No. 9 · año 2013

REVISTA

BITS DE CIENCIA

UNIVERSIDAD DE CHILE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Databases Algorithms Information Retrieval Networks Learning Data Structures Software Complexity Languages Cryptography Theory Artificial Intelligence

TEORÍA DE LA COMPUTACIÓN EN CHILE

- **"Kurt Gödel y Alan Turing: una nueva mirada a los límites de lo humano"**, Claudio Gutiérrez
- **"Discretización en ingeniería computacional y visualización científica"**, María Cecilia Rivara

Comité Editorial
Nelson Baloian, profesor.
Claudio Gutiérrez, profesor.
Alejandro Hevia, profesor.
Gonzalo Navarro, profesor.
Sergio Ochoa, profesor.

Editor General
Pablo Barceló, profesor.

Editora Periodística
Ana Gabriela Martínez.

Periodista
Karin Riquelme.

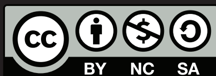
Diseño y Diagramación
Anzuelo Creativo.

Imagen Portada
Anzuelo Creativo.

Fotografías
Comunicaciones DCC U. de Chile.
Comunicaciones FCFM U de Chile.
Comunicaciones DCC PUC.
Anzuelo Creativo.

Dirección
Departamento de Ciencias de la Computación
Avda. Blanco Encalada 2120, 3º piso
Santiago, Chile.
837-0459 Santiago
www.dcc.uchile.cl
Teléfono: 56-2-29780652
Fax: 56-2-26895531
revista@dcc.uchile.cl

Revista BITS de Ciencia del Departamento de Ciencias de la Computación de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile se encuentra bajo Licencia Creative Commons Atribución-NoComercial-CompartirIgual 3.0 Chile. Basada en una obra en www.dcc.uchile.cl



Revista Bits de Ciencia Nº 9
ISSN 0718-8005 (versión impresa)

www.dcc.uchile.cl/revista
ISSN 0717-8013 (versión en línea)

CONTENIDOS

04 ***INVESTIGACIÓN DESTACADA***
Discretización en ingeniería computacional y visualización científica
/ *María Cecilia Rivara*

10 ***COMPUTACIÓN Y SOCIEDAD***
Historia del desarrollo de la Computación en la Universidad de Concepción (1960 – 1980)
/ *Yussef Farrán, José Durán*

14 U-papers: accediendo a las publicaciones científicas del DCC
/ *Felipe Chacón, José A. Pino*

18 Kurt Gödel y Alan Turing: una nueva mirada a los límites de lo humano
/ *Claudio Gutiérrez*

28 ***TEORÍA DE LA COMPUTACIÓN EN CHILE***
Teoría de la Información
/ *Gonzalo Navarro*

34 Planificación, preferencias y conocimiento: motivación y problemas abiertos
/ *Jorge Baier*

38 Tipos: garantías a medida
/ *Éric Tanter*

46 Teoría de Bases de Datos
/ *Jorge Pérez*

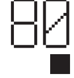
52 Indexación y Compresión de Datos para motores de búsqueda
/ *Diego Arroyuelo*

60 Algoritmos, Estructuras y Datos y mousse de chocolate
/ *Jérémy Barbay*

64 Demostraciones de Nula Divulgación: o cómo convencer a alguien que mi sudoku tiene solución, sin revelarla
/ *Alejandro Hevia*

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

72 ***SURVEY***
Problemas de satisfacción de restricciones
/ *Miguel Romero*


80 ***CONVERSACIONES***
Entrevista a Roberto Camhi
/ *Benjamín Bustos, Claudio Gutiérrez*

82 ***GRUPOS DE INVESTIGACIÓN***
UC Temuco: la investigación como elemento que fortalece la docencia. Una mirada desde la región
/ *Oriel Herrera, Marcos Lévano*

EDITORIAL

La Teoría de la Computación es una maravillosa disciplina que constituye los fundamentos de la Computación como tal. Su importancia es doble: por un lado nos permite entender los límites de lo computable, es decir, qué problemas pueden ser resueltos utilizando un algoritmo y, por otro, nos ayuda a diseñar herramientas para la resolución más eficiente de problemas en función de recursos limitados (memoria, tiempo, número de procesadores, etc.).

En su acepción tradicional, la Teoría de la Computación estudia qué problemas pueden ser computados y a qué costo. Para este número, sin embargo, hemos decidido utilizar una definición amplia del área, que incluya además de los temas más ortodoxos –como algoritmos, computabilidad y complejidad– otros que utilicen una buena dosis de teoría en su estudio, como por ejemplo Inteligencia Artificial, Lenguajes, Recuperación de la Información o Bases de Datos.

Nuestra idea ha sido dar una "vista de pájaro" al estudio de la Teoría de la Computación en Chile. Esto incluye colaboraciones de Gonzalo Navarro sobre Teoría de la Información; Jorge Baier sobre Planificación; Diego Arroyuelo sobre Indexación y Compresión; Jorge Pérez sobre Lenguajes de Consulta para Bases de Datos; Alejandro Hevia sobre Criptografía; Éric Tanter sobre Sistemas de Tipos para Lenguajes de Programación; y Jérémy Barbay sobre Algoritmos. Una de las conclusiones que podemos sacar de todos estos interesantes artículos es que la Teoría de la Computación que se hace en Chile es de excelente nivel, pero que es indispensable una mayor masa crítica para lograr convertirnos en referentes mundiales.

Además de los artículos sobre Teoría de la Computación, la Revista incluye las secciones usuales:

• Investigación Destacada, con un artículo de María Cecilia Rivara sobre su trabajo en Visualización Computacional.

• Computación y Sociedad, que contiene un artículo de Yussef Farrán y José Durán sobre la historia del Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción; otro de Claudio Gutiérrez con un análisis comparativo de la vida y obra de Kurt Gödel y Alan Turing; y finalmente uno de Felipe Chacón y José A. Pino sobre el sistema U-papers, que nuestro DCC ha desarrollado para que la comunidad acceda a sus publicaciones.

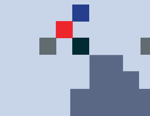
• Survey , donde el alumno de Doctorado Miguel Romero nos habla de la resolución de problemas bajo restricciones.

• Grupos de Investigación, a cargo de Oriel Herrera y Marcos Lévano, donde nos cuentan sobre la experiencia de realizar docencia a través de la investigación en la Universidad Católica de Temuco.

En este número inauguramos una nueva sección que incluye entrevistas a nuestros ex alumnos. Esta vez entrevistamos a Roberto Camhi, creador y actual gerente de Mapcity.

Ojalá que disfruten esta edición de Bits de Ciencia tanto como nosotros disfrutamos preparándola. BITS

Pablo Barceló
Editor General Revista Bits de Ciencia



Discretización en ingeniería computacional y visualización científica

María Cecilia Rivara
Profesora Titular Departamento de Ciencias de la Computación, Universidad de Chile. Dr. in Applied Sciences (1984) y Master of Engineering (1980), Katholieke Universiteit Leuven, Leuven, Bélgica; Ingeniera Matemática, Universidad de Chile (1973). Intereses en investigación: Mallas Geométricas y aplicaciones, Algoritmos para Triangulaciones, Algoritmos Paralelos, Métodos Numéricos, Computación Gráfica, Métodos de Elementos Finitos, Modelación Geométrica, Visualización Científica, Ingeniería y Ciencia Computacional, aplicaciones a Ingeniería, aplicaciones Médicas.
mcrivara@dcc.uchile.cl

Discretización es un anglicismo que aún no es aceptado por la Real Academia Española. Es un concepto importante que cruza todos los campos relacionados con Ingeniería y Ciencia Computacional, y que da soporte a la computación gráfica, la visualización científica, la visualización realista, el diseño de las arquitecturas de hardware gráficas, y todas las aplicaciones de estos temas. Se refiere a la necesidad de modelar objetos geométricos continuos en base a una cantidad finita y adecuada de información para los fines requeridos por una aplicación específica. En este artículo revisamos el uso interdisciplinario de estas ideas y lo relacionamos con investigación realizada en el Departamento de Ciencias de la Computación de la Universidad de Chile.

DISCRETIZACIÓN DE SUPERFICIES

Para introducir el tema consideremos una geometría simple: una placa plana rectangular y sin grosor en dos dimensiones. La modelación computacional más simple de esta geometría consiste en seleccionar un conjunto finito y ordenado de puntos (arreglo o matriz) sobre la superficie de la placa que se usa como base para construir una aproximación del objeto geométrico. Con estos datos se puede usar este mismo modelo discreto simple para aproximar la geometría continua (ver Figura 1a), lo que permite definir funciones discretas sobre estos puntos como los usados en los métodos de diferencias finitas. Una alternativa más compleja es construir una aproximación poligonal de la geometría de la placa en base a cuadriláteros o triángulos (Figuras 1b y 1c). En estos dos últimos casos se tiene una aproximación continua de la superficie construida en base a los puntos discretos de la Figura 1a que se transforman en vértices de la malla de cuadriláteros (Figura 1b) y de la triangulación (Figura 1c). Las aproximaciones poligonales tienen la ventaja de ser superficies continuas, donde cada polígono está definido por la ecuación de un plano; tiene área definida y normal asociada. Sobre estas mallas de polígonos se pueden definir funciones con distintos grados de continuidad, de acuerdo a los requerimientos de aplicaciones complejas. También permite realizar trabajo sofisticado de visualización, que requiere de modelos de iluminación que necesitan las normales a la geometría para pintar el objeto.

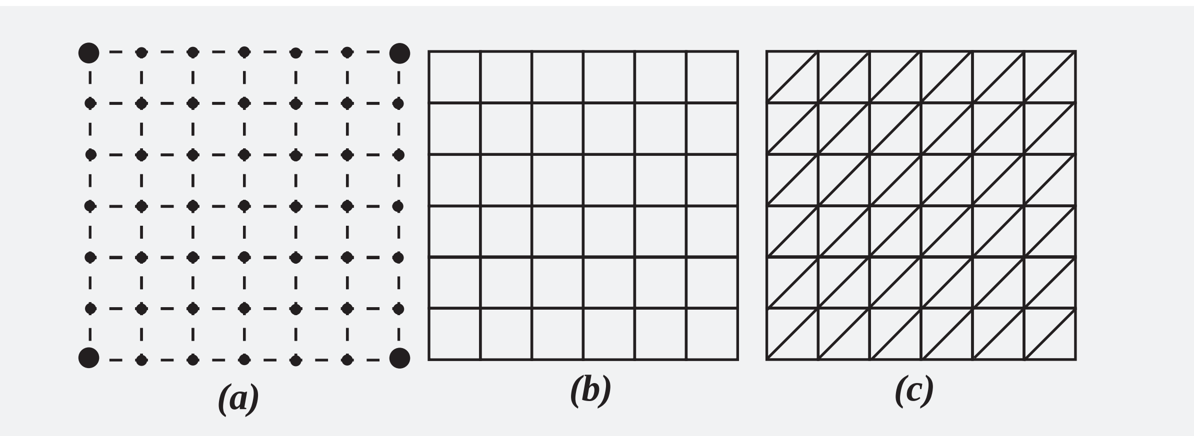
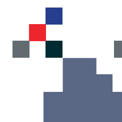


Figura 1 • (a) Modelo discreto: conjunto de puntos. (b) Malla de cuadriláteros. (c) Triangulación (malla de triángulos).



EJEMPLOS DE DISCRETIZACIONES POLIGONALES

Modelos poligonales más avanzados se ilustran en las Figuras 2 y 3. La Figura 2 muestra dos triangulaciones de la superficie del Lago Superior, compartido entre Canadá y Estados Unidos. La triangulación 2a se ha construido en base a una aproximación poligonal del borde de dicho lago. Observe que los vértices de la triangulación corresponden sólo a los vértices del polígono y por esta razón hay muchos triángulos tipo agujas que son inaceptables para muchas aplicaciones. Para la Figura 2b se ha seleccionado automáticamente un conjunto irregular de puntos interiores para modelar la superficie del Lago Superior con triángulos de buena calidad (ángulo pequeño mayor a 30°) mediante el algoritmo Lepp-Delaunay centroide desarrollado e implementado por nuestro grupo de investigación. La Figura 3 ilustra la modelación de un terreno con triangulaciones. La triangulación se ha obtenido usando un algoritmo de simplificación a partir de un reticulado de datos obtenidos por satélite. Se ha utilizado sólo el 5% de los datos para construir una triangulación de buena calidad geométrica bien adaptada a la topografía.

MÉTODOS NUMÉRICOS PARA ECUACIONES DIFERENCIALES

La gran mayoría de los métodos numéricos para analizar o simular computacionalmente fenómenos físicos, requieren de la discretización de una geometría asociada, como las requeridas para fenómenos físicos modelados por ecuaciones diferenciales ordinarias y por ecuaciones diferenciales parciales.

En el caso de problemas modelados por Ecuaciones Diferenciales Parciales (EDP) en dos dimensiones, es necesario discretizar el dominio o geometría bidimensional asociada. Los métodos numéricos más usados para problemas modelados por EDPs, son el método de diferencias finitas y el método de elementos finitos que ejemplificaremos con problemas de estado estacionario (ecuación de Laplace, ecuación de Poisson, ecuaciones elípticas en general). El método de diferencias finitas es intuitivo y simple de explicar y entender. Para el caso particular de un dominio rectangular se utiliza una discretización como la ilustrada en la Figura 1a (conjunto ordenado de puntos discretos), y se aproxima la ecuación diferencial por una ecuación de diferencias sobre cada punto de este reticulado, lo que conduce a resolver numéricamente un sistema de ecuaciones. La solución numérica de la EDP es una función discreta definida sobre el dominio discreto.

El método de elementos finitos en cambio "resuelve" numéricamente un problema matemáticamente más complejo, equivalente al problema de EDP. Éste requiere que el dominio sea aproximado por una malla de polígonos, sobre el cual se construye una solución numérica con mayor grado de continuidad. Tiene la ventaja de manejar adecuadamente geometrías más complejas mediante mallas

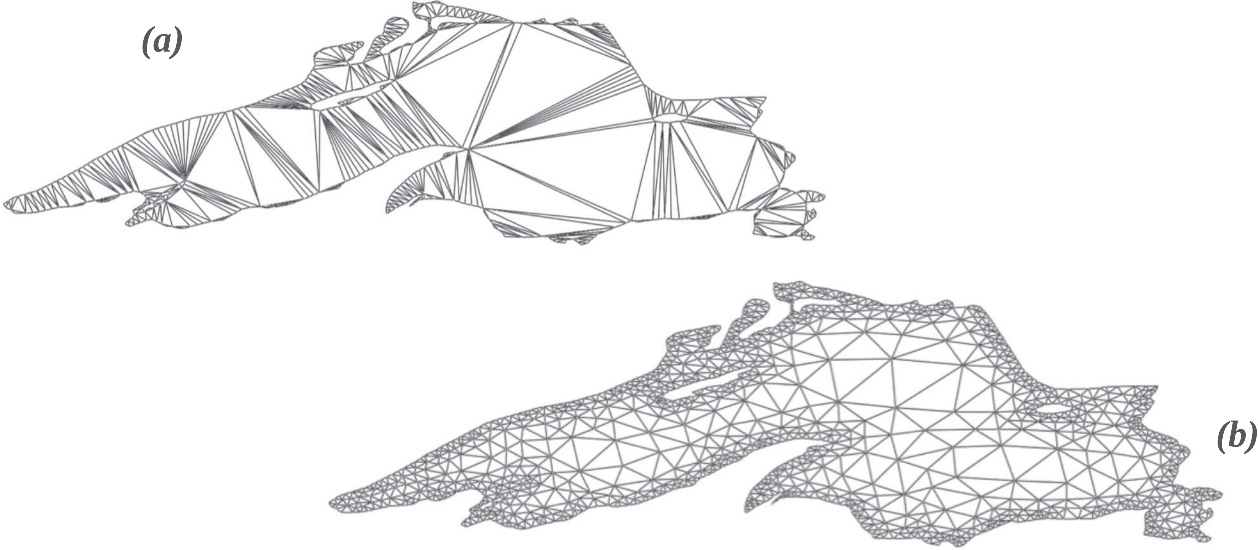


Figura 2 • Lago Superior: (a) Triangulación del borde del polígono. (b) Triangulación de buena calidad (ángulos mayores o iguales a 30°), obtenida con algoritmo Lepp-Delaunay centroide (Rodríguez, Rivara).

