una lo suficientemente general para servir en un amplio tipo de problemas. Como ejemplo de aplicación, se usa para resolver el problema de las N-Damas, especificándose primero los criterios de aceptación en *Cucumber*, para pasar luego al programa propiamente dicho en lenguaje *Ruby*⁴⁸.

```
# language: es
# encoding: utf-8
# Archivo: VerificarNDamasGA.feature
# Autor: Ángel García Baños
# Email: angarciaba@gmail.com
# Fecha creación: 2015-05-20
# Fecha última modificación: 2015-05-20
# Versión: 0.1
# Licencia: GPL
```

Característica: Verificar que funciona la evaluación de un Cromosoma en el algoritmo genético para la N-Damas. Nota: los tableros aquí indicados no deben tener conflictos en filas ni en columnas, pues eso no se verifica.

Escenario: Ningún conflicto

Cuando el tablero es

X		X	
X			
	X		

Entonces al evaluarlo debe indicar 0 conflictos

Escenario: Un conflicto en diagonal principal

Cuando el tablero es

X			
	X		

Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Un conflicto en diagonal principal

Cuando el tablero es

	X		
		X	

Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Un conflicto en diagonal secundaria

Cuando el tablero es

		X	
			X

48 Disponible en <https://github.com/angarciaba/libroVA>

| | | | |

Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Un conflicto en diagonal secundaria

Cuando el tablero es

	X			
X				

Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Dos conflictos

Cuando el tablero es

			X
	X		
		X	

Entonces al evaluarlo debe indicar 2 conflictos

Escenario: Dos conflictos

Cuando el tablero es

X			
	X		
		X	

Entonces al evaluarlo debe indicar 2 conflictos

Escenario: Tres conflictos

Cuando el tablero es

X			
	X		
		X	
			X

Entonces al evaluarlo debe indicar 3 conflictos

Escenario: Tres conflictos

Cuando el tablero es

			X
X			
	X		
		X	

Entonces al evaluarlo debe indicar 3 conflictos

Escenario: Cuatro conflictos

Cuando el tablero es

		X	
			X
X			
	X		

Entonces al evaluarlo debe indicar 4 conflictos

```
# language: es
# encoding: utf-8
# Archivo: VerificarNDamasCromosoma.feature
# Autor: Ángel García Baños
# Email: angarciaiba@gmail.com
# Fecha creación: 2014-11-08
# Fecha última modificación: 2015-03-24
# Versión: 0.2
# Licencia: GPL
```

Característica: Verificar el correcto funcionamiento de los Cromosomas.

Antecedentes: Crear unos cuantos cromosomas para poder trabajar con ellos

Dado que los Cromosomas van a ser de 50 genes

Y tengo un Cromosoma1 con todos los genes distintos

Y copio el Cromosoma1 al Cromosoma2

Escenario: El Cromosoma debe estar bien formado

Entonces todos los genes del Cromosoma1 deben ser distintos

Escenario: La mutación funciona
Cuando muto el Cromosoma1 10 veces
Entonces todos los genes del Cromosoma1 deben ser distintos
Y el Cromosoma1 debe ser distinto al Cromosoma2

Escenario: La mutación funciona
Cuando muto el Cromosoma1 1 veces
Entonces todos los genes del Cromosoma1 deben ser distintos
Y la diferencia entre el Cromosoma1 y el Cromosoma2 deben ser 2 genes

Escenario: El cruce funciona
Cuando copio el Cromosoma1 al Cromosoma3
Y muto el Cromosoma3 50 veces
Y cruzo el Cromosoma1 con el Cromosoma3 dando como resultado el Cromosoma4
Entonces el Cromosoma4 debe tener los genes del Cromosoma1 o el Cromosoma3
Y el Cromosoma4 debe ser distinto al Cromosoma1
Y el Cromosoma4 debe ser distinto al Cromosoma3

```
# encoding: utf-8
# Archivo: VerificarNDamas_steps.rb
# Autor: Ángel García Baños
# Email: angarciaiba@gmail.com
# Fecha creación: 2015-05-20
# Fecha última modificación: 2015-05-20
# Versión: 0.1
# Licencia: GPL
```

```
Cuando(/^el tablero es$/) do |tabla|
  tablero = tabla.raw
  @cromosoma = Cromosoma.new(0)
  tablero.each_index do |fila|
    tablero[fila].each_index { |columna| @cromosoma[columna] = fila if tablero[fila][columna] and not tablero[fila][columna].empty? }
  end
end
```

```
Entonces(/^al evaluarlo debe indicar (\d+) conflictos$/) do |numeroConflictos|
  algoritmoGeneticoTest = AlgoritmoGeneticoTest.new
  aptitud = algoritmoGeneticoTest.evaluar(@cromosoma)
  expect(-aptitud).to eq(numeroConflictos.to_i)
end
```

```
# encoding: utf-8
# Archivo: VerificarNDamas_steps.rb
# Autor: Ángel García Baños
# Email: angarciaiba@gmail.com
# Fecha creación: 2014-11-08
# Fecha última modificación: 2015-05-19
# Versión: 0.1
# Licencia: GPL
```

```
Dado /^que los Cromosomas van a ser de (\d+) genes$/ do |numeroDeGenes|
  @numeroDeGenes = numeroDeGenes.to_i
  @cromosomas = Hash.new(Cromosoma.new(@numeroDeGenes))
end
```

```
Y /^tengo un Cromosoma(.+?) con todos los genes distintos$/ do |nombreDeUnCromosoma|
  @cromosomas[nombreDeUnCromosoma] = Cromosoma.new(@numeroDeGenes)
end
```

```
Y /^copio el Cromosoma(.+?) al Cromosoma(.+?)$/ do |nombreDeUnCromosoma, nombreDelOtroCromosoma|
  @cromosomas[nombreDelOtroCromosoma] = @cromosomas[nombreDeUnCromosoma].clone
end
```

```
Cuando /^muto el Cromosoma(.+?) (\d+) veces$/ do |nombreDeUnCromosoma, numeroDeVeces|
  numeroDeVeces.to_i.times { @cromosomas[nombreDeUnCromosoma].mutar! }
end
```

```
Entonces /^todos los genes del Cromosoma(.+?) deben ser distintos$/ do |nombreDeUnCromosoma|
  vecesQueEstaCadaGen = Hash.new(0)
```

```

@cromosomas[nombreDeUnCromosoma].each { |gen| vecesQueEstaCadaGen[gen] += 1 }
vecesQueEstaCadaGen.each_value { |veces| expect(veces).to eq(1) }
end

Y /^el Cromosoma(.+?) debe ser distinto al Cromosoma(.+?)$/ do |nombreDeUnCromosoma, nombreDelOtroCromosoma|
  iguales = @cromosomas[nombreDeUnCromosoma].zip(@cromosomas[nombreDelOtroCromosoma]).reduce(true) { |
    acumulado, genes| genes[0] == genes[1] ? acumulado : false }
    expect(iguales).not_to eq(true)
end

Entonces(/^la diferencia entre el Cromosoma(.+?) y el Cromosoma(.+?) deben ser (.+?) genes$/)
do |nombreDeUnCromosoma, nombreDelOtroCromosoma, numeroDeGenesDistintos|
  diferencia = @cromosomas[nombreDeUnCromosoma].zip(@cromosomas[nombreDelOtroCromosoma]).reduce(0) { |
    acumulado, genes| genes[0] == genes[1] ? acumulado : acumulado+1 }
    expect(diferencia).to eq(numeroDeGenesDistintos.to_i)
end

Cuando(/^cruzo el Cromosoma(.+?) con el Cromosoma(.+?) dando como resultado el Cromosoma(\d+)/$)
do |nombreDeUnCromosoma, nombreDelOtroCromosoma, nombreDelCromosomaHijo|
  @cromosomas[nombreDelCromosomaHijo] =
  @cromosomas[nombreDeUnCromosoma].cruzar(@cromosomas[nombreDelOtroCromosoma])
end

Entonces(/^el Cromosoma(.+?) debe tener los genes del Cromosoma(.+?) o el Cromosoma(.+?)$/)
do |nombreDelCromosomaHijo, nombreDeUnCromosoma, nombreDelOtroCromosoma|
  correcto =
  @cromosomas[nombreDeUnCromosoma].zip(@cromosomas[nombreDelOtroCromosoma]).zip(@cromosomas[nombreDelCromosomaHijo]).reduce(true) { |acumulado, genes| genes[1] == genes[0][0] or genes[1] == genes[0][1] ? acumulado : false }
  expect(correcto).to eq(true)
end

#!/usr/bin/env ruby
# encoding: utf-8
# Programa: NDamas.rb
# Autor: Ángel García Baños
# Email: angarciaba@gmail.com
# Fecha creación: 2014-11-07
# Fecha última modificación: 2014-11-08
# Versión: 0.3

#####
# Utilidad: Enseñar a programar en Ruby. Hacer un algoritmo genético para resolver las N-Damas
#####
# VERSIONES
# 0.3 rdoc
# 0.2 Se simplificaron las clases a solo dos (Cromosoma y AlgoritmoGeneticoNDamas), para que fuera un ejemplo sencillo y didáctico.
# 0.1 La primera. Con clases Gen, Cromosoma, AlgoritmoGenetico y NDamas. Demasiado complicado.
#####
# Para ayudar a depurar:
def dd(expresion,entorno,mensaje="")
  p "# {expresion}=# {entorno.eval(expresion)} # {mensaje}"
end
# Ejemplo de uso:
# a="Hola"
# dd("a",binding)
#####
# Para que funcione bundler (que traiga las gemas especificadas en Gemfile):
require 'rubygems'
require 'bundler/setup'
#####
=begin
require 'ruby-prof'
RubyProf.start
$result = RubyProf.stop
printer = RubyProf::FlatPrinter.new($result)
printer.print(STDOUT)
=end
#####
# El Gen es un entero, por lo que no merece la pena hacer una clase para ello

```

```

# El Cromosoma es un Array de Genes (enteros)
class Cromosoma < Array
  # El Cromosoma tiene una aptitud que se puede leer y escribir
  attr_accessor :aptitud

  # Se define cuantosGenes va a tener el Cromosoma
  # Este constructor depende del problema a resolver, en este caso, las NDamas. Por ello, lo que hace es
  # construir un Cromosoma de genes enteros en un Array, cuya posición indica la columna y cuyos alelos
  # codifican el número de la fila donde se ubica cada reina
  def initialize(cuantosGenes=0)
    super
    cuantosGenes.times { |n| self[n] = n } # Para asegurar que no haya alelos (valores de genes) repetidos
    sort_by! {rand} # Se permutan los genes al azar
  end

  # La mutación se hace intercambiando dos genes cualesquiera
  def mutar!
    cualGen1, cualGen2 = rand(size), rand(size)
    self[cualGen1], self[cualGen2] = self[cualGen2], self[cualGen1]
    @aptitud = nil
    self
  end

  # El cruce se hace uniforme, eligiendo al azar un gen del padre o de la madre, para cada posición
  def cruzar(otroCromosoma)
    hijo = Cromosoma.new
    self.zip(otroCromosoma).each { |genPadre, genMadre| hijo << (rand < 0.5 ? genPadre : genMadre) }
    hijo
  end

  # No se puede usar el operador = porque hace una copia superficial (por referencia), y lo que queremos
  # es una copia profunda (por valor).
  # Ojo: El operador = no se puede redefinir porque es global (se puede usar con cualquier tipo de dato), por
  # lo que no pertenece a ninguna clase específica.
  def clone
    otroCromosoma = Cromosoma.new
    otroCromosoma.replace(self)
    otroCromosoma.aptitud = self.aptitud
    otroCromosoma
  end
end

# Esta clase implementa un algoritmo genético para resolver el problema de las NDamas. Es un Array de Cromosomas.
class AlgoritmoGeneticoNDamas < Array
  attr_reader :mejorCromosoma, :numeroDeEvaluaciones

  # En el constructor hay que especificar cuantas damas tiene el problema y cuantos Cromosomas se desea tener
  # en la población
  def initialize(cuantasDamas, cuantosCromosomas=100)
    cuantosCromosomas.times { self << Cromosoma.new(cuantasDamas) }
    @mejorCromosoma = Cromosoma.new
    @numeroDeEvaluaciones = 0
  end

  # Se ejecuta el número de generaciones que se especifique aquí
  def ejecutar(cuantasGeneraciones=1000)
    cuantasGeneraciones.times do
      cromosoma1 = seleccionarPorTorneo
      cromosoma2 = seleccionarPorTorneo
      #   cromosomaHijo = cromosoma1.cruzar(cromosoma2) # El cruce no se debe usar, porque genera cromosomas
      # inválidos
      cromosomaHijo = cromosoma1
      cromosoma1.mutar!.mutar!
      cromosoma2.mutar!
      reemplazar(cromosoma1)
      reemplazar(cromosoma2)
      reemplazar(cromosomaHijo)
    end
    self
  end
end

```

```

private

# Evalúa un Cromosoma, retornando su aptitud.
# La función de evaluación depende del problema concreto a resolver. En este caso, las NDamas, se
# puntúa negativamente cada colisión entre damas, en los dos tipos de diagonales
# En las filas y en las columnas es imposible que haya colisiones debido a la forma de codificar el Cromosoma
# Dos damas A y B están en la misma diagonal si la recta que definen tiene pendiente +1 o -1, o sea:
# (yA-yB)/(xA-xB) = +1
# ==> (yA-yB)=+(xA-xB) ==> (yA+-xA) = (yB+-xB)
def evaluar(cromosoma)
  return cromosoma.aptitud if cromosoma.aptitud
  @numeroDeEvaluaciones += 1

coordenadas = []
cromosoma.each_index do |fila|
  coordenadas[fila] = []
  coordenadas[fila][0] = fila+cromosoma[fila] if cromosoma[fila]
  coordenadas[fila][1] = fila-cromosoma[fila] if cromosoma[fila]
end
repetidos = coordenadas.inject([Hash.new(0),Hash.new(0)]) { |hash, item| hash[0][item[0]] += 1; hash[1][item[1]] += 1; hash}
conflictos = 0
repetidos.each do |r|
  conflictos += r.inject(0) { |acumulador, item| acumulador+item[1]-1 }
end

cromosoma.aptitud = -conflictos
@mejorCromosoma = cromosoma.clone if not @mejorCromosoma.aptitud or @mejorCromosoma.aptitud <
cromosoma.aptitud # Copia profunda
cromosoma.aptitud
end

# La selección por torneo elige dos Cromosomas al azar y retorna el que tenga mejor aptitud
def seleccionarPorTorneo
  cromosoma1 = sample
  cromosoma2 = sample
  if evaluar(cromosoma1) > evaluar(cromosoma2) then
    cromosoma1
  else
    cromosoma2
  end
end

# El reemplazo recibe un nuevo Cromosoma y lo inserta en un lugar al azar en la población, eliminando al
# cromosoma que estuviera allí
def reemplazar(cromosoma) # Eliminando otro al azar
  self[rand(size)] = cromosoma.clone # Copia profunda
end
end

if $0 == __FILE__
#####
# INPUTS
numeroCromosomas=200
numeroGeneraciones=100000
numeroDamas=12
#####
espacioBusqueda=1
1.upto(numeroDamas) { |i| espacioBusqueda *= i }
nDamas = AlgoritmoGeneticoNDamas.new(numeroDamas,numeroCromosomas)
nDamas.ejecutar(numeroGeneraciones)
p "===== SOLUCION PERFECTA =====" if nDamas.mejorCromosoma.aptitud == 0
p "El mejor cromosoma es #{nDamas.mejorCromosoma} que tiene #{-nDamas.mejorCromosoma.aptitud} conflictos."
p "El espacio de búsqueda es #{espacioBusqueda} y se hicieron #{nDamas.numeroDeEvaluaciones} evaluaciones
  (#{(100.0*nDamas.numeroDeEvaluaciones)/espacioBusqueda} %)"
end

```

Resumen

La evolución fue descubierta por Darwin en biología, pero puede ocurrir en cualquier otro ambiente, incluyendo el computacional, siempre y cuando se den cuatro condiciones: que haya una población de entes, que saquen copias de sí mismos (reproducción), pero que las copias no sean exactas (variabilidad) y que estén sujetos a una presión selectiva.

Los algoritmos evolutivos sirven para hacer búsqueda, optimización y diseños creativos. Son algoritmos muy simples y generales, y de allí su potencia. A cambio, son lentos. Los usaremos como última opción, es decir, cuando no se conozca ningún otro algoritmo razonable, cuando las entradas estén mal especificadas, cuando los requerimientos cambien con el tiempo o cuando el objetivo a lograr no esté muy claro.

A lo largo de los años se han diseñado varios tipos de algoritmos evolutivos, dependiendo de cuál sea el objeto que se quiera hacer evolucionar. El más sencillo es el algoritmo genético, que hace evolucionar un conjunto de datos simples como enteros, bits o *strings*. Las estrategias evolutivas hacen evolucionar un conjunto de parámetros flotantes, y el enfriamiento simulado un conjunto de estructuras de datos cualesquiera. Por otro lado, también se pueden hacer evolucionar programas de computador, y eso es lo que hace la programación genética, las gramáticas evolutivas y la programación por expresión genética. La programación evolutiva hace evolucionar máquinas de estado, que se pueden considerar un tipo limitado de programas. Y los sistemas clasificadores hacen evolucionar un sistema de reglas que se pueden ver también como datos o como programas.

Los algoritmos evolutivos se han estudiado en mayor detalle, porque muchas de sus partes se aplican a los demás algoritmos. En particular hemos visto la selección, la reproducción y el reemplazo, así como los principales problemas que suelen aparecer. Recordemos que lo importante de la selección es que sea probabilista, dando mayor prioridad a los mejores pero sin descuidar a los peores. Y que el operador de reproducción más importante es la mutación, que debe producir cambios pequeños y que, si se aplica infinitas veces, debe poder generar todos los cromosomas posibles.

También hemos visto la búsqueda multiobjetivo, usando varios criterios de dominancia de Pareto.

Hay otras variantes de los algoritmos anteriores que se muestran porque son populares: la evolución diferencial y los algoritmos híbridos de Taguchi.

Además, se explica lo que es el concepto de la coevolución, cuando evolucionan simultáneamente tanto el problema como el algoritmo que lo soluciona.

Al final se incluye el código en *Ruby* de un algoritmo genético optimizado, junto con su especificación de pruebas en *Cucumber*, aplicado a resolver el problema de las N-Damas.

Para saber más

- **Anil Menon et ál. (2004). *Frontiers of evolutionary computation*. New York: Kluwer Academic Publishers.**
Un resumen de las técnicas evolutivas, donde se hace énfasis en los aspectos a mejorar y en los que desconocemos por completo.
- **David E. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Massachusetts: Addison-Wesley Publishing Company Inc.**
Escrito por un alumno de Holland, es el primer libro que habla de algoritmos evolutivos y sus numerosas aplicaciones. En particular muestra su uso para optimizar las tuberías de distribución de gas.
- **Melanie Mitchel (1999). *An introduction to genetic algorithms*. Cambridge: The MIT Press.**
Cuenta lo básico de los algoritmos genéticos, donde se incluyen distintas variantes y ejemplos. Es de particular interés un tipo de algoritmo coevolutivo, diseñado por Hillis.
- **Richard J. Bauer Jr. (1994). *Genetic Algorithms and Investment Strategies*. New York: John Wiley & Sons Inc.**
Habla de algoritmos genéticos, caos, redes neuronales, y cómo analizar series de datos financieras en busca de patrones fractales. Explica muchos detalles sobre las estrategias de inversión. Independientemente del título, el libro no devela el secreto de cómo hacerse rico, pero sí propone ideas para investigar y está repleto de citas bibliográficas para consultar estudios más detallados.

- **Una-May O'Reilly, Tina Yu, Rick Riolo y Bill Worzel (2005). *Genetic Programming: Theory and practice II*. Boston: Springer.**
Se centra en la programación genética, es decir, generar programas de forma evolutiva. Contiene varios artículos individuales y muchos ejemplos.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Arcuri, A. y Yao, X. (2014). Co-evolutionary automatic programming for software development. *Information Sciences*, 259(20), pp. 412-432. Elsevier. DOI: <https://doi.org/10.1016/j.ins.2009.12.019>

Bentley, P. J. (1999). *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

Bongard, J. y Lipson, H. (2007). Automated reverse engineering of nonlinear dynamical systems. *PNAS*, 104(24), pp. 9943-9948. DOI: <https://doi.org/10.1073/pnas.0609476104>

Carroll, L. (1992). *Alicia en el país de las maravillas: A través del espejo*. Traducción de Ramón Buckley. Madrid: Ediciones Cátedra.

Delgado, C. A., García, Á. y Bucheli, V. A. (2017). *Perception of rankings: risks and opportunities*. Enviado a publicación.

Draves, S. (2017). *Scott Draves - Software Artist*. Recuperado el 17 de agosto de 2017. Disponible en <http://scottdraves.com/sheep.html>

Ferreira, C. (2002). Combinatorial Optimization by Gene Expression Programming: Inversion Revisited. *Proceedings of the Argentine Symposium on Artificial Intelligence*, pp. 160-174. Argentina.

Fogel, D. B. (1995). *Evolutionary Computation. Toward a New Philosophy of Machine Intelligence*. Piscataway, New Jersey: IEEE Press.

Fogel, L. J., Owens, A. J. y Walsh, M. J. (1967). *Artificial Intelligence Through Simulated Evolution*. New York: John Wiley & Sons Inc.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine*

Learning. Boston: Addison-Wesley.

Jacob, F. (1977). Evolution and Tinkering. *Science*, 196(4295), pp. 1161-1166.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, Massachusetts: MIT Press.

Koza, J. R., Bennett, F. H., Andre, D. y Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco: Morgan Kaufmann.

Latham, W. (2017). *William Latham - Software*. Recuperado el 17 de agosto de 2017. Disponible en <http://latham-mutator.com/category/software/>

Lovelock, J. (2000). *Las edades de Gaia*. Barcelona: Tusquets Editores.

Maes, P. (1996). *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*. Cambridge, Massachusetts: MIT Press.

Otten, R. H. J. M. y Van Ginneken, L. P. P. P. (1989). *The Annealing Algorithm*. Boston: Kluwer Academic Publishers.

Poorjandaghi, S. S. y Afshar, A. (2014). A Robust Evolutionary Algorithm for Large Scale Optimization. Proceedings from the 19th IFAC World Congress, pp. 7037-7042. Cape Town: Elsevier Ltd.

Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms*. [PhD Thesis]. Department of Computer Science, University of Pittsburgh.

Sorkin, G. B. (1990). Simulated Annealing on Fractals: Theoretical Analysis and Relevance for Combinatorial Optimization. In Dally J.W., 6th MIT Conference on Advanced Research in VLSI, pp. 331-351. Cambridge, Massachusetts: The MIT Press.

Spector, L. (ed.) (2001). Proceedings from the 10th Genetic and Evolutionary Computation Conference - GECCO. San Francisco, California: Morgan Kaufmann.

Torres, A. R. y García, A. (2013). *Optimización de puntajes de admisión a una carrera universitaria, el caso de Ingeniería de Sistemas*. CLEI-2013, Venezuela: IEEE.

Tsai, J., Liu, T. y Chou, J. (2004). Hybrid Taguchi-genetic algorithm for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 8(4), pp.

Whitley, L. D., Goldberg, D. E., Cantú-Paz, E., Spector, L., Parmee, I. C., y Beyer, H. G. (ed.) (2000). *Proceedings from the 9th Genetic and Evolutionary Computation Conference - GECCO*. Las Vegas, Nevada: Morgan Kaufmann.

Yang, Ch. I., Jyh-Horng Ch. y Ching-Kao Ch. (2013). Hybrid Taguchi-based genetic algorithm for flowshop scheduling problem. *International Journal of Innovative Computing, Information and Control*, 9(3), pp. 1045-1063.

PELÍCULAS Y VIDEOS

ABCUPM (2013). *Araña robótica con aprendizaje de la marcha mediante Algoritmo Genético*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=exHTTYhpevk>

ArmoredSandwich (2010). *Genetic Algorithm and Robocode*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=Hp6bhARBGc4>

Bacalov, D. (2010). *Demo de Algoritmos Genéticos*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=KdrfFFWwWiU>

CornellCCSL (2014). *Evolved Electrophysiological Soft Robots*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=HgWQ-gPlvt4>

Mann, C. (2010). *Pirouette - Evolved Virtual Creature*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=GS18h-h6IM>

OptimalEnergetics (2011). *Evolutionary Algorithms and Building Design*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=FB49qgz4BuA>

preddəqS (2013). *Genetic evolution of a wheeled vehicle with Box2d*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=uxourrlPlf8>

Seč, V. (2012). *Karl Sims - Evolving Virtual Creatures With Genetic Algorithms*. Recuperado el 2 de septiembre de 2017. Disponible en:

<https://www.youtube.com/watch?v=bBt0imn77Zg>

Virtualspecies (2012). *Genetic Programming - "Santa Fe Trail" problem.* Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=BKF7pGw8qbY>

TESIS Y TRABAJOS DE GRADO EN EVALAB

Arias, C. G. (2010). *Descubrimiento automático de equivalencias entre esquemas de bases de datos relacionales.* [Tesis Maestría]. Cali: Universidad del Valle.

Barón, R. (1999). *Planificador de horarios de clases automático.* [Tesis Maestría]. Cali: Universidad del Valle.

Barona, M. A. (2013). *Música evolutiva.* Cali: Universidad del Valle.

Cabezas, I. M. (2004). *Algoritmos genéticos híbridos distribuidos.* Cali: Universidad del Valle.

Camayo, J. M. (2009). *Software para diseño evolutivo usando Ruby orientado a web 2.0.* Cali: Universidad del Valle.

Castrillón, J. (2015). *Módulo de asignación de agendas basado en algoritmos genéticos.* Cali: Universidad del Valle.

Chaparro, J. C. (2009). *Objetos virtuales para la enseñanza de la computación evolutiva -OVACE-.* Cali: Universidad del Valle.

Cossio, O. (2011). *Objeto virtual de aprendizaje para programación por expresión genética.* Cali: Universidad del Valle.

Cruz, M. A. (2014). *Búsqueda de solapamiento en clusters, usando técnicas de computación evolutiva.* Cali: Universidad del Valle.

Fuertes, C. E. y Tigreros, Ó. I. (2017). *Análisis, Implementación y Comparativas entre un Algoritmo Genético Tradicional, el Algoritmo Híbrido Genético-Taguchi y el Algoritmo Híbrido Genético-Taguchi Cooperativo-Coevolutivo para Problemas de Optimización Numérica Global.* Cali: Universidad del Valle.

Gómez, J. A. (2001). Desarrollo de una Máquina de Estados Finitos (FSM) evolutiva mediante algoritmos genéticos. Cali: Universidad del Valle.

Guzmán, J. (2008). *Desarrollo de una aplicación para la optimización del*

aprovechamiento de telas en la actividad de trazado en la industria de la confección mediante algoritmos genéticos. [Tesis Meritoria]. Cali: Universidad del Valle.

Lourido, A. K. y Solano, C. (2004). *Desarrollo de una aplicación que colabora en la búsqueda de imágenes usando un algoritmo genético.* Cali: Universidad del Valle.

Pineda, D. L. y Estacio, I. (2003). *Desarrollo de una herramienta para la implementación de algoritmos evolutivos paralelos utilizando modelo de islas o celular.* Cali: Universidad del Valle.

Posada, J. M. (2014). *Aplicación de algoritmos evolutivos en un videojuego competitivo usando un lenguaje de muy alto nivel.* Cali: Universidad del Valle.

Posada, L. E. y Benítez, A. (2006). *Diseño e implementación de la interfaz de usuario para la interacción web con la biblioteca evolutiva.* Cali: Universidad del Valle.

Triana, J. (2013). *Desarrollo de un agente para el videojuego Tetris basado en técnicas de inteligencia artificial.* Cali: Universidad del Valle.

Villate, D. F. (2012). *Comparación de técnicas de inteligencia artificial aplicadas al juego Otelo.* Cali: Universidad del Valle.

Villegas, J. A. (2001). *Asignación de aulas empleando algoritmos genéticos paralelos.* Cali: Universidad del Valle.

TEORÍA DE JUEGOS

Como tantas otras cosas relacionadas con estos temas, la teoría de juegos fue inventada por uno de los grandes de la ciencia y la ingeniería, von Neumann (con la colaboración de Morgensten) en 1944. Aquí la palabra “juego” hay que entenderla en un sentido amplio, como una situación donde interactúan dos o más personas, cada una tratando de conseguir sus objetivos.

Entonces abarca no solamente los juegos propiamente dichos como el ajedrez o las damas, sino también:

- Juegos económicos. Fijar precios para comprar o vender bienes muebles o inmuebles. Hacer ofertas de compra o venta de acciones y divisas. Hacer préstamos y fijar los intereses. Negociar los salarios o el salario mínimo.
- Juegos políticos. Decidir el tipo de publicidad durante campañas electorales. Buscar cooperación o confrontación con otros países.
- Cualquier otra interacción, como conducir un automóvil en una calle muy transitada, decidir con tu pareja en qué barrio vivirán o a qué restaurante irán esta noche.

La teoría de juegos en muchos casos se relaciona con la evolución, ya que los animales cuyo comportamiento no esté adaptado al medio, desaparecen. Lo mismo pasa con las empresas que toman decisiones equivocadas. La teoría de juegos nos dice matemáticamente cuáles son las estrategias óptimas y, si nos desviamos de ellas, nuestro grado de supervivencia y reproducción irá disminuyendo⁴⁹.

Vamos a ver entonces una breve introducción a la teoría de juegos, lo justo para saber usar las matrices de pagos que son el mecanismo que nos ayudará a entender cómo surge la cooperación. Por otro lado, la teoría de juegos hoy día abarca otros temas interesantes como las creencias, la aversión al riesgo,

⁴⁹ Las empresas también se reproducen cuando les va bien: abren sucursales, con comportamientos, objetivos y funcionalidades muy similares a la empresa original.

objetivos borrosos o incluso distintos para cada jugador, y la psicología de los jugadores. Pues aunque nos atribuimos mucha inteligencia, los experimentos sociales muestran que somos muy irracionales (Ariely, 2008). No vamos a entrar a detallar nada de ello, pero animo al interesado a hacerlo buscando en los libros indicados en la bibliografía.

En teoría de juegos lo primero que hay que aprender es que para uno lograr sus objetivos no se puede ser ingenuo. Hay que pensar estratégicamente.

Pongamos un ejemplo clásico: las votaciones. Ana, Braulio y Cecilia conforman un club y, según el orden del día, deben votar a ver si aceptan a Yuri como nueva miembro. Sin embargo, aparece otra propuesta, que es sustituir la candidata Yuri por Zoila. Entonces han llegado a un acuerdo sobre cómo desarrollar la votación, que va a tener dos fases: primero se decidirá cuál de las dos (Yuri o Zoila) se presenta como candidata. Y después se votará a ver si se acepta esa candidata.

Supongamos que las preferencias de cada votante son las de la tabla 16.

Orden de preferencia	Ana	Braulio	Cecilia
Primero:	Yuri	nadie	Zoila
Segundo:	nadie	Yuri	Yuri
Tercero:	Zoila	Zoila	nadie

Tabla 16: Ejemplo de preferencias en una votación.

La votación tiene dos fases como se muestra en el diagrama de la figura 171. Esta manera de presentar un juego, en **forma de árbol** con todas sus posibilidades, se llama **forma extensiva**. Si todos los integrantes del club votan ingenuamente, entonces en la primera votación entre Yuri o Zoila ganará Yuri. Y en la segunda votación entre Yuri o nadie, también ganará Yuri, que será entonces admitida al club. En este diagrama se muestran todas las posibilidades y se marca en color rojo la rama que ganará sabiendo las preferencias de los votantes. Entonces, en el hipotético caso de que se vote entre Zoila y nadie, ganará nadie, por lo que se marca ese arco de color rojo.

Hay un único camino rojo que va desde el inicio hasta el final, y conduce a que la ganadora será Yuri. Entonces, con base en estas preferencias, si todos votan de forma ingenua, ganará Yuri.

Pero como todos conocen las preferencias de los demás, Braulio se dará cuenta que no tiene sentido votar contra Zoila en la primera votación, ya que si Zoila gana en la primera votación, “nadie” ganará en la segunda, que es lo que quiere

Braulio.

Además, Cecilia se puede dar cuenta de lo que Braulio intentará hacer. Sabe que Zoila no puede ganar, por lo que votará por Yuri, que lo prefiere sobre “nadie”.

Está claro que si unos votan estratégicamente y otros ingenuamente, los

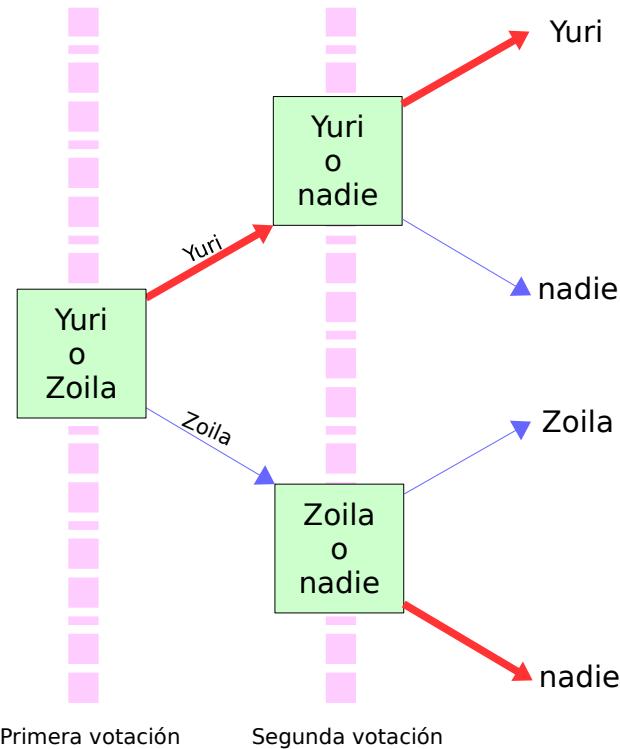


Figura 171: Árbol del juego.

resultados pueden ser distintos a los esperados.

El razonamiento de Braulio y el de Cecilia es una inducción hacia atrás, también llamado algoritmo de Zermelo.

Las votaciones son ejemplo de relaciones no-transitivas. Un sistema perfecto de votación democrática por parejas de candidatos es, en principio, imposible según demostró el premio Nobel de economía Kenneth J. Arrow. Esto es muy fácil de ver: en el ejemplo de la tabla 17, si la votación la hacemos seleccionando candidatos de dos en dos, puede salir cualquier resultado, dependiendo del orden de selección.

Orden de preferencia	Ana	Braulio	Cecilia
Primero:	Yuri	nadie	Zoila
Segundo:	Zoila	Yuri	nadie
Tercero:	nadie	Zoila	Yuri

Tabla 17: Empate en preferencias que hace que pueda ganar cualquiera.

Sin ir más lejos, en las últimas elecciones presidenciales de USA de 2016 hemos visto cómo ganó un candidato de forma inesperada debido al protocolo de votación en varias fases. Si el protocolo hubiera sido distinto (por ejemplo, con voto ciudadano directo), habría ganado el otro candidato.

Pongamos otro ejemplo con estrategias de empresas. Hay dos fábricas de helados *A* y *B*, compitiendo en el mismo mercado. Hay una gama de clientes (uniformemente repartidos), que demandan distintas proporciones de leche en el helado. Supongamos que primero *A* decide la proporción de leche de sus helados y luego lo hace *B*. Ambos quieren maximizar el número de clientes. Se supone que un cliente decide comprar helados *A* o helados *B*, según el que esté más cerca de sus gustos personales.

Aquí también se puede hacer un razonamiento hacia atrás: supongamos que *A* decide fabricar helados con 20% de leche. Entonces *B* maximiza sus clientes si fabrica al 21%, ya que *A* se queda con 20% de clientes y *B* con el 80% (figura 172).

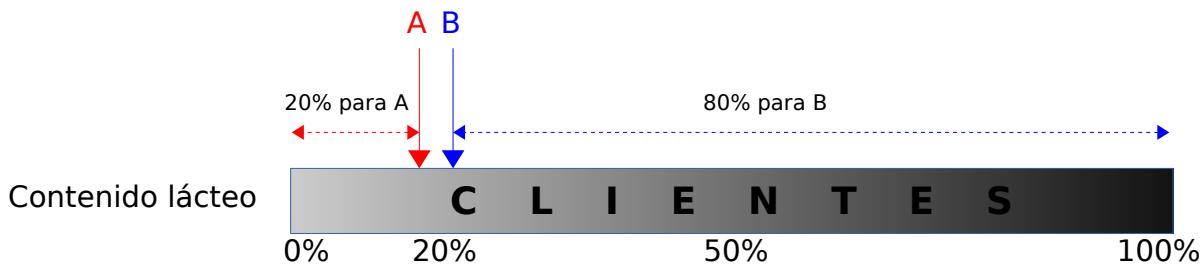


Figura 172: Dos marcas de helados.

De ahí la empresa *A* puede deducir que su óptimo es fabricar al 50% de contenido lácteo. Entonces la respuesta óptima de *B* será fabricar con un porcentaje ligeramente mayor o con uno ligeramente menor. De esta forma, se reparten el mercado, la mitad para cada uno (figura 173).

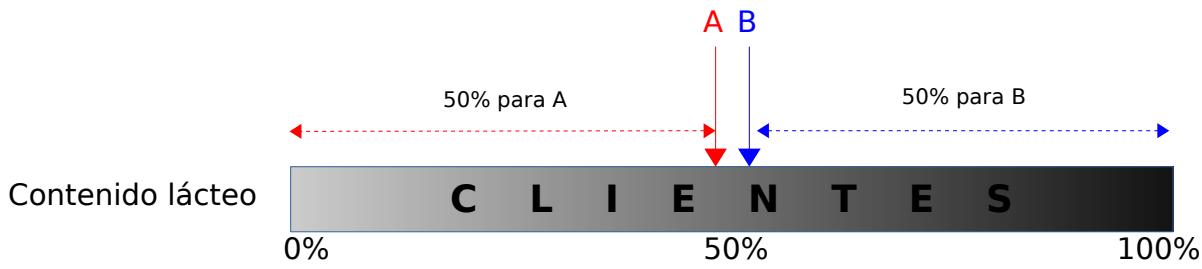


Figura 173: Dos marcas de helados con competencia óptima.

El ejemplo puede hacerse más complicado suponiendo que aparece una tercera empresa C que comienza a preparar sus fábricas. A o B corren el riesgo de perder todos sus clientes si C se posiciona justo a su lado (figura 174).

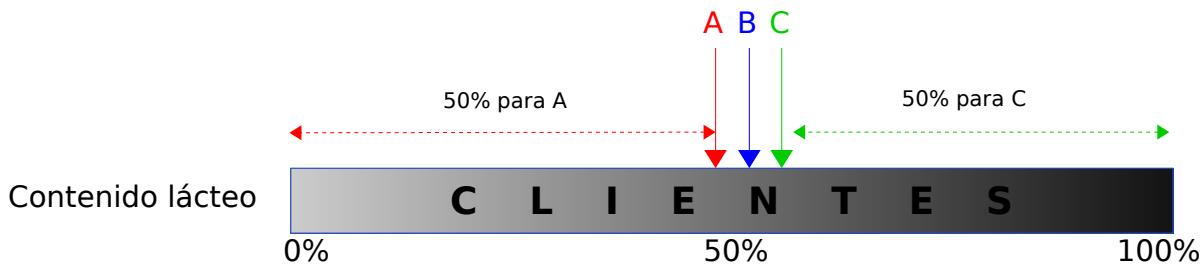


Figura 174: Aparece una tercera empresa.

Dado que ambas empresas están en riesgo de perder todos sus clientes, lo que deben hacer es cambiar su estrategia alejándose del centro (figura 175). De este modo, una empresa C que trate de introducirse, haga lo que haga, no conseguirá más del 25% del mercado.

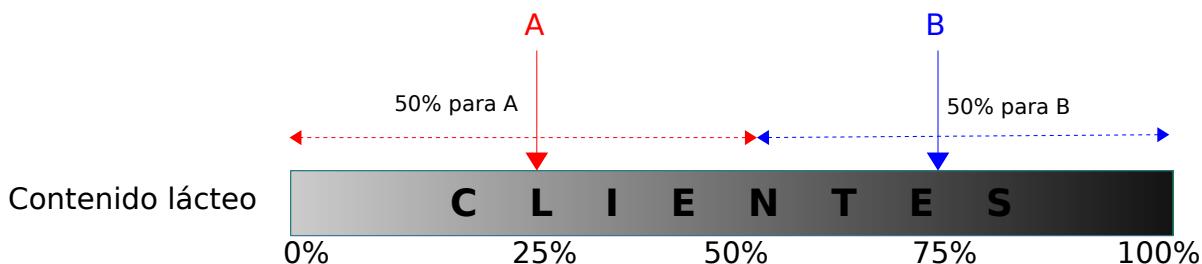


Figura 175: Nueva estrategia cuando una tercera empresa amenaza con competir.

Un tercer ejemplo es el juego del triqui, también llamado tres en raya. En la figura 176 podemos ver un árbol parcial (el árbol completo es demasiado grande para dibujarlo aquí, pues consta de $9!=362\ 880$ tableros). Se parte del tablero de 3x3 casillas vacías y se exploran todas las posibles jugadas del jugador rojo, y todas las posibles respuestas del jugador azul, y así hasta que lleguemos a un tablero donde haya un ganador (enmarcado en un círculo del respectivo color) o un empate (enmarcado con línea negra a trazos). Este árbol es similar al de la figura

177, pero mucho más grande. Y la idea es la misma: buscar si existe una forma de llegar a un tablero donde yo gane, que sea siempre alcanzable por mí, independientemente de lo que haga el otro jugador. En este juego se puede demostrar que no existe tal estrategia ganadora para ninguno de los dos jugadores, y lo máximo que pueden garantizar es empatar si ambos juegan correctamente.

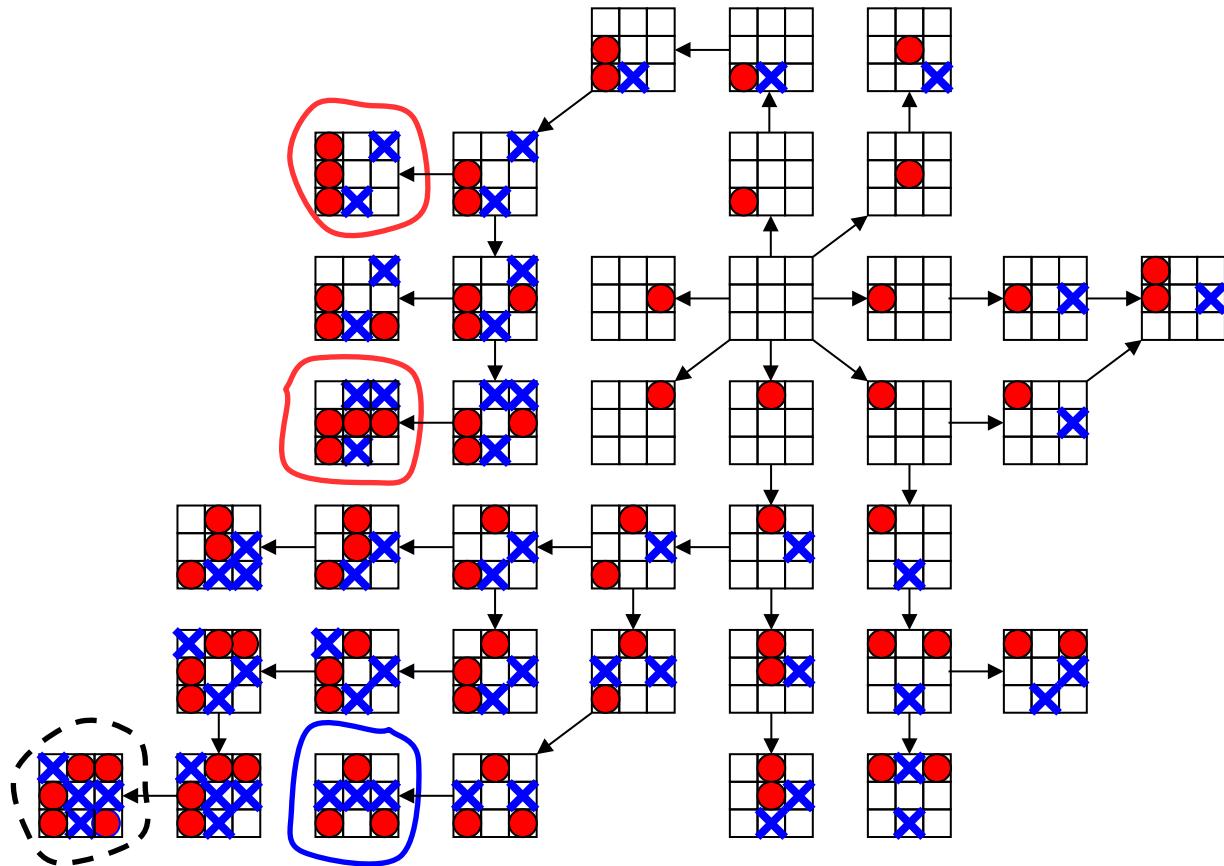


Figura 176: Árbol parcial del triqui.

Una última forma de representar un juego es por su **matriz de pagos**, que también se llama **forma normal** o **estratégica**. Esto es particularmente útil cuando son solo dos jugadores, el número de jugadas es pequeño y se conoce cuánto gana y pierde cada jugador en función del resultado del juego. Un ejemplo es el juego de niños “piedra, papel, tijera” donde hay dos jugadores que deben seleccionar simultáneamente (y sin que el otro lo vea) uno de estos tres objetos. La regla del juego es que la piedra gana a la tijera, la tijera al papel y el papel a la piedra. Entonces, su matriz de pagos viene dada en la figura 177. Los números en cada casilla indican lo que el jugador *B* debe pagar al jugador *A*.

Por ejemplo, si el jugador *A* elige piedra y el jugador *B* elige tijera, entonces *A* gana y ello implica que *B* debe pagar 1 (una moneda, un punto...) a *A*. Otro caso: si *A* elige piedra y *B* elige papel, entonces *B* debe pagar -1 a *A*, lo que se entiende

		Jugador B		
		Piedra	Papel	Tijera
Jugador A	Piedra	0	-1	1
	Papel	1	0	-1
	Tijera	-1	1	0

Figura 177: Matriz de pagos para el juego piedra, papel, tijera.

como que es A el que debe pagar 1 a B.

Antes de continuar, conviene también saber que hay dos tipos básicos de juego:

- **Estratégico** (o no-cooperativo). Cada jugador busca una estrategia óptima para sí mismo. Si hay solo dos jugadores, y lo que es bueno para un jugador es igual de malo para el otro. Se le llama juego estrictamente competitivo, o **de suma cero**, porque lo que gana uno lo pierde el otro.
- **Coalicional** (o cooperativo). Hay muchos jugadores. La teoría de juegos trata de modelar las posibles coaliciones que pueden surgir.

También se pueden clasificar así:

- **Con información perfecta.** Todos los jugadores conocen el estado pasado y actual del juego. Por ejemplo, en el ajedrez.
- **Con información imperfecta.** Justo lo contrario. Por ejemplo, el póquer debido a que cada jugador no puede ver las cartas del otro y a que el orden de las cartas del mazo depende del azar.

Entonces a partir de ahora nos vamos a centrar en juegos de dos jugadores, estratégicos y con información perfecta. En el ejemplo anterior, la matriz de pagos era de suma cero, pero en futuros ejemplos podría no ser así.

Veamos un ejemplo adicional con una matriz de pagos en un juego de suma cero que servirá para introducir el concepto de “dominancia”, muy parecido a la dominancia de Pareto que hemos visto en el capítulo de algoritmos evolutivos. El jugador A elige una jugada entre $\{A_1, A_2, A_3\}$ y el jugador B, sin haber visto lo que hace A, elige otra entre $\{B_1, B_2, B_3, B_4\}$. Hechas las elecciones, B paga a A la

cantidad indicada en la matriz de pagos de la figura 178:

		Jugador B			
		B ₁	B ₂	B ₃	B ₄
Jugador A	A ₁	2	1	10	11
	A ₂	0	-1	1	2
	A ₃	-3	-5	-1	2

Figura 178: Matriz de pagos de juego de suma cero.

Por ejemplo, si A elige A₃ mientras que B elige B₂, entonces B paga -5 a A. Es decir, A paga 5 unidades a B.

Sería genial saber de antemano cuál es la mejor jugada para cada jugador, y ello puede hacerse eliminando las filas y columnas dominadas, esto es, aquellas donde las puntuaciones que se obtienen son peores o iguales a las de otra fila o columna. B desea puntuaciones altas pues son los pagos que recibe. En ese sentido, observemos que cada una de las puntuaciones de la segunda fila son mayores o iguales a las de la tercera fila ($0 \geq -3$, $-1 \geq -5$, $1 \geq -1$ y $2 \geq 2$). Se dice que la estrategia A₂ domina a A₃, de modo que el jugador A siempre preferirá A₂ a A₃, independientemente de lo que haga el jugador B. Por eso, podemos tachar A₃.

Siguiendo el mismo método, vemos que A₁ domina a A₂, por lo que tachamos también A₂ y nos quedamos con una única jugada racional para el jugador A, que es A₁.

Por el contrario, el jugador B prefiere puntuaciones bajas (pues es lo que le corresponde pagar), y vemos que la columna B₂ domina a B₁ (ya que $1 \leq 2$, $-1 \leq 0$ y $-5 \leq -3$). Ningún jugador racional elegiría B₁, por lo que esa columna se puede eliminar.

De la misma manera vemos que B₂ domina a B₃ y que B₂ domina a B₄. La única jugada que queda para B es B₂.

Es importante entender que estas son las jugadas óptimas para cada jugador, independientemente de lo que haga el otro, y se les llama **estrategias estrictamente dominantes** (A₁ y B₂), por lo que podemos predecir que eso es lo que harán y el resultado será que B deberá pagar 1 unidad a A. También es importante entender que A tiene garantizado ganar al menos 1 unidad, y B tiene

garantizado no perder más de 1 unidad. Cualquier jugador que elija una jugada distinta a estas obtendrá menos de lo garantizado.

Entonces, dada una matriz de pagos p_{ij} siendo i el índice de las filas y j el de las columnas, en general, si el jugador A elige la fila i , sabe que ganará al menos

$$\min_j p_{ij}$$

Ec. 51

Como puede elegir cualquier i , es natural que elija el que dé valor máximo

$$\max_i \min_j p_{ij}$$

Ec. 52

De la misma manera, la mejor elección de B es la columna j que le da

$$\max_j \min_i p_{ij} = -\min_i \max_j p_{ij}$$

Ec. 53

Luego, A puede conseguir al menos

$$\max_i \min_j p_{ij}$$

Ec. 54

y B le puede impedir que gane más de

$$\min_j \max_i p_{ij}$$

Ec. 55

En el caso particular que se verifique que son iguales

$$\max_i \min_j p_{ij} = \min_j \max_i p_{ij} = p$$

Ec. 56

entonces al punto (i,j) cuyo valor es p se le llama **punto de silla**, y p es el valor del juego, o sea, lo que obtendrán si ambos juegan razonablemente (A obtiene p y B obtiene $-p$). La prueba de existencia del punto de silla es lo que se conoce como teorema del *minimax* y fue enunciado por John von Neumann en 1928. A los puntos de silla se les llama también **equilibrios de Nash**⁵⁰, pero este último concepto es más amplio pues abarca también a juegos de suma no cero y a juegos con estrategias mixtas, que veremos enseguida.

No vamos a entrar a estudiar las condiciones que impone el teorema del *minimax*, pero no siempre se dan. Por eso, en un juego puede haber más de un punto de silla pero también podría no haber ninguno y eso vuelve el juego mucho más interesante. Veamos por qué con un ejemplo (figura 179).

50 Descubierto por el matemático John Forbes Nash.

		Jugador B			
		B ₁	B ₂	B ₃	B ₄
Jugador A	A ₁	1	3	3	4
	A ₂	4	2	5	6
	A ₃	0	1	2	1
	A ₄	1	2	2	3

Figura 179: Otra matriz de pagos.

Una vez que eliminamos las filas dominadas (A_2 y A_3) y las columnas dominadas (B_2 y B_3), nos queda la matriz de pagos de la figura 180.

		Jugador B	
		B ₁	B ₂
Jugador A	A ₁	1	3
	A ₂	4	2

Figura 180: Matriz de pago sin punto de silla.

En este caso, ninguna columna domina a la otra y ninguna fila domina a la otra. Un análisis rápido diría que al jugador B le gustaría ganar esas 4 unidades para lo cual debe elegir B_2 . Pero si el jugador A adivina sus intenciones, entonces él elegirá A_2 , con lo cual las ganancias de B serían solo de 2 unidades. Un razonamiento similar se puede hacer con las preferencias de A , que pueden ser adivinadas por B .

Resulta que se vuelve esencial poder predecir las intenciones del otro. Y ello no tiene nada extraño pues en el siguiente libro veremos que la predicción es el fundamento de la inteligencia.

Este juego lo va a ganar el más inteligente, el que pueda predecir mejor lo que va a hacer el otro. Pero, a la vez, debe evitar ser predicho por el otro y, para ello, necesita tener libertad de elección. Porque si un jugador no tiene libertad de elección, si su comportamiento es mecánico, entonces se puede predecir fácilmente lo que va a hacer. Esto tiene mucha relación con el concepto de

libertad que veremos en el siguiente libro.

Si el juego se va a repetir muchas veces tiene sentido no elegir una jugada fija, sino seleccionar una al azar cada vez que se juega. A esto se le llama seguir una **estrategia mixta**, por contraposición con las **estrategias puras** que veíamos hasta ahora, donde las jugadas eran fijas sin tener que aleatorizar entre varias.

Cuando la estrategia es mixta queda por calcular con qué probabilidad elegir cada jugada, y eso es lo que vamos a ver a continuación⁵¹. Para ello supongamos que *A* elige A_1 con probabilidad r y A_2 con probabilidad $(1-r)$, mientras que *B* elige B_1 con probabilidad s y B_2 con probabilidad $(1-s)$. Entonces la esperanza matemática de *A*, o sea, lo que va a ganar *A* en promedio si se juega muchas veces, es:

$$\begin{aligned} E_A(r,s) &= -E_B(r,s) = 1rs + 3r(1-s) + 4(1-r)s + 2(1-r)(1-s) = \\ &= -4rs + r + 2s + 2 = \\ &= -4(r-1/2)(s-1/4) + 5/2 \end{aligned} \quad Ec. 57$$

En la última línea se ha hecho una factorización astuta de términos que permite calcular fácilmente el resultado. Porque si *A* hace $r=1/2$ entonces queda $E_A=5/2$, es decir, independientemente de lo que haga *B*, el jugador *A* puede garantizar que va a ganar $5/2$. De la misma manera, si *B* hace $s=1/4$ entonces $E_B=-5/2$. Es decir, independientemente de lo que haga *A*, el jugador *B* puede garantizar que solo va a perder $5/2$.

Esto es lo que se conoce como equilibrio de Nash⁵² con estrategias mixtas: cualquier jugador que se aparte de su estrategia óptima, suponiendo que el otro jugador no lo haga, y todos tengan información completa del juego, no logrará mejorar sus ganancias.

Es posible incluso que un juego tenga varios equilibrios de Nash y uno de ellos sea el óptimo global, es decir, que sea el mejor resultado para todos los jugadores⁵³. Sin embargo, si se cayó inicialmente en otro equilibrio de Nash, no se podrá salir de allí a no ser que se realice una negociación por fuera del juego para poner de acuerdo a todos los jugadores, porque las maniobras de un solo jugador se traducirán en pérdidas para él.

Otro aspecto a considerar es cómo asegurarnos que elegimos las jugadas al azar, pues hay varios experimentos que muestran lo malos que somos los humanos

51 Las estrategias mixtas también pueden existir simultáneamente con estrategias puras.

52 Antes que Nash, von Neumann ya había llegado al mismo resultado para juegos de suma cero.

53 Estamos trabajando con solo 2 jugadores porque es más didáctico, pero todo ello es aplicable a juegos de muchos jugadores, solo que el análisis es más complicado.

para generar eventos estocásticos⁵⁴. Cuando hay que usar estrategias mixtas no debemos confiar en nuestra capacidad para hacerlo, sino que es mejor lanzar monedas al aire u otro tipo de generador aleatorio. La esencia del proceso es, precisamente, que no sea posible para nuestro contrincante adivinar cómo va a caer nuestra moneda, de modo que no nos pueda predecir para sacar provecho.

Veamos otro ejemplo un poco más elaborado (figura 181-a).

		Jugador B			
		B ₁	B ₂	B ₃	B ₄
Jugador A	A ₁	1	-3	1	4
	A ₂	5	-3	2	3
	A ₃	1	1	2	2
	A ₄	6	5	1	3

(a)

		Jugador B			
		B ₁	B ₂	B ₃	B ₄
Jugador A	A ₁	1	-3	1	4
	A ₂	5	-3	2	3
	A ₃	1	1	2	2
	A ₄	6	5	1	3

(b)

		Jugador B	
		B ₁	B ₄
Jugador A	A ₁	1	4
	A ₂	5	3
	A ₃	1	2
	A ₄	6	3

(c)

		Jugador B	
		s	1-s
Jugador A	A ₁	1	4
	A ₄	6	3

(d)

Figura 181: Otra matriz de pagos más complicada.

Aquí B_1 domina a B_2 y B_4 domina a B_3 por lo que podemos eliminar las columnas B_2 y B_3 (figura 181-b). Y entonces A_4 domina a A_3 y a A_2 , por lo que podemos eliminarlas también (figura 181-c). Nos quedamos entonces con una matriz de pagos de 2×2 , sin filas ni columnas dominadas (figura 181-d) en la que no hay puntos de silla, por lo que no hay estrategias fijas ganadoras. No obstante, se puede averiguar cuáles son las estrategias mixtas para cada jugador, asignando probabilidad r a jugar A_1 y probabilidad $(1-r)$ a jugar A_4 así como probabilidad s a

⁵⁴ Una anécdota: Claude Shannon creó un sencillo predictor de secuencias con una máquina de 8 estados y retó a un amigo suyo matemático a probarlo. El amigo debería generar una secuencia de ceros y unos, mientras que el software de Shannon trataba de predecirla. El software ganó, porque los humanos tendemos a repetir patrones. A continuación, Shannon mostró a su amigo cómo estaba hecho el software y después volvieron a jugar. El software volvió a ganar.

jugar B_1 y probabilidad $(1-s)$ a jugar B_4 . Entonces, si el juego se repite muchas veces, la esperanza matemática de A es:

$$\begin{aligned} E_A(r,s) &= -E_B(r,s) = 1rs + 6(1-r)s + 4r(1-s) + 3(1-r)(1-s) = \\ &= rs + 6s - 6rs + 4r - 4rs + 3 - 3r - 3s + 3rs = \\ &= -6rs + r + 3s + 3 \end{aligned} \quad Ec. 58$$

Se desea factorizar de la forma:

$$-6rs + r + 3s + 3 = a(r-b)(s-c) + d \quad Ec. 59$$

que, desarrollando, queda:

$$a(r-b)(s-c) + d = ars - acr - abs + abc + d \quad Ec. 60$$

Identificando términos, queda:

$$\begin{aligned} a &= -6 \\ -ac &= 1 \\ -ab &= 3 \\ abc + d &= 3 \end{aligned} \quad Ec. 61$$

De donde sale:

$$\begin{aligned} a &= -6 \\ b &= 1/2 \\ c &= 1/6 \\ d &= 7/2 \end{aligned} \quad Ec. 62$$

Insertando estos valores en la ecuación 59, queda:

$$E_A(r,s) = -E_B(r,s) = -6(r-1/2)(s-1/6) + 7/2 \quad Ec. 63$$

Entonces el jugador A puede garantizar que sus ganancias serán al menos $7/2$ si hace $r=1/2$. Y el jugador B puede garantizar que sus pérdidas serán a lo sumo $7/2$ si hace $s=1/6$.

Como conclusión, la estrategia mixta de A es jugar A_1 con probabilidad $1/2$ y A_4 con probabilidad $1/2$, mientras que la estrategia mixta de B es jugar B_1 con probabilidad $1/6$ y B_4 con probabilidad $5/6$.

A continuación explicaremos una nueva característica de la teoría de juegos. Si el juego es de suma cero, lo que un jugador gana es lo que el otro pierde. Pero si no es de suma cero, entonces en cada casilla de la matriz hay que especificar lo que

cada jugador gana. Veámoslo con un ejemplo (figura 182): aquí, si el jugador A elige A_3 y el jugador B elige B_2 , entonces el pago será $(1, 4)$ que significa que A ganará 1 y B ganará 4.

		Jugador B			
		B_1	B_2	B_3	B_4
Jugador A	A_1	1, 2	-3, -4	1, 0	4, 3
	A_2	5, 1	-3, 7	2, -1	-5, -3
	A_3	1, 0	1, 4	2, 5	1, 0
	A_4	6, 3	5, 1	1, 2	3, 1

Figura 182: Juego de suma no cero con tres equilibrios de Nash señalados en color verde.

Ahora ya no podemos hablar de puntos de silla sino solo de equilibrios de Nash, y el algoritmo para encontrarlos es muy simple: para cada columna se busca el pago máximo para el jugador A y se verifica si el correspondiente pago para el jugador B es el máximo de esa fila. En caso de que lo sea, en esa casilla hay un equilibrio de Nash. En la figura anterior podemos observar que la matriz de pagos tiene tres equilibrios de Nash. Por ejemplo, (A_4, B_1) con pago $(6, 3)$ es equilibrio de Nash porque $\max_{\text{COLUMNA}1} \{1, 5, 1, 6\} = 6$ y $\max_{\text{FILA}4} \{3, 1, 2, 1\} = 3$.

Sin embargo, en la columna 2 no hay ningún equilibrio de Nash porque el pago máximo para el jugador A es $\max_{\text{COLUMNA}2} \{-3, -3, 1, 5\} = 5$ que se da en la casilla (A_4, B_2) pero en ella el jugador B gana 1 que no es el $\max_{\text{FILA}2} \{3, 1, 2, 1\} = 3$.

Los equilibrios de Nash se alcanzan en juegos de información perfecta donde cada jugador puede calcular cuáles son sus preferencias, y sabe que esos cálculos los estarán haciendo también los demás jugadores. Un jugador B muy primitivo podría querer jugar B_2 con la esperanza de ganar 7 puntos si A jugase A_2 . Pero eso no va a ocurrir. Mientras que un jugador inteligente sabe que a A no le interesa jugar A_2 , de modo que se centraría únicamente en los equilibrios de Nash.