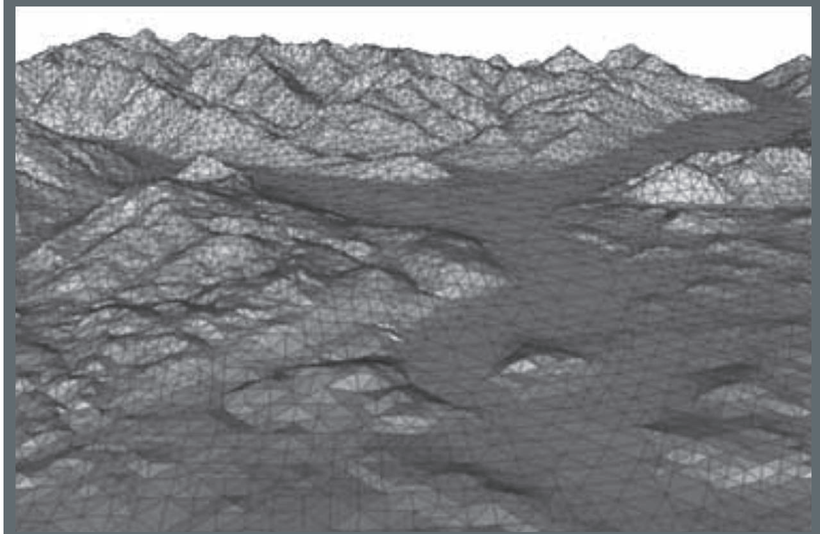


Figura 3 • Triangulación obtenida mediante un algoritmo de simplificación a partir de datos de satélite (usa el 5% de los datos). Trabajo conjunto de la profesora Rivara con investigadores de la Universidad de Gerona (Marité Guerrieri, Narcín Coll, Antoni Sellarès).

irregulares. En la Figura 5 se muestra la triangulación y la solución lineal de elementos finitos del problema descrito por la ecuación de Poisson y las condiciones de borde de la Figura 4 sobre un cuadrado unitario. Para obtener esta solución numérica con buena precisión, se ha usado un método de elementos finitos adaptativo que a partir de una triangulación simple del dominio (cuadrado dividido en dos triángulos rectángulos) construye automáticamente una discretización adaptada a la solución del problema. Estos resultados se han obtenido con el software desarrollado por Eduardo Mercader, en su memoria de Ingeniería Civil en Computación. La Figura 6a muestra una triangulación en tres dimensiones (malla de tetraedros) que aproxima la geometría del corazón, gentileza de Chandrajit Bajaj de la Universidad de Texas en Austin, Estados Unidos.

PANTALLAS O TERMINALES GRÁFICOS EN 2D

Son dispositivos tecnológicos diseñados y construidos también en base al concepto de discretización para visualizar imágenes computacionales. Las pantallas son en esencia un mundo discreto constituido por un arreglo bidimensional de píxeles. Cada píxel es el elemento mínimo del dibujo (con área mayor que 0) que se accede y pinta independientemente mediante la activación del fósforo ubicado en el píxel. La imagen se construye en base a estas contribuciones. Por supuesto, la calidad de la imagen (ilusión de continuidad) depende de la resolución del dispositivo que corresponde al tamaño de la matriz de píxeles.

COMPUTACIÓN GRÁFICA EN 3D

Es un área de la Ciencia de la Computación cuyo objetivo es visualizar en un dispositivo o pantalla computacional en dos dimensiones, escenas de un “mundo” tridimensional modelado computacionalmente. Es una mezcla de tecnología y modelación matemático/computacional, a lo que se agrega el uso de una secuencia de algoritmos y técnicas (agrupados bajo el nombre técnico de *rendering*), que permiten generar imágenes de buena calidad para el sistema visual humano. Este trabajo se basa en modelar la superficie de los objetos mediante mallas de polígonos, donde es fundamental el uso de las normales para aplicar modelos de iluminación adecuados.

VISUALIZACIÓN REALISTA

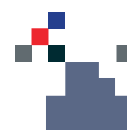
Esta expresión se usa principalmente en las aplicaciones del área de entretención, juegos, y animación 3D (obtención de imágenes fotorrealistas). En general utiliza técnicas avanzadas de computación gráfica. Se inicia con la modelación matemática y computacional de una escena compleja en 3D, continúa con el proceso de rendering que corresponde a la secuencia de pasos necesaria para obtener una visualización aceptable que incluye transformaciones geométricas, proyec-

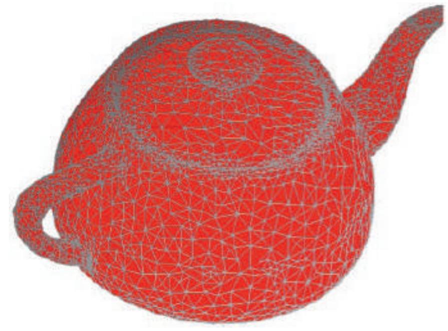
$$-\Delta u = f, en \quad \Omega = [0,1]x[0,1],$$
$$u = 0 \quad en \quad \partial\Omega$$

donde la solución exacta es la función

$$u = x(x-1)y\,(y-1)e^{-100((x-0,5)^2+(y-0,117)^2)}$$

Figura 4

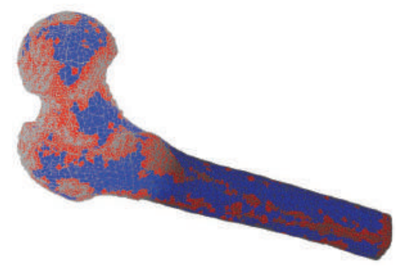




ción, uso de modelos sofisticados de iluminación y un conjunto de algoritmos y técnicas computacionales. Este proceso culmina con una imagen en el dispositivo que impresiona por su realismo al observador humano, como la que se muestra en la Figura 7, obtenida mediante *ray tracing.*

VISUALIZACIÓN CIENTÍFICA

El objetivo es usar técnicas gráficas para ayudar al analista a comprender o analizar la validez de los resultados de una aplicación de Ingeniería o Ciencia Computacional. Se usan técnicas de visualización simples y esquemáticas que resalten las características inherentes al estudio científico. En esta página se visualizan mallas de tetraedros (sólo la superficie exterior) de un fémur humano y de la tetera de té de Utah obtenidas por nuestro grupo de investigación. En el fémur se han pintado con distinto color los tetraedros según su calidad geométrica. Ejemplos de otros tipos de visualización científica se muestran en la Figura 6.



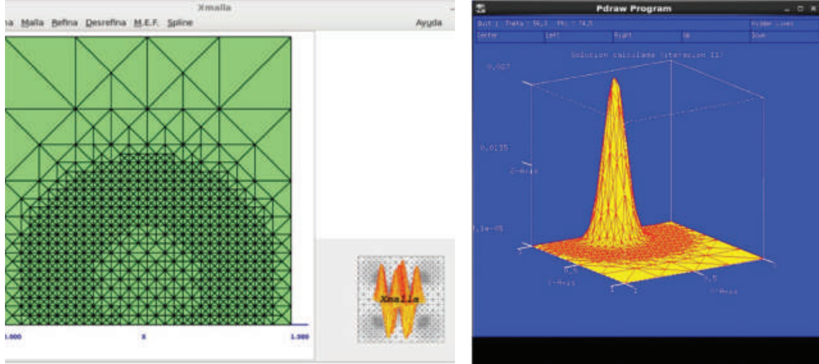


Figura 5 • Discretización y visualización de la solución de elementos finitos del problema de la Figura 4, obtenida con el software desarrollado en el DCC por Eduardo Mercader.

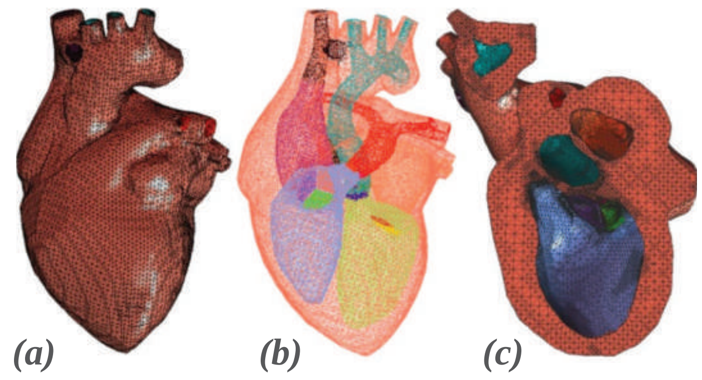


Figura 6 • Discretización del corazón y visualización de éste, gentileza de Chandrajit Bajaj, Universidad de Texas en Austin: (a) malla de tetraedros de elementos finitos; (b) visualización tipo alambre del corazón y sus componentes; (c) las válvulas y las cámaras del corazón se han pintado con distintos colores para diferenciarlas.

Figura 7 • Glasses from Wikipedia Commons: imagen fotorrealista de excelente calidad obtenida con el algoritmo de *ray tracing.*

INVESTIGACIÓN SOBRE TRIANGULACIONES EN EL DCC DE LA UNIVERSIDAD DE CHILE

Actualmente trabajamos en dos líneas principales de investigación.

• Estudio teórico de aspectos abiertos en los algoritmos Lepp-bisección y Lepp-Delaunay para refinar triangulaciones (obtención de triangulaciones más finas en zonas de interés de la geometría). Los algoritmos Lepp-bisección son formulaciones eficientes de los algoritmos basados en la bisección de triángulos (tetraedros) por la arista más larga. Permiten refinar local e iterativamente triangulaciones, manteniendo la calidad de la triangulación inicial. Se utilizan principalmente para métodos de elementos finitos. En trabajo previo se ha demostrado que estos algoritmos son robustos y eficientes (de costo lineal en el número de puntos insertados). También se han utilizado ampliamente en aplicaciones de Ingeniería. Recientemente hemos demostrado que en dos dimensiones las triangulaciones que se obtienen con el algoritmo Lepp-bisección son de tamaño óptimo (no más refinadas que lo necesario). En investigación en curso estamos trabajando también en demostrar que a partir de una triangulación con triángulos con ángulos muy pequeños (como los de la Figura 2a) se obtienen triangulaciones de buena calidad de tamaño óptimo con el algoritmo Lepp-Delaunay (ver Figura 2b).

• Diseño e implementación de algoritmos paralelos de refinamiento de triangulaciones. El objetivo es permitir el procesamiento de mallas muy grandes (decenas y centenas de millones de tetraedros) que son difíciles de manejar con algoritmos seriales (presentan problemas de memoria y tiempo de procesamiento). Recientemente hemos desarrollado algoritmos paralelos Lepp-bisección y Lepp-Delaunay 2D y 3D en ambiente multicore, que aprovechan las arquitecturas con varios núcleos de los computadores actuales. Estos algoritmos usan “randomización” para que el procesamiento de los triángulos sea independiente del orden; y “pre-fetching” para que el algoritmo sea independiente de la arquitectura del hardware. También se han desarrollado algoritmos paralelos de refinamiento de mallas sobre sistemas paralelos con memoria distribuida.

Las imágenes del encabezado de este artículo corresponden a refinamiento paralelo con el algoritmo Lepp-bisección en 3D (Rodríguez, Rivara).

Han contribuido a la investigación realizada en el DCC:

• Pedro Rodríguez, Ingeniero Civil Informático de la Universidad de Concepción, Académico de la Universidad del Bío Bío. Actualmente realiza su Tesis de Doctorado, en Ciencias de la Computación en la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile sobre la paralelización de algoritmos Lepp-bisección y Lepp-Delaunay.

• Carlos Bedregal, Bachiller en Ingeniería Informática de la Universidad Católica San Pablo del Perú, Ingeniero Informático de la misma Universidad. Actualmente realiza su Tesis de Doctorado en Ciencias de la Computación en la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile, en el tema Análisis de algoritmos Lepp-bisección y Lepp-Delaunay.

• Francisca Gallardo, Ingeniera Civil en Computación, Universidad de Chile, 2012.

• Eduardo Mercader, Ingeniero Civil en Computación, Universidad de Chile, 2012. BITS

Referencias

• M.C. Rivara, Lepp-bisection algorithms, applications and mathematical properties, Applied Numerical Mathematics, 59 (2009), 2218-2235.

• M.C. Rivara, C. Calderón, Lepp terminal centroid method for quality triangulation, Computer-Aided Design, 42 (2010) 58-66.

• D. Azócar M. Elgueta, M.C. Rivara, Automatic LEFM crack propagation method based on local Lepp-Delaunay mesh refinement, Advances in Engineering Software 41 (2010) 111-119.

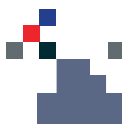
• N. Coll, M. Guerrieri, M.C. Rivara, J.A. Sellares, Adaptive simplification of huge sets of terrain grid data for geosciences applications J. Computational Applied Mathematics 236 (2011) 1410-1422.

• M.C. Rivara, P. Rodríguez, R. Montenegro, G. Jorquera, Multithread parallelization of Lepp bisection algorithms, Applied Numerical Mathematics 62 (2012) 473-488.

• P. Rodríguez, M.C. Rivara, Isaac D. Sherson, Exploiting the memory hierarchy of multicore systems for parallel triangulation refinement, Parallel Processing Letters, Vol. 22, N°3, 9 pages, Septiembre 2012.

• C. Bedregal, M.C. Rivara, A study on size-optimal longest edge refinement algorithms, 21th International Meshing Roundtable, California, 7-10 October 2012, 15 pages.

• C. Bedregal, M.C. Rivara, Longest edge algorithms for quality Delaunay refinement, Computational Geometry: Young Research Form, CG: YRF, Río de Janeiro, Brasil, Junio 17-20, 2013.



Integrantes del Departamento de Ingeniería Informática y Ciencias de la Computación de la UdeC: Andreas Polyméris, Jorge López, Gonzalo Rojas, John Atkinson, Loreto Bravo, Ricardo Contreras, Yussef Farrán, María Angélica Pinninghoff, Diego Seco, Marcela Varas, Andrea Rodríguez, Lilian Salinas, Cecilia Hernández, Carolina Rocha, Leo Ferres, Javier Vidal, Pamela Pinto, Juan Martínez y Hugo Novoa.