De los tres equilibrios de Nash, ¿cuál es el que se elegirá? Todo va a depender de si hay comunicación entre los jugadores, si pueden negociar por fuera del juego, o si uno puede coaccionar al otro. Si el juego se repite muchas veces y no hay comunicación (por ejemplo, si el juego se juega entre bacterias o entre humanos con distinto idioma) entonces se llegará a algún equilibrio de Nash al azar. Y una vez allí, a nadie le conviene cambiar su jugada porque saldrá perdiendo. Pero si

hay comunicación y posibilidad de negociar, a A le conviene (A_4, B_1) mientras que a B le conviene (A_3, B_3) y quizás lleguen al acuerdo de jugar (A_4, B_1) porque la ganancia total allí ($6+3=9$) es mayor que en el otro ($2+5=7$). Incluso quizás convenga a A pagar algo a B por fuera del juego, para llegar a ese acuerdo. ¿Cuánto? Ese es otro juego que también podría analizarse.

Otro concepto todavía más interesante son las **estrategias evolutivamente estables**. Resulta que en sistemas evolutivos con una gran población de seres vivos compitiendo entre sí, los jugadores no se comunican ni negocian entre ellos, ni son conscientes de las alternativas óptimas de los otros. Además, habitualmente los genes imponen estrategias fijas a sus portadores. En este ambiente, las mutaciones producen, por así decir, los cambios de estrategia de cada jugador. Y sucede exactamente lo mismo que antes: puede haber varios óptimos, y el sistema quedará atrapado en cualquiera de ellos. En este ámbito biológico, algunos equilibrios de Nash son también estrategias evolutivamente estables. Es decir, son estrategias que —una vez adoptadas por la mayoría de la población— no se dejan invadir por otras estrategias mutantes. La selección natural mantiene la estabilidad. Este concepto fue acuñado por John Maynard Smith y George Price en 1973. Todas las estrategias evolutivamente estables son equilibrios de Nash, pero no al revés. La pequeña diferencia está en que a las estrategias evolutivamente estables se les exige también que el pago que recibe un jugador cuando el otro cambia de estrategia siga siendo mayor que cuando ambos cambian.

Cuando un juego se juega una sola vez, tiene sentido buscar los equilibrios de Nash. Pero estos no necesariamente son estables. Por tanto, si el juego se juega muchas veces, las jugadas óptimas vendrán dadas por las estrategias evolutivamente estables (ESS_Wiki, 2017).

Y cuando el juego es evolutivo, es decir, con poblaciones de agentes que difieren en pequeñas mutaciones, se suele plantear que todos los agentes son similares y cada estrategia corresponde a una mutación. En este caso, la matriz de pagos es simétrica. Veamos un ejemplo en la figura 183, donde podemos observar que hay dos equilibrios de Nash, marcados en color verde.

		Jugador con mutaciones			
		M₁	M₂	M₃	M₄
Jugador con mutaciones	M₁	1, 1	7, 5	0, 1	3, 6
	M₂	5, 7	7, 7	5, 1	1, 5
	M₃	1, 0	1, 5	5, 5	3, 2
	M₄	6, 3	5, 1	2, 3	2, 2

Figura 183: Matriz de pagos con una población de jugadores, jugando de 2 en 2.

Los equilibrios de Nash se pueden definir también como aquellas estrategias M_N tales que:

$$E(M_N, M_N) \geq E(M_N, M_X) \text{ siendo } M_X \text{ cualquier otra estrategia} \quad \text{Ec. 64}$$

En este caso, M_2 es un equilibrio de Nash porque lo que gana la estrategia M_2 contra sí misma es 7, y si por mutaciones aparece otra estrategia M_x , la nueva ganaría menos de 7, por lo que el proceso de selección de la evolución la rechazaría rápidamente.

De la misma forma, si en la población solo existiera la estrategia M_3 , entonces se cumple que $E(M_3, M_3) \geq E(M_3, M_x)$, es decir, lo que gana al jugar contra sí misma es 5. Por otra parte, si aparece alguna mutación será rechazada porque la nueva ganaría menos de 5.

Para que una estrategia M_E sea evolutivamente estable debe cumplirse una de estas dos condiciones:

$$\begin{aligned} & E(M_E, M_E) > E(M_E, M_X) \\ & \text{o que:} \\ & E(M_E, M_E) = E(M_E, M_X) \text{ y } E(M_E, M_X) > E(M_X, M_X) \end{aligned} \quad \text{Ec. 65}$$

siendo M_X cualquier otra estrategia

La estrategia M_3 no es evolutivamente estable porque si aparece por mutación M_2 , entonces M_3 gana lo mismo (5) jugando contra sí misma que contra M_2 . Pero M_2 gana mucho más jugando contra sí misma (7) que contra M_3 (1). Por tanto, la mutación M_2 invadirá la población.

A su vez, M_2 sí es evolutivamente estable, es decir, no se deja invadir por otras

mutaciones.

Como toda herramienta matemática, la teoría de juegos con sus matrices de pagos sirve para analizar situaciones con rigurosidad, eliminando las ambigüedades del lenguaje hablado. En el problema 2 vemos una situación de este tipo.

Problema 2: DINERO DE BOLSILLO

Jorgito y Luisita apuestan a ver quién lleva menos dinero en su billetera. El ganador (el que lleve menos), se quedará con lo que lleve el otro. Si hay empate nadie se lleva nada del otro.

Parece algo sencillo y es de esperar que Jorgito razoné así: *"si yo tuviera más que Luisita, ella ganará todo lo mío. En cambio, si ella ha traído más que yo, ganaré más de lo que tengo. Es decir, lo que puedo perder es menos de lo que puedo ganar. ¡Yo llevo ventaja en este juego!"*.

Luisita razona de manera análoga.

Pero entonces, ¿cómo pueden ambos jugadores llevar ventaja?

Sin embargo, también puede ocurrir al revés. Hay muchas situaciones aparentemente sencillas que se pueden modelar como juegos con su matriz de pagos, y que al final generan resultados inesperados, como un tipo de emergencia muy básico, donde interactúan pocos agentes, posiblemente solo dos. Vamos a ver a continuación algunos ejemplos.

Dilema del prisionero

En la figura figura 184 vemos la matriz de pagos del famoso dilema del prisionero, inventado por Dresher y Flood en 1950. La historia es la siguiente: la policía captura dos presuntos ladrones de un banco y los mete en celdas separadas, incomunicados entre sí. A cada uno le promete que lo dejará libre si confiesa el robo que acaban de cometer, y al otro le darán cinco años de cárcel si calla. Si ninguno confiesa, hay pruebas circunstanciales que permitirían meter a la cárcel a ambos durante dos años. Si los dos confiesan, entonces la policía no tendría problema en encarcelarlos durante cuatro años a cada uno. En la figura, "confesar" es "traicionar" al otro, y "callar" es "cooperar" con el otro.

		Jugador B	
		Cooperar	Traicionar
Jugador A	Cooperar	-2, -2	-5, 0
	Traicionar	0, -5	-4, -4

Figura 184: Matriz de pagos del dilema del prisionero.

El análisis es sencillo porque la fila con la jugada “traicionar” domina a “cooperar” para el jugador A, es decir, $0 \geq -2$ y $-4 \geq -5$. Además, la columna con la jugada “traicionar” también domina a “cooperar” para el jugador B, es decir, $0 \geq -2$ y $-4 \geq -5$, como vemos en la figura 185. Entonces las jugadas racionales para ambos jugadores son traicionar al otro y son estrategias estrictamente dominantes, por lo que conviene seguir las independientemente de lo que haga el otro.

Recordemos que en teoría de juegos se supone que cada jugador busca egoístamente lo mejor para él mismo, y que la suerte del otro le da igual. No hay odio para el otro, pero tampoco hay ningún tipo de consideración.

		Jugador B	
		Cooperar	Traicionar
Jugador A	Cooperar	-2, -2	-5, 0
	Traicionar	0, -5	-4, -4

Figura 185: Las jugadas dominantes están en amarillo.

Aquí se entiende la razón del nombre “dilema”, ya que lo que ambos van a obtener es una pena bastante alta, de 4 años de cárcel para cada uno. El óptimo global está en la casilla (-2, -2) donde ambos cooperan, pero como no pueden comunicarse ni hacer tratos o promesas entre ellos, la buena intención de cooperar de uno puede llevarle a algo peor, 5 años de cárcel, si el otro decide traicionar. Y ambos lo saben. De modo que ese óptimo global es inalcanzable. El egoísmo se impone y el resultado es malo para ambos.

Como es molesto trabajar con números negativos, se le puede sumar una constante a todos los pagos y el juego sigue siendo el dilema del prisionero. Esto es lo que hemos hecho en la figura 186.

		Jugador B	
		Cooperar	Traicionar
Jugador A	Cooperar	3, 3	0, 5
	Traicionar	5, 0	1, 1

Figura 186: Dilema del prisionero con todos los pagos positivos.

De hecho, el juego es un dilema del prisionero siempre que tengamos una matriz como la de la figura 187 donde se cumpla:

$$\begin{aligned} T &> R > P > S \\ \frac{(T+S)}{2} &< R \end{aligned}$$

Ec. 66

La segunda desigualdad evita que los jugadores se pongan de acuerdo en alternarse en traicionar y cooperar, y así ganar más que cooperando, cuando el juego se juega muchas veces.

		Jugador B	
		Cooperar	Traicionar
Jugador A	Cooperar	R, R	S, T
	Traicionar	T, S	P, P

Figura 187: Dilema del prisionero generalizado.

En 1740, el filósofo David Hume adelantaba que incluso el individuo más egoísta puede descubrir que cooperar es la mejor solución a largo plazo. Pero ¿cómo lograrlo con esta matriz de pagos tan adversa? Pues resulta que si permitimos que los dos prisioneros se comuniquen es trivial que lleguen a un acuerdo de cooperar con lo cual ambos saldrían ganando. Pero si no pueden comunicarse, o no pueden firmar un contrato vinculante sobre el acuerdo que desean llevar a cabo, las matemáticas son inexorables y la única posibilidad para ambos es traicionarse.

En un programa de televisión donde se jugaba *Split or Steal*, un juego parecido al dilema del prisionero, podemos observar un ejemplo de negociación y engaño

bastante espectacular (Spinout3, 2012).

Tragedia de los comunes

Este dilema fue ideado por William Foster Lloyd en 1833 para rebatir una conjetura del economista Adam Smith, y fue retomado por Garrett Hardin en 1968. Se cuenta así: unos campesinos pueden llevar cada uno tantas vacas como quieran a pastar a un prado comunal. Cada uno maximiza su ganancia llevando cada vez más y más vacas. Pero el resultado final es que el prado compartido acaba destrozado por sobreuso.

La conclusión a la que llegan es que los recursos comunes deberían ser administrados por una autoridad central.

Paradoja de Braess

Surge cuando hay un recurso común cuyo costo de uso se incrementa conforme aumenta el número de individuos que lo utilizan. Inicialmente, a cada individuo se le asigna un recurso al azar. Después, cada individuo puede cambiar al recurso que le es más barato, y entonces el costo para cada individuo, paradójicamente, puede aumentar. Esto sucede, por ejemplo, cuando modelamos el tráfico en una carretera o en Internet, como en la figura 188: hay agentes entrando a una red

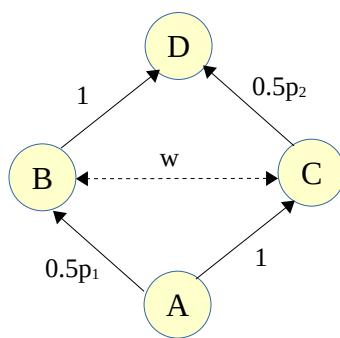


Figura 188: Circuito para la paradoja de Braess.

por el punto A, y el objetivo es salir por D eligiendo libremente la ruta. Cada tramo de la ruta es un recurso que tiene un costo.

Hay rutas de costo constante, como AC y BD, y rutas de costo proporcional al

porcentaje p de agentes que viajan por ellas.

Si la ruta BC no existiera, los costos de ABD y ACD serían iguales, por lo que la mitad de los agentes viajaría por cada ruta.

Parece lógico que al aumentar el número de rutas el costo total debe bajar, pero no es así: al añadir la ruta BC con costo $1/4 < w < 1/2$, todos los agentes elegirán egoístamente la ruta $ABCD$, cuyo costo por agente es $(1+w)$, que es mayor que 1.25 (el costo cuando no existía BC).

Este problema no es solo teórico. En libros como el de Vanderbilt (2010) están documentados varios casos reales: al aumentar el número de autopistas a veces se fomenta la demanda, con lo que empeora la congestión.

Paradoja del votante

Cuando hay que participar en una votación, si acudes a votar te beneficias por el resultado de la votación. Pero si no acudes —dado que la población de votantes es muy grande— tu voto apenas cuenta, y entonces te beneficias más, pues el resultado no cambia pero no has gastado tu tiempo. Dicho con otras palabras: si muchos votan como yo, mi voto es superfluo, mientras que si pocos votan como yo, mi voto no vale para nada. Mi voto tiene importancia únicamente en caso de empate, pero la probabilidad de ello es insignificante. El problema de este tipo de razonamientos es que si todo el mundo hace lo mismo el sistema de votaciones deja de funcionar.

Otra paradoja ocurre en las votaciones con varias opciones X , Y , Z . Vamos a suponer que mis preferencias son $X > Y > Z$. Es habitual tener pensamientos como: “quisiera votar por X , pero como no va a ganar mi voto será inútil. Por ello, prefiero votar por Y , para que no gane Z ”. Esto se denomina “voto útil” pero en ocasiones tiene como consecuencia que si muchos piensan de ese modo, X perderá misteriosamente.

Señalización

En el caso del dilema del prisionero, y en muchos otros, sería útil poder indicar nuestras intenciones al otro jugador. A veces se puede hacer de forma verbal, o firmando contratos. Aunque a veces no, bien porque la comunicación no sea fiable

si los agentes pueden mentir, bien porque ni siquiera haya comunicación, como en el caso del dilema del prisionero o en el caso de que los agentes sean animales o plantas sin posibilidad de comunicación verbal.

En muchos casos, para que surja la cooperación o para que cada uno encuentre su estrategia óptima, es necesario que los individuos sean capaces de reconocerse unos a otros o, más bien, de reconocer las intenciones de los demás y de dar a conocer las suyas.

A ese lenguaje no verbal que permite comunicar las intenciones de un agente a los otros se le conoce en biología como señalización. En teoría de juegos se mantiene tanto el término como su significado. Por ejemplo, en biología encontramos aves, como el pavo real, que desarrollan plumas de muchos colores para atraer pareja, indicando lo buenos padres que van a ser. Otros animales, entre ellos las ranas, desarrollan vivos colores para indicar a sus predadores que son venenosos. También hay crías de aves que pían mucho cuando tienen hambre. En los humanos ocurre la señalización cuando empleamos vestidos o adornos costosos para indicar nuestro estatus social.

La señalización es un sistema rudimentario de comunicación. Sin embargo, hay algo muy importante en ello: para que tenga éxito, para que sea creíble y no parezca el resultado de una técnica de engaño, quien señala debe gastar mucha energía e incluso poner en riesgo su vida. La señalización solo funciona si el mensaje que se envía es más valioso que el costo de falsificarlo.

Por ejemplo, cuando una cría de pájaro pía sin tener hambre, atrae también la atención de predadores, poniendo en riesgo su vida sin obtener a cambio ningún beneficio. Mientras que la cría que pía para llamar la atención de sus padres porque realmente tiene hambre, sí obtiene como beneficio la comida, es decir, evita la muerte por inanición. Eso significa que los genes de “piar sin tener hambre” van a encontrarse en menor proporción en la población de pájaros respecto a los genes de “piar teniendo hambre”. Al generar un costo mayor en los individuos que mienten, la evolución va a presionar para que el mensaje diga la verdad.

En el caso del pavo real macho, las plumas llamativas y lo aparatoso de la cola también atraen a los predadores, por lo que el mensaje que envía a las hembras es algo así como “soy muy fuerte gracias a que tengo genes muy buenos, y si no fuera así, los predadores ya me habrían devorado”. El mensaje es cierto porque el costo de engañar es alto. Los genes que producen colas llamativas y animales débiles dejaron de existir hace mucho.

Conviene aclarar que la señalización para mejorar la supervivencia, como la de las crías de pájaro, es un juego de dos (predador y presa), fácil de analizar. Por otro lado, la señalización cuyo objetivo es mejorar la reproducción, como el pavo real, es un juego de tres (macho, hembra y predador), bastante más complicado de analizar y en el que todavía quedan detalles en los que los biólogos no se ponen de acuerdo.

Las gacelas, que viven en manadas y dan grandes saltos sin moverse del sitio cuando ven un predador, están señalizando su fuerza para que el predador no las persiga a ellas y se centre en otras más débiles que no puedan saltar. En este caso la señalización es imposible de falsificar.

Sin embargo, realmente solo podemos hablar de probabilidades. Es poco probable que mensajes de señalización falsos prosperen. Pero entra dentro de lo posible — e incluso puede ser que matemática y biológicamente no haya otra alternativa—. Es lo que ocurre con las bandadas de pájaros (Dawkins, 1994) en las que algunos de ellos se quedan como centinelas en los árboles, encargados de alertar de peligros mientras los demás bajan al suelo a comer lombrices. El mensaje de señalización es el graznido que emiten cuando ven algún peligro. Si el mensaje fuera siempre falso, la bandada habría dejado de existir, devorada por predadores. Pero si siempre fuera cierto, los pájaros que se quedan como centinelas se morirían de hambre. La evolución encontró el equilibrio cuando estos animales emiten un 15% de sus avisos falsos.

Las serpientes coral son venenosas y tienen unos vivos anillos de color con los que alertan a sus predadores para que las dejen tranquilas. Los predadores que no tienen genes para entenderlo así morirán con bastante probabilidad, eliminando esos genes (que no relacionan el color con el peligro) de la población. Aquí se puede ver que la señalización es un protolenguaje, con un emisor y un receptor que se han puesto de acuerdo en el contenido (veneno), a lo largo de la evolución por generaciones, mientras que las formas son irrelevantes (color), producto de algún accidente.

Con el tiempo, otras serpientes logran desarrollar anillos de colores similares sin tener que invertir en la fabricación más costosa del veneno. Son las falsas corales, que han conseguido falsificar ese tipo de señalización porque el costo de hacerlo era favorable (protegerse de depredadores).

Es decir, que asociado a una señalización también suele surgir después el engaño. Y si el engaño se generaliza, se hace fácil de realizar y se extiende a toda una población, entonces la señalización se destruye.

Se piensa que los lenguajes humanos también comenzaron como formas de señalización.

En economía se han planteado varios escenarios donde la señalización es útil. Uno de ellos es el juego de la reputación que ilustraremos detalladamente con el siguiente ejemplo: el jugador A es un vendedor que monopoliza su sector y que gana así 7 millones. El jugador B amenaza con entrar al mercado a vender lo mismo, pero sabe que gana 1 millón si no entra. Si B decide entrar, el jugador A puede hacer dos cosas:

- Inundar el mercado con sus productos de modo que ambos ganen 0.
- Repartirse a medias un mercado que va a resentirse bastante. Concretamente, A ganará 2 y B ganará 2.

Podemos ver el juego en su forma extensa en la figura 189. Se trata de un juego no competitivo, por lo que la puntuación de cada jugador es independiente de la del otro. A ambos solo les interesa maximizar la propia puntuación.

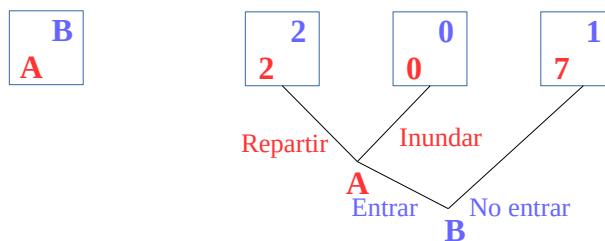


Figura 189: Juego de la reputación en forma extensa.

En la figura 190 lo hemos convertido a su forma normal.

Está claro que las puntuaciones de “repartir” dominan a las de “inundar” para el jugador A. Por tanto, el jugador B no percibe “inundar” como una seria amenaza y lo racional es que se arriesgue a jugar “entrar”.

		Jugador B	
		No entrar	Entrar
Jugador A	Repartir	1 7	2 2
	Inundar	1 7	0 0

Figura 190: Matriz de pagos del juego de la reputación.

Sin embargo, el jugador A puede señalizar su intención irrevocable de jugar “inundar” haciendo un fuerte gasto previo al juego, o sea, añadiendo una nueva jugada. Lo puede lograr haciendo una inversión irreversible para aumentar su capacidad de producción. Esto conlleva para A una pérdida de 2 millones que se restan sobre lo previsto en el caso anterior si no utiliza esta capacidad extra. La única oportunidad de utilizar la capacidad extra es luchar contra B si se decide a entrar, en cuyo caso ganará 1 millón, en vez de 0, porque la capacidad extra abaratará su producción que inundará el mercado. Las ganancias de B no cambian. El nuevo árbol de decisiones lo vemos en la figura 191.

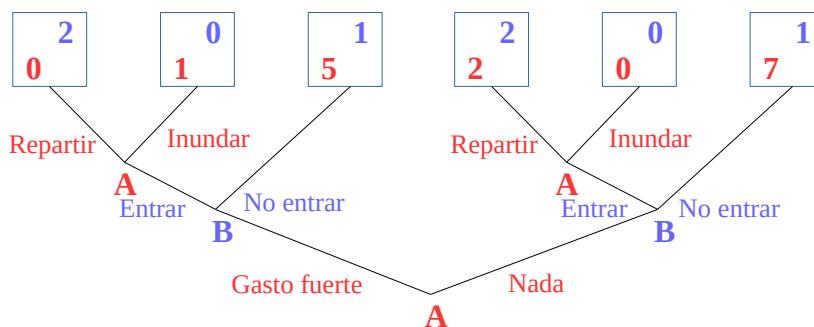


Figura 191: Señalización en el juego de la reputación, en forma extendida.

La matriz de pagos queda entonces como se indica en la figura 192. Si el jugador A hace el gasto fuerte, es decir, si emite la señalización, el resto del juego se desarrollará en la mitad de abajo de la matriz de pagos. Entonces, la fila “gasto fuerte e inundar” es dominante para el jugador A, por lo que esa será su jugada racional, y el jugador B lo sabe. Ante esas condiciones, la jugada racional de B es “no entrar”.

		Jugador B	
		No entrar	Entrar
Jugador A	Nada y Repartir	1 7	2 2
	Nada e Inundar	1 7	0 0
	Gasto fuerte y Repartir	1 5	2 0
	Gasto fuerte e Inundar	1 5	0 1

EN EL CASO DE QUE HAYA SEÑALIZACIÓN

Figura 192: Matriz de pagos del juego de la reputación con señalización.

Ahora podemos entender con cifras concretas lo que significa que una señalización sea creíble. Si el jugador A se limita a anunciar en los periódicos que

va a inundar los mercados, pero sin realizar ninguna inversión previa, eso no es creíble pues nos lleva de nuevo a la figura 190.

Resumen

Hemos visto que la teoría de juegos se encarga de analizar matemáticamente las interacciones entre distintos agentes (humanos, animales, software), buscando la jugada óptima de cada uno. Se abordaron varias clasificaciones y cómo expresar los juegos en forma de árbol o en forma de matriz de pagos. En este último caso se pueden descartar las jugadas dominadas de cada jugador y si ello conduce a una única jugada, se la llama estrategia estrictamente dominante donde se gana más con ella, independientemente de lo que haga el otro jugador. En el dilema del prisionero, cada jugador tiene una estrategia estrictamente dominante, que es traicionar al otro. Cuando esto ocurre, se dice que es una estrategia pura. Pero en el caso de que al eliminar todas las jugadas dominadas queden varias (dos o más), entonces habrá que elegir al azar entre ellas, con una probabilidad para cada jugada que puede ser calculada fácilmente. Aleatorizar entre varias jugadas es lo que se conoce como estrategia mixta.

Luego definimos los equilibrios de Nash como aquellas jugadas donde a cada jugador no le conviene cambiar porque saldría perdiendo si todos los demás jugadores se mantienen en su jugada. Son, por así decir, óptimos locales. Solo se puede salir de ellos si los jugadores se ponen de acuerdo en cambiar de jugada todos a la vez, lo cual requiere procesos de negociación externos al juego.

Y también vimos las estrategias evolutivamente estables, cuya definición es casi idéntica a los equilibrios de Nash. Todas estas definiciones son muy parecidas e incluso se han ido afinando bastante recientemente, por lo que aún persisten algunos errores en ciertos medios (Talwalkar, 2016).

Después se presenta el dilema del prisionero como un juego aparentemente competitivo y que lleva inevitablemente a la traición, cuando se juega una sola vez o cuando no hay suficiente inteligencia en los jugadores. Pero si se juega muchas veces y los jugadores disponen de un mínimo de inteligencia (memoria para reconocer al adversario y las jugadas pasadas) entonces emerge la cooperación que, aunque no es estable en todas las circunstancias, sí es bastante robusta. Este ambiente inteligente es el que suele darse cuando hay evolución, por lo que la cooperación requiere aproximadamente la misma complejidad que la evolución.

Para terminar se muestran otras paradojas y cómo emerge la comunicación en su forma más primitiva, es decir, la señalización. Y con ella, el engaño.

Para saber más

- **Robert Wright (2001). *Non Zero. The logic of human destiny.* New York: Vintage Books.**

Un libro algo especulativo acerca del destino que le espera a la humanidad. Sin embargo, lo recomiendo porque muestra muchos ejemplos reales de juegos de suma no-cero. La tesis del autor es que a través de estos juegos la humanidad evolucionó mejorándose cada día más. La mayoría de los juegos sociales en que intervenimos son de suma no cero, es decir, todos los jugadores pueden ganar a la vez. Solo con ello ya estaría asegurada la mejora continua de la sociedad. Lo malo es que también hay juegos de suma cero (como repartirse una cosecha, pues lo que uno gana es lo que otro pierde), pero para evitar problemas la misma sociedad ha desarrollado herramientas que empujan a la cooperación o, al menos, evitan el abuso. Estas herramientas son las leyes, las religiones y las costumbres.

Además explica las mayores transiciones, desde las tribus de la edad de piedra, el cacigazgo, los reinos medievales y los estados actuales (quién sabe qué venga después). Muchas de estas transiciones fueron apoyadas por descubrimientos o invenciones tecnológicas, como la agricultura (con la que comienza el capitalismo, pues es posible acumular algo precioso), la escritura (que permite guardar la historia, escribir contratos o no perder futuros hallazgos científicos, artísticos o tecnológicos, lo que da lugar a los sistemas educativos modernos) en varias fases (primitiva sobre una variedad de materiales, la imprenta automatizada y los medios electrónicos que culminan con Internet) y la revolución industrial, seguida de la revolución de la información.

- **Mark van Vugt y Anjana Ahuja (2012). *Naturalmente seleccionados.* São Paulo: Editora Cultrix.**

Algunos temas del libro anterior se explican aquí desde otra óptica: no debe haber genes de liderazgo (ya que ello implica tomar riesgos) sino de “seguidazgo” que son más útiles (repetir lo que otros han hecho con éxito). Las pocas personas que tienen versiones débiles de estos genes son los

más propensos a ser líderes. Además, el texto muestra las características, retos, ventajas e inconvenientes de ser líder en las distintas sociedades humanas (tribus, cacicazgos, reinos y estados).

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Ariely, D. (2008). *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. USA: Harper Collins.

Binmore, K. (1994). *Teoría de juegos*. Madrid: McGraw Hill.

Corning, P. A. (2004). The evolution of politics. In *Handbook of Evolution*, 1. Wemheim: Wiley-VCH Verlag GmbH & Co.

Dawkins, R. (1994). *El gen egoísta*. Barcelona: Salvat.

Dugatkin, L. A. y Reeve, H. K. (1998). Game Theory and Animal Behaviour. Oxford: Oxford University Press.

ESS_Wiki (2017). Evolutionary Stable Strategy. Wikipedia. Recuperado el 18 de agosto de 2017. Disponible en: https://en.wikipedia.org/wiki/Evolutionarily_stable_strategy

Kaneko, M. (2005). *Game Theory and Mutual Misunderstanding*. Berlin: Springer.

Maynard, J. y Price, G. R. (1973). The Logic of Animal Conflict. *Nature*, 246(5427), pp. 15-18. DOI: <https://dx.doi.org/10.1038/246015a0>

Monsalve, S. y Arévalo, J. (2005). *Un curso de teoría de juegos clásica*. Bogotá: Universidad Externado de Colombia.

Nash, J. F. (1951). Non-Cooperative Games. *The Annals of Mathematics*, 54, pp. 286-295. DOI: <https://dx.doi.org/10.2307/1969529>

Shubik, M. (1982). *Teoría de juegos en las ciencias sociales*. México: Fondo de Cultura Económica.

Talwalkar, P. (2016). *SciShow Incorrectly Explains The Nash Equilibrium*. *Game Theory Tuesdays*. Recuperado el 11 de agosto de 2017. Disponible en:

<https://mindyourdecisions.com/blog/2016/10/11/scishow-incorrectly-explains-the-nash-equilibrium-game-theory-tuesdays/>

Vanderbilt, T. (2010). *Tráfico*. Barcelona: Random House Mondadori.

von Neumann, J. y Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. United States: Princeton University Press.

PELÍCULAS Y VIDEOS

Spinout3 (2012). *Golden Balls. The weirdest split or steal ever!* Recuperado el 26 de agosto de 2017. Disponible en: <https://www.youtube.com/watch?v=S0qjK3TWZE8>

TESIS Y TRABAJOS DE GRADO EN EVALAB

Hoyos, J. D. (2009). *Objeto virtual de aprendizaje para la teoría de juegos*. Cali: Universidad del Valle.

AUTÓMATAS CELULARES

Los autómatas celulares son una estructura computacional que permite añadir el concepto de “espacio” a los algoritmos. Un autómata celular es una matriz de celdas (o células) donde cada una de ellas está comunicada únicamente con sus vecinas. Todas las celdas reciben datos de entrada que les ofrecen sus vecinas, realizan un cómputo cambiando su estado interno y generan datos de salida para sus vecinas, de forma sincrónica, es decir, todas exactamente a la vez. No hay un control central, como en los sistemas de cómputo tradicionales. Todo es distribuido y las decisiones se toman localmente.



Fuente: Fotografía de dominio público. Los Alamos National Laboratory (1945).
https://commons.wikimedia.org/wiki/File:Stanislaw_Ulam.tif

Personaje 4

STANISLAW ULAM (1909-1984)

Stanislaw Ulam fue un matemático nacido en el imperio Austro-Húngaro (actual Polonia) y nacionalizado en USA. Trabajó con Teller, Fermi, von Neumann en el diseño de la primera bomba atómica, y con Metrópolis en la creación del método de simulación de Monte Carlo, usando el ENIAC, uno de los primeros computadores existentes. También trabajó en numerosas ramas de las matemáticas, incluyendo la teoría del caos y los autómatas celulares.

Él fue quien recibió la visita de Paul Erdős mientras estaba hospitalizado con encefalitis, quien le propuso diversos problemas matemáticos para verificar si su cerebro aún funcionaba bien (Ulam, 1976, p. 184).

Aunque no se sabe muy bien quién los inventó, la primera referencia que se tiene es de Stanislaw Ulam, pues en 1950 le recomendó a John von Neumann su uso para el trabajo que quería desarrollar sobre autorreplicación.

Los autómatas celulares tienen múltiples aplicaciones: como modelo teórico de computación abstracta (la computación sistólica que tiene bastante utilidad en el

proceso digital de señales de audio y de vídeo); como una clase de sistemas dinámicos discretos; como herramienta para simulación de vida artificial (por ejemplo, añaden la noción de espacio a las ecuaciones de predador-presa de Lotka Volterra, para así lograr modelar que los conejos puedan escapar de los zorros y estos puedan perseguir a aquellos); como pasatiempos matemáticos (hay mucha gente trabajando en sus ratos libres en el diseño de patrones con propiedades sorprendentes, para el juego de *LIFE* que veremos enseguida); como estructuras electrónicas (o de *software*), por ejemplo, para generar secuencias de números seudoaleatorias usando *Linear Feedback Shift Registers*. En EVALAB los hemos usado para modelar tráfico peatonal, en el 2016 como tesis de maestría de Luz Estela Muñoz.

Y en nuestro caso, los autómatas celulares tienen un gran interés porque permiten investigar propiedades fundamentales en mundos artificiales, como la computación universal, la autoduplicación (ambos temas los veremos en el presente capítulo) y la fabricación de mundos físicos similares o distintos al nuestro.

Definiciones

Como hemos dicho, un autómata celular es una matriz de celdas que puede ser de una dimensión (figura 193), dos (figura 194), tres o más y se extiende en todas direcciones de forma indefinida. Esto se hace para evitar bordes (celdas finales, sin vecinas a continuación) que pueden crear artefactos indeseados. Otra forma de evitar este problema es definir un tamaño máximo y conectar el borde derecho con el izquierdo y el de arriba con el de abajo, según una topología toroidal (figura 195).

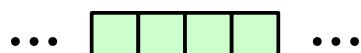


Figura 193: Autómata celular
1D.

Los autómatas celulares tienen entonces un tamaño indefinido, pero no infinito, y esto, en *software*, se puede lograr utilizando matrices dispersas.

Por cierto que la matriz no tiene por qué ser de celdas cuadradas. Pueden ser también triangulares o hexagonales, es decir, las figuras geométricas regulares que logran teselar el plano.

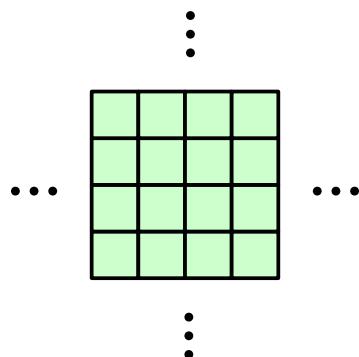


Figura 194: Autómata celular 2D.

Por otro lado, hay que especificar lo que significa “celdas vecinas”, dando un conjunto de coordenadas relativas. Es decir, a las coordenadas de una celda cualquiera se le suman estas coordenadas relativas y así se obtienen sus celdas vecinas.

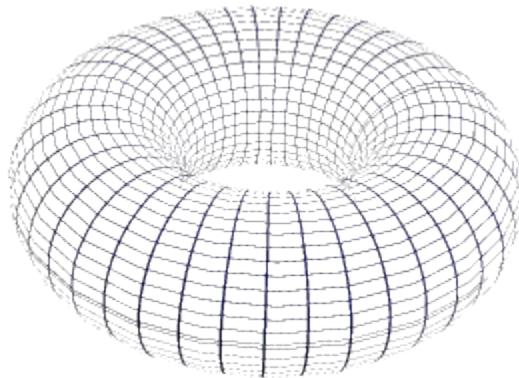


Figura 195: Autómata celular 2D finito sin borde (cada cuadradito es una celda).

Es importante entender que todas las celdas ejecutan el mismo algoritmo simultáneamente, y que si cada celda puede ofrecer resultados distintos es porque su estado interno o sus entradas son distintas. El algoritmo se puede escribir libremente en cualquier lenguaje de programación, aunque en los primeros trabajos sobre autómatas celulares siempre se usaba una máquina de estados finitos (de allí el nombre de “autómata”). La razón de hacerlo así era doble: por un lado, se conseguía sincronismo entre todas las celdas, ya que un autómata correctamente implementado requiere siempre la misma cantidad de tiempo para pasar del estado actual al siguiente. Por otro lado, se evitaba caer en el problema de la parada que podría ocurrir si le damos potencia de cómputo

universal a cada celda.

Como hemos mencionado (figuras 193 y 194), el número de celdas puede crecer indefinidamente, y potencialmente puede llegar a ser infinito. Para evitarlo se añade una restricción adicional buscando que el conjunto sea computable en un tiempo finito: existe un estado de inactividad # (también llamado estado quiescente), donde no es necesario realizar ningún cómputo si todas sus celdas vecinas también están en el mismo estado. Además, por definición, en todo momento, el número de estados no-quiescentes de un autómata celular debe ser finito. O sea, aunque el número de celdas podemos imaginarlo como infinito, el número de celdas donde hay que realizar cómputo debe ser finito.

Con ello no solo se logra modelar el espacio, sino que también se consigue un dispositivo de cómputo no centralizado, diferente a la implementación clásica de una Máquina de Turing Universal (aunque, en últimas, es matemáticamente equivalente a ella).

Un autómata celular se define formalmente como una tupla $C = \langle d, r, Q, \#, V, \delta \rangle$ donde:

- $d \geq 1$ es la dimensión del autómata. Ec. 67
- $r \geq 0$ es su índice de localidad. Ec. 68
- Q es el conjunto de estados. Ec. 69
- $V = (z_1, z_2 \dots z_r) \subset (Z^d)^r$ es un vector de vecindad, que contiene r elementos distintos de Z^d . Ec. 70
- $\delta : Q^{r+1} \rightarrow Q$ es la regla de transición del autómata. Ec. 71
- $\# \subset Q$ es el estado quiescente. Como hemos dicho, por definición $\delta(\#, \#, \#, \dots \#) = \#$. Este estado implica ausencia de actividad. Ec. 72
- Si $V = (z_1, z_2 \dots z_r)$ entonces las r celdas vecinas de la celda w se obtienen mediante sumas vectoriales: $(w+z_1, w+z_2 \dots w+z_r)$. Ec. 73

Las vecindades más utilizadas son la de Moore de rango k , $M(d,k)$, y la de von Neumann $V(d)$, que se definen:

$$M(d,k) = \{ (x_1, x_2 \dots x_d) : \forall i \ (1 \leq i \leq d) \rightarrow (-k \leq x_i \leq k) \} - (0,0\dots 0) \quad \text{Ec. 74}$$

$$V(d) = \{ (x_1, x_2 \dots x_d) : \exists i \ (|x_i|=1 \ \wedge \ \forall j \ (j \neq i \rightarrow x_j=0)) \} \quad \text{Ec. 75}$$

Obsérvese que $V(d) \equiv M(d,1)$. En la figura 196 podemos ver algunos ejemplos. Las más usadas y que emplearemos aquí son $M(2,1)$ y $M(1,1) \equiv V(1)$.

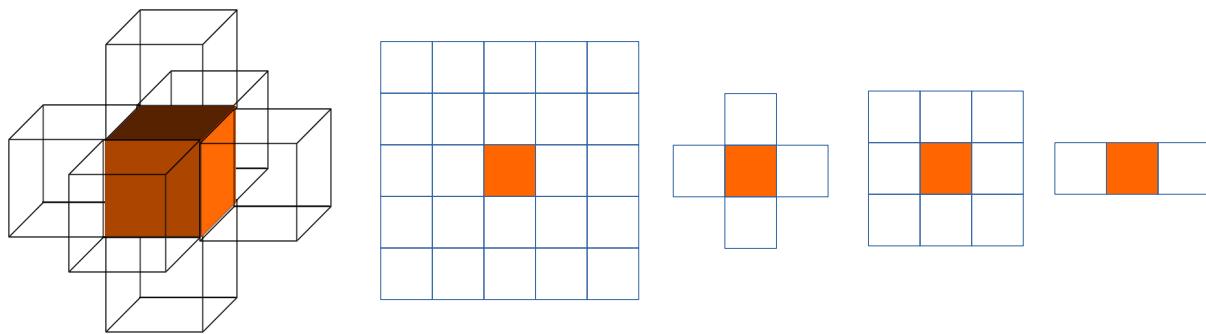


Figura 196: Ejemplos de vecindades $V(3)$, $M(2,2)$, $V(2)$, $M(2,1)$ y $V(1)$, donde la celda q_0 que estamos estudiando está de color naranja y sus vecinas no tienen color.

Dado un autómata celular C , se llama configuración de C a cualquier función $A: \mathbb{Z}^d \rightarrow Q$ tal que $A^{-1}(\#)$ sea un conjunto cofinito. Es decir, en cada instante debe haber un número finito de estados distintos al quiescente. Esto es importante para asegurar la computabilidad del autómata, ya que así solo hay que calcular el estado siguiente de un número finito de celdas.

Sea $C = \langle d, r, Q, \#, V, \delta \rangle$ un autómata celular con $V = (z_1, z_2, \dots, z_r)$, y sea A una configuración de C . Definimos la configuración siguiente $\delta(A)$ como:

$$\forall y \in \mathbb{Z}^d : (\delta(A)(y) = \delta(A(y), A(y+z_1), A(y+z_2), \dots, A(y+z_r))) \quad \text{Ec. 76}$$

Si llamamos A_0 a la configuración inicial, el autómata celular irá pasando sucesivamente por las configuraciones $A_0, A_1, A_2, \dots, A_n$. O sea,

$$A_i = \delta^i(A_0) \quad \text{Ec. 77}$$

Las reglas de transición más interesantes son las isotrópicas, es decir, aquellas en las que no hay direcciones privilegiadas para que ocurra algo. Dentro de ellas se suele trabajar con reglas totalísticas, cuyas transiciones solo dependen de la suma de celdas vecinas que cumplen con alguna propiedad.

LIFE

Un autómata celular muy interesante, que además es un mundo artificial muy conocido, es el del juego de *LIFE*, ideado por John Conway en 1970. Se define así:

$C=(2, 8, \{0,1\}, 0, V, \delta)$, o sea, es un mundo de 2 dimensiones con estados binarios, y vecindad $M(2,1)$: $V=\{(-1,0), (-1,1), (0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1)\}$. Y cuya función de transición es:

$$\delta(q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8) = \begin{cases} q_0 & \text{si } \sum_{i=1}^8 q_i = 2 \\ 1 & \text{si } \sum_{i=1}^8 q_i = 3 \\ 0 & \text{en los demás casos} \end{cases} \quad Ec. 78$$



Fuente: CC BY 2.0, Thane Plambeck (2005). Disponible en <http://www.flickr.com/photos/thane/20366806> y <https://commons.wikimedia.org/w/index.php?curid=13076802>

Personaje 5

JOHN CONWAY (1937-)

John Horton Conway es un matemático británico que trabaja en diversos campos incluyendo la teoría de juegos de información perfecta. Se le conoce sobre todo por la invención de *LIFE* (el Juego de la Vida) que se popularizó rápidamente porque apareció en la columna de pasatiempos matemáticos de Martin Gardner, en la revista *Scientific American* en 1970. En aquella época no existían los computadores personales, por lo que las simulaciones de este juego las hacía a mano, en una cuadrícula de papel sobre la que situaba pequeñas piedras.

Cada celda solo puede tener dos estados: muerta o quiescente (de color negro) o viva (de color rojo). Y regla de transición de *LIFE* de la ecuación 78 se puede enunciar también de una manera menos matemática y más literaria:

- Si una celda está en contacto con 2 celdas vivas entonces su estado se mantiene.
- Si está en contacto con 3 celdas vivas entonces pasa a viva. A ese nacimiento a la vida se le llama reproducción.
- Si está en contacto con 4 o más celdas vivas entonces muere, se dice que por superpoblación.
- Si está en contacto con 1 o menos celdas vivas entonces muere, se dice que

por aislamiento.

Es decir, de alguna manera *LIFE* modela un sistema de seres vivos muy elemental. Pero esto no es lo importante. Lo verdaderamente sorprendente es que con unas reglas de transición tan simples emerge un comportamiento terriblemente complejo a nivel global. En la figura 197 podemos ver una transición de una configuración a la siguiente, después de un tic del reloj síncrono. Hay algunas celdas que permanecen, otras nacen y otras mueren.

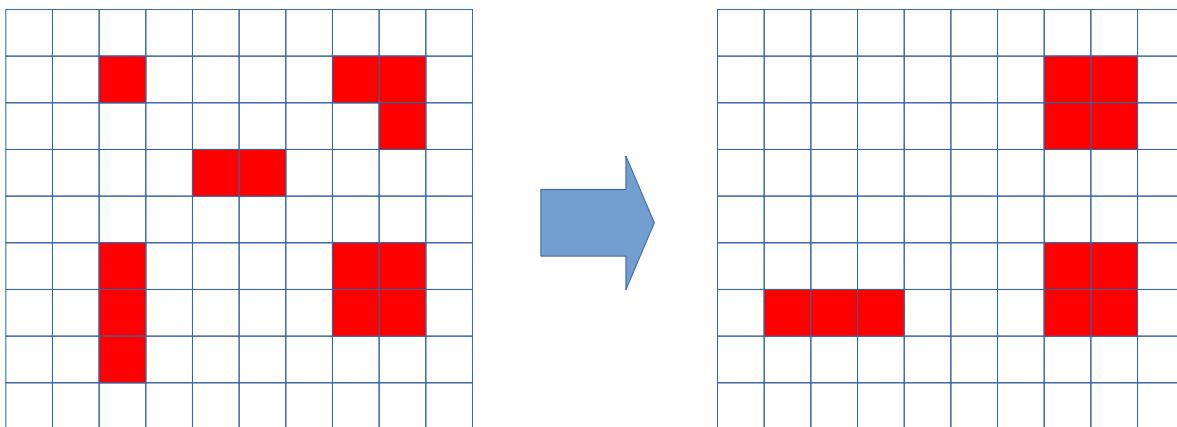


Figura 197: Una transición en *LIFE*.

Se llama patrón a un subconjunto de una configuración, aislado del resto por celdas quiescentes (en la figura 197 hay inicialmente cinco patrones pero en el estado siguiente solo quedan tres). A veces se puede descomponer una configuración en un conjunto de patrones, pero a veces no. La gente que trabaja en *LIFE* ha ido descubriendo y dando nombre a un gran número de patrones.

Dependiendo de las condiciones iniciales, pueden observarse patrones que:

- Son estáticos y no cambian. Por ejemplo, en la figura 197, el de la esquina inferior derecha, de 2x2 celdas.

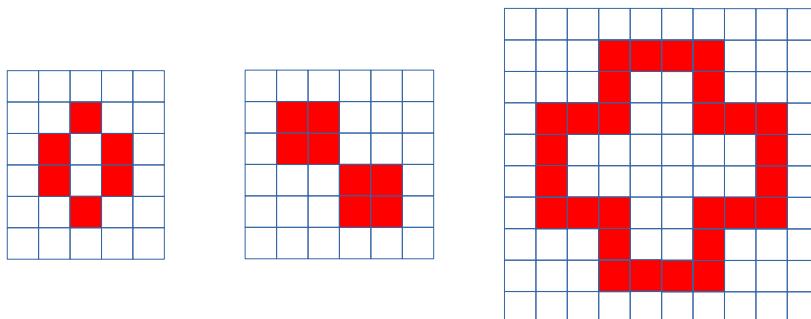


Figura 198: Otros osciladores de periodo 1 (estático), 2 y 3.

- Oscilan periódicamente por varios estados. Por ejemplo, en la figura 197, el de la esquina inferior izquierda de 3 celdas verticales que se convierte a 3 celdas horizontales y luego a 3 verticales. Se pueden construir osciladores con cualquier periodo (figura 198).
- Tienen una evolución larga o incluso desaparecen. En la figura 197 desaparecen dos patrones. Para ver evoluciones largas es mejor usar algún *software* de autómatas celulares.

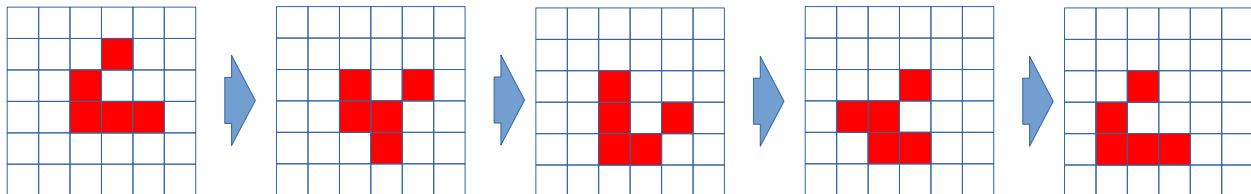


Figura 199: Un *glider* se mueve en diagonal hacia abajo a la izquierda, dando un paso en 4 ticks de reloj.

- Se mueven espacialmente. Un ejemplo es el *glider* de la figura 199 y otro el *spaceship* de la figura 200. Hay más complejos, como *rakes* y *puffers*. Otro patrón importante son los *guns*, que disparan continuamente *gliders* (figura 201).

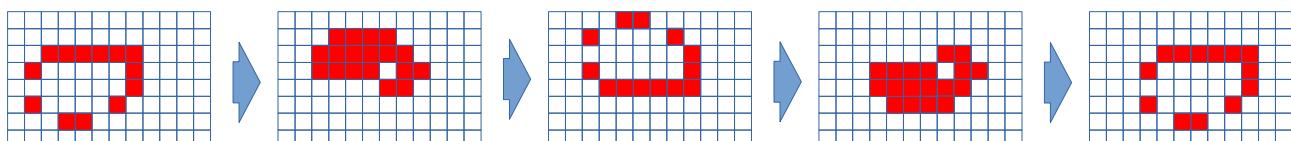
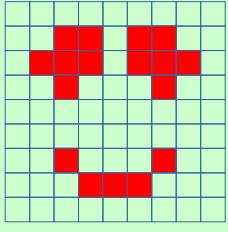


Figura 200: *Spaceship* que se mueve 2 pasos en horizontal hacia la derecha, en 4 ticks de reloj.



Problema 3: NAVE ESPACIAL

Hay un *spaceship* que se mueve horizontalmente y es todavía más pequeño que el descrito en la Figura 200. ¿Puede usted diseñarlo?

- Todas estas estructuras son un ejemplo de emergencia: en las reglas que gobiernan este mundo artificial (es decir, la ecuación 78), no hay nada que force, o ni siquiera sugiera, la existencia de los *gliders*. Sin embargo, si

activas al azar muchas celdas y pones en marcha el autómata celular, es muy probable que aparezcan. Lo mismo ocurre con los osciladores básicos y muchos otros pequeños patrones. Emergen aunque nadie los ha diseñado intencionalmente.

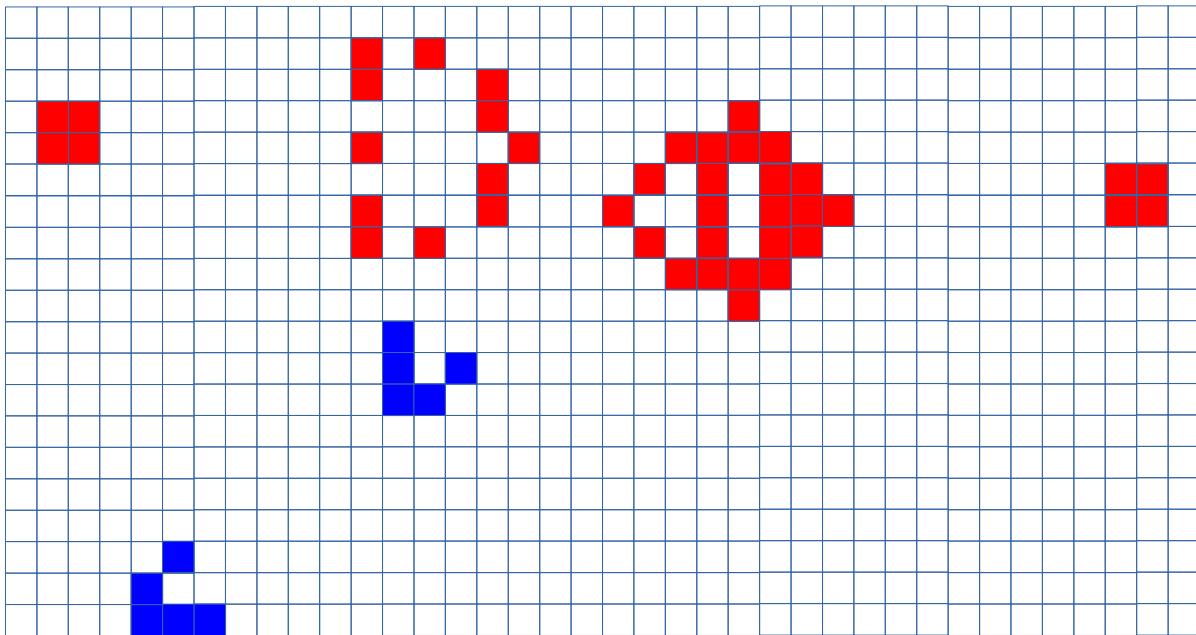


Figura 201: Patrón gun de Bill Gosper que está disparando continuamente gliders en diagonal hacia abajo a la izquierda (resaltados en color azul).

- Interacción entre estructuras. Un ejemplo es el de la figura 202, donde un *glider* choca contra un *eater* y es destruido. También hay espejos donde los *gliders* rebotan; y existen muchos más casos.

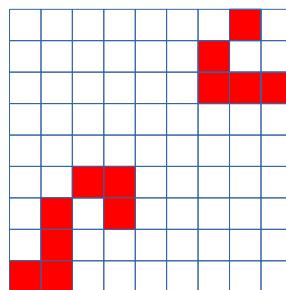


Figura 202: Patrón eater (abajo a la izquierda). Un glider (arriba a la derecha) se va a destruir al chocar contra él.

Existe mucho *software* para jugar con *LIFE*. Actualmente el más completo puede ser *GOLLY*, que es gratuito y multiplataforma, y viene con muchos patrones preprogramados, donde se puede observar todo lo mencionado hasta ahora y mucho más. Es conveniente interactuar un rato con alguno de estos programas,

para darnos cuenta del tipo de mundos que estamos creando. Una característica es que son mundos muy frágiles, pues basta una celda viva de más o de menos para que las configuraciones no lleven a cabo su cometido y muy probablemente se destruyan. También se da uno cuenta de que el diseño de cualquier cosa es muy laborioso. No suelen haber atajos. Únicamente por el método de prueba y error puede uno construir *spaceships*, *guns*, *puffers*, *rakes*, por nombrar algunos. Hay patrones que fabrican otros patrones (*syntethizers*). Y, lo más importante para nuestro objetivo, se pueden construir puertas lógicas (*AND*, *OR*, *NOT* y bits de memoria) donde un '1' es la presencia de un *glider* y un '0' es su ausencia, así como formas para enrutar estas señales en la dirección que uno desee. La puerta NOT es trivial, pues basta lanzar una secuencia de *gliders* que se choquen contra un flujo continuo de otros *gliders* generados por un *gun*. El choque los destruye mutuamente, generando un '0' donde en la secuencia original había un '1' o dejando pasar un '1' procedente del *gun* cuando en la secuencia de entrada hay '0'. Las otras puertas son más laboriosas de hacer, por lo que es mejor verlas en acción en (Bellos, 2016).

Que sea posible implementar cualquier circuito lógico en *LIFE* tiene una consecuencia trascendental: se puede construir una Máquina de Turing Universal (es un buen momento para cargar la que viene en *GOLLY*). La conclusión es que con algo tan simple y natural como una estructura de 2 dimensiones espacialmente distribuida con vecindad trivial $M(2,1)$ y con la sencilla regla de transición dada por la ecuación 78 se consigue computación completa.

Problema 4: VELOCIDAD DE LA LUZ

En un autómata celular de vecindad $k=1$ (como el de *LIFE*), en un tic de reloj la información puede propagarse desde una celda hasta sus vecinas inmediatas. No hay forma de transmitir nada más rápido. O, al menos, eso es lo que dice la teoría. Esto es lo que se ha dado en llamar la velocidad de la luz de ese mundo artificial. Pero ¿será que hay manera de mover un patrón más rápidamente? ¿Qué patrón es el más rápido de todos?

Una vez alcanzado este objetivo, nada impide repetirlo todas las veces que se deseé: dentro de una Máquina de Turing Universal (el computador que está encima de tu mesa) se puede construir un autómata celular que implemente la regla de *LIFE*, en donde hemos diseñado una Máquina de Turing Universal, dentro de la cual hay un autómata celular, y así indefinidamente. Ya se ha implementado esto en *Minecraft*, un juego que también posee capacidad de cómputo universal. Y otro ejemplo divertido e inquietante lo puedes encontrar en el video de

Bradbury (2016).

¿Será que se puede conseguir una Máquina de Turing Universal de alguna manera más sencilla? Enseguida daremos una respuesta afirmativa a esta cuestión, pero antes, terminemos de explorar los autómatas $M(2,1)$. El número de reglas de transición posibles a investigar es enorme. Concretamente es $k^{k(r+1)}$ siendo k el número de estados de una celda y r su número de vecinos. Para $k=2$ y $r=8$ son $2^{512} \approx 10^{154}$ reglas de transición posibles. Cada una de ellas da lugar a un mundo artificial. En muchos de ellos (como la regla *B2/* o la *B3678/S34678*)⁵⁵ hay un único atractor (o unos pocos) que lleva a cualquier condición inicial a desaparecer, a estabilizarse en osciladores o a explotar y llenarlo todo. En otros se producen largos transitorios dependiendo de las condiciones iniciales (como la regla *B36/S23*), que puede ser indicio de que también es posible lograr allí la computación completa. Otros tienen propiedades curiosas, como en el que cualquier estructura evoluciona hasta formar un rombo (regla *B35678/S5678*) y, si hay varios, cuando se tocan se engullen unos a otros. Lo más interesante es observar cómo el patrón que se forma lucha por mantener sus lados diagonales, por muy grande que sea el rombo y a pesar de que la regla es local (cada celda cambia su estado solo en función de sus vecinas). Y esto es solo en autómatas celulares de 2 dimensiones. Imaginen las posibilidades en 3 o más dimensiones.

Stephen Wolfram hizo lo contrario. Bajó a una dimensión con $N(1,1)$ porque allí solo hay $2^8 = 256$ funciones posibles y las analizó todas.

Autómatas celulares 1D

Cuando la dimensión es 1 y la vecindad es 1, el estado siguiente de cada célula solo depende del estado actual de esa célula y de las dos vecinas (la que tiene a la derecha y la que tiene a la izquierda). Como los estados son binarios {0=muerta, 1=viva}, la tabla de transición de estados es como la de la figura 203.

En esta figura, las columnas del estado actual, en color azul, tienen todas las combinaciones posibles de valores binarios. Queda por llenar la columna del estado siguiente, que tiene $2^3=8$ casillas. Esas 8 casillas binarias se pueden llenar de $2^8=256$ formas posibles, y esta es la razón por la que hay 256 reglas

⁵⁵ Las reglas totalísticas en autómatas celulares binarios 2D se identifican con dos listas de números separadas por una barra inclinada: primero (después de la B que significa *birth*, nacimientos), el número de celdas vecinas vivas que hace que la celda actual pase a viva. Y después (después de la S que significa *survive*) el número de vecinas vivas que hace que la celda actual mantenga su estado. Por ejemplo, la regla de *LIFE* es *B3/S23*.

nada más. En la figura 204 podemos ver una de ellas que, por cierto, es la regla 6.

ESTADO ACTUAL			ESTADO SIGUIENTE
Celda izquierda	Celda actual	Celda derecha	Celda actual
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Figura 203: Tabla de transición de estados de un autómata celular $M(1,1)$.

Se llama así porque, si leemos la columna del estado siguiente de abajo a arriba, sale '00000110', un número binario que, convertido a decimal, da 6.

ESTADO ACTUAL			ESTADO SIGUIENTE
Celda izquierda	Celda actual	Celda derecha	Celda actual
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figura 204: Regla 6.

Trabajar con autómatas celulares de una dimensión tiene otra ventaja: como el papel (o la pantalla del computador) tiene dos dimensiones, es posible utilizar la otra para el tiempo. Es decir, en horizontal vamos a ver el espacio 1D y en vertical

el paso del tiempo, de arriba a abajo. El estado inicial puede ser cualquiera, pero para ver qué ocurre en el caso más simple, vamos a poner solo una celda viva.

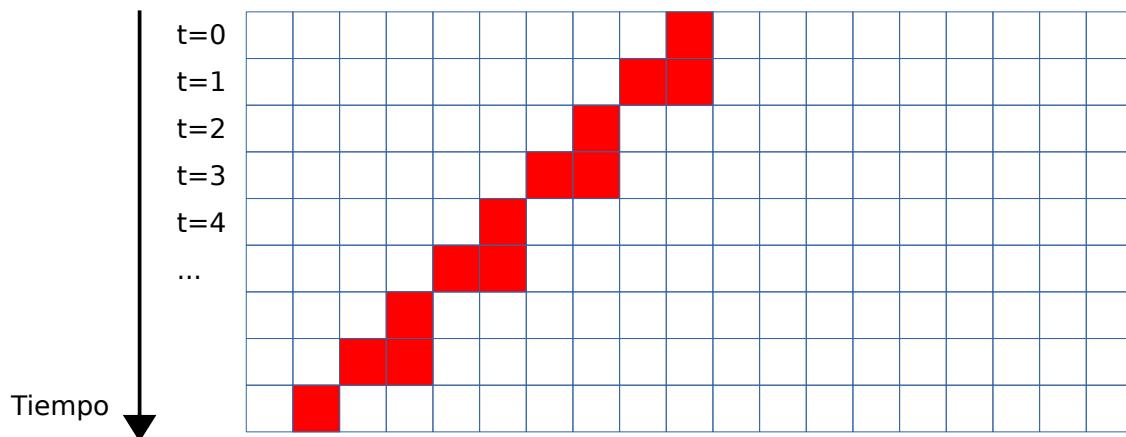


Figura 205: Desarrollo en el tiempo de la regla 6 con una semilla de una única celda viva.

Entonces podemos ver en la figura 205 que en $t=0$ solo la celda central está viva. Aplicando la regla 6, se obtiene el estado de las celdas en $t=1$. Por ejemplo, la celda que está viva hace la transición $010 \rightarrow 1$. O sea, pasa a viva, mientras que la celda a su izquierda hace la transición $001 \rightarrow 1$ (viva) y la de su derecha $100 \rightarrow 0$ (muerta). Las demás no cambian debido a que $000 \rightarrow 0$, por lo que siguen muertas. Después se vuelve a repetir lo mismo para calcular el estado de las celdas en $t=2$ a partir de su estado en $t=1$, etc. Puede verse que el patrón que sale es completamente regular y predecible a largo plazo. Para otras condiciones iniciales saldrá un patrón distinto, pero también muy predecible.

ESTADO ACTUAL			ESTADO SIGUIENTE
Celda izquierda	Celda actual	Celda derecha	Celda actual
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Figura 206: Regla 90.

Una regla de transición más interesante es la 90, que podemos observar en la figura 206. En la figura 207 se observa cómo se desarrolla esta regla a lo largo del tiempo. Sorprendentemente, es el fractal de Sierpinski que se vimos en un capítulo anterior.