Historia del desarrollo de la Computación en la Universidad de Concepción (1960 - 1980)

Yussef Farrán

Académico del Departamento de Ingeniería Informática y Ciencias de la Computación (DIICC), Facultad de Ingeniería, Universidad de Concepción (UdeC). M.Sc. Informática en la PUC/J. Ha sido Jefe de Carrera y Director del Departamento de Ingeniería de Sistemas y luego del DIICC de la UdeC.

yfarran@udec.cl

José Durán

Profesor (JP) de la Facultad de Ingeniería y Director del Programa de Educación a Distancia, de la Universidad de Concepción. Posee estudios de Ingeniería Eléctrica en la Universidad Técnica Federico Santa María; M.Sc. Matemática de la Universidad de Waterloo, Canadá, y estudios de Postgrado en Ingeniería Eléctrica en la Universidad de Stanford. Co-fundador y ex Director del Centro de Computación de la Universidad de Concepción, ex Director del Centro de Computación de la Universidad de Santiago y ex Director del Departamento de Ingeniería Informática de la Universidad Técnica Federico Santa María.

jduran@udec.cl

En este artículo queremos dar una rápida revisión al desarrollo de la Computación en la Universidad de Concepción (UdeC), entre los años 1960 y 1980. El período posterior será tratado en una próxima presentación. Miraremos cómo evolucionó el desarrollo de Ciencias de Computación e Informática desde el punto de vista organizacional, académico y humano. Las fuentes de información son las memorias anuales de la UdeC, el Diario El Sur de Concepción, el libro sobre la Historia de la Facultad de Ingeniería y documentos personales de los autores.

La Universidad de Concepción se ha caracterizado permanentemente por mantenerse al día con los avances tecnológicos y metodológicos que requieren la docencia, investigación y administración propia, ello en un ambiente académico que procura calificarlo de excelencia.

Una prueba de ello se encuentra en el uso de la computación tempranamente. Durante la década de los cincuenta contaba ya con equipos precursores de la computación, aunque únicamente para fines administrativos, en una oficina de procesamiento de datos. Para ello operaba equipos IBM Unit Record (tarjetas perforadas), programables con tableros cableados. Uno de sus últimos directores fue el académico del Departamento de Física, profesor Phénix Ramírez, quien más tarde jugó un papel importante junto con el Sr. Patricio Léniz y el profesor Deforest Trautmann, (UNESCO) en la creación del Centro de Cómputos.

A comienzos de la década de 1960, en la Universidad se implementó el Proyecto del Fondo Especial de Naciones Unidas "Plan de Desarrollo de la Escuela de Ingeniería", cuyos objetivos principales fueron: mejorar y desarrollar la enseñanza en los niveles de técnicos e ingenieros, y estimular y promover la investigación científica como herramientas para el desarrollo del personal docente y de los alumnos. Uno de los logros de ese Plan fue la creación del Centro de Cómputos, que ordenó en 1965 el primer computador digital, un IBM 1620-II que llegó el 23 de enero de 1966, comenzando a operar en marzo del mismo año. Esta adquisición fue financiada con aportes económicos de la UdeC, de la UNESCO, e importantes descuentos dados por IBM para financiar su costo total del orden de 1,5 millones de dólares.

Este computador fue seleccionado por sus características de orientación científica, por el equipo de trabajo dirigido por Patricio Léniz.

Con la llegada del computador, se inició de inmediato un programa de capacitación para docentes, alumnos y profesionales externos sobre los lenguajes de programación Fortran, SPS y Pactolus (aplicado en simulación). Un dato anecdótico: la matrícula para el curso de programación Fortran costaba 100 "escudos", moneda nacional de la época.

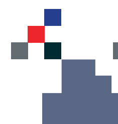
En ese período fue muy valiosa la colaboración del profesor Silvio O. Navarro, Director del Departamento de Ciencias de Computación de la Universidad de Kentucky, Estados Unidos, quien además puso a disposición una serie de programas de computación de aplicaciones variadas, que él mismo trajera. Permaneció en Concepción por tres meses durante el año 1965 y luego otro período en 1966.

El computador IBM 1620-II junto con el IBM 1401 de la Compañía de Acero del Pacífico, CAP Huachipato, recibido en 1964, fueron los primeros computadores digitales en llegar a la región, y están dentro de los ocho primeros en llegar al país. Las primeras aplicaciones de estos computadores fueron en el área administrativa (lenguajes de programación de bajo nivel: Autocoder y SPS) y Fortran en el área de ingeniería. Algunos nombres de pioneros del desarrollo de Computación en la industria (Huachipato), en la década de los sesenta: Waldo Muñoz, Óscar Sáez, Sergio Castillo, José Durán y Hernán Ibarra; mientras en la Universidad: Patricio Léniz, Phénix Ramírez, Renán Donoso y Marcelo Pardo. Es interesante destacar además, que el desarrollo de la computación en la Universidad contó en los primeros años con el apoyo de dos especialistas norteamericanos del Cuerpo de Paz de los Estados Unidos: Alfred J. Maley y Frank Pender.

En 1967 el Centro de Cómputos dio origen a una nueva unidad: el Centro de Ciencias de Computación e Información (CCCI) anexando a los encargados de los equipos UR de la Casa Central, luego se creó su división de docencia para la capacitación y dictación de asignaturas.

Las actividades de capacitación, para la utilización del IBM 1620-II, se orientaron principalmente a docentes y estudiantes de la Escuela de Ingeniería, en el lenguaje de alto nivel con que se contaba, Fortran II-D, y luego Kingston Fortran II, que se trajo de la Universidad de Waterloo en Canadá.

A fines de la década de los sesenta, se inició un proceso de discusión en la educación superior, conocido como "La Reforma", que trajo consigo cambios en la manera de administrar las Universidades.



De izquierda a derecha: José Durán, Waldo Muñoz y Óscar Sáez.

La Universidad de Concepción fue uno de los focos de mayor acción en el concierto nacional. Aprovechando esa contingencia un grupo de profesionales del Centro de Computación: Renán Donoso, José Durán, Atendolfo Pereda, Carlos Le Fort, y algunos más, lograron a fines de la década de los sesenta, que se creara el CCCI como una Unidad Académica con participación en el Consejo Superior de la Universidad. Sus primeros directores fueron Renán Donoso V. (1968), José Durán R. (1969-1974), Jorge González R. y Humberto Zúñiga.

Es interesante destacar en el ámbito de la extensión una importante actividad llevada a cabo aproximadamente en 1968, por el Centro de Computación con el apoyo de UNESCO, que realizó uno de los primeros Simposios Latinoamericanos de Computación Internacional. Para su organización contó con el apoyo de dos profesores extranjeros, Abou Thaleb de Egipto y Sergio Beltrán, Director del Centro de Cálculo Electrónico de la UNAM, México. En él se presentaron trabajos nacionales y del extranjero.

Destacamos de esta época, una iniciativa de colaboración de la Compañía de Acero del Pacífico, a fines de la década de los sesenta, al donar al CCCI un sistema de capacitación de Analistas de Sistemas de Información, adquirido en Gran Bretaña en aproximadamente tres mil libras esterlinas. Este aporte tuvo un impacto muy significativo en preparar los Programas de capacitación de Programadores y de Analistas de Sistemas de Información, ya que permitió no sólo potenciar la formación de profesionales en computación en la Universidad de Concepción (un programa de postítulo en Análisis de Sistemas que se dictó hasta 1985), sino que además sirvió de base para apoyar en alguna medida la estructuración del Plan Nacional de Capacitación Intensiva en Procesamiento de Datos que fue impulsado en 1975 por las Universidades de Chile, Católica de Chile y la UTE, con la Empresa Nacional de Computación (ECOM) en Santiago, y se inició en 1976 con la UdeC y Crecic en Concepción. En cuanto a los aspectos históricos del uso del computador IBM 1620–II, se orientó inicialmente a la capacitación de profesores y estudiantes, y al apoyo de la investigación. En paralelo se dio curso a la conversión de sistemas de información administrativos convencionales (Abastecimiento, Remuneraciones, etc.). Capítulo especial lo constituyó el sistema de selección de alumnos de la UdeC, aplicación que permitía a pocas horas de anunciarse los resultados de las pruebas de admisión a las Universidades entregados por el Centro de Computación de la Universidad de Chile, procesar los datos básicos en el IBM 1620-II, y obtener las listas de los seleccionados. En esos tiempos, un avión de la U. de Concepción esperaba en el aeródromo de Tobalaba a uno de sus profesionales quien con la cinta magnética bajo el brazo, que contenía la nomina de postulantes a la Universidad de Concepción con sus respectivos puntajes, se dirigía de inmediato de regreso a Concepción, donde se le esperaba para dar paso al procesamiento de datos.

Otra aplicación notable la constituyó un sistema de selección de alumnos del primer año propedéutico, al segundo año, para todas las carreras que ofrecía la Universidad (una experiencia docente innovadora).

Este sistema consistía en presentarle a los alumnos, ordenados según la nota promedio final de primer año (de mayor a menor), aquellas carreras a las que había previamente postulado. Una vez que el alumno decidía en qué carrera se inscribiría, se procedía a restar uno del cupo de la carrera seleccionada. Y así se continuaba con el siguiente, que esperaba su turno.

Lo novedoso del sistema, consistía en transmitir desde el computador, ubicado en el Centro de Computación, la oferta de carreras para dicho alumno y luego que éste decidía en qué carrera se inscribía, se comunicaba al operador del computador, quien reiniciaba el proceso de datos para actualizar los cupos disponibles. Para esto se captaba el resultado desde la impresora del computador con una cámara de televisión (no existían terminales en dicho sistema computacional) y se transmitía a un televisor ubicado en la sala donde estaban los alumnos. La confirmación de la carrera seleccionada se hacía transmitiendo con un micrófono la decisión del estudiante al operador del computador, a la sala en que se encontraban los alumnos. El sistema operaba maravillosamente bien en cuanto a rapidez y precisión, esto sucedía al comienzo y hasta más allá del proceso de selección, sin embargo una vez que comenzaron a agotarse los cupos de las carreras, comenzó a generarse un ambiente de amargura y frustración en los alumnos que no lograban satisfacer sus aspiraciones. Este fenómeno generó alguna impopularidad por ello. No obstante fue un buen anticipo de sistemas de información remotos, en tiempo real e interactivos.

Desde el punto de vista del perfeccionamiento docente, es importante destacar que varios de los profesionales del CCCI obtuvieron su postgrado en universidades del extranjero, como consecuencia de una política de desarrollo impulsada por ellos mismos en aquella época.

Así podemos citar a: José Durán (1969) en la Universidad de Waterloo, Canadá; Marcelo Pardo (1970), Renán Donoso, Atendolfo Pereda y Héctor Rodríguez en (1975), en la Pontificia Universidad Católica de Río de Janeiro (PUC/RJ); y, posteriormente, Atendolfo Pereda, quien obtuvo además el grado de Doctor en 1979 en la misma PUC/RJ. Ese mismo año cursaba su primer año del Magíster, Yussef Farrán. Luego en 1980, Eduardo Méndez, y en 1981, Ricardo Contreras, todos en la PUC/RJ. Posteriormente Gastón Leiva, Cecil Álvarez y Jorge López siguieron programas de postgrado en España.

Especial mención debo hacer de mi profesor, amigo y compadre Dr. Atendolfo Pereda, quien falleció trágicamente en un accidente automovilístico en diciembre de 2005 (y no producto de tortura u otros hechos que se acuñaron en la mitología urbana extranjera) y quien el 1 de marzo de 2013 hubiera cumplido setenta años. Este profesor es gratamente recordado principalmente como un mentor y amigo por sus ex alumnos en los encuentros anuales que estos realizan. Atendolfo o Fito como le llamaban sus familiares y amigos cercanos, fue uno de los dos primeros Doctores en Computación que regresó a Chile a compartir sus conocimientos y experiencia en el área.

Otras personas que pasaron por el área docente del CCCI fueron: Jorge Guzmán, Tatiana Aldea, Nelson Plaza, Héctor Correa, María Teresa Rosende, Juan Durán, Rodrigo Benavente, Waldo Muñoz, Óscar Bull, Sergio Bravo, y Hugh Small (un inglés que llegó producto de un intercambio)¸ además de Claudio Vivaldi, Manuel Riffo, Marco Abusleme, Juan de Dios Cáceres, Alain de Trancalie, Óscar Gericke, Pedro Vergara, Carlos Gallardo, Alfredo Goecke, Norie Fujie, Luis Sotomayor, Alfonso Peirano, Jorge Venegas, Miguel Alehuanlli, Eduardo Vargas, Marcelo Rodríguez, Hugo Molina, Eduardo Méndez, Claudia Jiménez, Pablo Sáez, Esteban Osses, Tomás Arredondo, Luis Rueda, Daniel Campos, Hernán Campos, Patricio Catalán, David Morrison, Christian Weldt, entre otros.

Respecto a la formación de profesionales, en 1970 se creó la carrera de Programador de Computación, nivel Técnico, con duración de tres años vespertino, para trabajadores, y así formar recursos para apoyar a los centros de computación, que algunos le llamaron "Unidad IBM", que comenzaron a aparecer en diversas instituciones y empresas. Luego se reordenó a 2,5 años en régimen diurno para aceptar postulantes egresados de la educación media o secundaria. El último ingreso fue en 1979 y de esta carrera se titularon 157 alumnos muchos de los cuales hoy continúan sus funciones como altos ejecutivos de empresas.

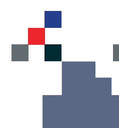
El año 1977 se creó una unidad académica separada de la unidad de servicios de computación, la que pasó a llamarse Instituto de Ciencias de Computación e Informática (ICCI), y, desde 1980, Departamento de Ingeniería de Sistemas producto de la reforma universitaria de la época, siendo su primer Director, Héctor Rodríguez E., sucediéndole Cecil Álvarez, Yussef Farrán y Gastón Leiva. En 1993 se creó el Depto. de Ingeniería Informática y Ciencias de la Computación (DIICC) que ha sido dirigido por Andreas Polymeris, Yussef Farrán, Ricardo Contreras, John Atkinson, Javier Vidal y Andrea Rodríguez.

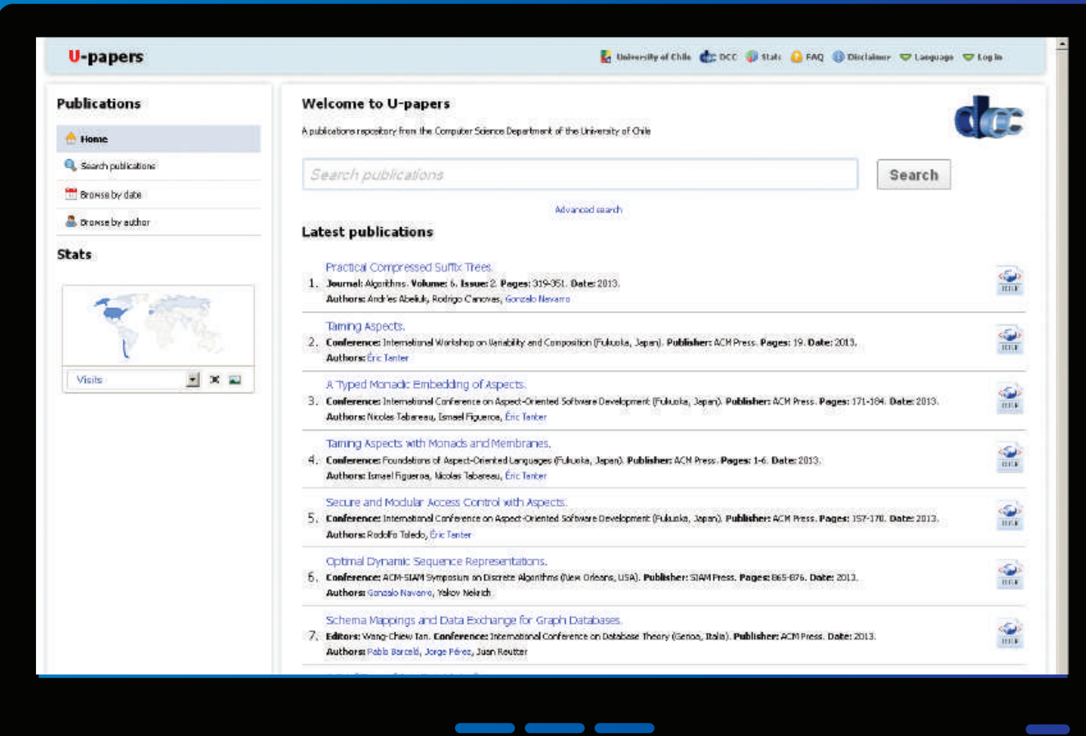
Finalmente, los grandes hitos que se detallarán en un artículo posterior. A partir de 1980 se inicia la carrera de Ingeniería de Ejecución en Computación e Informática con último ingreso el año 1982, carrera de la cual se titularon 277 profesionales. Luego en 1982 se inició la carrera de Ingeniería Civil en Informática teniendo sus primeros titulados en 1985 producto de un plan especial para Ingenieros de Ejecución, y a marzo de 2013 se han titulado de Ingeniería Civil en Informática más de 700 profesionales.