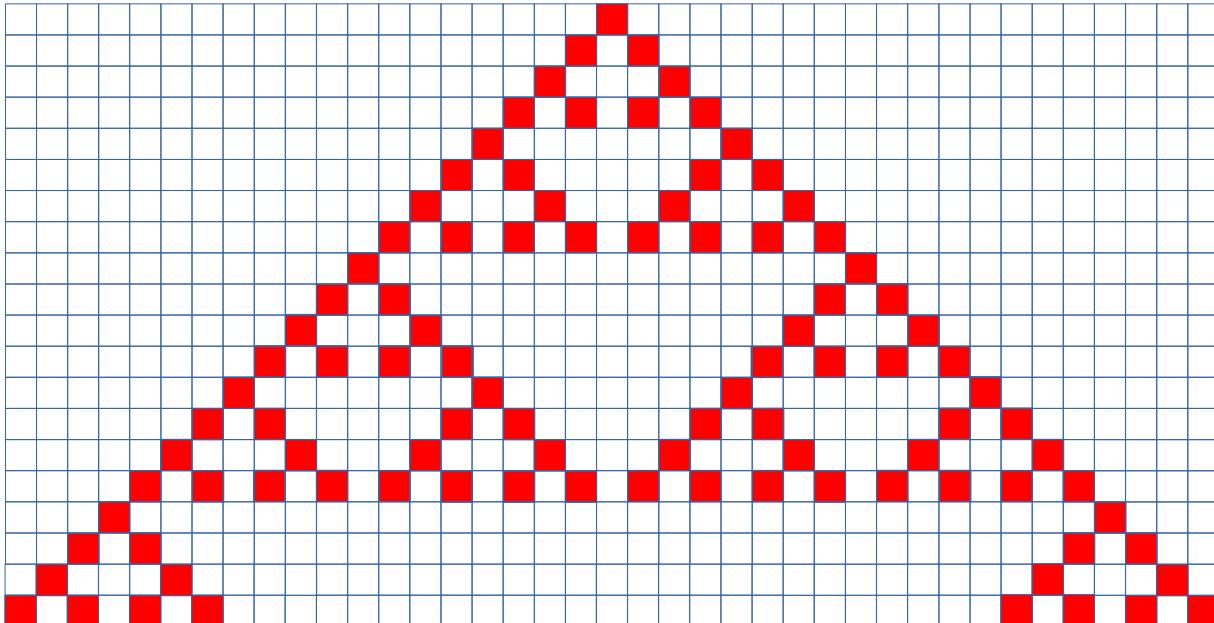


Figura 207: Desarrollo en el tiempo de la regla 90 con una semilla de una única celda viva.

Otra regla interesante es la 30 de la figura 208, y cuya evolución en el tiempo se ve en la figura 209.

ESTADO ACTUAL			ESTADO SIGUIENTE
Celda izquierda	Celda actual	Celda derecha	Celda actual
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figura 208: Regla 30.

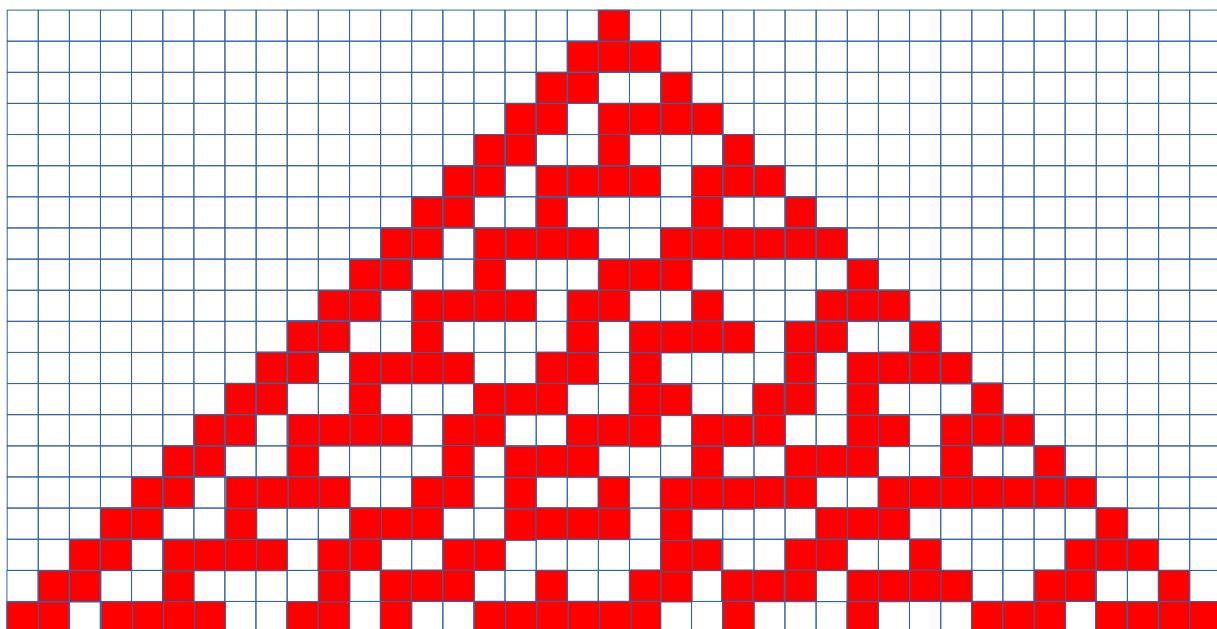


Figura 209: Desarrollo en el tiempo de la regla 30 con una semilla de una única celda viva.

Puede verse que el patrón que sale es regular en ciertos sitios (en los bordes diagonales) e irregular en otros (en el centro). Si te pido que me digas qué va a salir 4 tics de reloj más allá de lo que aparece en la figura, la única forma de saberlo es calculando el estado de todas las celdas tic a tic. No hay atajos. Esto significa que el patrón debe ser estocástico, seudoaleatorio o caótico. Estocástico

no es, pues la regla de transición y las condiciones iniciales se conocen bien y son deterministas. Por tanto, es seudoaleatorio o caótico. Devaney demostró que el sistema cumple las tres condiciones matemáticas del caos, aunque todo ello sigue siendo objeto de controversia, como puede verse en el texto de Cattaneo (1999).

ESTADO ACTUAL			ESTADO SIGUIENTE
Celda izquierda	Celda actual	Celda derecha	Celda actual
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figura 210: Regla 110.

Para terminar, veamos la regla 110 cuya tabla de transición está en la figura 210 y cuya evolución en el tiempo está en la figura 211.

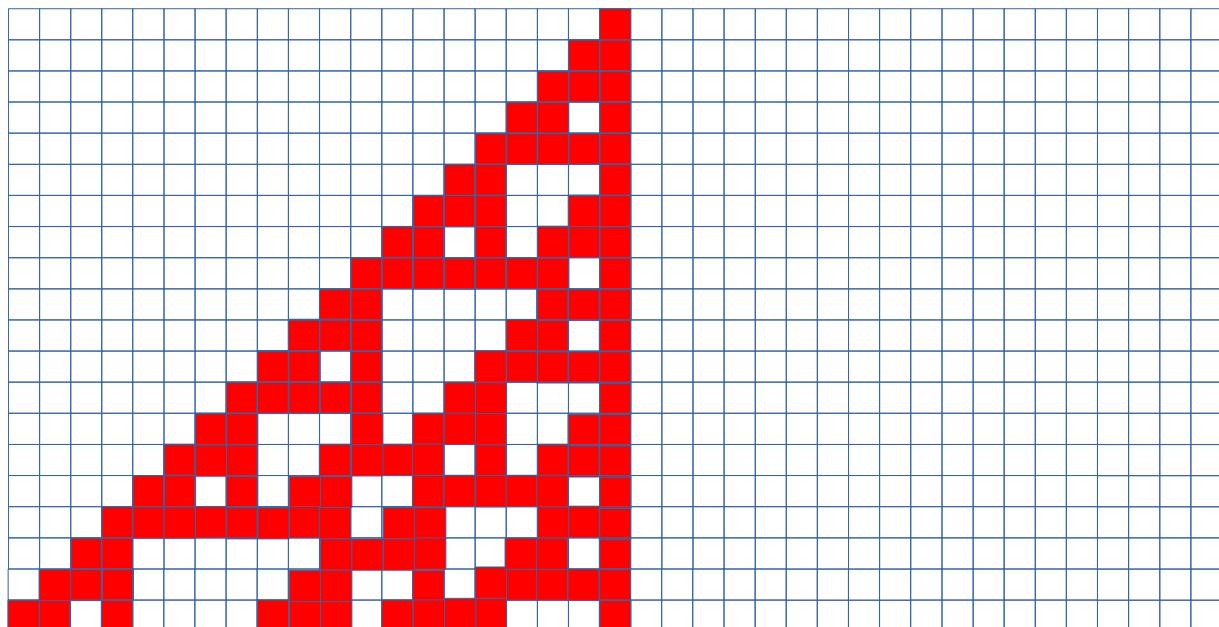


Figura 211: Desarrollo en el tiempo de la regla 110 con una semilla de una única celda viva.

Wolfram clasificó estos autómatas de la siguiente manera, dependiendo de sus reglas de transición y para configuraciones iniciales aleatorias:

- **Clase 1.** Todas las configuraciones iniciales evolucionan hacia una única configuración muy sencilla, destruyéndose la información del pasado. El sistema es totalmente predecible, pues tiende a un único atractor. La regla 0 es así, donde todos los sucesivos estados son 0 (ya que la tabla de transiciones es $*** \rightarrow 0$), independientemente de las condiciones iniciales.
- **Clase 2.** Hay varios atractores y el resultado final es bastante predecible. Las condiciones iniciales dictan hacia cuál de ellos se dirigirá el sistema. Cambios pequeños en las condiciones iniciales no cambian la cuenca de atracción. El sistema es predecible si se conocen *grossos modo* las condiciones iniciales. Un ejemplo de ello es la regla 6.
- **Clase 3.** La evolución es seudoaleatoria. Solo sabiendo con toda exactitud cuáles fueron las condiciones iniciales, se podrá predecir el estado futuro del autómata celular. Y la forma más rápida de predecirlo es dejando que se ejecute. Un ejemplo de ello es la regla 30.
- **Clase 4.** La evolución es caótica, es decir, predecible a corto plazo, pero no a largo plazo. Hay largos transitorios que desembocan en cuencas de atracción, generando estructuras periódicas, no periódicas, que se mueven espacialmente. Es el tipo más complejo e interesante. Son difíciles de predecir. En algunas configuraciones se sabe que estos autómatas celulares son dispositivos de cómputo universal. Un ejemplo de ello es la regla 110.

Cristopher Langton estudió estos autómatas celulares en función de un parámetro *lambda* (λ) que definió así:

$$\lambda = \frac{\text{número de transiciones que producen un estado no quiescente}}{\text{número total de transiciones}} \quad \text{Ec. 79}$$

Este parámetro mide de forma grosera el índice de actividad. Es el típico dial con el que vamos barriendo el espacio de configuraciones, igual que vimos en el capítulo de caos. Al ir variando λ de 0 a 1 obtuvo los siguientes resultados aproximados:

- Valores pequeños de λ indican poca actividad, o sea, mucho orden. Estos son los autómatas celulares de clase 1, según la clasificación de Wolfram.
- Al aumentar λ aparecen estructuras estables, o sea, sigue habiendo orden y

predictibilidad. Estos son los autómatas celulares de clase 2.

- Al aumentar un poco más λ aparecen estructuras metaestables, o sea, casi-estables, disminuyendo la predictibilidad. Estos son los de clase 4.
- Y al llegar a los valores más altos de λ aparece el desorden, lo estocástico, o sea los de la clase 3.

Luego estudió con más detalle la clase 4 y vio que era el resultado de una transición de fase entre orden y desorden, que es lo que se conoce como borde del caos. Si ya has leído el capítulo sobre el caos, esto te sonará bastante.

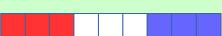
Mientras tanto, Wolfram intuía que la regla 110 tenía capacidad de cómputo universal. No obstante, fue su empleado Mathew Cook en 2004 quien logró demostrarlo publicando los resultados, contrariando bastante a su jefe. Parece ser que finalmente llegaron a un acuerdo sobre la autoría de estos trabajos. Pero lo importante para nosotros es que la regla 110 es equivalente a una Máquina de Turing Universal. Programarla no es nada fácil. Por ejemplo, si quieres sacar la lista de números primos o cualquier otra cosa, debes poner ciertas celdas vivas o muertas en la fila inicial (o sea, en $t=0$). ¿Cuáles? Eso es lo difícil, pues no se trata de un lenguaje ensamblador ni nada a lo que estemos acostumbrados. Podría usarse un algoritmo genético, dado que se trata de encontrar un vector binario que realice cierta tarea. Lo habitual es decidir que la salida de este “computador” esté en la columna central. Por cierto, las reglas 124, 137 y 193 también son Máquinas de Turing Universales, pues son equivalentes a la regla 110 sin más que hacer algunas simetrías y complementos.

Este es uno de los resultados más importantes en lo que se refiere a los objetivos de este libro: con muy poca infraestructura (un conjunto de celdas interconectado linealmente de forma local, y una pequeña máquina de estados en cada celda), de cada 256 configuraciones, cuatro son computadores. Es, definitivamente, sorprendente. La probabilidad de encontrar un computador cuando se tiene cierta complejidad es bastante alta, alrededor del 2%. Uno creería que los computadores son bastante complejos, y si abres uno te encontrarás con procesadores de miles de millones de transistores, memoria RAM, disco duro, buses y periféricos. Sin embargo, en lo básico, para construir un computador requieres del orden de 8 bits de información donde almacenar la tabla de transiciones y una estructura de conexiones simple, local y repetitiva.

Este es uno de los hitos de la complejidad, en el que emerge la capacidad de cómputo universal. Y emerge en un punto a medio camino entre el orden y el desorden, o el borde del caos. Es lógico que así ocurra: en sistemas ordenados no

aparece nunca nada nuevo. Todo es estático o periódico. Allí es imposible lograr la flexibilidad de la computación. Por el contrario, en sistemas completamente desordenados no hay nada fiable, si intentas repetir un proceso sale cada vez distinto. El cómputo también es allí imposible. La computación requiere algo de orden (por ejemplo, un dato que guardas en memoria debería continuar allí cuando vayas a buscarlo), pero no tanto que impida la flexibilidad y creatividad de los algoritmos.

Por otro lado, la comunicación local es algo muy natural, no solo biológicamente hablando, sino matemáticamente, pues apenas se requiere geometría. Pensemos que en un cristal las vibraciones de un átomo afectan a sus vecinos; las células de un órgano afectan y son afectadas por las vecinas; en una comunidad de animales, los virus, los conocimientos, la comida y la información se propagan a los vecinos; las placas tectónicas empujan y son empujadas por las vecinas; y así podríamos poner infinidad de ejemplos.



Problema 5: BANDERA FRANCESA

Tenemos un autómata celular de una dimensión de tipo $M(1,1)$, inicializado con un segmento de células en el mismo estado I , y donde cada célula exhibe un color que depende de su estado (estados distintos podrían tener el mismo color). Como el autómata es infinito, las demás células están en estado quiescente Q .

A partir de una perturbación inicial P en uno de los extremos, debe desarrollarse la bandera francesa: un tercio rojo, un tercio blanco, un tercio azul.

Etc. Finalmente debe quedar:

Y si el segmento se divide en dos (poniendo alguna celda en estado P), cada parte debe regenerar su propia bandera, orientada en el mismo sentido que la original. Esto se puede repetir indefinidamente.

Su misión consiste en desarrollar el *software* que implemente todo esto. Recuerde que todas las celdas tienen el mismo algoritmo.

Si en un ambiente así disponemos de 8 bits de decisión y generamos las

combinaciones posibles, al lanzar los dados, en el 2% de las veces saldrá un computador. ¡Sorprendente! Y nadie negará que, teniendo un computador a mano, las posibilidades de generar más complejidad se expanden dramáticamente, saltando a un nuevo nivel.

Autoduplicación

Con los autómatas celulares también se puede lograr autorreplicación⁵⁶. Este fenómeno es importante por varias razones:

- Es un elemento imprescindible de la evolución. Al principio del capítulo “Algoritmos evolutivos” mencionamos los cuatro factores necesarios para que surja la evolución, y el autocopiado es uno de ellos. No solo eso, sino que es el más difícil de lograr de los cuatro. Podemos simplificar diciendo que cuando aparece el autocopiado, surge también la evolución. Y la evolución es un excelente generador de complejidad. Por eso nos interesa saber qué se requiere y cuál es el umbral necesario para que aparezca la autoduplicación.

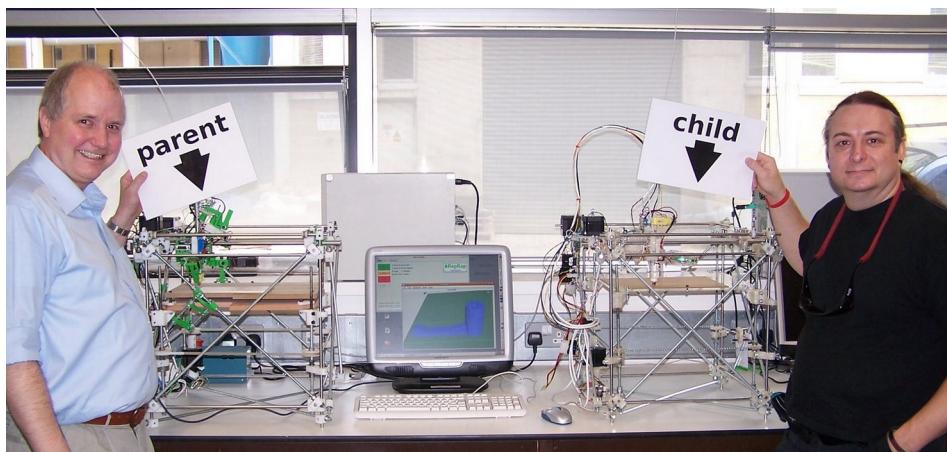


Figura 212: RepRap fue el primer objeto industrial autorreplicante.

Fotografía bajo el dominio CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=7844781>

- Es el sueño de todo ingeniero para no volver a trabajar. Ya comienzan a aparecer las primeras máquinas de autocopia, como las impresoras 3D con las cuales puedes fabricar cualquier objeto, incluyendo otra impresora 3D (bueno, ¡vale!, las partes metálicas y electrónicas todavía no, pero solo es

⁵⁶ Que puede sacar una copia de sí mismo. Usaremos como sinónimos: autorreplicación, autoduplicación, autocopia y autorreproducción.

cuestión de tiempo). La impresora 3D *RepRap* de Adrian Bowyer (figura 212) fue la primera construida con esas intenciones que es, además, de bajo costo y de acceso libre a los planos de construcción. Este tipo de máquinas nos permitirá en un futuro hacer ingeniería a nivel planetario o aún más ambiciosa, como fabricar una esfera de Dyson⁵⁷. Como hecho curioso, el escritor de ciencia ficción Arthur Clarke se adelantaba a ello en la segunda parte de su novela *2001: una odisea en el espacio*, donde proponía como convertir el planeta Júpiter en una estrella, usando máquinas autorreplicantes.

- Por último, la autorreplicación es una realimentación positiva, que da lugar a crecimientos exponenciales y en muchos casos puede producir fenómenos caóticos.

En la actualidad la autorreplicación ya se lleva a cabo rutinariamente con robots como el del trabajo de Cornell (2009), formado por 4 cubos idénticos que se conectan entre sí con electroimanes, se transmiten entre ellos información y tienen un eje de rotación en su diagonal. Un robot de 4 cubitos puede fabricar en muy poco tiempo otro robot de 4 cubitos a su lado, como se ve en el video mencionado. Sin embargo, ¡hay un truco!, comenta mucha gente cuando lo ve: los cubitos ya están fabricados previamente, y lo único que hace el robot es ensamblarlos en el orden correcto. Aunque esto es verdad, no le quita mérito ni interés al trabajo por la siguiente razón: los animales también sacamos copias de nosotros mismos, pero para lograrlo tenemos que ingerir moléculas largas previamente fabricadas procedentes de otros animales y vegetales. Ningún animal come exclusivamente tierra para alimentarse. Lo mismo pasa con estos robots. Falta únicamente dirigir trabajos de desarrollo para conseguir que las piezas de partida sean cada vez más básicas, hasta lograr que haga autocopia usando minerales que se encuentren en el suelo. En cierto modo, esto ya está muy avanzado, pues las fábricas humanas de metalmecánica, microelectrónica y plásticos, cada vez están más automatizadas, de modo que las piezas básicas que se entreguen a los robots autorreplicantes procederán también de procesos que pueden replicarse. Otra línea de trabajo prometedora es la nanotecnología, que permitiría copiar objetos átomo a átomo.

Ni que decir tiene que el futuro que nos espera puede ser asombroso o aterrador, en el momento en que un robot pueda sacar copias de sí mismo. Pero el momento está muy cerca.

57 Una hipotética estructura espacial que se construiría rodeando por completo a una estrella con el objetivo de capturar toda la energía que emitía.

Después de explicar la importancia del proceso de autocopia, nos debemos estar preguntando qué se requiere para que surja espontáneamente en la naturaleza (o dentro del computador). Por suerte para nosotros, John von Neumann ya hizo este estudio de forma teórica, y aunque no alcanzó a publicar sus resultados, un amigo suyo si lo hizo (von Neumann y Burks, 1966).

				U	Estado quiescente (desprogramado)
\uparrow	\downarrow	\rightarrow	\leftarrow		Caminos normales
(\uparrow)	(\downarrow)	(\rightarrow)	(\leftarrow)		Caminos normales con una señal
($\uparrow\downarrow$)	($\downarrow\uparrow$)	($\rightarrow\Rightarrow$)	($\leftarrow\Leftarrow$)		Caminos alternativos
($\uparrow\downarrow$)	($\downarrow\uparrow$)	($\Rightarrow\rightarrow$)	($\Leftarrow\leftarrow$)		Caminos alternativos con una señal
C_{00}	C_{01}	C_{10}	C_{11}		Confluencia (puerta AND con memoria)
S_θ	S_0	S_1	S_{00}		Estados intermedios constructivos
S_{01}	S_{10}	S_{11}	S_{000}		

Figura 213: Los 29 estados de von Neumann.

Él empleó un autómata celular de tipo V(2), donde cada célula podía tener uno de los 29 estados que aparecen en la figura 213. Como en todo autómata celular, hay un estado quiescente *U*. Luego hay cuatro estados que sirven para trazar caminos por donde circulen señales. Están también esos cuatro caminos pero con la señal circulando (un círculo alrededor de la flecha). Esta situación puede causar confusión al principio, pues en ambientes de programación orientados a objetos, hay un camino y hay un objeto aparte que recorre el camino. Recordemos que aquí el estado es todo: existe el mismo algoritmo en todas las celdas y lo único que las diferencia es el estado. Por eso, tanto la señal como el camino se codifican dentro del estado. En la figura 214 se muestra un claro ejemplo de lo que ello significa. La señal inicialmente está en la casilla 'b2', y en el siguiente tic de reloj la casilla 'b1' ve que a su derecha hay una señal en un camino que apunta a la izquierda, por lo que ella se convierte en señal, sin cambiar el camino que ya tiene apuntando a la izquierda. En resumen, cada celda mira si a su alrededor hay una señal con un camino apuntando hacia ella, en cuyo caso, en el tic siguiente, pasa ella misma al estado 'señal'. También mira si ella está en estado señal y, en caso afirmativo, en el estado siguiente la señal desaparece. Con estos dos pasos se simula el movimiento de señales por caminos.

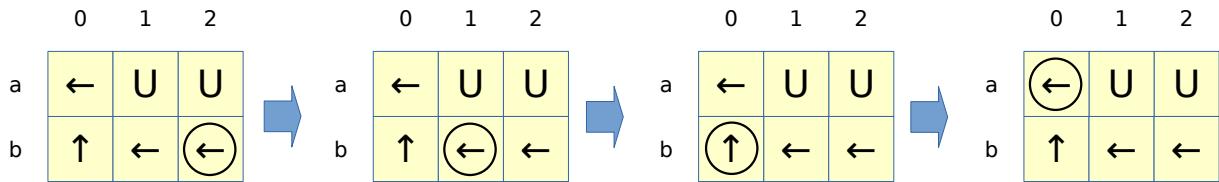


Figura 214: Una señal moviéndose tranquilamente por un camino.

Las flechas dobles con sus respectivas señales funcionan exactamente de la misma manera. La única particularidad con los dos tipos de flechas es que si hay una señal viajando por un camino de flechas simples y tropieza con un camino de flechas dobles (o al revés), entonces la celda se desprograma, pasando a estado quiescente U, como puede observarse en la figura 215.

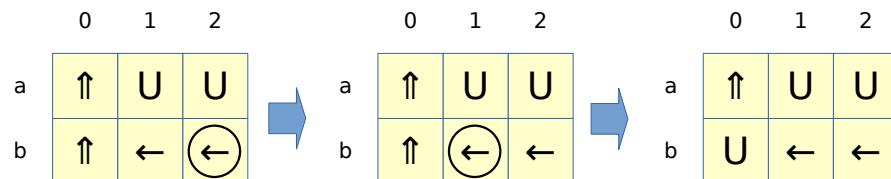


Figura 215: Choque de caminos.

Los siguientes cuatro estados configuran una puerta AND secuencial, con memoria. La idea es que cuando todas sus entradas (todos los caminos que apunten a la puerta) reciben a la vez una señal, entonces la salida generará también una señal, pero un tic de reloj más tarde. Al haber ese retraso, podría ocurrir que la puerta esté memorizando que le toca activarse en el siguiente tic de reloj y a la vez que le lleguen nuevas señales de activación. Esta es la razón por la que se requieren más estados internos, cuyas transiciones se explican en la figura 216.

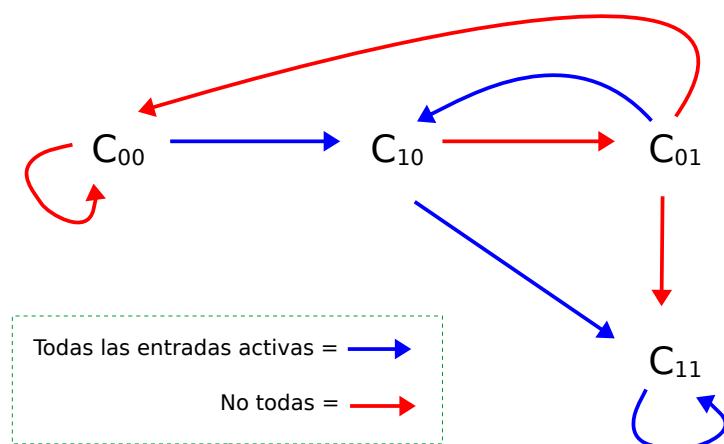


Figura 216: Transiciones posibles en una puerta AND con memoria.

Por último, hay ocho estados intermedios que permiten la construcción de celdas con un cierto estado inicial, a partir de la llegada secuencial de señales que choquen contra una celda en estado quiescente, según se explica en la figura 217. Las celdas que se pueden construir así son nueve (las cuatro flechas simples, las cuatro flechas dobles y la puerta AND en su estado inicial). Para entender mejor la figura supongamos, por ejemplo, que a una celda desprogramada (en estado U) le llega la secuencia 1101 (donde 1 significa una señal y 0 ausencia de ella). Entonces al final de 4 tics de reloj quedará programada como una flecha doble hacia la izquierda (\Leftarrow).

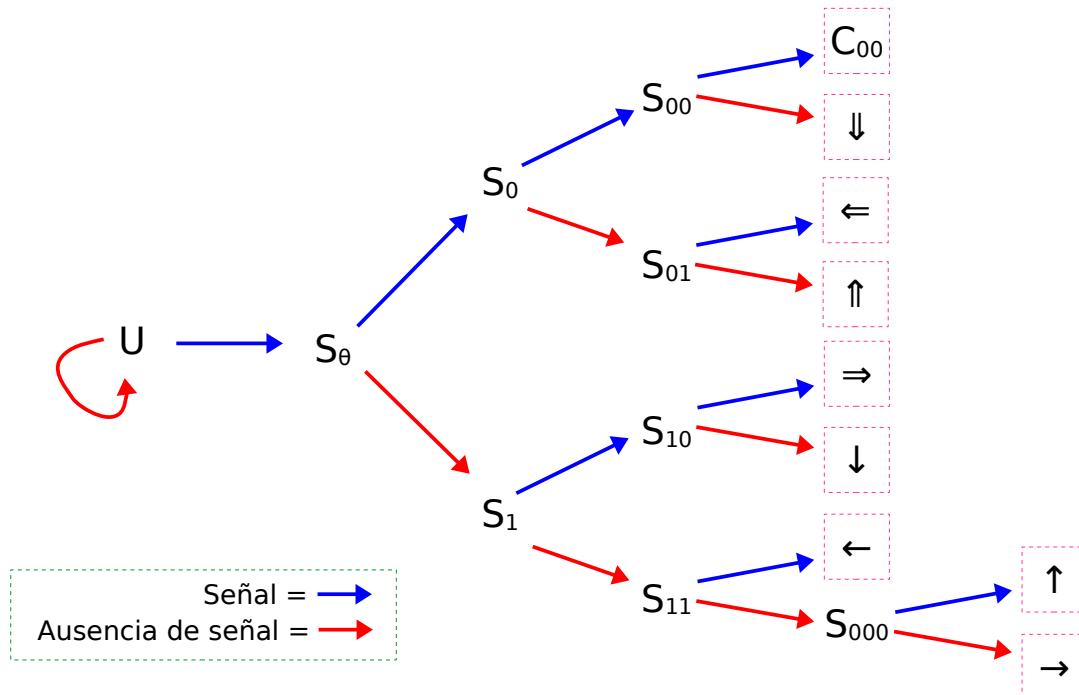


Figura 217: Programación del estado de una celda previamente desprogramada.

Para lograr una mejor comprensión de todo esto, veamos un ejemplo en la figura 218, donde hay nueve señales circulando:

- La de 'c1', que se va a perder por no haber otra en 'b0' para alimentar la puerta AND.
 - La de 'a4' que junto con la de 'b3' alimentarán la puerta AND que hay en 'b4' llevándola al estado C_{10} .
 - La que hay en espera internamente en la puerta AND de 'b1', que saldrá por 'b2' en el siguiente tic de reloj dejando 'b1' en el estado C_{00} .
 - En 'd0' hay dos señales: una que acaba de entrar a la puerta AND y otra

que entró en el tic anterior. En el siguiente tic de reloj, 'd0' quedará en estado C_{01} y saldrán señales simultáneamente por 'c0' y 'd1'. Y en el siguiente tic de reloj pasará a C_{00} y saldrá la otra señal acumulada, por el mismo sitio.

- Finalmente, la secuencia de señales que hay en 'd5', 'd4' y 'd2' chocarán en sucesivos tics de reloj con la flecha doble que hay en 'd6' (concretamente llegará la secuencia 11010). Como el camino por el que vienen esas señales es de tipo flecha simple, la celda 'd6' quedará desprogramada (en estado U) al llegar el primer 1. Las demás señales 1010 convertirán la celda 'd6' en una flecha simple hacia abajo (\downarrow).

	0	1	2	3	4	5	6
a	\uparrow	\leftarrow	\leftarrow	\downarrow	$\circlearrowright \downarrow$	\Leftarrow	U
b	\rightarrow	C_{01}	\Rightarrow	$\circlearrowright \Rightarrow$	C_{00}	\Rightarrow	\Downarrow
c	\uparrow	$\circlearrowright \uparrow$	U	\Leftarrow	\Leftarrow	\Leftarrow	\Downarrow
d	C_{11}	\rightarrow	\circlearrowright	\rightarrow	\circlearrowright	\circlearrowright	\uparrow

Figura 218: Un trozo de autómata de von Neumann.