En 1994 inició sus actividades el Magíster en Ciencias de la Computación, formando a más de 33 graduados a la fecha y en 2010 se inició el programa de Doctorado en Ciencias de Computación.

El actual cuerpo académico está compuesto por quince docentes de los cuales nueve tienen Doctorado y seis tienen Magíster, estos son: Andrea Rodríguez, Yussef Farrán, Loreto Bravo, Cecilia Hernández, Marcela Varas, Lilian Salinas, Javier Vidal, Andreas Polymeris, Ricardo Contreras, María Angélica Pinninghoff, Jorge López, John Atkinson, Leo Ferres, Diego Seco y Gonzalo Rojas. BITS

De izquierda a derecha: Óscar Gericke, Yussef Farrán y Eduardo Vargas.





U-papers: Accediendo a las publicaciones científicas del DCC

Felipe Chacón

Ingeniero de Software, Trabajando.com. Ingeniero Civil en Computación, Universidad de Chile (2012). Líneas de especialización: Desarrollo de Sistemas Web, Ingeniería de Software, Desarrollo Web Front-end.

felipe.chacon@gmail.com

José A. Pino

Profesor Titular Departamento de Ciencias de la Computación, Universidad de Chile. Master y calificado a Ph.D. in Computer Science (1977); Master of Science in Engineering, University of Michigan (1972); Ingeniero Matemático, Universidad de Chile (1970). Líneas de Investigación: Sistemas Colaborativos, Sistemas de Apoyo a Decisiones, Interacción Humano-Computador, Educación apoyada con tecnología.

jpino@dcc.uchile.cl

Una de las tareas esenciales de la Universidad es hacer investigación científica, avanzando en la creación de conocimiento. Los resultados de ese quehacer son las publicaciones científicas, que en el caso de la Ciencia de la Computación se reportan principalmente en revistas (*journals)*, conferencias y capítulos de libro. Naturalmente, el interés de los autores y de la Universidad es que esas publicaciones puedan ser leídas por quienes puedan encontrarlas útiles, especialmente si son del país. Esto último por cuanto es la Nación la que de manera directa o indirecta ha financiado parcial o totalmente las investigaciones.

El acceso a los medios de publicación en algunos casos es gratuito. Por ejemplo, hay revistas que están en línea y que no cobran por leer o descargar artículos. Sin embargo, la gran mayoría de los medios cobra por ese acceso y, en algunos casos, el monto es significativo, especialmente si lo está pagando un lector de su bolsillo. Desafortunadamente, muchas veces el lector sólo puede descubrir la relevancia de un artículo para lo que le interesa después de haberlo leído completo.

Para facilitar el acceso a publicaciones, tanto originadas en Chile como en el extranjero, algunas instituciones pagan por la eventual lectura irrestricta de parte de todos sus miembros. Es así como el DCC y/o la Universidad de Chile adquiere todos los años el acceso a interesantes publicaciones, como son aquellas editadas por ACM Press, IEEE Computer Society Press y la serie de libros Lecture Notes in Computer Science de la editorial Springer. Por otro lado, Conicyt paga por el acceso desde las universidades a una gran cantidad de revistas publicadas por grandes editoriales internacionales. En consecuencia, cualquier alumno o profesor de la Universidad de Chile puede leer estas publicaciones sin costo desde un computador conectado a los servidores institucionales.

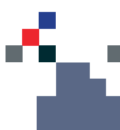
Sin embargo, ¿qué pasa con personas que no son alumnos o profesores de la Universidad de Chile? El acceso para ellos sigue siendo oneroso. Es por esto que diseñamos una manera de proveer acceso gratuito a publicaciones generadas por autores del Departamento de Ciencias de la Computación (DCC) de la U. de Chile. La idea básica fue crear un repositorio que tuviera como contenido las publicaciones del DCC y proveyera acceso irrestricto. Un obstáculo a la existencia del repositorio lo constituyen los derechos de autor, que han sido cedidos a las editoriales que publican los artículos. No obstante, en muchos casos, las editoriales permiten que los artículos sean mostrados en la página web de los autores o de sus departamentos. Además, se podría almacenar en el repositorio sólo borradores de los artículos, no sus versiones finales. Estos borradores puede que sean suficientes para el lector, o provean elementos para que el lector decida si vale la pena comprar la versión definitiva. En todo caso, en el repositorio se incluirían borradores después de un período prudente de tiempo. En el intertanto, la editorial tendría la exclusividad.

U-PAPERS

Con estas ideas en mente, se desarrolló U-papers, un repositorio que lista todas las publicaciones cuyos autores son académicos de jornada completa del DCC. Los académicos deben informar sus publicaciones al Departamento, así es que de la misma declaración se captura la información bibliográfica correspondiente. Además, los autores pueden decidir si quieren que un borrador de la publicación se almacene en el repositorio para acceso universal o no.

El diseño de la interfaz del software es simple, tanto en inglés como en español. El sistema tiene el objetivo de permitir obtener eficientemente lo que el lector busque. Es así como se ofrece búsquedas por autor, por fechas, por palabras clave, palabras del título y/o tipo de publicación.

Pruebe buscar publicaciones de acuerdo a sus intereses: https://upapers.dcc.uchile.cl



EL DESARROLLO DEL SISTEMA

U-papers fue desarrollado usando herramientas tecnológicas open source, trabajo que tomó seis meses, entre agosto de 2011 y enero de 2012. Fue realizado por un alumno del DCC a modo de memoria de título.

Las tecnologías involucradas en el desarrollo de U-papers fueron:

1. Sistema:

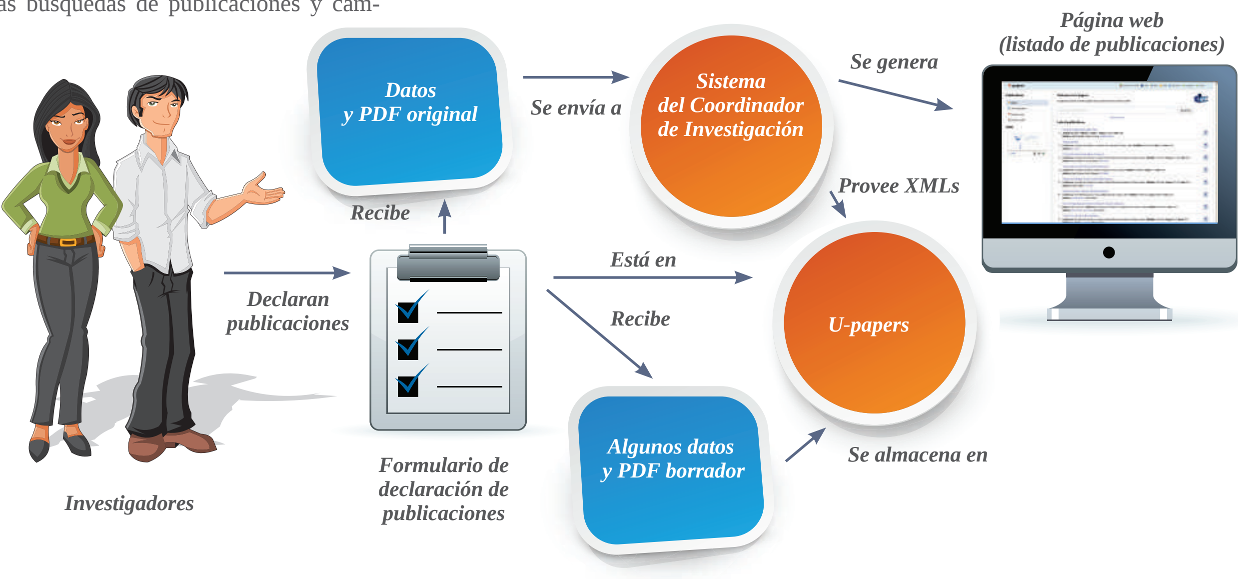
- **Apache HTTP Server:** servidor HTTP usado para la comunicación entre el servidor y el cliente (browser).
- **MySQL:** motor de base de datos. En el caso de U-papers se usó un modelo de datos relacional.
- **PHP:** lenguaje de programación usado para el desarrollo de las páginas dinámicas.

2. Software:

- **CodeIgniter:** Framework PHP para el desarrollo rápido del sistema. Permite el uso ordenado del patrón MVC.
- **jQuery:** librería Javascript que permite realizar acciones en el lado del cliente.

U-papers es un sistema web que utiliza una arquitectura de tres capas: Presentación, Negocios y Datos. Está relacionada directamente con el patrón MVC, en donde la Presentación se asocia con las Vistas; la de Negocios con el Controlador y los Datos con el Modelo. A continuación un esquema de lo mencionado:

Si bien lo anterior permitió el desarrollo completo del sistema, su utilización no fue lo desafiante. Lo interesante fue implementar una herramienta que facilitara las búsquedas de publicaciones y cambiara las prácticas en el DCC respecto a la declaración de las mismas. Debía ser un sistema que se las arreglara con la información ya existente y se acoplara a la forma en que ésta se podía entregar. En modo gráfico, la incorporación de la información en U-papers se hace de la siguiente manera:



País	Visitas
Chile	703
Estados Unidos	158
España	83
México	30
Francia	12

Figura 1 • Cantidad de visitantes por país.

El sistema U-papers es el relevo de la información que se entregaba en la página institucional del DCC. Al principio el sistema iba a ser alimentado con la información existente en una página web privada, pero esa idea fue desechada. Finalmente, se optó que fueran los mismos académicos los gestores directos del poblamiento de la base de datos de U-papers. Se construyó un subsistema en el que los académicos declaran sus publicaciones y al cabo de un tiempo, se publican sus trabajos en el sitio principal.

Ahora, aproximadamente a un año de su lanzamiento, U-papers cuenta con 23 académicos activos, más de 500 publicaciones y visitas desde 33 países distintos. En las Figuras 1, 2 y 3 se presentan algunos datos relevantes.

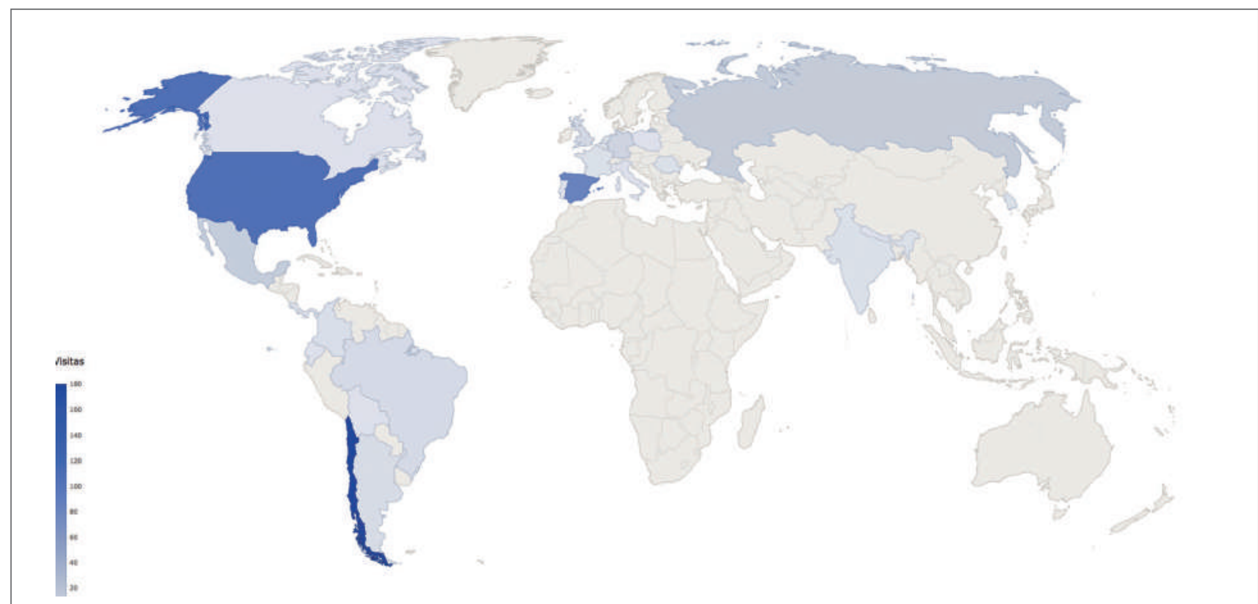
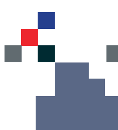


Figura 2 • Distribución de visitantes (más azul → más visitantes).

Artículos de journal	*30*
Artículos de conferencias	*59*
Capítulos de libros	*2*
Publicaciones totales	*91*

Figura 3 • Publicaciones totales por tipo (año 2012).

Con estos datos se puede observar que U-papers es un sistema en crecimiento, cuyo primer año de vida se puede catalogar como exitoso. Se espera que esos números aumenten y que los académicos permitan el acceso a la mayor cantidad de publicaciones posibles para que así U-papers cumpla su objetivo de dar visibilidad a la investigación realizada en el DCC. BITS



Kurt Gödel y Alan Turing: una nueva mirada a los límites de lo humano

Claudio Gutiérrez

Profesor Asociado Departamento de Ciencias de la Computación, Universidad de Chile.
Ph.D. Computer Science, Wesleyan University; Magíster en Lógica Matemática, Pontificia Universidad Católica de Chile; Licenciatura en Matemáticas, Universidad de Chile. Áreas de investigación: Fundamentos de la Computación, Lógica aplicada a la Computación, Bases de Datos, Semántica de la Web, Máquinas Sociales.
cgutierr@dcc.uchile.cl

¿Cuál es la contribución intelectual que Gödel y Turing hicieron a la humanidad? Usualmente se sostiene que Gödel demostró que los sistemas formales son febles, esto es, los programas totalizadores de formalización del razonamiento están inevitablemente destinados a fracasar. Y en la misma línea, usualmente se sostiene que el gran aporte de Turing fue clarificar el fundamento teórico que subyace a los computadores y sus relaciones con los sistemas formales. En otras palabras, sus aportes habrían sido mostrar los alcances y las limitaciones intrínsecas del uso de lenguajes formales para representar el mundo que nos rodea.