Hasta aquí la carpintería, pero ¿para qué le sirvió todo esto a von Neumann? Pues resulta que con ello pudo crear dentro del autómata un brazo constructor de cualquier longitud, como se muestra en la figura 219. Una señal que se inyecte por la fila 'b' chocará al final con la flecha doble que hay en 'a4', desprogramándola. Enviando después una secuencia adecuada de señales podemos programar cualquier cosa en 'a4'. Algo similar haremos con 'b4', dejando el brazo retraído en una celda, y listo para repetir lo mismo con 'a3' y 'b3'.

	0	1	2	3	4
a	\Rightarrow	\Rightarrow	\Rightarrow	\Rightarrow	\Rightarrow
b	\rightarrow	\rightarrow	\rightarrow	\rightarrow	\uparrow

Figura 219: Brazo constructor.

El proceso completo es simple pero laborioso y lo podemos ver en la figura 220. En el último paso vemos que se han escrito dos símbolos cualesquiera (representados por α y β) en las casillas del extremo, y el brazo constructor se ha retraído. De modo que el proceso puede continuar indefinidamente.

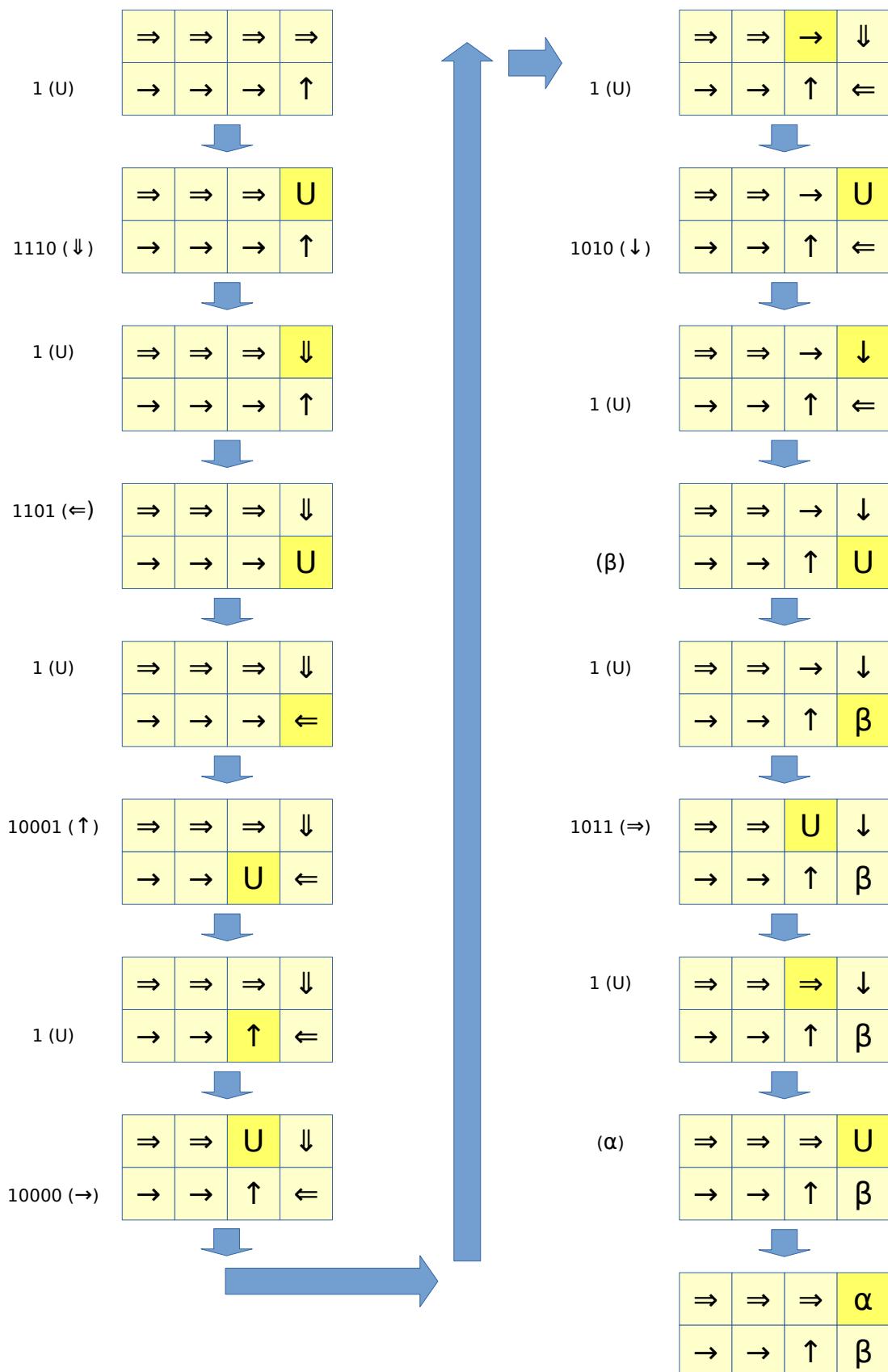
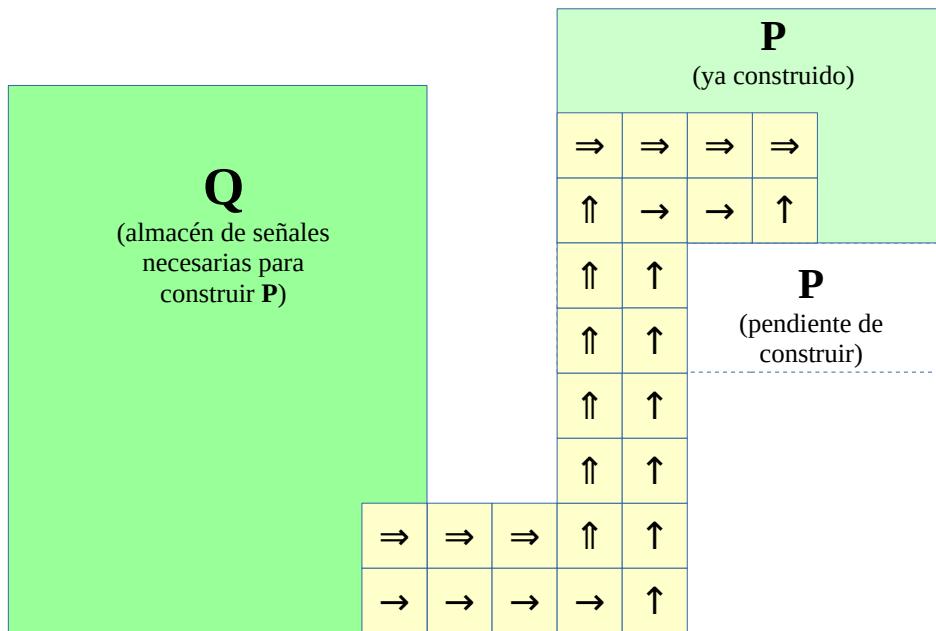


Figura 220: Uso del brazo constructor.



*Figura 221: Un autómata **Q** construyendo otro autómata **P**.*

De forma análoga se puede hacer un brazo constructor en dirección vertical, y combinando ambos se logra que un autómata celular **Q** convenientemente diseñado, inyecte señales de manera adecuada para construir un rectángulo **P** de celdas con cualquier símbolo en ellas (figura 221). Es decir, un autómata puede construir otro autómata. ¡Estamos a punto de lograr la autoduplicación!

Pero las cosas no son tan fáciles porque el autómata **Q** es muy complejo, concretamente **Q** sale bastante más grande que **P**. Afortunadamente a von Neumann se le ocurrió una forma de reducir el tamaño, separando por un lado el brazo constructor con sus circuitos de control y, por otro, la información de descripción del nuevo autómata en una cinta⁵⁸. La cinta está formada por cuatro filas de celdas (figura 222): las dos primeras **A** y **B** emiten señales para leer la

A	→	→	→	↓									
B	⇒	⇒	⇒	↓									
C	U	U	↓	?	↓	↓	U	↓	↓	U	↓	U	
D	←	←	←	←	←	←	←	←	←	←	←	←	←

Figura 222: Cinta de información.

cinta, para reconstruir su contenido, pues la lectura es destructiva, y para

58 Una cinta funcionalmente idéntica a la de las máquinas de Turing.

retraerse a la posición anterior y leer la siguiente celda. En últimas es un brazo constructor también. La siguiente C es donde está la información de descripción del nuevo autómata celular P que se va a construir. Su formato es binario siguiendo el convenio de que U significa 0 y \downarrow significa 1. Por la última fila D se recibe la información leída de la cinta.

Supongamos que queremos leer la información binaria que hay en la celda resaltada con otro color. Como no sabemos lo que hay, le hemos puesto '?'. Para hacerlo, el proceso es enviar 10101 por la fila A . Si en D se recibe lo mismo (10101) es que la celda contiene 1 (o sea, \downarrow , lo cual es obvio). Mientras que si se recibe 1 significa que la celda contiene 0, o sea, U , lo cual ya no es tan obvio y para entenderlo tendremos que recurrir a la figura 217.

Entonces la estrategia a seguir dado un patrón P que queramos construir, es codificar sus dimensiones seguido de los estados de cada célula (en un cierto orden). A ello lo llamaremos $d(P)$, o sea, descripción de P . En la cinta se coloca $d(P)$ y el constructor universal M irá leyéndola, para construir P (figura 223).

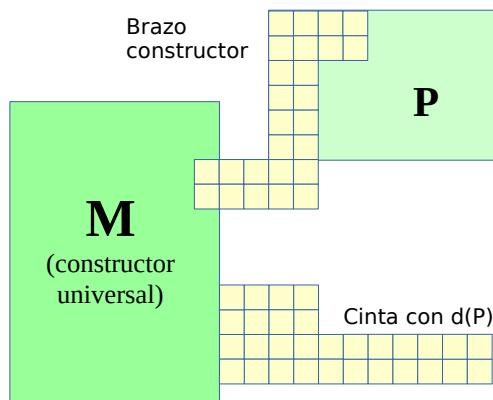


Figura 223: Constructor universal M creando otro autómata celular P .

Para lograr la autoduplicación hacemos $P=M$, de manera que en la cinta está $d(M)$. Aunque esto no es suficiente por dos razones: $M+d(M)$ construyen M . O sea, la réplica no es exacta, pues le falta la cinta. Y el nuevo M no está “vivo”, ya que se genera con todos sus estados inactivos (es decir, sin señales circulando), de modo que no podrá a su vez reproducirse.

Para solventar estos problemas, se modifica M a un M' que no solo interpreta la cinta sino que, al finalizar, saca también una copia de ella. Además, se diseña un bloque de modo que todas las señales de M' se generen a partir de una única señal inicial. Así, al terminar la copia únicamente debe enviarse esa señal desencadenante a la copia. Lo cierto es que esto último no es posible en todos los

casos de modo que el constructor de von Neumann no llega a ser completamente universal, si bien lo es a todos los efectos prácticos.

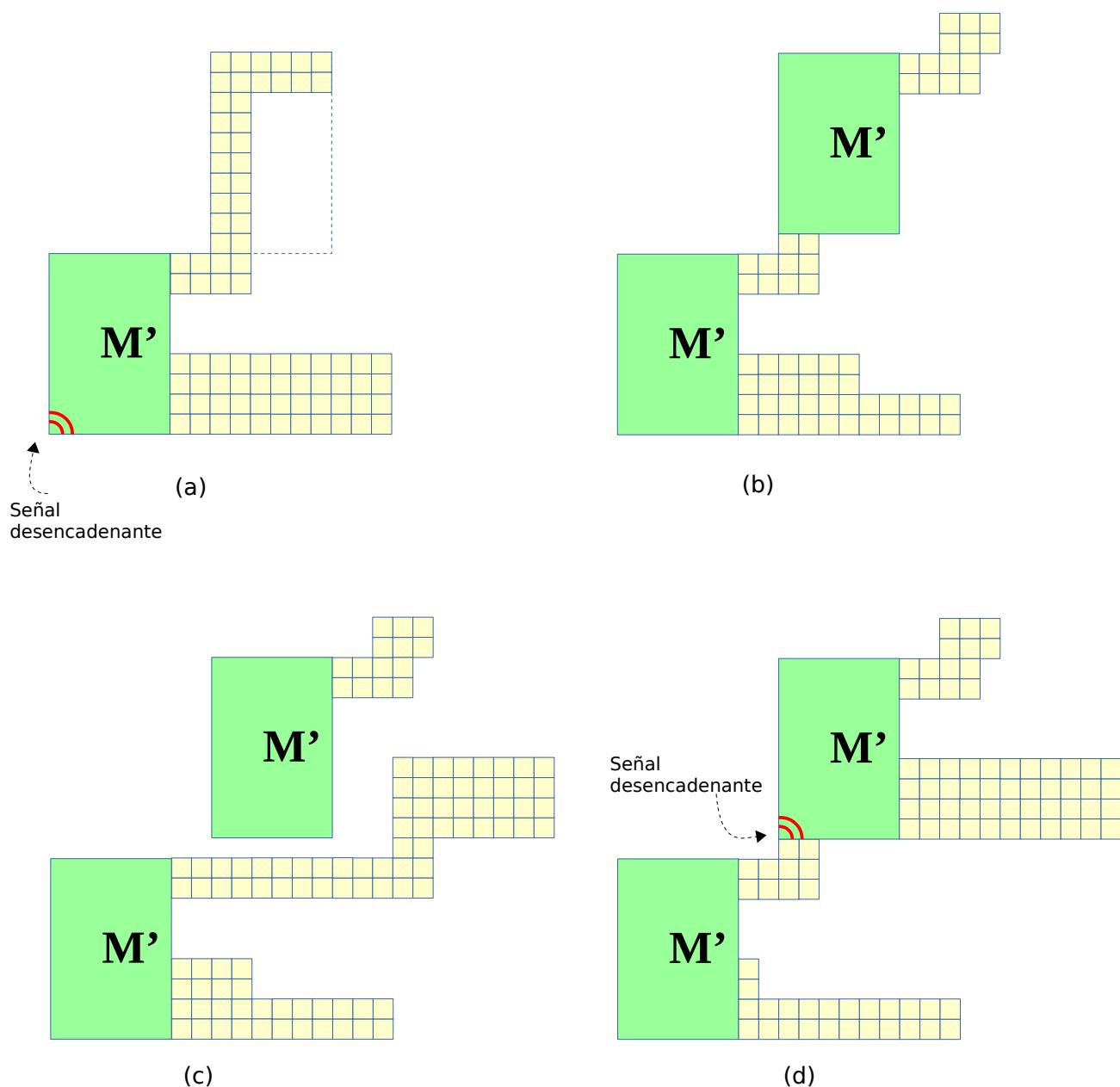


Figura 224: Funcionamiento del constructor universal.

En la figura 224 podemos ver las fases de funcionamiento. En la primera fase (a) el constructor universal M' recibe la señal para ponerse en marcha, lo cual hace que en (b) lea la cinta y saque una copia de sí mismo. En (c) vuelve a leer la cinta y la copia literalmente en la máquina ya creada. Y en (d) envía a la nueva máquina una señal para activarla y que comience ella a sacar su propia copia. Es importante recalcar que la cinta juega dos papeles:

- Su información se interpreta activamente para construir una copia de M' .

- Su información se interpreta pasivamente como datos a ser copiados y anexados a la copia de M' .

Esto es análogo a lo que hace el ADN fabricando proteínas pero también sacando copias de sí mismo. También es similar al proceso de gödelización de la aritmética, donde por un lado las cadenas de símbolos operan para construir teoremas pero, por otro, hablan de sí mismos. Es un caso más de autorreferencia.

El propio von Neumann se dio cuenta de era más eficiente para un objeto sacar una copia de sí mismo a partir de su descripción simbólica (figura 225) que a partir de su introspección material. Es decir, aunque este trabajo sea muy teórico, tiene consecuencias cuando queramos fabricar robots autorreplicantes.

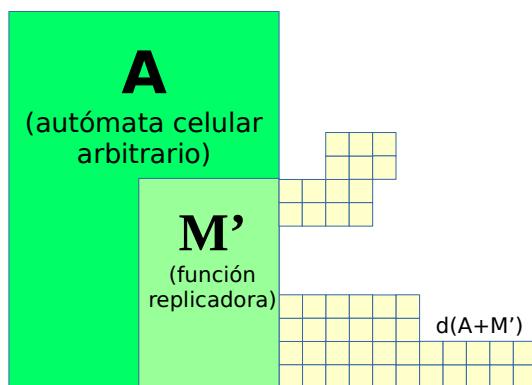


Figura 225: Cómo se logra la copia universal.

Lo que acabamos de ver es solo un esbozo del trabajo que von Neumann realizó en 1949, pues hay muchos más detalles a resolver. Lo importante es que necesitó 40 000 células y 29 estados. Dado que 29 posibilidades necesitan 5 bits para ser codificadas, entonces la cantidad de información requerida para lograr autocopia es de $40\ 000 \times 5 = 200\ 000$ bits de complejidad.

En 1968, Codd necesitó 8 estados y 100 000 000 células. En 1970, Devore las redujo a 87 500. Pero la complejidad medida en bits es similar.

Con ello llegamos a entender otro hito de la emergencia. Para lograr autocopia, y con ella poner en marcha la evolución, se requieren del orden de 2×10^5 bits. ¿Es mucho o es poco? Por un lado, los primeros discos duros que aparecieron en 1971 eran de 80 kilobytes, es decir, más o menos 6×10^6 bits, un orden de magnitud mayor que el número que tenemos entre manos. No menciono la capacidad de almacenamiento de hoy día, pues es muchos órdenes de magnitud mayor. O sea que, desde el punto de vista tecnológico, 2×10^5 bits no son gran cosa.



Personaje 6

VON NEUMANN (1903-1957)

John von Neumann fue un matemático húngaro que trabajó la mayoría de los temas considerados en este libro. En el presente capítulo destacamos su investigación en sistemas autorreplicantes usando autómatas celulares, pero es también uno de los inventores de la teoría de juegos, que veremos en otro capítulo. Además de lo anterior, también realizó aportes valiosos en la creación de la mecánica cuántica, en el diseño de la primera bomba atómica y en el diseño de los primeros computadores. A la arquitectura computacional más común hoy día que tiene CPU, memoria y entradas-salidas usando buses de datos y de memoria separados, se le llama “arquitectura de von Neumann” para distinguirla de las “arquitecturas no-von Neumann”, también de su invención (por lo menos, las primeras, que incluían la noción de varias CPU trabajando en paralelo). El que se conoce como

“segundo teorema de Gödel” realmente es de von Neumann. Posiblemente hiciera sugerencias a Claude Shannon y Alan Turing, en sus respectivos trabajos.

Otro dato curioso es que von Neumann fue un genio pero no el único de su época. En las mismas coordenadas espaciotemporales surgieron otros científicos como Leo Szilard, Edward Teller, Eugene Wigner y Paul Erdős. Les llamaban los húngaros extraterrestres. ¿A qué se debe esta coincidencia? Bueno, ya se ha descubierto lo que tenían en común la mayoría de ellos: el mismo profesor de instituto, Lászlo Rátz, que daba clases muy personalizadas y con lo que hoy llamaríamos metodología constructivista. ¡Hay que invertir más en educación, en buena educación!

Pero, por otro lado, si pensamos en hacer lo mismo que con la capacidad de cómputo universal, la probabilidad de acertar a generar al azar una estructura con la propiedad de autocopiado es terriblemente baja ($1/2^{200000}$), cero a todos los efectos, incluso si consideramos todas las reacciones químicas que se hayan podido dar en los océanos desde el origen de nuestro planeta. El trabajo de von Neumann es meritorio pero no nos ayuda a entender cómo pudo surgir espontáneamente el autocopiado en la naturaleza (o dentro del computador). No obstante, recordemos que, como toda medida de complejidad, lo único que tenemos es una cota superior y quizás se descubra más adelante una forma de reducirla.

También hay que entender que seguramente al principio no se requería una

capacidad de copiado universal, sino simplemente un objeto que tuviera capacidad de autocopiado. Con eso es suficiente para comenzar una tímida evolución, que tendrá que hacer muchos intentos por prueba y error para lograr las primeras mejoras. Es razonable suponer que la capacidad de autocopiado mejore con el tiempo, pues la evolución consiste en eso precisamente. Pero no podemos deducir cuánto tiempo se requiere para que ello ocurra. En este sentido se han hecho trabajos teóricos donde la capacidad de copia es limitada.

En los autómatas 1D de Wolfram, la regla 90 es autorreplicante en el sentido de que si hay una configuración inicial cuya anchura W es tal que $W < 2^n$ (siendo n un número entero), entonces esta regla saca copias de la configuración inicial cada 2^n tics de reloj. En la figura 226 podemos ver un ejemplo, donde el patrón inicial es 1100101, que ocupa $W=7$ celdas siendo $W=7 < 2^3 = 8$. Entonces cada 8 tics de

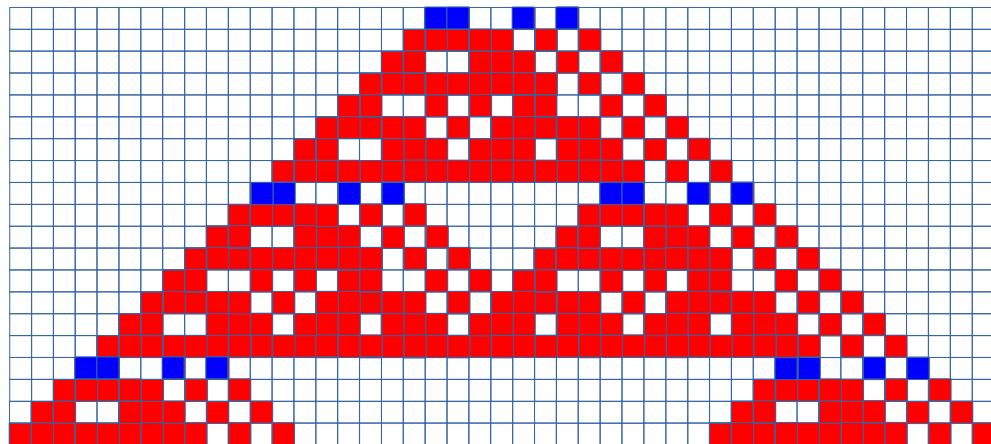


Figura 226: Regla 90 como replicadora.

reloj se repite el patrón, que se ha coloreado distinto para ayudar a verlo.

En los autómatas 2D de von Neumann, la regla *B1357/S1357* es replicante en el sentido de que cualquier configuración inicial que se ponga, termina por aparecer copiada múltiples veces, conforme pasa el tiempo.

Además, hay muchas otras funciones de copia triviales, como por ejemplo la regla *B12345678/* en 2D (figura 227): una celda pasa a viva si tiene a su lado alguna viva, y pasa a muerta en otro caso (muy parecida a la regla 50 en 1D).

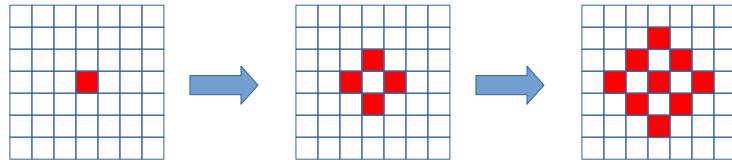


Figura 227: Regla B12345678 de replicación trivial.

Como un ejemplo de sistema de copiado no trivial, pero tampoco universal, está el que diseñó Langton en 1983, basado en una estructura definida por Codd, compuesta por células “camino” encerradas entre dos “cunetas” que guían las señales (figura 228). Cada señal produce un efecto distinto cuando llega al final del camino, modificándolo (añadiendo curva a la derecha, a la izquierda o alargándolo). Cada señal lleva detrás una “estela” que sirve para romper la simetría adelante-atrás, y así recordar la dirección que llevaba.

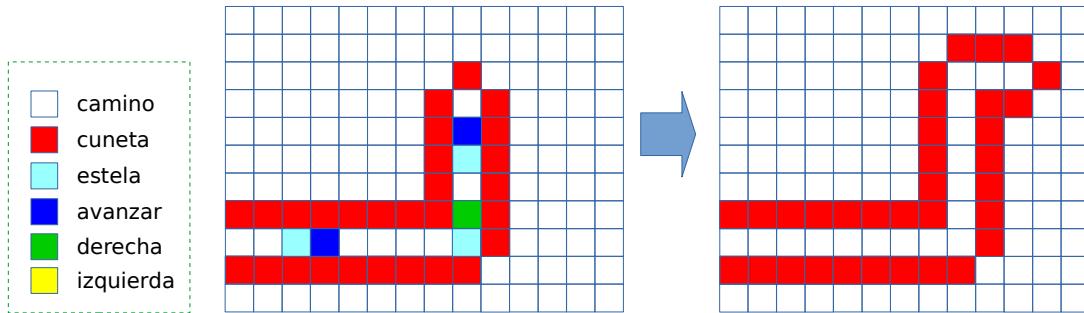


Figura 228: Autómata celular de Langton.

Sin embargo, ocurre el mismo problema de antes: las señales constructoras deben ser almacenadas en una estructura cuyo tamaño es tres veces mayor que el del patrón que se va a construir. Para solucionarlo, Langton creó un camino en bucle cuadrado (figura 229). De esta forma, su descripción requiere 1/4 de señales, respecto al modelo anterior (ya que el cuadrado es 4 veces un giro seguido de un avance). Las señales circulan cuatro veces por el bucle, y en la bifurcación se duplican: una copia sale para afuera mientras la otra sigue

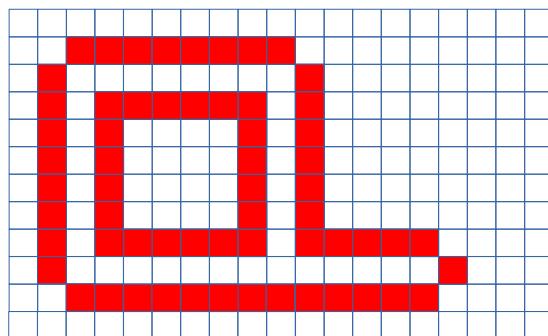


Figura 229: Bucle autorreproductor de Langton.

recirculando. Hubo que añadir estados especiales para regenerar el brazo, pero rotado 90 grados. El resultado se asemeja al crecimiento del coral, en el sentido de que los bucles que van quedando en el centro no se pueden volver a reproducir, y solo está viva la parte exterior (figura 230).

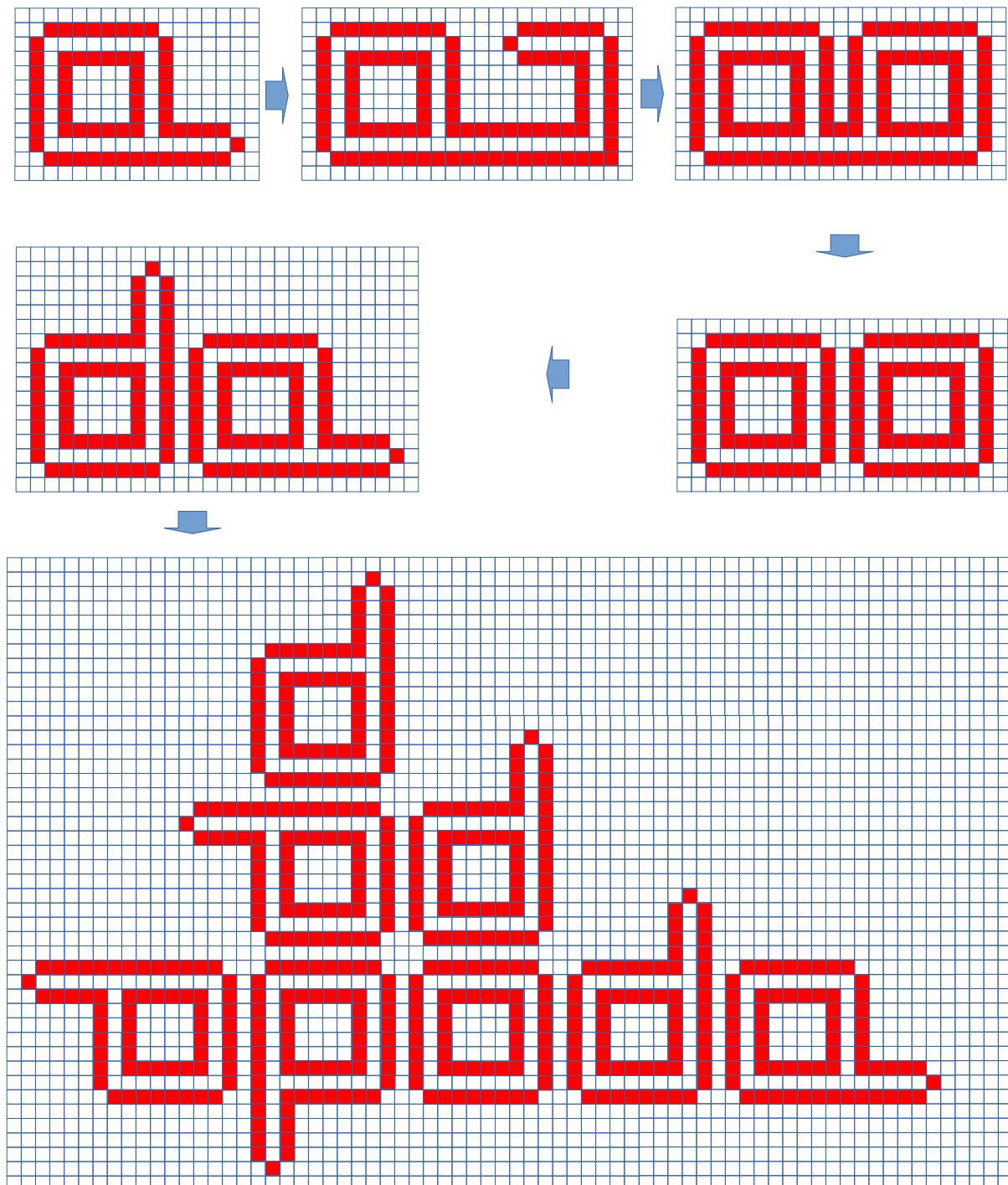


Figura 230: Crecimiento exponencial del bucle de Langton.

Para realizar todo esto Langton necesitó 8 estados y 150 células, bastante menos que von Neumann. Por aquí puede estar la solución al dilema, pues la complejidad requerida es de apenas de $\log_2(8)*150 = 450$ bits. No obstante, sigue siendo un número grande.

Merece la pena aclarar que se pueden hacer muchos sistemas de copia trivial, pero hay que entender cuál es la diferencia con una copia no trivial. Las copias triviales replican dibujos, formas, adornos, o pequeñas colecciones de objetos, aunque sin sus funcionalidades ni relaciones entre ellos. Las copias no triviales también replican las funcionalidades y las relaciones entre esos objetos de modo que, en la copia, siguen siendo objetos activos.

Para lograr copias no triviales, y a la vez universales, el patrón replicante debe contener una descripción de sí mismo que trabaje de dos maneras:

- Activamente, guiando la operación de copia (traducción de la información en acciones).
- Pasivamente, dejándose copiar (transcripción de la información).