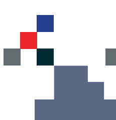
En lo que sigue argumentaré que Gödel y Turing fueron mucho más allá de eso: revolucionaron nuestra manera de conceptualizar el entendimiento de lo humano. Cabe a Kurt Gödel haber llevado los argumentos sobre el *razonamiento humano* hasta sus últimas consecuencias, haber disecado esa noción hasta entender su esencia misma, haber desentrañado su mecanismo escondido. El gran aporte de Gödel fue demostrar –usando la lógica matemática para ello– que el razonamiento humano está indisolublemente ligado a una noción material, mecánica, la de *procedimiento*. Un procedimiento es una serie de acciones materiales que conducen a un resultado dado. Casi en paralelo, Alan Turing analiza la noción de procedimiento y, a partir de un análisis profundo de las acciones que normalmente lleva a cabo un humano para calcular, enlaza esa noción material con la noción abstracta de *función recursiva*, que el mismo Gödel había formulado en el intertanto. Así, entre ambos, enlazan la noción de acción humana, de hacer humano, con una familia de formalismos, con un lenguaje de especificación. Podríamos decir que enlazan la materia con el espíritu, o para entendernos mejor, enlazan el *hardware* y el *software* humanos.

Como ocurre con todos los procesos verdaderamente revolucionarios, quien hace un forado al dique que sujeta el viejo sistema, inesperadamente es arrastrado por el subsiguiente aluvión sin que generalmente alcance a entender qué cambió. Es lo que le ocurrió a Gödel y a Turing. Entre ambos –sin proponérselo explícitamente– trenzaron sus ideas estableciendo los fundamentos y los alcances del razonamiento humano y una de las hipótesis más relevantes que tenemos hoy sobre las relaciones entre la materia y el espíritu. De esta historia hablaremos en lo que sigue.

GÖDEL Y TURING: CERCANÍAS Y DESENCUENTROS

Partamos presentando a nuestros dos protagonistas. Kurt Gödel (1906-1978) fue un matemático austríaco que hoy es considerado el lógico más relevante desde Aristóteles. Muy joven se interesó por la lógica y los fundamentos de las matemáticas. En 1931, cuando tenía 25 años, publicó su trabajo sobre la incompletitud de los sistemas formales que cambiaría para siempre nuestro entendimiento de los sistemas formales y la lógica. Gödel era una persona muy reservada, profunda y obsesiva (en su infancia su familia le apodaba "Señor Por qué"). Hay una anécdota que muestra muy bien su carácter. Al llegar a Estados Unidos arrancando del nazismo que había anexado Austria, para nacionalizarse debió estudiar la constitución de los Estados Unidos. Detallista, descubrió que este entramado lógico-legal permitiría una dictadura como la nazi. Sus amigos Einstein y Morgenstern intentaron desesperadamente convencerlo que no mencione eso en público, que era un caso muy poco probable de darse. Durante el examen de inmigración, apareció el tema de las formas de gobierno y Gödel orgulloso le indica al examinador que tiene una demostración de la posibilidad de una dictadura. Por suerte el funcionario conocía a Einstein y las extravagancias de estos personajes, y rápidamente le cambió de tema. Gödel vivió tranquilo el resto de su vida en Princeton y al final de sus días se obsesionó con que alguien lo podía envenenar; así murió de malnutrición crónica. Una biografía completa de Gödel es la de Dawson [4]. Los recuerdos de su amigo Hao Wang [14] sobre él son muy interesantes también.

Alan Turing (1912-1954) fue un matemático inglés, de amplios intereses científicos. También desde muy joven mostró signos de genio en el estudio de las matemáticas y motivado por los resultados de Gödel se interesó en la lógica matemática. Captó rápidamente la esencia misma del resultado de Gödel y por su cuenta, sin guía formal, dio el siguiente paso (que no vislumbró Gödel) desarrollando su famoso formalismo (conocido hoy como "máquina de Turing") que captura la noción de procedimiento "efectivo" (esto es, que efectivamente puede realizarse). Con esos antecedentes se va a Princeton donde continúa estudiando con Alonzo Church y se doctora en 1938. Vuelve a Inglaterra en los años iniciales de la Segunda Guerra y se enrola entre los científicos que trabajaban



en asuntos militares, en su caso, descifrando códigos enemigos. Después de la guerra, se dedica a construir el primer computador inglés. Escribe sobre diversas áreas. El Gobierno inglés lo persiguió por su condición homosexual y debido a ello, también muere trágicamente. Una excelente biografía es [10]. Su obra está en línea en http://www.turing.org.uk/.

Aunque Turing y Gödel nunca se encontraron físicamente (al parecer tampoco intercambiaron correspondencia), cada uno "descubrió" al otro a su manera. Cuando estaba en Princeton, Turing escribía a su madre: *"Aquí hay un gran número de los más distinguidos matemáticos: J. v. Neumann, Weyl, Courant, Hardy, Einstein, Lefschetz y otros. Desafortunadamente no hay muchos lógicos. Church está aquí, pero Gödel, Kleene, Rosser y Bernays que estuvieron el año pasado, se fueron. No creo que me importe mucho ninguno excepto Gödel"*. En la constelación de genios de Princeton, Einstein y Von Neumann incluídos, a Turing sólo le interesó Gödel.

La admiración (secreta) era recíproca. También a Gödel lo que más le llamó la atención de la impresionante obra lógica de los años treinta, fue la obra de Turing. Ese formalismo logró convencerlo que finalmente se había encontrado la representación de la noción de procedimiento efectivo que los matemáticos andaban buscando hace mucho tiempo. En particular, Gödel le vió una consecuencia muy relevante que se deducía de él y escribió: *"Debido al trabajo de A. M. Turing, es posible dar ahora una definición precisa e incuestionable del concepto general de sistema formal"*. Ahora es Gödel quien ve más allá del propio Turing. Cuando Turing se adentraba en otros campos dejando atrás sus investigaciones teóricas sobre lo computable, Gödel continuó la reflexión sobre ello.

ANTECEDENTES

Debemos fijar un comienzo, y lo haremos en el cambio del siglo XIX al XX. Probablemente se trata del período fundacional de lo que será el gran salto conceptual que darán Gödel y Turing en los años treinta del siglo XX.

El origen de esta revolución conceptual puede remontarse a la concepción del mundo que se inicia con la filosofía y la ciencia moderna, y a su programa para instalar el análisis, la descomposición de un fenómeno complejo en sus partes simples constituyentes, como metodología para evitar errores en el razonamiento y profundizar el conocimiento. Los pensadores y científicos advierten sobre los problemas que trae al conocimiento la vaguedad del lenguaje natural y la falta de precisión en los conceptos.

En el área de la lógica, el fondo de estas inquietudes podemos presentarlo hoy de la siguiente forma. Si el razonamiento no es más que una sucesión finita de argumentos muy precisos, cada uno encadenado al siguiente por mecanismos también precisos, lo que los matemáticos llaman una demostración; ¿es posible llevar cada uno de esos encadenamientos, cada uno de esos eslabones, a un extremo de claridad y precisión de tal forma que sean "evidentes" (esto es, no se necesite "inteligencia" para verificarlos)? De esta pregunta surgieron dos temas que desarrollaremos en lo que sigue: por una parte, *los alcances de la formalización tanto del lenguaje como de la lógica* y, por otro, *los mecanismos de descripción de conceptos y objetos matemáticos* de forma que sean susceptibles de ser "construidos" a partir de otros básicos y evidentes.

La formalización del lenguaje

A comienzos del siglo XX la comunidad lógico-matemática trabajaba febrilmente en un programa propuesto por el renombrado matemático David Hilbert, cuyos antecedentes se pueden rastrear en los escritos de lógicos medievales como Raimundo Lulio, luego en Leibniz, y en quienes soñaron con "mecanizar" el razonamiento. Este programa consistía en la *formalización total del razonamiento matemático* y la reducción del razonamiento a sus partes atómicas evidentes. El siguiente párrafo de Hilbert ilustra muy bien sus motivaciones:

> "Si se quiere que la inferencia lógica sea confiable, debe ser posible inspeccionar esos objetos [concretos matemáticos] completamente en todas sus partes, y que junto con esos objetos, se determine inmediata e intuitivamente el hecho de que estén allí, que difieren unos de otros, y que uno sigue a otro, o que estén concatenados, como algo que no puede ser reducido a ninguna otra cosa que requiera reducción. Ésta es la posición filosófica básica que considero requisito para las matemáticas, y en general, para todo pensamiento, entendimiento y comunicación científica [9]".

Como vemos, el programa tenía fuertes motivaciones prácticas. En particular, los fundamentos del análisis matemático, especialmente el tratamiento del sospechoso concepto de infinito y de número infinitesimal hacía imperiosa la necesidad de contar con algún sistema formal que permita razonar sobre estos objetos sin incurrir en errores (por ejemplo, las llamadas paradojas). Los fundamentos mismos de la ciencia "más segura", la ciencia "exacta" por antonomasia, se veían temblorosos.

A comienzos del siglo XX se habían conseguido grandes avances. El monumental tratado *Principia Mathematica* de Whitehead y Russell por un lado, y la emergente *teoría de conjuntos* de Zermelo-Fraenkel por otro, entregaban sistemas formales en los cuales se podía expresar formalmente toda la matemá-

tica conocida. Sólo quedaba entonces demostrar que esos sistemas eran coherentes, esto es, no tenían inconsistencias internas. Es ésta la tarea que se propuso el joven Kurt Gödel a comienzos de los años treinta, y cuyo sorprendente resultado analizaremos más adelante.

Transformaciones efectivamente calculables

Junto al infinito, hay otra noción que transformó las matemáticas modernas: el concepto de función. Aunque ahora nos parezca trivial, la noción de función, el tratarla como un objeto en sí, no es nada sencillo. Lo que se está abstrayendo realmente es la noción de transformación, que tanto dolor de cabeza ha producido a los filósofos.

Y cuando se junta la noción de función con la de infinito, aparecen problemas serios. Una función (en matemáticas) transforma números (en general objetos matemáticos) en otros. ¿Cómo especificar una tal transformación, especialmente si la cantidad de objetos sobre los que se aplica es infinita? Fue Richard Dedekind en 1888 el primero que propuso un método –muy elemental por lo demás– que pasaría a la historia como *definición inductiva o recursiva*. Dedekind ya hablaba así: “Por una transformación T entenderemos una ley según la cual a cada elemento determinado s de un sistema S le corresponde una cosa determinada” y propone que para transformaciones sobre los números naturales, se especifique su valor para 1 y se reduzca su cómputo para un número cualquiera al cómputo del anterior mediante otra función Θ de la siguiente manera[1]:

$$T(1) = \omega,$$
$$T(n+1) = \Theta(T(n)).$$

La definición es algo limitada. Por ejemplo, no es posible especificar en este formalismo la función factorial $f(n+1) = (n+1) f(n)$. Peano generaliza esta definición y establece en 1889 una noción más amplia conocida posteriormente como *recursión primitiva*. Una función f de varias variables se especifica así:

$$(1)\; f(0, y_1,\ldots, y_n) = h(y_1,\ldots, y_n)$$
$$(2)\; f(x+1, y_1,\ldots, y_n) = g(x, f(x, y_1,\ldots,y_n), y_n,\ldots, y_n)$$

donde g y h son funciones definidas previamente por recursión también. Nótese que esta definición es mucho más general y poderosa. En particular, la función factorial se puede especificar ahora poniendo $f(0) = 1$ y $f(x+1) = mult(x, f(x))$, donde *mult* es la multiplicación usual de enteros que se puede definir a su vez por las fórmulas (1) y (2)[2].

La gracia de todas estas definiciones es que dadas la entradas (el *input*), es posible *mecánicamente*, esto es, sin gran esfuerzo mental y sin necesidad de “ingenio”, calcular el valor de la función para esa entrada en un número finito de pasos. Esto es lo que en la época le llamaban una *función efectivamente calculable*. ¿Cuánto es posible generalizar esta idea? ¿Es posible tener un formalismo en el que se pueda especificar todas las transformaciones que son efectivamente calculables? Este problema está estrechamente ligado al anterior sobre los *procedimientos* efectivos en un razonamiento. La transformación (función) aparece allí como el eslabón que permitiría enganchar un argumento con otro en una demostración o razonamiento.

EL COMIENZO: GÖDEL 1931

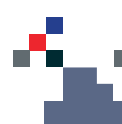
Hacia 1930 el joven Kurt Gödel trabajaba en el programa de Hilbert. En particular, intentaba demostrar la consistencia de la aritmética, que se consideraba el fundamento del análisis matemático clásico. Fue él quien se tomó más en serio ese programa y fue también quien terminó por destruirlo. Esto es, intentando demostrar la consistencia de la aritmética, descubrió que ese era un objetivo imposible.

El trabajo de Gödel, titulado *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme* (“Sobre sentencias formalmente indecidibles de Principia Mathematica y Sistemas afines”), de modestas 25 páginas, fue escrito el año 1930 y publicado en 1931 en la revista *Monatshefte für Mathematik und Physik*. Allí Gödel se propone como objetivo principal demostrar lo que hoy se conoce como el “Teorema de incompletitud de Gödel”. Dejemos que él mismo nos explique, con las palabras introductorias de su artículo, en qué consiste su contribución:

> “Como es sabido, el progreso de la matemática hacia una exactitud cada vez mayor ha llevado a la formalización de amplias partes de ella, de tal modo que las deducciones pueden

[1] No es difícil mostrar que es suficiente contar con una buena definición sólo para los números naturales. Para quienes quieran conocer estas ideas fundacionales de Dedekind, recomiendo el excelente librito de R. Chuaqui: “¿Qué son los números? El Método Axiomático”. Edit. Univ. 1980. Allí traduce y comenta la obra central de Dedekind sobre este tema. La cita es la Definición 21, p. 45, y la definición de función recursiva es el Teorema de la Definición por Inducción, p. 88.

[2] Ejercicio para el lector: esto último no es totalmente cierto. Hay una linda historia asociada a este detalle. Resulta que con la formulación de Peano no es posible definir la multiplicación pues necesita recursión doble. Nadie lo había advertido, hasta que Karl Grandjot –la historia transcurre en Göttingen– puso la observación en los márgenes de un libro que su profesor Edmund Landau le prestó para hacer las clases. El genial Karl Grandjot luego se vino a Chile y fue profesor de la Universidad de Chile durante el resto de su vida. Ver [8].



llevarse a cabo según unas pocas reglas mecánicas. Los sistemas formales más amplios construídos hasta ahora son el sistema de *Principia Mathematica* (PM) y la teoría de conjuntos de Zermelo-Fraenkel (desarrollada aún más por J. von Neumann).

Estos dos sistemas son tan amplios que todos los métodos usados hoy día en la matemática pueden ser formalizados en ellos, es decir, pueden ser reducidos a unos pocos axiomas y reglas de inferencia. Resulta por tanto natural la conjetura de que estos axiomas y reglas basten para decidir *todas* las cuestiones matemáticas que puedan ser formuladas en dichos sistemas[3]. En lo que sigue se muestra que esto no es así, sino que por el contrario, en ambos sistemas hay problemas relativamente simples de la teoría de los números naturales que no pueden ser decididos con sus axiomas (y reglas)".

Su resultado formalmente dice así:

Teorema (de Incompletitud). Sea K un conjunto recursivo primitivo de fórmulas, y S un sistema deductivo para K que cumple ciertos requisitos mínimos[4], y sea $dem(G)$ el conjunto de proposiciones que se pueden demostrar en el sistema formal a partir del conjunto de fórmulas G. Entonces hay una proposición A en K tal que $\sim A$ pertenecen a $dem(K)$.

Expresemos el significado profundo, las consecuencias, de este resultado con sus propias palabras:

"Todos ustedes conocen las famosas palabras de Hilbert sobre que todo matemático está convencido de que para cada pregunta matemática precisamente formulada es posible encontrar una única respuesta y que es exactamente esta convicción el estímulo fundamental del trabajo de investigación matemático. Hilbert mismo estaba tan convencido de esto que aún pensaba que era posible que se diera una demostración matemática de esto, al menos en el dominio de la teoría de los números.

¿Cómo podemos imaginar que una tal demostración pueda ser obtenida? Para buscarla primero tenemos que analizar el significado del teorema a ser demostrado. Para cada hombre desprejuiciado esto sólo puede significar lo siguiente: dada una proposición matemática cualquiera A, existe una demostración para A o para no-A, donde por "demostración" se entiende algo que parta de axiomas evidentes y proceda por inferencias evidentes. Ahora, formulado de esta manera, el problema no es susceptible de tratamiento matemático pues involucra las nociones no-matemáticas de evidencia. Luego lo primero que hay que hacer es explicitar esta noción a través del análisis de las demostraciones matemáticas reales. Si eso se hace –y ha sido hecho por la lógica matemática y por la *Teoría de la Demostración* de Hilbert– entonces nuestro problema puede ser tratado matemáticamente y la respuesta resulta ser negativa aún en el dominio de la teoría de los números.

Pero es claro que esta respuesta negativa tiene dos significados diferentes: (1) puede significar que el problema en su formulación original tiene una respuesta negativa, o (2) puede significar que algo se perdió en el proceso de transición de la evidencia al formalismo. Es fácil ver que realmente ocurrió lo segundo, puesto que las preguntas de la teoría de los números que son indecidibles en un formalismo dado son siempre decidibles por inferencias evidentes que no son expresables en el formalismo dado. Respecto de la evidencia de estas nuevas inferencias, ellas resultan ser tan evidentes como las del formalismo dado. Luego el resultado es más bien que no es posible formalizar la evidencia matemática aún en el dominio de la teoría de los números, pero la convicción acerca de la cual Hilbert hablaba permanece enteramente intocada".

Es importante destacar los asuntos cruciales que están involucrados en la argumentación anterior: la noción de sistema formal y la de inferencia. Está implícito aquí el hecho que esta inferencia y este sistema formal deben expresarse por medio de transformaciones "efectivas".

INTERMEDIO: FUNCIONES RECURSIVAS Y TESIS DE CHURCH

En paralelo al espectacular resultado de Gödel, continuaba la búsqueda de formalismos que especificaran la noción de función efectiva (ver los

[3] Un sistema que cumpla esto, o sea, que pueda decidir todas las cuestiones que puedan ser formuladas en él se dice *completo*. En 1930 el mismo Gödel había demostrado en otro célebre teorema, que la lógica (pura) de primer orden (la lógica "usual") era completa. Esto significa que los axiomas y reglas de deducción usuales son suficientes para decidir todas las cuestiones lógicas (puras) que pueden ser formuladas allí. Cuando se le agrega la teoría de números eso ya deja de ser cierto, y el sistema se torna incompleto.

[4] Estoy evitando detalles técnicos engorrosos. "Requisito mínimo" refiere esencialmente a la aritmética. Para una exposición simple de la idea central de la demostración ver [7]. Para la demostración misma, ver [6].

detalles de esta entretenida historia en [1]). Volvamos a la definición (algo ingenua) de Peano, y veamos qué más necesitamos para especificar completamente el sistema. Después de un análisis simple, surge la necesidad de definir las siguientes nociones:

(3) La función sucesor
$s(x) = x + 1$,
que aparece como noción primitiva (no definida);
(4) La función constante
$f(x_1,\ldots, x_n) = 0$
(que también aparece como noción primitiva; no es difícil ver que cualquier otra función constante se puede definir a partir de ésta);
(5) La función proyección
$\pi_k(x_1,\ldots, x_n) = x_k$
(para pasar subconjuntos de argumentos a otra función);
(6) La composición de funciones u operador de substitución (que se usa en varias partes).

Agregando las nociones (3) a (6) a la definición de Peano (ecuaciones (1) y (2) más arriba), queda completamente definida la noción de *función recursiva primitiva.*

Pronto quedó en evidencia que aún este formalismo no lograba capturar todo lo que era efectivamente calculable. Uno de los contraejemplos más conocidos es la función de Ackermann (un alumno de Hilbert). Esta extraña función de tres argumentos $\varphi(m,n,p)$ no se puede especificar con el formalismo de funciones recursivas primitivas, y declara, por iteración, operaciones aritméticas cada vez más complejas:

$$\varphi(m, n, 0) = m + 1 + \cdots + 1 \ (n \text{ veces}) = m + n \ (suma)$$
$$\varphi(m, n, 1) = m + \cdots + m \ (n \text{ veces}) = m \cdot n \ (multiplicación)$$
$$\varphi(m, n, 2) = m \cdots m \ (n \text{ veces}) = m^n \ (exponenciación)$$
$$\varphi(m, n, 3) = m^{m^{\cdot^{\cdot^{m}}}} \ (n \text{ veces}) = \cdots \ (\text{sin nombre conocido})$$
$$\varphi(m, n, 4) = \text{ejercicio para el lector}$$

Una vez más Gödel, esta vez inspirado por sugerencias del lógico francés Jacques Herbrand, consigue desarrollar un formalismo (basado en ecuaciones funcionales) que logra extender la noción de función efectiva para incluir funciones más generales como la de Ackermann; formalismo que hoy se conoce como *funciones recursivas.*

El lógico Stephen Kleene simplificó la formulación de Gödel, agregando a la especificación que tenemos (ecuaciones (1) - (6)) el siguiente operador, al que bautizó como operador μ, y que está definido para una función primitiva recursiva $f(x, y_1,\ldots,y_n)$ de la siguiente manera:

$$(7)\ \mu(x) f(x, y_1,\ldots, y_n) = \min_{z\geq 0} \exists y_0 \ldots \exists y_n f(z, y_1,\ldots, y_n) = 0$$

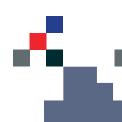
En castellano: el valor de la función μf se obtiene "buscando", a partir de $z = 0$ números $y_1,\ldots, y_n$ que satisfacen $f(z_0, y_1,\ldots,y_n) = 0$. Si no existen tales números para $z = 0$, seguimos buscando con $z = 1$, y así sucesivamente, hasta encontrar los primeros $z_0, y_1,\ldots, y_n$ que hacen $f(z_0,y_1,\ldots,y_n) = 0$. Allí la búsqueda se detiene y se retorna z_0. Nótese que con esto se introduce la posibilidad de que una función no retorne ningún valor; esto es, que no sea total[mente] definida para todos los valores de sus argumentos. Por ello muchas veces se llama a este conjunto funciones recursivas *parciales.*

Tesis de Church. En la búsqueda de formalismos que capturaran la noción de transformación, Alonzo Church, de Princeton, propuso poco después que Gödel y Kleene formalizaran la noción anterior, el *cálculo lambda*. Este cálculo captura la noción abstracta de función a partir de un análisis de la noción de evaluación [2].

Consideremos la función que saca la raíz cuadrada de un número. Cuando se escribe $\sqrt{3}$, lo que el común de los matemáticos entiende es que éste es el valor que retorna la función al evaluarla en 3. Pero, ¿cómo escribir –especificar– entonces la sola aplicación de la función "sacar raíz cuadrada" aplicada a *a*, pero sin evaluarla aún? Para ello, en primer lugar, hay que tener un formalismo para describir funciones (sin que necesariamente se apliquen a nada aún). Church usa la notación $\lambda x[M]$, que interpretada como programa computacional, consta de λx, que indica una "variable" *x,* y *M* que especifica el "código" o "programa" que usa esa variable. En segundo lugar, es necesario definir la noción de aplicación de la función (nuestro $\lambda x[M]$) sobre un argumento (que denotaremos *A*, y que es otra expresión cualquiera del cálculo lambda), y que Church denotó $\{\lambda x[M]\}(A)$. La evaluación entonces sera el "procedimiento" (¡aquí aparece la noción de procedimiento mecánico!) que consiste en reemplazar cada ocurrencia de la variable *x* en *M* por el argumento *A*. Por ejemplo al evaluar $\{\lambda x\ \sqrt{x^2+2x}\}(3)$ resulta, $\sqrt{3^2+2\bullet 3}$, y si evaluamos las operaciones aritméticas básicas en un segundo paso llegamos a $\sqrt{15}$. Nótese que con este formalismo se pueden aplicar funciones a funciones, etc[5].

En un artículo publicado en 1936, *An unsolvable problem of elementary number theory* ("Un problema insoluble de la teoría elemental de los números"), basado en ideas presentadas informalmente el año anterior, Church demuestra que este forma-

[5] Y aparecen también expresiones "infinitamente recursivas", esto es, que no terminan nunca de evaluarse. Por ejemplo {λx[{x}(x)]}(λx[{x}(x)]) en la notación horrible de Church, o (λx(xx))(λx(xx)) en notación actual. Nótese que al evaluar esta expresión, esto es al reemplazar la variable x en la expresión de la izquierda por el argumento λx(xx), queda la misma expresión. Luego hay que evaluarla de nuevo y así sucesivamente.



lismo es equivalente a (puede especificar las mismas funciones que) el conjunto de las funciones recursivas parciales. Lo que quedó para la historia es que a partir de esta "evidencia", Church avanza la audaz tesis de que esta noción captura precisamente el concepto de función efectivamente calculable (Gödel tiempo antes había comentado en privado sobre las relaciones entre el formalismo de las funciones recursivas y lo efectivamente calculable, pero consideró que la evidencia recogida hasta ese momento no era suficiente para asociarlas). Church formula así su tesis:

> "El propósito del presente artículo es proponer una definición de calculabilidad efectiva que se piensa que corresponde satisfactoriamente con la algo vaga intuición en términos de los cuales los problemas [para los cuales se requiere una función efectivamente calculable] son formulados, y mostrar, por medio de un ejemplo, que no todos los problemas de esta clase son solubles".

La afirmación anterior es lo que se conoce como Tesis de Church. "Tesis" porque esta afirmación no es susceptible de demostración, pues relaciona un formalismo matemático con una noción ("efectividad") que es, digamos, filosófica. Como indicamos, aunque Gödel la intuyó antes, seguía considerando que no existía suficiente evidencia para sostenerla. Esto cambió cuando apareció el artículo de Turing de 1936.

FINALE: TURING 1936

En 1936 Alan Turing –de sólo 25 años– publicó, casi simultáneamente con el artículo de Church mencionado anteriormente, su trabajo titulado *On Computable Numbers with an Application to the Entscheidungsproblem* ("Sobre números computables con una aplicación al problema de decisión"). En él analiza en detalle el proceso de razonamiento (de "cómputo") humano y en base a ello propone una máquina que simula ese mismo proceso. A continuación, demuestra que ese formalismo es equivalente con la noción de función recursiva.

Consideramos interesante reproducir ese análisis de Turing, que hoy sorprende por su sencillez (el artículo original está reproducido en [3], pero es fácil encontrarlo en la Web):

> "Un cómputo se hace normalmente escribiendo ciertos símbolos en papel. Podemos suponer que este papel está normalmente dividido en cuadrados, como los cuadernos de aritmética de los niños. En aritmética elemental se usa a veces el carácter bidimensional del paper. Pero tal uso es evitable, y creo que se puede convenir que este carácter bidimensional no es esencial para el cómputo. Asumo entonces que el cómputo se lleva a cabo en un papel de una dimensión, esto es, en una cinta dividida en cuadrados.
>
> Supondré también que el número de símbolos que se puede imprimir es finito. Si permitiéramos un número infinito de símbolos, entonces habría símbolos que difieren en un grado arbitrariamente pequeño. El efecto de esta restricción sobre el número de símbolos no es muy seria. Siempre es posible usar secuencias de símbolos en lugar de símbolos individuales. Luego un numeral arábico como *17 o 999999999999999* se tratará normalmente como un símbolo. Similarmente en cualquier lengua europea las palabras son tratadas como símbolos individuales (el chino, sin embargo, intenta tener una cantidad infinitamente enumerable de símbolos). Las diferencias, desde nuestro punto de vista, entre símbolos individuales y compuestos es que los compuestos, si son muy largos, no pueden ser observados con una sola mirada. Esto es de acuerdo a mi experiencia. No podemos decir con una mirada si *9999999999999999 o 999999999999999* son iguales.
>
> El comportamiento de quien computa en cada momento está determinado por los símbolos que está observando, y su "estado mental" en ese momento. Podemos suponer que hay una cota B al número de símbolos o cuadrados que puede observar en cada momento quien computa. Si quiere observar más, debe usar observaciones sucesivas. Podemos suponer también que el número de estados mentales que necesitan ser considerados es finito. Las razones para ello son del mismo carácter que restringen el número de símbolos. Si admitiéramos infinitos estados mentales, algunos de ellos estarían "arbitrariamente cercanos" y se confundirían. De nuevo, esta restricción no afecta seriamente el cómputo, puesto que el uso de estados más complejos se puede evitar escribiendo más símbolos en la cinta.
>
> Imaginemos ahora las operaciones que hace quien computa descompuestas en "operaciones simples" que son tan elementales que no sea fácil imaginarlas divididas más aún. Cada tal operación consiste en algún cambio de estado del sistema, el cual se compone de quien computa y su cinta. Conocemos el estado del sistema si conocemos la secuencia de símbolos sobre la cinta, cuáles de ésos son observados por quien computa (posiblemente en algún orden especial), y el estado mental de quien computa. Podemos suponer que en una operación simple no se altera más de un símbolo. Cualquier otro cambio puede descomponerse en cambios simples de este tipo. La situación respecto a los cuadrados cuyos símbolos pueden ser alterados de esta manera, es la misma que la de aquellos cuadrados

que sólo se observan. Podemos, por consiguiente, asumir sin pérdida de generalidad que los cuadrados cuyos símbolos se cambian son siempre cuadrados "observados".

Aparte de estos cambios de símbolos, las operaciones simples deben incluir cambios de distribución de los cuadrados observados. Los nuevos cuadrados observados deben ser inmediatamente reconocibles por quien computa. Pienso que es razonable suponer que ellos pueden ser sólo cuadrados cuya distancia desde el más cercano de los cuadrados previamente observados no exceda cierta cantidad fija. Digamos que cada uno de los nuevos cuadrados observados está a L cuadrados [de distancia] de un cuadrado observado inmediatamente antes. [...]

Podemos ahora construir una máquina que haga el trabajo de quien computa. Para cada estado mental de quien computa corresponde una "m-configuración" de la máquina ["m" por máquina]. La máquina escudriña [escanea] B cuadrados correspondientes a los B cuadrados observados por quien computa. En cualquier movimiento la máquina puede cambiar un símbolo sobre un cuadrado escudriñado o puede cambiar cualquiera de los cuadrados escudriñados por otro cuadrado distante no más de L cuadrados de alguno de los otros cuadrados escudriñados. El movimiento que se hace, y la subsecuente configuración, están determinados por el símbolo leído y la m-configuración".