Aquí nos damos cuenta que el bucle de Langton no contiene en ningún sitio una descripción de sí mismo, y eso le impide ser universal. Solo sirve para copiarse él mismo. Si quisiéramos copiar una variante de un replicante universal, por ejemplo, procedente de una mutación evolutiva, basta con incluir las modificaciones en la descripción que tenga de sí mismo, lo cual es conceptualmente trivial. Es lo que hace el constructor universal de von Neumann, y también el ADN del mundo biológico. En el bucle de Langton simplemente no se puede.

Pensemos también en la potencia que se genera al combinar los dos hitos básicos. Tener un montón de puertas lógicas y bits de memoria capaces de sacar copias de sí mismas no es gran cosa. Pero tener las mismas puertas y bits con conexiones que les permitan realizar cómputo universal es algo mucho más poderoso porque, al poseer simultáneamente la capacidad de autocopia, habremos dado un salto cuantitativo en complejidad (figura 231). Dispondremos de computadores o robots, cada uno de ellos potencialmente con un algoritmo distinto, capaces de sacar copias de sí mismos. Este es precisamente el formalismo de la computación celular con membranas activas (Paun, 2001), cuya capacidad de cómputo excede a las Máquinas de Turing Universales, pues pueden resolver problemas NP en tiempo polinomial. Todavía no se sabe si es factible una implementación práctica, pues todo va a depender del desarrollo que haya en las máquinas de autorreproducción. Lo que se necesita básicamente es paralelismo

masivo para que, por así decir, cada celda del autómata celular trabaje simultáneamente con las demás. Esto quizás lo logremos por medio de tecnologías que ya están funcionando pero que hay que mejorar, como la computación por ADN, la nanoingeniería o la computación cuántica.

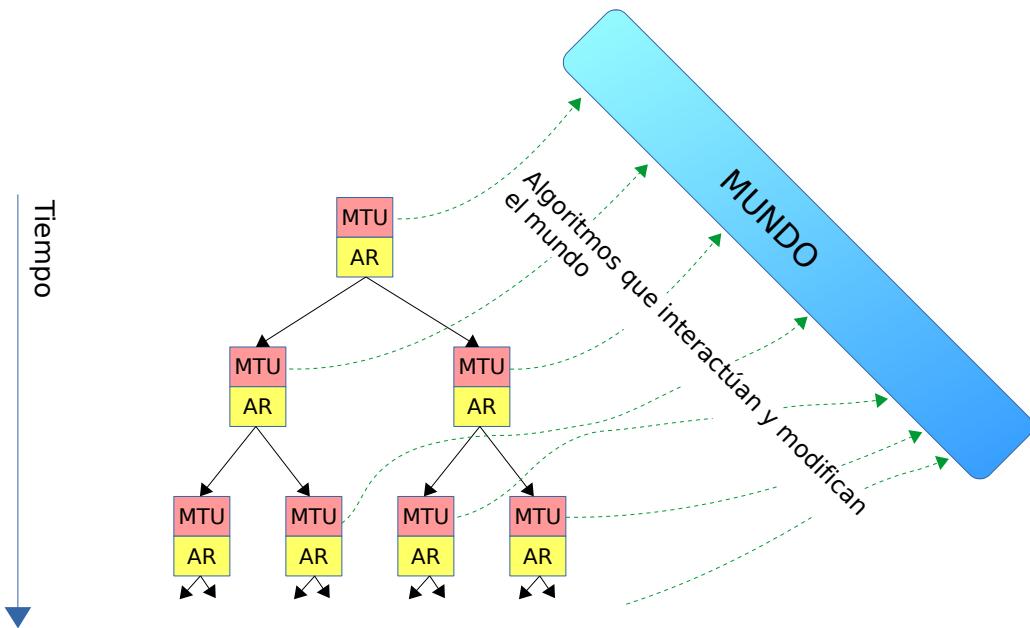


Figura 231: La combinación perfecta: MTU (Máquina de Turing Universal) con AR (autorreplicación).

Resumen

Los autómatas celulares sirven para modelar relaciones espaciales. El cómputo es masivamente distribuido y solo se comunica información de forma local, es decir, entre vecinos. Todas las celdas realizan sincrónicamente el mismo cómputo, que se codifica habitualmente en una regla de transición.

Variando la regla de transición se consiguen autómatas celulares muy simples y predecibles hasta totalmente impredecibles (seudoaleatorios). En un punto intermedio aparece la máxima complejidad (caos) donde el sistema tiene una mezcla de predictibilidad (*gliders*, *eaters*...) e unpredictabilidad. Ello le da al sistema la máxima libertad. Con una de esas reglas se construyó el sistema *LIFE* de dos dimensiones, que tiene capacidad de cómputo universal.

Los autómatas celulares se han utilizado para analizar la complejidad necesaria con el objetivo de alcanzar la capacidad de cómputo universal (es decir, la de una máquina de Turing universal), y resulta ser sorprendentemente baja, del orden de

8 bits. Eso indica que es fácilmente alcanzable por procesos al azar.

Los autómatas celulares también se han utilizado para analizar la complejidad necesaria con el fin de lograr la autocopia universal, que resulta ser del orden de 2×10^5 bits, que resulta ser demasiado alta para que aparezca en la Tierra por mero azar. Este valor es una cota superior, y lo que indica es que hay que hacer más estudios para ver cómo rebajarla.

Para saber más

- **Stephen Wolfram (2002). *A new kind of science*. Canadá: Wolfram Media Inc.**

La evolución no puede explicar satisfactoriamente por qué los sistemas simples interactúan para dar lugar a sistemas con un grado de complejidad ilimitado, antes de que aparecieran las primeras moléculas autorreplicantes. El autor nos muestra (aunque no lo dice explícitamente) que el proceso que lo consigue es matemático (es la computación completa de las máquinas de Turing) y es prácticamente inevitable que aparezca en sistemas generados al azar. El libro no es agradable de leer e incluso es farragoso, pero la información final que transmite es valiosa; merece la pena leer con detenimiento las páginas 1-113, 637-663, 690-691 y 772-846.

- **Leon O. Chua (2006). *A nonlinear dynamics perspective of Wolfram's new kind of science, volume I*. New Jersey, World Scientific.**

Muestra otra manera de implementar autómatas celulares, muy ingeniosa, usando ecuaciones diferenciales sencillas para la dinámica interna de cada celda, que dan un resultado discreto como salida hacia otras celdas. El trabajo de Chua es parecido al de Wolfram, pero anterior.

- **Andrew Ilachinski (2001). *Cellular automata: a discrete universe*. New Jersey: World Scientific.**

Repaso muy sintético (y probablemente con algunos errores) de fractales, grafos, grupos, anillos, campos, gramáticas. Cuenta que Erdős trabajó con grafos estocásticos, investigando la emergencia de nuevas propiedades conforme aumenta el tamaño del grafo. Gramáticas de Chomsky. Hace muchas reflexiones filosóficas y computacionales sobre los autómatas

celulares y el Juego de la Vida, explicando que se pueden usar para modelar cuestiones de Física. Explica varios paquetes de *software* de vida artificial. Hace muchas propuestas interesantes.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

- Adami, Ch. (1998). *Introduction to Artificial Life*. New York: Springer-Verlag.
- Bedau, M. A., McCaskill, J. S., Packard, N. H. y Rasmussen, S. (2000). *Artificial Life VII: Proceedings of the Seventh International Conference on Artificial Life*. Cambridge, Massachusetts: MIT Press.
- Cattaneo, G., Formenti, E., Margara, L. y Mauri, G. (1999). On the Dynamical Behavior of Chaotic Cellular Automata. *Theoretical Computer Science*, 217(1), pp. 31-51.
- Cook, M. (2004). Universality in Elementary Cellular Automata. *Complex Systems*, 15, pp. 1-40.
- Dennett, D. (1999). *La peligrosa idea de Darwin*. Madrid: Círculo de Lectores.
- Emmeche, C. (1998). *Vida simulada en el ordenador*. Barcelona: Gedisa editorial.
- Fernández, J. y Moreno, A. (1992). *Vida Artificial*. Madrid: Eudema S. A.
- Gardner, M. (1983). The Game of Life, Parts I-III. In *Wheels, Life, and other Mathematical Amusements*. New York: W. H. Freeman.
- Murray, J. D. (2002). *Mathematical Biology*. New York: Springer-Verlag.
- Paun, G. (2001). P Systems with Active Membranes: Attacking NP-Complete Problems. *Automata, Languages and Combinatorics*, 6(1), pp. 75-90.
- Penrose, R. (1994). *Las sombras de la mente*. Barcelona: Editorial Grijalbo Mondadori.
- Stewart, I. (1998). *El segundo secreto de la vida*. Barcelona: Colección Drakontos, Editorial Crítica.

Trevorrow, A. y Rokicki, T. (2017). GOLLY. Recuperado el 3 de septiembre de 2017. Disponible en: <http://golly.sourceforge.net/>

Ulam, S. M. (1976). Adventures of a Mathematician. New York: Charles Scribner's Sons.

von Neumann, J. y Burks, A. W. (1966). *Theory of Self-reproducing Automata*. Urbana: University of Illinois Press.

PELÍCULAS Y VIDEOS

Bellos, A. (2016). *Game of Life: Logic Gates*. Recuperado el 10 de octubre de 2016. Disponible en: <https://www.youtube.com/watch?v=vGWGeund3eA>

Bradbury, P. (2016). *Life in Life*. Recuperado el 24 de octubre de 2016. Disponible en: <https://www.youtube.com/watch?v=xP5-ileKXE8>

Heaton, J. (2014). *Finding interesting Cellular Automata by evolving universal constants using a genetic algorithm*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=Vphx4sYcl-o>

Hensel, A. (1999). *Conway's Game of Life*. Recuperado el 3 de septiembre de 2017. Disponible en: <http://www.ibiblio.org/lifepatterns/>

Hutton, T. (2013). *SmoothLifeL*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=KJe9H6qS82I>

pmav.eu. (2010). *Conway's Game of Life*. Recuperado el 2 de septiembre de 2017. Disponible en: <http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/>

Scientist, N. (2010). *Self-replicating machine*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=ZX-iJLHZt8M>

Scientist, N. (2008). *Modular robot reassembles when kicked apart*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=uIn-sMq8-Ls>

Southwell, R. (2013). *The Universe of 3D Cellular Automata*. Recuperado el 2 de septiembre de 2017. Disponible en: <https://www.youtube.com/watch?v=OxASD5xvgKI>

Zykov, V., Mytilinaios, E., Adams, B. y Lipson, H. (2009). *Self-replicating blocks from Cornell University*. Recuperado el 5 de febrero de 2010. Disponible en:

<https://www.youtube.com/watch?v=gZwTcLeelAY>

TESIS Y TRABAJOS DE GRADO EN EVALAB

Muñoz, L. E. (2016). *Herramienta para Modelar la Accesibilidad Peatonal al interior del Campus de la Universidad del Valle con Autómatas Celulares sobre una Plataforma SIG*. [Tesis Maestría]. Cali: Universidad del Valle.

SOLUCIÓN A LOS PROBLEMAS DE INGENIO

Lástima que hayas tenido que llegar hasta aquí. Pero no te preocupes, que no se lo diremos a nadie.

Torres de Hanoi

Podemos representar el estado del juego con un vector de N posiciones. La posición i -ésima representa la columna $\{1,2,3\}$ en la que se encuentra la anilla de radio i . Por ejemplo, para $N=4$ anillas el estado inicial es $(1,1,1,1)$, el estado objetivo final es $(3,3,3,3)$, y en la figura 232 también se representa el estado $(2,2,1,3)$.

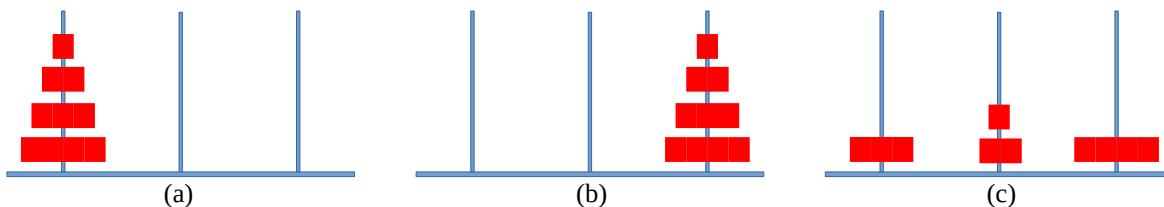


Figura 232: Juego de las Torres de Hanoi: (a) posición inicial; (b) posición final; (c) una posición intermedia válida.

Entonces, el algoritmo que nos lleva de la posición inicial a la final se puede obtener del grafo de la figura 233 (se ha dibujado solo con $N=3$ anillas), que representa los estados válidos y sus transiciones. Este grafo es similar al fractal de Sierpinski.

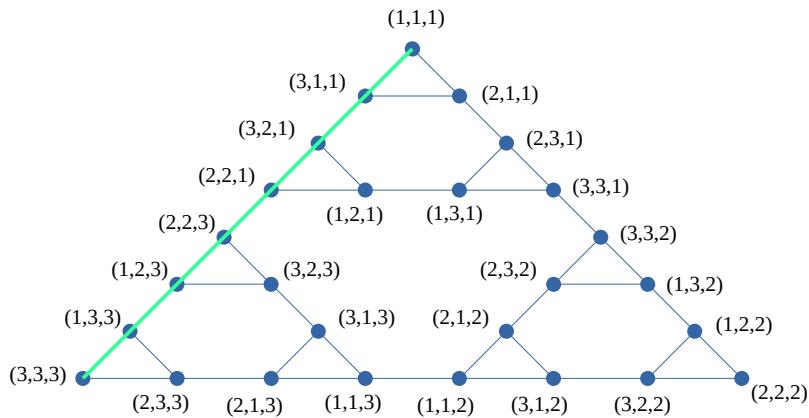


Figura 233: Diagrama de estados de las Torres de Hanoi. En color verde está la solución. El espacio de movimientos es análogo al fractal de Sierpinski.

Y resulta que los genes están organizados también así (unos genes regulan la expresión de otros), de modo que no debería sorprender que aparezcan fractales en la naturaleza, como veremos más adelante. Además, la naturaleza logra así fabricar cuerpos complejos a partir de especificaciones genéticas simples.

Dinero de bolsillo

Vamos a solucionar este problema eliminando las ambigüedades del lenguaje hablado y, para ello, nada mejor que plantear la matriz de pagos, en la figura 234, suponiendo —sin pérdida de generalidad— que cada uno puede llevar dinero por valor de 100, 200 o 300. Allí se pueden descartar las filas y columnas dominadas, con lo que se ve que la estrategia óptima para cada uno es llevar lo menos posible, en este caso 100, con lo cual quedarán empatados.

De donde se deduce que conviene llevar lo menos posible. La próxima vez que vayas a jugar este juego, no lleves nada de dinero en los bolsillos.

		Luisita			
		100	200	300	
Juanito		100	0	200	300
		200	-200	0	300
300		300	-300	-300	0

Figura 234: Matriz de pagos para dinero de bolsillo.

Nave espacial

Hay muchas formas de hacer naves espaciales (*spaceships*) que se muevan horizontalmente. La más pequeña de todas la encuentras en la figura 235 y es una variante de la mostrada en el enunciado del problema. Requiere 4 tics de reloj para avanzar dos celdas.

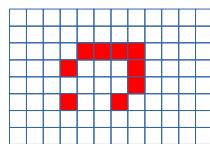


Figura 235: El spaceship de movimiento horizontal más pequeño que existe.

Velocidad de la luz

La teoría es correcta: la velocidad máxima teórica a la que puede viajar un objeto es una celda por tic de reloj. Sin embargo, durante algún tiempo no se encontró cómo hacerlo y se pensó que el *glider* era el objeto que podía viajar más rápido en el autómata celular *LIFE* en forma diagonal (una celda cada 4 tics) y el *spaceship* de la figura 235 en forma horizontal (dos celdas cada 4 tics). Después de todo, son los patrones más pequeños que se mueven sin destruirse. Además, patrones más grandes suelen requerir más tiempo para avanzar. Sin embargo, ya se conocen estructuras que alcanzan la máxima velocidad posible de 1 celda por tic y el truco está en que no se propagan en el “vacío” sino que requieren un cable previamente instalado. En la figura 236 podemos ver el cable y una señal desplazándose de derecha a izquierda. Está incompleta, pues hay que añadir un poco más de infraestructura para que el cable no se destruya en el proceso.

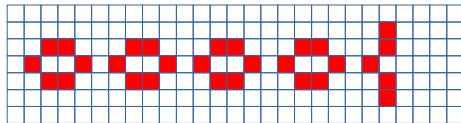


Figura 236: Patrón lightspeed-telegraph.

Y en la figura 237 podemos ver otra señal desplazándose de izquierda a derecha por el cable horizontal central.

Al respecto hay también una especie de broma, con un *spaceship* del que la gente afirma que se mueve más rápido que la luz. El patrón se llama *stargate*. Obviamente, no supera la velocidad de la luz, pero el truco es bonito y se basa en construir (o no) otro *spaceship* adelantado, en función de si llega (o no) el original. Búscalos en *GOLLY*.

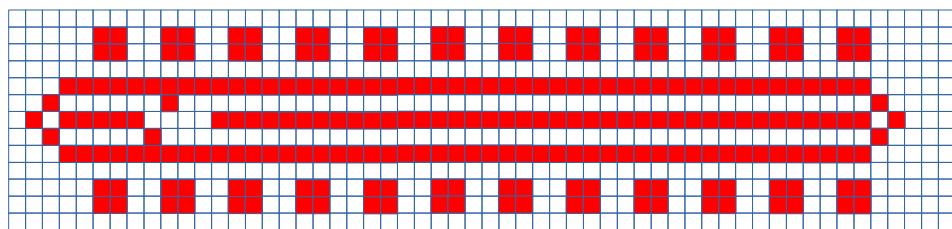


Figura 237: Patrón lightspeed-wire.

Bandera francesa

El *software* completo lo realizó el ingeniero Eider Falla, mientras era estudiante nuestro en la Universidad del Valle. Nos lo ha cedido amablemente escrito en Java y en tres versiones, de modo que si quieras ver cómo funciona, se encuentra publicado en <https://github.com/angarciaiba/libroVA> . Aquí vamos solamente a dar un esbozo de cómo hacerlo.

Definimos los estados básicos:

- Q: estado quiescente del entorno
- I: estado inicial del segmento
- P: perturbación inicial
- R: rojo
- B: blanco

- A: azul

Se parte de un segmento contiguo de células en estado I, con un extremo en estado P y rodeadas de todas las demás células en estado Q.

La función de transición de estados es sencilla, pero larguísima, concretamente muchos condicionales:

```
if estado == 'P' and celdalzquierda == 'Q' then estado = 'A'
```

```
if estado == 'I' and celdalzquierda == 'A' then estado = 'A'
```

Con estas dos líneas se logra que la perturbación inicial propague el color azul hacia la derecha. Hay que propagar más cosas, claro. Para ello se introducen estados adicionales que ayudan a propagar el color:

- Z: frente azul. Velocidad = 1 célula por ciclo
- N: frente blanco. Velocidad = 1 célula por cada 2 ciclos
- J: frente rojo. Velocidad = 1 célula por cada 5 ciclos
- E: estabilizador. Velocidad = 1 célula por ciclo

Z, N y J se generan a partir de la ocurrencia de P.

Para lograr la velocidad requerida, se desdobra N en {N1, N0}; y J en {J4, J3, J2, J1, J0}, que actuarán a manera de contadores. Y la señal E se genera en dirección contraria, en el momento en que el frente de onda Z llegue al extremo opuesto. Su misión es estabilizar N y J, como puede verse en la figura 238.

Queda por solucionar cómo lograr que se restaure la bandera cuando se rompa el segmento. Para ello hay que introducir más estados:

- TR: marcará el extremo rojo en el momento de salir la onda J.
- TA: marcará el extremo azul en el momento de salir la onda E.

Con estos estados se aísla el segmento de las células Q. Cuando el segmento se rompe, habrá dos células {R, B o A} en contacto con Q, y esta condición servirá para disparar el estado P en esas células y volver a empezar.

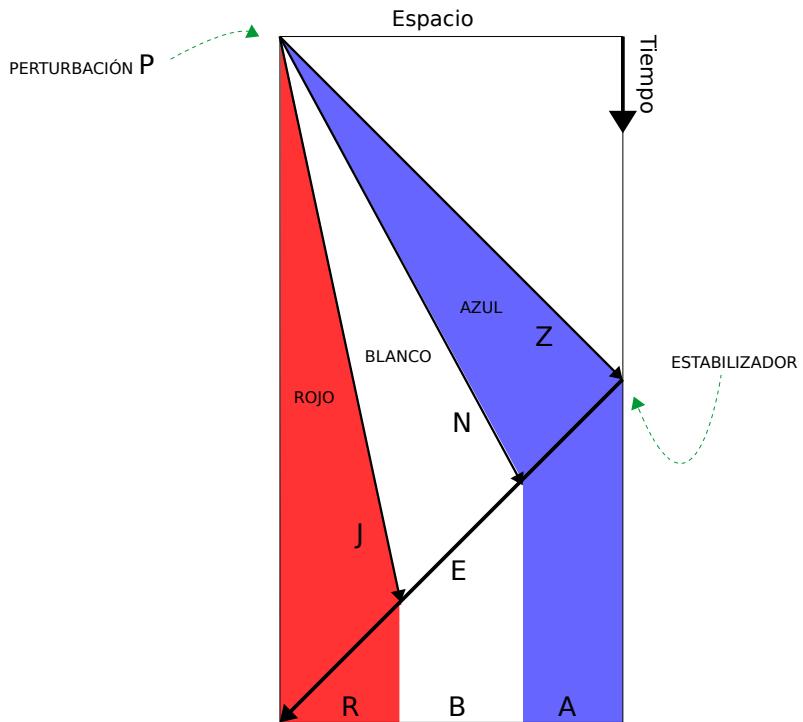


Figura 238: Señales que deben transmitirse en la bandera francesa.

Para respetar la polaridad de la bandera, si el frente azul llega al extremo TA, sabe que la orientación es la correcta. Mientras que, si llega al extremo TR, sabe que debe volver a empezar, cambiando TR por P para reiniciar el proceso en el extremo correcto.

Este problema es muy interesante porque simula varios fenómenos biológicos:

- La diferenciación de las células a partir de una única germinal.
- La distribución de tareas en una colonia de hormigas: un porcentaje fijo de hormigas se dedica a cada tarea, aunque se suprima una parte de la colonia. Piense que no hay ninguna hormiga contando a las demás para distribuirlas en grupos.
- La regeneración de una estrella de mar o una hidra, cuando sufren un corte. No hay un control centralizado que decida como volver a hacer crecer un miembro roto. Solo hay comunicación local entre células.
- La coloración de peces y otros animales como cebras, guepardos, tigres y un sinnúmero más. De nuevo el cerebro no se encarga de decidir el color de cada célula de la piel, pues sería un trabajo terrible. Ni como estirar las manchas conforme el animal va creciendo. Cada célula decide de qué color ponerse en función de lo que hagan sus vecinas. En muchos casos salen

dibujos fractales o similares a los autómatas celulares 1D como la regla 30 (figura 239). Aunque en otros casos proceden de un proceso evolutivo, como las mariposas que parece que tienen unos ojos pintados en las alas, con lo cual asustan a sus depredadores y sobreviven más, dejando más hijos parecidos a ellas.



Figura 239: El molusco *Conus Textile*.

Fuente: Richard Ling (richard@research.canon.com.au). Lugar: Cod Hole, Great Barrier Reef, Australia, CC BY-SA 3.0. Disponible en: <https://commons.wikimedia.org/w/index.php?curid=293495>

Los algoritmos distribuidos como estos son muy interesantes en vida artificial, física digital e inteligencia artificial. Dado que no hay un control central, es complicado diseñar algoritmos para esta arquitectura. A pesar de ello hay aportes interesantes en votaciones mayoritarias y sincronización de eventos (Cristian, 1989).

Para saber más

La mayoría de los autores de estos problemas son anónimos. Me los contaron frente a unas cañas o proceden de revistas de entretenimiento donde no aparecen referencias.

Los problemas de ingenio son un pasatiempo divertido, despiertan nuestra creatividad, sirven para aumentar nuestro coeficiente intelectual y mantienen joven nuestro cerebro. Los seleccionados aquí, demás, iluminan algunos de los temas que se abordan en este libro.

A continuación indico cuatro libros donde hay recopilaciones de estos problemas.

- **Juan José Rivera Gómez (1981). Comecocos I. Madrid: Editorial Álamo.**

Un libro muy viejo con muchos problemas de ingenio, la mayoría sencillos pero contiene también algunas joyas como “La mayor toca el piano”, que te propongo para que lo resuelvas y te anime a leer el libro completo: dos amigos matemáticos se encuentran por la calle después de mucho tiempo sin verse y se cuentan sus vidas.

—Tengo 3 hijas.

—¿De qué edades?

—Es muy sencillo. El producto de sus edades es 36.

—Necesito más datos.

—Si sumas sus edades te da el número de la calle en la que estamos.

El amigo mira hacia arriba, al cartel con el número de la calle y exclama: ¡con eso tampoco es suficiente!

—Tienes razón. Te daré una última pista: la mayor toca el piano.

—¡Ah! Ahora sí. Ya sé sus edades.

¿Cuáles son las edades de las tres hijas?

- **Lewis Carroll (1979). Matemática demente. Barcelona: Tusquets Editores.**

Del mismo autor de *Alicia a través del espejo*. Contiene un cuento espectacular, “Lo que le dijo la tortuga a Aquiles”, que demuestra que incluso para aplicar la lógica se necesita sentido común. Si hay reglas, se requieren metaregulas, para que las primeras no sean abusadas. Y hay luego unos pocos problemas de ingenio que hacen reflexionar sobre cosas

cotidianas como: ¿Por qué los espejos invierten izquierda y derecha, pero no arriba y abajo?

- **Martin Gardner (1992). *Inspiración Ajá*. Barcelona: Editorial Labor.**

Este libro es técnicamente el mejor, el más complejo y que toca muchos aspectos de las matemáticas, no solo la lógica.

- **Raymond Smullyan (1978). *¿Cómo se llama este libro? El enigma de Drácula y otros pasatiempos lógicos*. Madrid: Prentice Hall.**

Este libro también es muy bueno, por el tipo de preguntas que tienen respuesta lógica a pesar de que el escenario es muy desfavorable (personas que a veces mienten pero a veces no, personas que se olvidan de quienes son y cosas así).

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, 3(3), pp. 146-158. DOI: <https://doi.org/10.1007/BF01784024>

INTRODUCCIÓN A RUBY

"To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be"

Anónimo

"Si pones mil millones de micos tecleando en computadores, eventualmente uno escribirá un programa en Ruby. Todos los demás lo harán en Perl"

Anónimo

"Llevo usando vim desde hace dos años, principalmente porque no logro dar con el comando para salir"

Anónimo

No voy a comenzar aquí una soflama a favor o en contra de lenguajes de programación. Cada lenguaje es el resultado de un proceso histórico, unas ventajas en ciertos contextos, y unas facilidades que ofrece y promesas que hace (sí, también interviene aquí la publicidad y las modas). Hay lenguajes que son muy rápidos en ejecución, hay lenguajes que están pensados para que los programas puedan ser verificados con facilidad, hay lenguajes para la web, para ambientes empresariales, para juegos, fáciles de escribir, fáciles de leer, entre otros muchos. No suele existir un lenguaje bueno en todo a la vez.

Quiero presentar a Ruby como un lenguaje extremadamente lento en ejecución, que soporta los paradigmas procedural, orientado a objetos y funcional, y donde es muy fácil y rápido expresar ideas complejas. Es un lenguaje bueno para aprender a programar (si nos limitamos al paradigma procedural⁵⁹⁾) y es un lenguaje muy bueno para llevar a cabo pruebas con nuestras ideas complejas acerca de la complejidad de la vida artificial.

⁵⁹ Ruby es orientado a objetos (OO) puro, pero es muy fácil “engañoso” al estudiante que se inicia en la programación y no mostrarle nada OO, sino fingir que sus construcciones son procedurales (funciones, *if-else for, while*). La sintaxis de Ruby es tan limpia que el estudiante no nota que está programando dentro de la clase *Object*.

La migración desde cualquier lenguaje de programación orientado a objetos (C++, Java, Python, Perl...) es inmediata, aunque como todo en la vida, para adquirir mucha destreza se requiere tiempo. Pero comprender los conceptos no requiere esfuerzo. Y una vez que te hayas cambiado a este lenguaje, no querrás regresar.

La instalación de Ruby sobre Linux es como cualquier otro paquete:

```
sudo apt-get install ruby
```

Pero si vas a trabajar profesionalmente con este lenguaje, es recomendable hacer la instalación desde un manejador de versiones como **rvm**. De este modo puedes tener simultáneamente varios proyectos, cada uno con su propia versión del intérprete de Ruby y de sus gemas (las librerías), sin que interfieran unos con otros.

Para ejecutar un programa que hayas escrito en Ruby debes guardarlo en un archivo con extensión rb y luego dar el comando:

```
ruby archivo.rb
```

Similitudes entre Ruby y Python

Ruby tiene muchos aspectos sofisticados, pero cuando uno quiere aprender a programar (bajo el paradigma procedural) es casi idéntico a Python, solo que más sencillo.

- Las variables tienen tipado automático, como en Python.
- Hay un intérprete interactivo *irb* pero no lo recomiendo usar (por las mismas razones que en Python: al estudiante le confunde que las variables de pruebas anteriores siguen estando ahí).
- Las palabras clave para definiciones de funciones (*def*), estructuras condicionales (*if-else-elsif*) estructuras de repetición (*for in, while*) son las mismas.

Diferencias entre Ruby y Python, a este nivel

- No hay que poner dos puntos en ningún sitio.
- La indentación es recomendable, por estética y estilo, pero no tiene consecuencias semánticas. Es incómodo y hasta peligroso en *Python* que haya símbolos invisibles (el espacio y el tabulador) con consecuencias semánticas.
- Los bloques de código, como funciones, clases, bucles y condicionales terminan en *end* y pueden ir en una sola línea o en varias líneas. Si se desea poner todo en la misma línea se suele delimitar el bloque de código entre llaves { }

5.times do

p "Hola"

end

O bien:

5.times { p "Hola" }

Comentarios

Comentarios en una línea

=begin

Comentarios

que ocupan

varias líneas

=end

Funciones

Declaración de funciones:

```
def mayor(a, b)
```

```
    if a > b
```

```
        return a
```

```
    else
```

```
        return b
```

```
    end
```

```
end
```

Uso de funciones:

```
p mayor(3, 5)
```

Las funciones pueden retornar varios valores separados por comas:

```
def ordenar(a, b)
```

```
    if a > b
```

```
        return b, a
```

```
    else
```

```
        return a, b
```

```
    end
```

```
end
```

Y los resultados de estas funciones se recogen así:

```
elMenor, elMayor = ordenar(8, 5)
```

```
p elMenor
```

```
p elMayor
```

Paso de argumentos a funciones

Como en muchos lenguajes modernos es:

- Argumentos de entrada: paso por valor cuando los argumentos son simples (números enteros y flotantes).
- Argumentos de entrada-salida: paso por referencia cuando los argumentos son complejos (strings, arrays, estructuras de datos...).

La explicación técnica es más complicada: siempre se pasan “referencias por valor”, y los objetos enteros y flotantes son inmutables.