Luego de hecho este análisis, Turing propone el formalismo conocido hoy como "máquina de Turing", que es simplemente la abstracción de cada una de las nociones mencionadas anteriormente. Una cinta cuadriculada infinita de una dimensión, un alfabeto finito de símbolos, un cabezal que puede "leer" y "escribir" símbolos en cada uno de esos cuadrados, y que puede moverse de un cuadrado a su vecino a izquierda o derecha, y un sistema de "estados". La operación de la máquina está codificada esencialmente por una función que a cada par (símbolo-leído, estado) le asocia: un movimiento a izquierda o derecha en la cinta; la escritura de un símbolo en el nuevo cuadrado de la cinta; y un nuevo estado. Formalmente:

Una *máquina de Turing* es un cuádruple (K,S,e,δ) donde:
1. K es un conjunto finito de estados. Hay un estado especial h que se llama estado de detención *(halt state)*.
2. S es un conjunto de símbolos (el alfabeto de la máquina). Hay además tres símbolos extras: el símbolo # (que simula el cuadrado blanco), y otros dos, D e I (que indican el movimiento a derecha e izquierda respectivamente).
3. e es el estado inicial con el cual parte la máquina.
4. δ es una función de $K \times S \rightarrow (K \cup \{h\}) \times (S \cup \{D,I\})$.

La máquina funciona así: si está en el estado q y el cabezal leyendo el símbolo a, entonces el valor de la función $\delta(q,a)$ indica qué hacer de la siguiente manera: si $\delta(q,a)=(p,b)$, entonces se sobrescribe el símbolo a en el cuadrado que se está leyendo con el símbolo b y se cambia el estado a p. Si $\delta(q,a)=(p,D)$, se mueve el cabezal a la derecha y cambia el estado a p. Análogamente con $\delta(q,a)=(p,I)$. Finalmente, si $\delta(q,a)=(h,-)$, donde $-$ indica cualquier símbolo, la máquina se detiene.

Turing muestra –entre muchos otros resultados– que este formalismo es equivalente a los formalismos de Church y de Gödel, esto es, describe exactamente las funciones recursivas parciales.

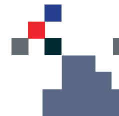
Es posible mirar esta contribución de Turing al menos desde dos perspectivas. La primera, como la presentación de un formalismo más (esta vez muy intuitivo y sencillo, tanto, que es el que hoy se usa para enseñar la teoría de computabilidad) equivalente a la noción de funciones recursivas parciales. La segunda, la que a nosotros nos interesa aquí, como un análisis cuidadoso de los límites de la noción de "procedimiento efectivo". Gödel expresa muy bien esto último:

"El trabajo de Turing da un análisis del concepto de "procedimiento mecánico" (alias "algoritmo" o "procedimiento computacional" o "procedimiento combinatorial finito"). Y *demuestra* que este concepto es equivalente con el de Máquina de Turing".

Destaqué el "demuestra" que usa Gödel. Esto es, Gödel –quien es el arquetipo de la precisión y rigurosidad lógica– considera que el análisis de Turing (particularmente la sección 9 de su artículo) *demuestra* que los procedimientos mecánicos son equivalentes a la noción de función recursiva. En otras palabras, que la noción de función recursiva captura la noción de procedimiento efectivo.

Muy poco después de conocer el resultado de Turing, Gödel reformula –generaliza– de la siguiente manera su resultado de incompletitud de 1931:

"No es posible mecanizar el razonamiento matemático, esto es, nunca será posible reemplazar los matemáticos por una máquina, aún si nos confinamos a los problemas de la teoría de números. Hay por supuesto, porciones de las matemáticas que pueden ser completamente mecanizadas y automatizadas; por ejemplo, la geometría elemental es una de ellas, pero la teoría de los números enteros ya no lo es".



¿Por qué no se podía afirmar esto antes del resultado de Turing de 1936? El mismo Gödel lo explica:

> "Cuando publiqué mi artículo sobre proposiciones indecidibles el resultado no podía ser expresado en esta generalidad, debido a que en ese tiempo no se habían dado definiciones matemáticamente satisfactorias de las nociones de procedimiento mecánico y sistema formal. Esta brecha ha sido llenada por Herbrand, Church y Turing. El punto esencial es definir qué es un procedimiento. Entonces la noción de sistema formal se sigue fácilmente puesto que una teoría formal está dada por tres cosas:
>
> 1. Un número finito de símbolos primitivos cuyas combinaciones finitas son llamadas expresiones;
> 2. Un conjunto finito de expresiones (llamadas axiomas);
> 3. Ciertas, así llamadas, "reglas de inferencia";
>
> Ahora, una regla de inferencia no es nada más que un procedimiento mecánico que permite a uno determinar si a partir de conjunto finito de expresiones es posible inferir algo por medio de la regla de inferencia usada, y así escribir la conclusión".

Pero la historia no termina todavía...

¿MÁS ALLÁ DE TURING?

Todos los esfuerzos posteriores por generalizar y ampliar la definición de función recursiva han confirmado la intuición de Church y los argumentos de Turing y Gödel que se había capturado –en la expresión de Gödel– la *noción epistemológica* de procedimiento efectivo. Aparte de los sistemas ecuacionales de Gödel-Herbrand, el cálculo lambda de Church, la *mu*-recursividad de Kleene, que hemos visto, se han propuesto muchos otros. Entre los más conocidos están los sistemas de Post (sistemas combinatoriales de símbolos); los sistemas de Markov (una suerte de gramáticas formales que capturan las funciones recursivas); los diagramas de flujo; la familia de máquinas con contadores *(counting machines)* que esencialmente agregan un registro para almacenar números enteros; las máquinas de acceso aleatorio (RAM) que pueden considerarse máquinas con contadores con la posibilidad de accesar registros de manera indirecta con instrucciones almacenadas.

En particular, la noción de máquina de Turing ha sido analizada desde múltiples perspectivas, intentando encontrar extensiones que pudieran capturar formas hoy desconocidas de "efectividad". Ninguna ha tenido éxito. Destaquemos aquí algunos de ellos. El más sistemático es probablemente el intento de Kolmogorov y su escuela, lo que se conoce como las máquinas de Kolmogorov-Uspensky. Las ideas esenciales que están detrás del análisis de Turing son (ver análisis en [15]):

> *1. Principio de localidad.* La única información relevante para decidir qué hacer en el siguiente paso de un procedimiento es la información local que se tiene en ese momento (en las palabras de Turing, la región bajo "escaneo" de la máquina y el "estado" mental).
>
> *2. Principio de finitud.* En cada momento, el estado (mente) es sólo capaz de almacenar y procesar un número finito de ítems (unidades atómicas de información).

Kolmogorov observa que hay otro supuesto en el análisis de Turing y es que la topología del espacio de información permanece invariable. Por lo tanto analiza la situación donde el espacio de trabajo es dinámico, esto es, la topología misma de la cinta cambia a medida que la máquina trabaja. Como toda investigación relevante, obtiene una serie de profundos resultados adicionales, pero no logra ir más allá de la máquina de Turing. Esto es, demuestra que este nuevo formalismo extendido de máquinas con espacio de trabajo dinámico tiene un poder expresivo igual a las máquinas de Turing [13].

El alumno y amigo de Turing, Robin Gandy, es quien probablemente ha llevado el análisis de la máquina de Turing más lejos. Propone la noción de *aparato mecánico discreto*, y axiomatiza la noción misma para luego estudiar (esta vez sí que formalmente, en contraposición a la tesis de Church) los límites y alcances de estos sistemas. El lógico alemán W. Sieg simplificó y mejoró esa idea haciéndola más entendible [12].

La historia no termina aquí. Más bien sólo está comenzando. En 1972, Gödel publicó una observación que denominó "Un error filosófico en el trabajo de Turing", donde expresa:

> "Turing en su artículo de 1937 [el texto largo reproducido aquí al comienzo de la sección "*Finale:* Turing 1936"] presenta un argumento que se supone que muestra que los procedimientos/procesos mentales no pueden ir más allá de los procedimientos/procesos mecánicos. Sin embargo, este argumento es inconclusivo. Lo que Turing deja completamente de lado es el hecho que la mente, en su uso, no es estática, sino está desarrollándose constantemente, esto es, que nosotros entendemos más y más los términos abstractos a medida que los vamos usando, y que más y más términos abstractos entran en la esfera de nuestro entendimiento. Pudieran existir métodos sistemáticos que actualizan

este desarrollo, los que pudieran formar parte de este procedimiento/proceso. Por lo tanto, aunque en cada etapa el número y precisión de los términos abstractos a nuestra disposición es finito, ambos (y, por consiguiente, también el número de estados indistinguibles de la mente que habla Turing) puede converger hacia el infinito en el curso de la aplicación de este procedimiento".

La profundidad de la observación de Gödel no se puede pasar por alto (aunque sea algo injusta con lo que Turing realmente quiere expresar). Lo que está indicando es la distorsión, la complejidad, los desafíos que introduce el tiempo, la dinámica al modelo clásico de Turing. De aquí surgen preguntas relevantes como las siguientes:

I. ¿Modela la máquina de Turing al individuo? Sí en un momento dado, fijo. No en su desarrollo existencial, en su desarrollo como persona. Gödel indica que falta un parámetro que es la dinámica de lo que en el modelo de Turing está fijo: el lenguaje, los símbolos, el conjunto de estados, etc.

II. ¿Modela la máquina de Turing el "espíritu" de la humanidad? No, contesta Gödel. Los argumentos son similares.

El debate que abrieron Gödel y Turing recién comienza. BITS

*Agradezco los comentarios de T. Carrasco, P. Alvarado, A. Martínez; el apoyo de P. Barceló, y al Proyecto Fondecyt 1110287.

Referencias

[1] R. Adams, An Early History of Recursive Functions and Computability. From Gödel to Turing. Docent Press, 2011.

[2] F. Cardone, J. R. Hindley, History of Lambda-calculus and Combinatory Logic. Swansea University Mathematics Department Research Report No. MRRS-05-06, 2006.

[3] M. Davis (Edit.) The Undecidable, Basic Papers on Undecidability Propositions, Unsolvable Problems and Computable Functions. Dover, 2004.

[4] J. W. Dawson Jr., Logical Dilemmas: The Life and Work of Kurt Gödel, A. K. Peters Ltd., 1997.

[5] K. Gödel, Collected Works, 3 volúmenes, 1986-1995. Oxford Univ. Press, 1995.

[6] K. Gödel, Obras Completas. Edición y traducción de J. Mosterín. Alianza Edit. 2006.

[7] C. Gutiérrez. "El Teorema de Gödel para no iniciados". Revista Cubo, Univ. La Frontera, Vol. 1, 1999 (dcc.uchile.cl/cgutierr/otros/godel.pdf).

[8] C. Gutiérrez, F. Gutiérrez, "Carlos Grandjot: tres décadas de matemáticas en Chile", Boletín Asoc. Mat. Venezolana, Vol. XI, No. 1 (2004), pp. 55 - 84.

[9] D. Hilbert,1926, "Über das Unendliche", Mathematische Annalen, 95: 161-90, 1926.

[10] A. Hodges, Alan Turing; the enigma. Vintage 1992.

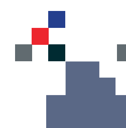
[11] P. Odifreddi, Classical Recursion Theory. The Theory of Functions and Sets of Natural Numbers. Elsevier, 1989.

[12] W. Sieg, "Church without dogma: axioms for computability". New Computational Paradigms (B. Lowe, A. Sorbi, B. Cooper, editors); Springer Verlag 2008, 139-152.

[13] V. A. Uspensky, A. L. Semenov. Algorithms: Main Ideas and Applications. Kluwer Academic Publishers, 1993.

[14] H. Wang, Reflections on Kurt Gödel. MIT Press, 1987.

[15] H. Wang, Popular Lectures in Mathematical Logic. Dover Publ., 1993.



Teoría de la Información

Gonzalo Navarro
Profesor Titular Departamento de Ciencias de la Computación, Universidad de Chile. Doctor en Ciencias Mención Computación, Magíster en Ciencias mención Computación, Universidad de Chile; Licenciatura en Informática, Universidad Nacional de La Plata y ESLAI, Argentina. Líneas de investigación: Diseño y Análisis de Algoritmos, Bases de Datos Textuales, Búsqueda por Similitud, Compresión.
gnavarro@dcc.uchile.cl

La Teoría de la Información puede definirse como el estudio de la cantidad de información que contiene un mensaje, o que puede codificarse en un mensaje. De la mera definición se hace evidente su relevancia en áreas tan amplias de la Computación como el almacenamiento eficiente de la información, la transmisión eficiente de mensajes, la compresión, e incluso la resistencia a errores en el medio de transmisión o almacenamiento.

Nuestra experiencia en el mundo moderno sería bastante distinta si la compresión no existiera. Sería impensable almacenar cientos de fotografías, vídeos y música en nuestro computador. Una película de una hora a un modesto 800x600 a 20 cuadros por segundo, que es bastante baja calidad, necesitaría casi 100 GB de almacenamiento. Con una conexión de 20 Mbits por segundo sólo podría ver vídeo a 2 cuadros por segundo. Una memoria de 1 GB en su cámara de 5 megapíxeles se llenaría con menos de 70 fotos. La transmisión de TV y vídeo en línea, analógica o digitalmente, sería probablemente un sueño imposible. Incluso en el caso menos dramático de la información textual, toda su navegación en Internet sería unas cuatro veces más lenta, y sus discos y USBs rendirían a un cuarto de lo que pueden rendir.

Pero, ¿cómo medir cuánta información contiene un mensaje? Ni siquiera es algo sencillo de definir. Consideremos secuencias binarias para no enredarnos con otros problemas. Tomemos como ejemplo la siguiente secuencia:
00000000000000000000000000000000...

Esta secuencia contiene muy poca información. Una descripción corta es "puros ceros". La siguiente es igualmente simple:
01010101010101010101010101010101...

y la siguiente algo más compleja, pero igualmente fácil de describir:
001011011101111011111011111110...

¿Se le ocurre cómo describirla? Podríamos explicar cómo generarla: poner un 0 y ningún 1, luego un 0 y un 1, luego un 0 y dos 1s, luego un 0 y tres 1s, etc. ¿Y la siguiente?
10110111111000010101000101100001...

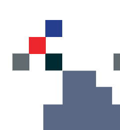
Ésta parece completamente aleatoria. Pero realmente no lo es. Son los primeros bits de la expansión fraccionaria del número e = 2.718..., la base de los logaritmos naturales. Basta esta descripción para reproducir la secuencia, no importa cuán larga sea.

Pero no todas las descripciones son igualmente útiles. Para dar un ejemplo un poco esotérico, imagine que listáramos todas las sentencias en lógica de predicados (la que se usa para describir teoremas matemáticos), ordenadas por largo creciente, y alfabéticamente dentro de cada largo. Entonces podríamos definir una secuencia donde el i-ésimo dígito es 1 si la afirmación es un teorema y 0 si no lo es. Como existen conjeturas matemáticas que aún no se sabe si son verdaderas o falsas, este tipo de descripción, si bien es clara y formal, no nos permite saber cuál es la secuencia (¡más bien, si tuviéramos la secuencia, tendríamos resueltos todos los problemas matemáticos!).

¿Cómo podemos definir un concepto de *descripción útil*? ¡Aquí entra la computación en juego! Andrey Kolmogorov (1903-1987) fue un matemático ruso que consideró este problema y propuso una medida de "complejidad" de las secuencias que se considera hoy en día la definición más clara, correcta y universal en término de descripciones útiles. *La Complejidad de Kolmogorov de una secuencia es la cantidad de bits que se necesita para representar un programa de computador que genere la secuencia*[1].