Ejemplos con argumentos simples:

```
def duplicar(numero)
```

```
    return numero*2
```

```
end
```

```
def triplicar(valor)
```

```
    valor = valor*3 # Se cambia el valor, pero eso no afecta a dato
```

```
    return valor
```

```
end
```

```
dato = 5  
  
resultado = duplicar(dato)  
  
otroResultado = triplicar(dato)  
  
p otroResultado # vale 15  
  
p resultado # vale 10  
  
p dato # sigue valiendo 5
```

Ejemplos con argumentos complejos

```
def modificar(array)  
  
    array[2] = "Hasta la vista"  
  
end
```

```
saludos = ["Hola", "Hello", "Hi", "Olá"]  
  
p saludos  
  
modificar(saludos)  
  
p saludos
```

Bucles

Las construcciones de control son las mismas, pero los rangos se definen con el operador dos puntos (valor final incluido) o tres puntos (valor final excluido). Ejemplos:

```
p "valor final incluido"  
  
for indice in 0..10
```

p indice

end

p "valor final excluido"

for indice in 0...10

p indice

end

contador=5

while contador< 10

p contador

contador += 1

end

No hay bucle do-while. En su lugar se hace lo siguiente:

loop

sentencias internas del bucle

break if condicion

end

Verdadero y falso

- Las palabras clave son *true* y *false*.

- *nil* también vale *false* (*nil* es el valor que toma un elemento inexistente dentro de una estructura de datos).
- Y cualquier cosa distinta de *false* es *true*.

Ejemplos:

verdad = 5>2

p verdad

mentira = 4==3

p mentira

datos = []

datos[3] = 100

p datos

datos = [9, 25, 14, 6]

contador = 0

while item=datos[contador] # Ojo: asignación

p item

contador += 1

end

No existe el operador incremento (*contador++*) o decremento (*contador--*) como en *C, C++ o Java*.

Condicionales

Las palabras clave son: *if elif else end* como en casi todos los lenguajes imperativos. Ejemplo:

```
ahorros=20_000_001    # El _ es solo una cosa estética  
  
if ahorros > 20_000_000  
  
    p "Puedo comprar un carro nuevo"  
  
elif ahorros > 10_000_000  
  
    p "Puedo comprar un carro usado"  
  
else  
  
    p "Me toca ir en el Masivo"  
  
end
```

Otra forma del condicional

Si la sentencia a ejecutar dentro de un *if* es solo una, se puede poner de esta otra manera más compacta y elegante: *sentencia if condición*

Por ejemplo:

```
if a < 0  
  
    a = -a  
  
end
```

Se puede poner de esta forma:

$a = -a \quad \text{if } a < 0$

Existe el *switch-case* de C, C++ y Java, pero se llama *case-when*, y es mucho más flexible y sofisticado, puesto que retorna un valor:

color="amarillo"

codigo =

case color

when "azul"

1

when "rojo"

2

when "amarillo"

3

else

0

end

Operadores lógicos

Existen las operaciones lógicas igual que en otros lenguajes populares: `&&`, `||` y `!`

`a,b=3,4`

`p "Error" if (a> 0 || b+a < 5) && a*b==12`

`p "Error" if !c`

También existen las palabras clave `and` o `or` y `not`, pero tienen la más baja prelación (menor que la del operador asignación), por lo que siempre deben encerrarse entre paréntesis externos que agrupen todo lo que se va a evaluar con estos operadores:

`p "Error" if ((a> 0 or b+a < 5) and a*b==12)`

`p "Error" if (not c)`

Arrays

Los *arrays* son dinámicos y con sintaxis muy sencilla y natural (al contrario que en *Python*):

```
letras = ["A", "B", "C"]
```

```
letras = letras << "D"      # Añadir al final
```

```
p letras
```

```
p letras[2]
```

```
letras[3] = "Hola"
```

```
p letras
```

```
letras[10] = "Chao"
```

```
p letras
```

```
p letras[8]
```

```
p letras[333]
```

```
p letras[-1]
```

```
p letras[-9]
```

Índices de los arrays

Los *arrays* no disparan excepciones con facilidad. Acabamos de ver que podemos salirnos fuera del *array* y los items no existentes los toma como *nil*. Se admiten también índices negativos, que significan

- -1 el último item
- -2 el penúltimo item
- y así sucesivamente.

Arrays de dos dimensiones

Los *arrays* de 2 o más dimensiones son triviales de hacer sabiendo que un elemento de un *array* puede ser otro *array*:

```
tablaMultiplicar = []          # Array vacío  
  
for fila in 0..10  
  
    tablaMultiplicar[fila] = []  # Cada elemento es otro array  
  
    for columna in 0..10  
  
        tablaMultiplicar[fila][columna] = fila * columna  
  
    end  
  
end  
  
p tablaMultiplicar
```

Biblioteca estándar

Ruby cuenta con una biblioteca estándar muy rica y con un *API* muy uniforme (sin sorpresas), y es mejor consultar un libro de referencia como (Thomas, 2005), para conocer cada detalle. Por ejemplo, para averiguar el tamaño de un *Array* (o de cualquier otro contenedor) se hace así:

```
a = [ [6, 5], [2, -5, 9, 8], [9] ]
```

```
p a.length
```

```
p a[0].length
```

```
p a[1].length
```

```
p a[2].length
```

Cómo imprimir en la pantalla

a = "Hola"

print a # No salta de línea al finalizar

print(a) # Es lo mismo que lo anterior

puts a # Sí salta de línea al finalizar

puts(a) # Es lo mismo que lo anterior

p a # Es lo mismo que lo anterior

Interpolación

Se pueden insertar variables dentro de un *string* con el operador `#{}{}` que lo que hace es ejecutar la expresión encerrada dentro de las llaves.

a = 5.5

b = 7

*c = "Si multiplico #{a} por #{b} me da #{a*b}"*

p c

Lo habitual es interpolar al imprimir:

*p "Si multiplico #\{a\} por #\{b\} me da #\{a*b\}"*

Cómo leer el teclado

Siempre lee *strings* que luego se pueden convertir a números enteros o flotantes.

a = gets

p a

Cómo convertir un string a entero, a flotante o a string

p "Introduce un número"

a = gets

a = a.to_i

p a

p "Introduce otro número"

b = gets

b = b.to_f

p b

c = b.to_s

p c

Ruby es OO puro

Todo, absolutamente todo en Ruby, es un objeto. Eso significa que todo es manipulable. Por ejemplo, los números son objetos y tienen un *API*:

```
-1.abs
```

Las clases también son objetos y tienen un *API* que las permite ser interrogadas y modificadas en tiempo de ejecución.

Ruby es un lenguaje orientado a objetos a la vez que funcional. Las funciones se pueden pasar como argumentos a otras funciones. Existen las funciones anónimas y la función *lambda*.

Los paréntesis no son necesarios, salvo para forzar prelaciones:

```
a.multiplicar_por 5
```

Toda construcción retorna un valor, que es el último valor calculado. Ejemplo:

```
a = if 5 < 7 then 3 else 4 end # a valdrá 3
```

Por ello el *return* se usa pocas veces. Ejemplo:

```
def duplicar(a)
```

```
  2*a
```

```
end
```

Iteradores

Ruby incluye iteradores en el lenguaje. Esta es la construcción que más confunde a las personas que ven *Ruby* por primera vez, pero es muy sencilla y potente. En *Ruby* no se suelen usar los bucles *for* para explorar *arrays*, sino iteradores:

Ejemplo sin iteradores:

```
a=[4, 2, -5, 3, 7]
```

```
for i in 0...a.length
```

```
item = a[i]
```

```
p item
```

```
end
```

Ejemplo con iteradores:

```
a.each do |item|
```

```
p item
```

```
end
```

O más abreviado:

```
a.each { |item| p item }
```

Hay iteradores que sirven para acumular:

```
a.inject(0) { |acumulado, item| acumulado + item }
```

each es un iterador que existe en el *API* de cualquier contenedor, y que explora uno a uno cada ítem y me lo entrega en el argumento que yo le ponga encerrado entre llaves verticales. El iterador *each* recibe como argumento una función que yo le doy encerrada entre llaves {} (recordemos que *Ruby* es también un lenguaje funcional). En esa función, recibo el (o los) argumentos encerrados entre las barras verticales y hago con ellos lo que desee, en este caso, imprimirllos.

Hay muchos iteradores estándar (*each*, *collect*, *inject*, *delete_if...*) y yo puedo crear otros si lo necesito.

Los iteradores hacen la programación muy elegante:

- Eliminan índices absurdos *i,j,k* que no tienen nada que ver con el algoritmo

si lo piensan bien, aunque nos hemos acostumbrado tanto a ellos que creemos que sí.

- Eliminan la necesidad de averiguar el valor inicial y el final de esos índices (con los consecuentes errores si me equivoco al hacerlo).
- El código resultante es mucho más corto y limpio.

Ruby incorpora automáticamente el polimorfismo, sin tener que hacer nada especial. No hace falta definir árboles de herencia, ni clases abstractas, ni *templates*, ni nada de nada. Eso es gracias a que las colecciones de objetos pueden contener objetos de cualquier clase.

Por ejemplo, suponiendo que las clases *Integer*, *Float*, *Array* y *Complex* tuvieran una función llamada *duplicar()* que multiplique por 2 el respectivo dato, entonces esto funciona sin problemas:

i = 5

f = 4.8

a = [3, 2, 10.1, 6.6]

c = Complex.new(2, -7)

[i, f, a, c].each { |item| item.duplicar }

Clases y objetos

Las clases se definen con la palabra *class* y terminan en *end*. Dentro están sus funciones, que se definen con *def* y terminan en *end*. El constructor siempre se llama *initialize*. Por convenio, el nombre de las clases comienza con letra mayúscula, mientras que el de los objetos comienza con letra minúscula.

```
class CuentaBancaria

  def initialize(valorInicial)
    @ahorros = valorInicial

  end
```

```

def ingresar(valor)

    @ahorros = @ahorros + valor

end

def retirar(valor)

    @ahorros = @ahorros - valor

end

def saldo

    @ahorros

end

end

```

Y los objetos se crean con el operador *new*, que llama al constructor de la clase.

```

miCuenta = CuentaBancaria.new(200)

miCuenta.ingresar(500)

miCuenta.retirar(100)

p miCuenta.saldo      # imprimirá 600

```

Las variables que comiencen por @ son atributos de objeto y las que comienzan por @@ son atributos de clase. Se crean en el momento en que se usan por primera vez, es decir, no hay que declararlas previamente. Las variables que no llevan ningún símbolo delante son variables locales. Las variables que comienzan por \$ son variables globales, aunque, como todo el mundo sabe, no conviene usarlas pues producen código no reentrant (que no funciona cuando hay varios hilos).

Se pueden crear automáticamente las funciones *getter* y *setter* para los atributos usando las palabras clave: *attr_reader*, *attr_writer*, *attr_accessor*

Por ejemplo, en vez de la función saldo del ejemplo anterior podríamos poner:

```

class CuentaBancaria

attr_reader :ahorros

def initialize(valorInicial)
    @ahorros = valorInicial
end

def ingresar(valor)
    @ahorros = @ahorros + valor
end

def retirar(valor)
    @ahorros = @ahorros - valor
end

```

Que se puede usar así:

```

miCuenta = CuentaBancaria.new(200)

miCuenta.ingresar(500)

miCuenta.retirar(100)

p miCuenta.ahorros      # Función getter. Imprimirá 600

```

Si se hubiera definido el atributo ahorros de tipo lectura-escritura:

```
attr_accesor :ahorros
```

Entonces también se podría escribir directamente así:

```
miCuenta.ahorros = 900  # Función setter
```

Las clases son también objetos y tienen un *API* por medio del cual pueden ser interrogadas y modificadas, permitiendo la metaprogramación. Estos *getter* y

setter son realmente ejemplos de metaprogramación.

Composición de objetos

- La herencia existe pero no es muy importante.
- Son más importantes los *mixins*, una especie de “herencia horizontal”. Responde a lo que hace años se llamaba “programación orientada a aspectos” y permite incluir en una clase los aspectos más relevantes que ya existen en otras clases. Por ejemplo, todas las colecciones de objetos (*Arrays*, *Hashes*, *Sets*...) incluyen el *mixin Enumerable*, que a su vez arrastra muchos iteradores como *each* e *inject*.

Metaprogramación

Ruby permite la metaprogramación muy fácil y ordenadamente, es decir, escribir programas que escriban otros programas. O que se automodifiquen. Eso sirve para:

- Crear nuevos idiomas dentro del lenguaje (nuevas sentencias como *attr_reader*, *attr_writer*, *attr_accesor*).
- Evitar la escritura de código repetitivo.
- Conectarse y manipular automáticamente bases de datos, como se hace al heredar de *ActiveRecord*, y entonces todos los atributos de la clase se mapean en una tabla, los objetos son filas, los atributos son columnas, y se logra automáticamente la persistencia.
- Mapearse automáticamente a *interfaces web* (como se hace en *Ruby on Rails*).
- Usar patrones de forma sencillísima (*singleton*, *delegación*, *decorador*, *función default*...).

Para saber más

Con estos dos libros se puede comenzar a programar Ruby, y seguir programando y aprendiendo por varios años. Son los mejores para empezar:

- **Dave Thomas, Chad Fowler y Andy Hunt (2005). *Programming Ruby. The Pragmatic Programmer's Guide (Second Edition)*. The Pragmatic Bookshelf.**

La primera parte te enseña poco a poco a programar en Ruby. Si ya sabes otro lenguaje orientado a objetos, mucho mejor, pero en caso contrario también te sirve. La segunda parte explica los accesorios de *Ruby*, su entorno, la línea de comandos, cómo conectarlo con otros lenguajes, etc. La tercera parte explica lo interno de Ruby, cuál es su filosofía y cómo se representan los objetos en memoria. Gracias a ello podrás entender la parte más difícil: la reflectividad, que permite a un programa interrogarse y modificarse a sí mismo. Ello lo convierte en un excelente libro incluso para programadores expertos. La cuarta parte es la librería estándar de Ruby, donde se explica cada clase y cada método, con varios ejemplos.

Lucas Carlson y Leonard Richardson (2006). *Ruby Cookbook*. USA: O'Reilly.

Ya sabes algo de Ruby, pero probablemente todavía no dominas todo su potencial. Todavía no piensas de la forma “*the Ruby way*”. Quieres hacer algo sencillo, pero te das cuenta de que estás empleando modelos mentales que arrastras de otros lenguajes como Java, Python o C++. ¿Será que hay una forma más simple de hacerlo en Ruby? Casi seguro que la respuesta es afirmativa, y en este libro te encontrarás multitud de ejemplos de trozos de código (unas pocas líneas) con las que realizar las tareas más habituales. Basta copiar, pegar y adaptarlo a tu problema específico. Cada trozo de código viene extensamente explicado y se dan ejemplos alternativos, para que no emplees solamente el *Copy&Paste* sino que sigas aprendiendo.

Referencias

Ruby tiene aplicaciones especializadas en muchas áreas, desde la programación web a la concurrencia, paralelismo, ambientes distribuidos y *testing*, por nombrar

las más conocidas. Estos libros te ayudarán a seguir aprendiendo en cada tema.

LIBROS, ARTÍCULOS Y ENLACES WEB

Baird, K. C. (2007). *Ruby by example*. San Francisco: No Starch Press.

Berube, D. (2007). *Practical Ruby Gems*. Berkeley: Apress.

Black, D. A. (2006). *Ruby for Rails*. Greenwich: Manning Publications.

Brown, G. T. (2009). *Ruby best practices*. Sebastopol: O'Reilly.

Burd, B. (2007). *Ruby on Rails for dummies*. New Jersey: Wiley.

Dees, I. (2008). *Scripted GUI testing with Ruby*. The Pragmatic Bookself.

Feldt, R., Johnson, L. y Neumann, M. (2002). *Ruby developer's guide*. USA: Syngress Publishing.

Fitzgerald, M. (2007). *Learning Ruby*. Sebastopol: O'Reilly.

Fisher, T. (2008). *Ruby on Rails bible*. Indianapolis: Wiley.

Fulton, H. (2007). *The Ruby Way: Solutions and Techniques in Ruby Programming, Second Edition*. Addison Wesley Professional. Boston: Pearson Education.

Gray II, J. E. (2006). *Best of Ruby quiz*. USA: The Pragmatic Bookself.

Grimm, A. (2013). *Confident Ruby*. USA: The Pragmatic Bookself.

__. (2013). *Exceptional Ruby*. USA: The Pragmatic Bookself.

Gutschmidt, T. (2003). Game programming with Python, Luan and Ruby. Premier Press.

Hal, F. (2002). *The Ruby Way*. USA: SAMS Publishing.

Harris, J. (2006). *Rubyisms in Rails*. Addison Wesley Professional.

Hartl, M. y Prochazka, A. (2008). *Rails space. Building a social networking website with Ruby on Rails*. Boston: Pearson Education.

Hellsten, C. y Laine, J. (2006). *Beginning Ruby on Rails E-Commerce. From Novice to Professional*. Berkeley: APress.

Lenz, P. (2007). *Build Your Own Ruby On Rails Web Applications*. Australia: SitePoint Pty.

Mahadevan, S. (2002). *Making use of Ruby*. Indianapolis: Wiley.

Marick, B. (2006). *Everyday scripting with Ruby for teams, testers and you*. USA: The Pragmatic Bookself.

Matsumoto, Y. (2001). *Ruby in a nutshell*. Sebastopol: O'Reilly.

Perrota, P. (2014). *Metaprogramming Ruby. Program like the Ruby Pros*. The Pragmatic Bookself.

Rappin, N. (2008). *Professional Ruby on Rails*. Indianapolis: Wiley.

Schmidt, M. (2006). *Enterprise Integration with Ruby*. USA: The Pragmatic Bookself.

Seki, M. (2012). *The DRuby book. Distributed and Parallel Computing with Ruby*. USA: The Pragmatic Bookself.

Tate, B. A. (2006). *From Java to Ruby. Things every manager should know*. USA: The Pragmatic Bookself.

__. y Hibbs, C. (2006). *Ruby on Rails up and running*. Sebastopol: O'Reilly.

Tennis, C. (2006). *Rapid GUI development with QtRuby*. USA: The Pragmatic Bookself.

Vohra, D. (2007). *Ruby on Rails for PHP and Java developers*. Berlin. Springer.

Williams, J. (2007). *Rails solutions. Ruby on Rails made easy*. Berkeley: Apress.

TESIS Y TRABAJOS DE GRADO EN EVALAB

Estamos empleando *Ruby* en muchos trabajos de grado. Aquí no los vamos a mencionar sino que los encontrarás en cada capítulo de este libro.

INTRODUCCIÓN A CUCUMBER

“El hombre es el único animal que tropieza dos veces con la misma piedra”
Anónimo

“Los humanos son los únicos animales que repiten manualmente dos veces la misma prueba”
Ángel E. García Baños

Alrededor de Ruby hay un colectivo enorme de excelentes programadores que han ido creando un ecosistema de herramientas siguiendo la misma filosofía que el lenguaje: fácil y potente. Entre ellas está el *framework* para programación en la web *Rails*, herramientas de *deployment* como *Capistrano* y la que vamos a presentar ahora, *Cucumber*, para hacer pruebas automáticas. Hay otras alternativas, para todos los gustos.

Habitualmente se usan los lenguajes livianos como Ruby bajo metodologías ágiles, concretamente BDD (*Behavior Driven Development*). Y en ellas las pruebas son algo fundamental. La recomendación es crear una batería de pruebas que debe pasar el *software*, y solo después escribir ese *software*. Así, de cierta manera, las pruebas se convierten en el equivalente a los requerimientos funcionales de las metodologías pesadas. De este modo, como programador puedo escribir las pruebas funcionales de las clases, las de integración, las del sistema y las de aceptación. Esto último es lo más importante: dado que las pruebas se escriben en español (o en inglés o cualquier otro idioma humano), se puede animar al cliente a que las redacte aunque no sepa nada de programación. Todas estas pruebas se convierten después a código ejecutable y un sistema de colores señalizará para cada una de ellas si se pasó o no se pasó (rojo=no se pasó, verde=sí se pasó, amarillo=pendiente de escribirse el código de la prueba). El cliente puede saber inmediatamente si el *software* que le estamos desarrollando cumple o no con sus expectativas.

Porque sabemos que es imposible garantizar teóricamente que el *software* que escribimos esté libre de errores (a no ser que limitemos la expresividad del lenguaje de programación, para que deje de ser Turing-completo). Pero con pruebas automáticas (sea con *Cucumber* o con cualquier otra herramienta) lo que logramos es que el número de errores disminuya de forma monótona, es decir, que tienda asintóticamente a cero. Las pruebas manuales no lo garantizan y mucho menos la ausencia de pruebas de ningún tipo.

La razón es que dentro de la metodología *BDD* se especifica que cada vez que se encuentre un error en nuestro *software* (bien sea en el ambiente de desarrollo o en el de producción) lo primero que se debe hacer es escribir una prueba que capture el problema. Al correr todas las pruebas, esa fallará, por supuesto. Ahora es el momento de reparar el error en el programa, corriendo las pruebas hasta que todas ellas pasen de nuevo.

En ambientes de prueba manuales es habitual que, al corregir un error en una parte del programa se genere un nuevo problema en otra parte. Y que cuando se detecte y lo corrijamos, puede ocurrir que el primer error resucite. Ello ocurre porque es muy laborioso ejecutar manualmente de nuevo todas las pruebas, y lo que se suele hacer es verificar únicamente que el nuevo error reportado haya desaparecido. Mientras que en un ambiente de pruebas automática es imposible que ocurra este problema porque las pruebas quedan escritas de forma acumulativa. Se introducen nuevas pruebas, pero no se borran las anteriores. De modo que, en cada ejecución de pruebas, si un error viejo resucita nos daremos cuenta y el *software* no saldrá a producción hasta que todas las pruebas se pasen en verde.

Diseñar las pruebas con *Cucumber* es muy sencillo. Supongamos que nuestro objetivo es diseñar una cuenta de ahorros bancaria:

```
class Banco
  def initialize(valorInicial)
    end

  def ingresar(valor)
    end

  def retirar(valor)
    end

  def saldo
    end
end
```

Como vemos, todavía no tenemos el código, pero sí las funcionalidades que ofrecerá. Entonces se requieren dos archivos para las pruebas:

Archivo.feature. Contiene la descripción de las pruebas en español. Hay algunas pocas palabras clave que están coloreadas y que sirven para orientar cómo escribir las pruebas, pero no tienen ninguna semántica asociada, es decir, se ignoran por el *software* de pruebas. Lo que se hace es definir cada característica del *software* que se va a probar y a continuación una serie de escenarios, tanto con los casos felices como los que especifican qué hacer cuando la contraseña no coincide o algún dato introducido no es correcto. A continuación vemos un ejemplo:

Característica: Verificar el correcto funcionamiento de una cuenta de ahorros.

Antecedentes: Crear una cuenta de ahorros bancaria con 1000 pesos

Dado que necesito ahorrar, abro una cuenta de ahorros con 1000 pesos iniciales

Escenario: Verificar que la cuenta tiene el saldo de 1000 pesos

Cuando pido el saldo

Entonces debe decirme que tengo 1000 pesos

Escenario: Introduzco más dinero

Dado que ingreso 3000 pesos

Cuando pido el saldo

Entonces debe decirme que tengo 4000 pesos

Escenario: Retirar dinero

Dado que retiro 400 pesos

Cuando pido el saldo

Entonces debe decirme que tengo 600 pesos

Escenario: Imposibilidad de retirar más dinero del que hay

Dado que retiro 5000 pesos

Cuando pido el saldo

Entonces debe decirme que no se pudo hacer ese retiro

Y debe decirme que tengo 1000 pesos

El texto posterior a la palabra **Característica** sirve solo para documentar y el *software* de pruebas la ignora. Lo mismo ocurre con el texto posterior a **Escenario**.

Todas las demás líneas de texto que comienzan por una palabra en color rojo (**Dado, Dada, Dados, Dadas, Cuando, Entonces, Y, E, Pero**) se convierten a un pequeño trozo de código que recrea en Ruby lo que se dice en español (enseguida explicaremos cómo se hace esta magia). Y los **Antecedentes** definen lo que se hace justo antes de que comience cada escenario. Los escenarios son independientes entre sí, es decir, los resultados de unos no afectan a los de otros. Precisamente los antecedentes sirven para dar valores iniciales y que todos los escenarios comiencen de la misma manera.

Archivo_steps.rb. Por cada archivo **feature** debe haber un archivo **steps** donde se especifique como se traduce cada sentencia en español a código Ruby. Es aquí donde se hace la magia, que consiste en que no hay ninguna magia, pues el programador debe hacer la traducción manualmente. Por ejemplo, para el caso anterior sería así:

```
Dado /^que necesito ahorrar, abro una cuenta de ahorros con 1000 pesos iniciales$/ do |valorInicial|
  @banco = Banco.new(valorInicial.to_i)
end
```

```
Cuando /^pido el saldo$/
@saldo = @banco.saldo
end
```

```
Entonces /^debe decirme que tengo (.+?) pesos$/ do |nuevoSaldo|
  expect(@saldo).to eq(nuevoSaldo.to_i)
end
```

```
Dado /^que ingreso (.+?) pesos$/ do |valor|
  @banco.ingreso(valor.to_i)
end
```

```
Dado /^que retiro (.+?) pesos$/ do |valor|
  @banco.retiro(valor.to_i)
end
```

Esto que estamos viendo son algo así como funciones en Ruby, excepto que en vez de tener un nombre de función tienen una expresión regular, que incluso puede capturar datos variables. Si son numéricos, hay que convertirlos de *string* a entero con la función **to_i**. La idea es que el cuerpo de la función se ejecuta cada vez que hay una coincidencia de la expresión regular con las líneas que especifican lo que hay que hacer en cada escenario del archivo **feature**. Estas funciones hacen llamadas al **Banco**, para crearlo, ingresar dinero y mirar el saldo. Pero también usan la librería **RSpec** (Chelimsky, 2010) para verificar expectativas. Allí podemos ver cómo se comparan dos números enteros por medio de la función **expect**. Si la expectativa se cumple, aparecerá la prueba en color verde, que significa que se pasó correctamente. Y si no se cumple, aparecerá de color rojo, que significa que nos hemos equivocado en algo y hay que revisar el código.

Inicialmente todas las pruebas saldrán en amarillo (no implementadas) o en rojo (fallidas) y ahora lo que toca hacer es añadir el código correcto a la clase Banco para que vaya pasando cada una de las pruebas.

Lo bonito de los pasos (**steps**) es que son reutilizables. Y así vemos que en el archivo *features* podemos usar el mismo paso varias veces con diferentes valores,

como en:

Dado que retiro 400 pesos

Dado que retiro 5000 pesos

En ambos casos se ejecutará el mismo paso en Ruby (la función disparada por la expresión regular:

```
/^que retiro (.+?) pesos$/
```

Pero en cada caso la expresión regular capturará un valor distinto (400 o 5000).

Hay algunas cosas más que saber para usar *Cucumber*, como que puede haber muchos archivos *feature* y *steps* y que se pueden especificar las características con tablas, para no hacer repetitivas las sentencias en español, etc.

También conviene saber que *Cucumber* se puede conectar a muchas otras herramientas. Una de las más interesantes que he encontrado es *Selenium* (Se, 2017), que permite hacer *scripts* que naveguen solos por la web. Verás el navegador dar saltos él solito de una página a otra, llenar campos y todo lo que un humano pueda hacer. No es la única herramienta para probar páginas web (hay otras como *watir*), pero sí es la más completa y profesional. Además, no solo sirve para hacer *testing* de las *interfaces* de usuario web, sino que también puedes automatizar procesos en páginas web que no ofrezcan ninguna otra alternativa como *Corba* o *Web Services*. Usa directamente la página web sobre los navegadores *Chromium* o *Mozilla*, imitando los clic de *mouse* que pueda hacer un usuario, llenando formularios de forma programática y leyendo los resultados que entregue la página web. Al interactuar con el navegador, ejecuta cualquier *script* de *Javascript* que pudiera haber allí (esa es una ventaja respecto a *Watir*, que se limita a interactuar con la aplicación web usando mensajes *HTML*, sin levantar primero un navegador, con lo que se pierden todas las funcionalidades de los *scripts* del lado del cliente).

Para saber más

- **David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp y Dan North (2010). *The RSpec Book. Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Bookshelf.**

Cucumber puede conectarse a muchas otras herramientas. Con *RSpec* se pueden hacer aserciones sobre condiciones que debe cumplir alguna variable del programa.

- **Matt Wynne y Aslak Hellesøy (2012). *The Cucumber Book. Behaviour-Driven Development for Testers and Developers*. The Pragmatic Bookself.**

Este es el mejor libro que conozco sobre *Cucumber*. Viene todo muy bien explicado. Introduce progresivamente los temas, primero lo más sencillo. Y tiene al menos un ejemplo de cada tema.

Referencias

LIBROS, ARTÍCULOS Y ENLACES WEB

Dees, I., Wynne, M. y Hellesøy, A. (2013). *Cucumber recipes. Automate anything with BDD tools and techniques*. Dallas: The Pragmatic Bookself.

Se (2017). SeleniumHQ Browser Automation. Recuperado el 14 de septiembre de 2017. Disponible en: <http://www.seleniumhq.org/>

Ye, W. (2013). *Cucumber BDD. How-to*. Birmingham: Packt Publishing.

AC: Autómata celular. Un grafo regular e infinito, donde cada nodo está conectado bidireccionalmente solo con sus vecinos.

FSM: *Finite State Machine*, máquina de estados finitos. Grafo dirigido con una única marca indicando el estado activo. La marca puede moverse a otro estado a través de algún arco saliente de ese estado, si se cumple la condición indicada en el arco. Si hay más de una marca activa, se llama Red de Petri.

Estocástico: una secuencia es estocástica si cada término es imposible de predecir conociendo los anteriores. Las secuencias estocásticas también se pueden llamar al azar o no-deterministas. No es lo mismo que aleatorio, aunque mucha gente los confunde.

Aleatorio: una secuencia es aleatoria si no se puede comprimir. No es lo mismo que estocástico, aunque mucha gente los confunde.

Seudoaleatorio: es similar a un proceso caótico digital, con horizonte de predicción de una unidad de tiempo (o sea, solo se puede predecir la salida actual, pero no las siguientes, a partir de todas las entradas pasadas y de la fórmula) y donde permanece oculta la fórmula de conversión de entradas en salidas. El hecho de que sea digital implica que no hay ruido en las entradas, de modo que estrictamente hablando no es caótico. Pero la idea de un cortísimo horizonte de predictibilidad permanece.

Caótico: la definición rigurosa puede verse en el correspondiente capítulo, pero aproximadamente se puede decir que un proceso es caótico si es muy sensible a sus entradas, es decir, con un minúsculo cambio de la entrada, la salida cambia mucho. Eso hace que sean difíciles de predecir y suele haber un horizonte de predicción a partir del cual los errores acumulados hacen imposible anticipar el futuro del sistema.

Determinista: una secuencia de datos (o un proceso) es determinista si las sucesivas salidas (o estados) están completamente determinadas por las salidas anteriores (o estados anteriores). Los procesos deterministas convierten sus entradas en salidas por medio de una fórmula o un algoritmo que no contiene ningún elemento de azar. De modo que si se conocen las condiciones iniciales, se pueden predecir las salidas sucesivas. Y si las entradas se repiten, también lo

hacen las salidas.

Fractal: un objeto geométrico cuya dimensión de Hausdorff-Besicovitch no coincide con su dimensión topológica. Informalmente se dice cuando un objeto tiene infinita rugosidad y autosemejanza en todas las escalas.