Esta definición restringe las descripciones a aspectos *computables*, es decir, que la secuencia se pueda producir mecánicamente a partir de la descripción. Esto

[1] Si bien la definición parece depender del lenguaje en que escribimos el programa y cómo lo codificamos en bits, este hecho se hace irrelevante cuando consideramos secuencias suficientemente largas, pues el programa puede incluir un decodificador y un intérprete de otro lenguaje, y éstos son de largo fijo.



incluye nuestros primeros ejemplos pero excluye el último, donde la descripción no permitía obtener la secuencia. De hecho, todos nuestros primeros ejemplos tienen una complejidad de Kolmogorov muy baja: se pueden hacer programas muy cortos para generar secuencias muy largas. Más aún, considere un texto de largo n que se pueda comprimir a m bits usando un compresor cualquiera. Entonces su complejidad de Kolmogorov no es mayor a $c+m$, donde c es el largo del programa descompresor. Es decir, esta complejidad abarca cualquier técnica de compresión conocida o por conocer.

Por otro lado, si no existe ningún programa de largo menor a n que genere una cierta secuencia de largo n, entonces su complejidad de Kolmogorov será a lo menos n y diremos que la secuencia es completamente aleatoria. ¿Existen estas secuencias completamente aleatorias? Sí, siempre existe al menos una, pues existen 2^n secuencias binarias de largo n, pero el conjunto de todos los programas posibles de largo menor a n (codificados en binario) suma 2^n-1 . En realidad, por cada programa de largo m que emite una secuencia de largo n, hay otros 2^{n-m} programas invalidados, por lo que en realidad la gran mayoría de las secuencias son *completamente aleatorias*. ¿Y cuán larga puede ser la complejidad de Kolmogorov de una secuencia? No tan mala: basta un programa ligeramente más largo que n, que diga "Imprimir 01001100111...."

Seguramente ha escuchado, para explicar la Teoría de la Evolución, que si sentara a un mono frente a un teclado por suficiente tiempo, éste terminaría escribiendo las obras completas de Shakespeare. Digamos que éstas tienen un largo n en bits. Entonces el mono requeriría en promedio 2^n intentos para dar con el texto correcto. Como existen compresores capaces de comprimir estas obras a un cuarto de su tamaño original, la complejidad de Kolmogorov de las obras de Shakespeare es a lo más $n/4$. Por ello, si sentáramos al mono frente a un computador a escribir programas, daría mucho antes que el que usa la máquina de escribir, en unos $2^{n/4}$ intentos, con un *programa* que generara estas obras como output. Es decir, un computador funciona como un dispositivo que aumenta las probabilidades de producir secuencias interesantes. Volveré pronto sobre este importante punto.

La complejidad de Kolmogorov ha sido sumamente útil en muchos aspectos, por ejemplo para calcular cuánto se tienen que parecer dos secuencias genéticas para determinar si existe entre ellas alguna relación o no. Y debería conducirnos al *compresor universal* óptimo, que nos entregara la descripción más corta posible para cualquier secuencia. Sin embargo, esta medida tiene una limitante muy grave: no es *computable*. Es decir, dada una secuencia, ningún programa de computador puede determinar cuál es su complejidad de Kolmogorov[2]. Lamentablemente necesitamos utilizar un modelo menos general para poder medir la cantidad de información y diseñar un compresor acorde a ella.

Claude Shannon (1916-2001) fue un matemático estadounidense, considerado el padre de la Teoría de la Información. Shannon atacó el problema desde otro punto de vista: consideremos una fuente infinita de símbolos sobre un alfabeto de tamaño s. ¿Cuántos bits necesitamos para codificar cada símbolo? Bien, si la fuente emite los símbolos en forma equiprobable, necesitaremos lg(s) bits, donde lg denota el logaritmo en base 2. Es decir, si la fuente tiene cuatro símbolos, digamos A, B, C y D, necesitaremos dos bits para codificar cada símbolo (por ejemplo podríamos codificar A como 00, B como 01, C como 10, y D como 11).

Pero ¿qué ocurre si los símbolos no son equiprobables? Digamos que la mitad de las veces se emite A, la cuarta parte de las veces se emite B, un octavo de las veces se emite C y otro octavo se emite D. Entonces podríamos usar el siguiente código: 0 denota A, 10 denota B, 110 denota C y 111 denota D. Como ninguno de los cuatro códigos es prefijo de otro, podemos concatenar los códigos en una secuencia y no tendremos problemas para decodificarlos. Haciendo las cuentas, resulta que un mensaje de largo n con estas frecuencias requerirá $1.75n$ bits, en vez de los $2n$ que necesitaríamos si los codificáramos con 2 bits por símbolo.

Shannon demostró que lo óptimo es utilizar $\lg(\frac{1}{p})$ bits para codificar un símbolo que aparece con frecuencia relativa p, lo que da lugar a su definición de *entropía de una fuente*. La entropía de una fuente que emite símbolos con probabilidades $p_1, p_2, \ldots, p_s$ es $H = \sum_{i=1}^{s} p_i \lg(\frac{1}{p_i})$.

Dicho de otro modo, Shannon demostró que, si una fuente infinita genera símbolos con esas frecuencias (y no tiene otras regularidades aparte de ésa), entonces es imposible codificar un mensaje de esa fuente utilizando menos de H bits por símbolo. Con el tiempo aparecieron codificadores (como la codificación aritmética) que se acercan mucho a ese mínimo teóri-

[2] Tal vez se le ocurra que podría considerar todos los programas posibles, de largos crecientes, hasta dar con el primero que produzca la secuencia que le interesa. Desengáñese: un programa puede ejecutar durante un tiempo arbitrariamente largo antes de emitir su output, y tampoco es computable saber si alguna vez emitirá algo.

co: nH+2 bits para un mensaje de largo n con entropía H.

La medida de entropía de Shannon no es tan universal como la de Kolmogorov. Sólo se aplica cuando la frecuencia es la única regularidad que tiene una fuente. Por ejemplo, la secuencia de la expansión del número e se ve como completamente aleatoria en este sentido, si bien su complejidad de Kolmogorov es muy baja. En cambio, la complejidad de Kolmogorov de una secuencia que tiene entropía de Shannon H es muy cercana a Hn: como ya mencionamos, basta comprimir la secuencia con un codificador aritmético, cuyo programa mida c bits, y el programa sería el descompresor más la secuencia comprimida, que ocupa a lo más $c+Hn+2$ bits. A medida que n crece, el extra de c+2 bits se hace menos relevante.

Es posible extender la entropía de Shannon a casos más complejos, como aquellos en los que la fuente tiene una cierta cantidad de memoria finita. Es decir, la probabilidad de los símbolos no es fija sino que depende de los últimos símbolos emitidos. Esto modela muy bien los lenguajes humanos, donde por ejemplo la probabilidad de ver una consonante en español es muy baja si las dos letras previas ya fueron consonantes. Los compresores que se basan en modelar los textos como si vinieran de este tipo de fuente se llaman compresores estadísticos. Un ejemplo son los compresores tipo PPM. Instale, por ejemplo, ppmdi en su computador, y verá tasas de compresión de 4-5 veces en sus textos.

Otro tipo de medida de entropía, menos conocida por ser menos elegante, pero muy utilizada en los compresores, fue definida implícitamente por Abraham Lempel y Jacob Ziv en 1976. Se basa en el hecho de que un texto donde muchos pasajes son copias de pasajes previos es fácilmente compresible. Lempel y Ziv dan un algoritmo que va procesando el texto, y en cada paso lee la mayor secuencia posible que ya se haya visto antes. El número de "frases" así formadas es una medida de entropía. Mientras que se puede probar que, en las secuencias que consideró Shannon, la entropía de Lempel-Ziv converge (si bien algo lentamente) a la de Shannon, existen casos en que la entropía de Lempel-Ziv es mucho más baja, por ejemplo cuando la secuencia es muy repetitiva (imagínese el conjunto de versiones de un artículo, por ejemplo, o de un sistema de software). Los compresores tipo "zip", ampliamente disponibles en Linux y Windows, se basan en variantes de este tipo de compresión.

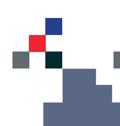
¿Existe un compresor que funcione siempre? ¿Todas las secuencias son compresibles? Tal como ocurría con la complejidad de Kolmogorov, y con más razón en estos modelos más restrictivos de compresibilidad, la respuesta es un rotundo no. De hecho, la gran mayoría de las secuencias son incompresibles. Por cada secuencia compresible que hay, muchas otras no lo son.

¿Por qué entonces en la vida práctica los compresores parecen funcionar siempre? Porque los aplicamos siempre sobre cierto tipo de secuencias. Más aún, a falta de un compresor universal de Kolmogorov, utilizamos compresores adecuados al tipo de compresibilidad que esperamos ver. Los compresores estadísticos y tipo Lempel-Ziv son adecuados a secuencias tipeadas por humanos. Las imágenes, audio y vídeo necesitan otro tipo de compresor[3]. Los textos procesables automáticamente, como XML o programas, pueden comprimirse utilizando gramáticas de lenguajes formales. Las secuencias biológicas necesitan compresores especializados para detectar copias de bloques complementadas e invertidas, y así.

Sin embargo, todo esto es posible solamente porque los *textos, audios, imágenes y vídeos que nos interesan tienen una complejidad de Kolmogorov muy baja*. Si los humanos encontráramos que el ruido blanco de la radio es apasionante, las estaciones de radio tendrían serios problemas para distribuir esa "música" en forma comprimida. Es decir, *lo que es interesante para nuestro cerebro resulta ser "interesante" algorítmicamente, pues es compresible*. Esto es una indicación muy fuerte de que el procesamiento de nuestro cerebro es esencialmente algorítmico, en el sentido de lo que entendemos por algoritmo. O dicho de otra manera, nuestra noción de algoritmo parece capturar correctamente lo que hace nuestro cerebro.

Si tomamos otra fuente de secuencias, como el ADN, por ejemplo, encontramos que la situación es muy distinta. El ADN bacteriano, en particular, es muy difícil de comprimir. Y por una buena razón: en una bacteria no sobra espacio para moléculas muy largas, de modo que la funcionalidad de la bacteria debe ser expresada en una secuencia lo más corta posible. La evolución ha hecho el resto, favoreciendo a las bacterias capaces de codificar

[3] Estos compresores en general admiten una cierta diferencia entre lo que comprimieron y lo que reproducen, que se busca que sea mínima para el receptor. Esto introduce otra dimensión que no abordo en este artículo, que es el compromiso entre tasa de compresión y distorsión permitida al recuperar el mensaje. La entropía de Shannon también regula hasta cuánta compresión se puede obtener por el precio de cuánta distorsión. En mensajes textuales normalmente no se admite ninguna distorsión.



comportamiento más sofisticado en el mismo espacio. El resultado es un "programa" muy corto que hace mucho; algo cercano al ideal de Kolmogorov.

ESTRUCTURAS DE DATOS COMPRIMIDAS

Antes de terminar este artículo quisiera hablar sobre una rama reciente derivada de la compresión, que se llama *estructuras de datos comprimidas.* Hablamos de la utilidad de la compresión para ahorrar espacio y tiempo de comunicación. Esta nueva área, en cambio, se relaciona con el hecho de que las memorias más rápidas son mucho más pequeñas que las memorias más lentas. Las memorias cachés son decenas de veces más rápidas que la RAM, y ésta es miles a millones de veces más rápida que el disco. Un programa que pueda funcionar en una memoria menor gozará de accesos a memoria mucho más rápidos, y esto influenciará su tiempo de ejecución. Y si, en cambio, estamos dispuestos a costearnos un sistema distribuido con tantas memorias RAM como necesitemos (como hacen los grandes buscadores de Internet), el utilizar menos espacio redundará en comprar menos máquinas, reducir el tiempo de comunicación entre ellas, y ahorrar en la energía que consumen. Hoy en día la energía es una parte muy alta de la factura que pagan los grandes centros de datos.

Para lograr esto necesitamos más que compresión. No basta comprimir los datos si es que para procesarlos vamos a necesitar descomprimirlos primero. Necesitamos *poder operar los datos en forma comprimida.* Y no sólo los datos, sino las estructuras de datos que usamos para organizarlos. Si tenemos números en un árbol binario, necesitamos poder representar el árbol binario en forma comprimida de modo que nos permita buscar, y tal vez insertar y borrar elementos, *sin descomprimirlo.*

Éste es el campo de estudio de las estructuras de datos comprimidas, que combina la Teoría de la Información con los Algoritmos y Estructuras de Datos, y ofrece desafíos apasionantes. Daré algunos ejemplos para ilustrarlo. Considere un árbol general de n nodos. Una implementación normal utilizará, como mínimo, n punteros para representarlo en memoria. Como un puntero tiene que por lo menos distinguir entre los n nodos del árbol, estaremos utilizando $n \lg n$ bits de memoria. Pero ¿es ésta una representación eficiente? La Teoría de la Información nos dice que no. El total de árboles distintos de n nodos es cercano a $\frac{4^n}{n^{3/2}}$.[4]

Esto significa que, incluso considerando a todos los árboles igualmente probables, bastarán $\lg\left(\frac{4^n}{n^{3/2}}\right) \leq 2n$ bits para representar cualquier árbol general. ¡Esto es mucho, mucho menos, que los $n \lg n$ bits que se suelen usar! Para un árbol de un millón de nodos, es ¡diez veces menos!

¿Se atreve a diseñar una representación comprimida de árboles que use $2n$ bits? No es tan difícil en realidad. Recorra el árbol en profundidad, recursivamente. Cada vez que llegue a un nodo por primera vez, abra un paréntesis. Cada vez que lo abandone, cierre un paréntesis. La secuencia resultante es de largo $2n$, y como se compone de paréntesis abiertos y cerrados, se puede representar usando $2n$ bits. Viendo la secuencia, puede fácilmente reconstruir el árbol. ¡Felicitaciones, ha diseñado un excelente compresor especializado en árboles! Pero aún no tiene una estructura de datos comprimida para árboles. Para esto, debería ser capaz de navegar en esta secuencia de paréntesis, por ejemplo encontrando el padre de un nodo (podemos identificar un nodo, por ejemplo, con la posición de su paréntesis que abre), bajando al i-ésimo hijo, calculando el tamaño de un subárbol, la profundidad o la altura de un nodo, el ancestro común más bajo entre dos nodos, y decenas de otras operaciones útiles. Hoy en día se sabe cómo realizar todas estas operaciones en tiempo constante, y utilizando esencialmente los $2n$ bits. Existen implementaciones que usan unos $2.3n$ bits y resuelven todas esas consultas en microsegundos.

Un segundo ejemplo son los grafos Web, es decir subconjuntos de la Web, donde los nodos son páginas y las aristas son links. Estos se utilizan con múltiples propósitos, como calcular relevancia de páginas, encontrar autoridades, detectar fábricas de spam, distinguir comunidades, etc. Para representar un grafo dirigido de n nodos y e aristas en forma clásica, por ejemplo como listas de adyacencia, se necesitan unos $n \lg e + e \lg n$ bits. Esta vez la Teoría de la Información no nos ayuda mucho: la cantidad de grafos distintos es tan grande que el número de bits que se necesita para representarlos no es muy distinto del que se usa con una lista de adyacencia. Pero tal como ocurre con las imágenes, los grafos Web no son grafos cualquiera. Tienen algunas características que los hacen "interesantes" y algorítmicamente compresibles. Existen hoy en día estructuras de datos comprimidas para grafos Web que usan poco más de un bit por arista (es decir, veinte a treinta veces menos que una representa-

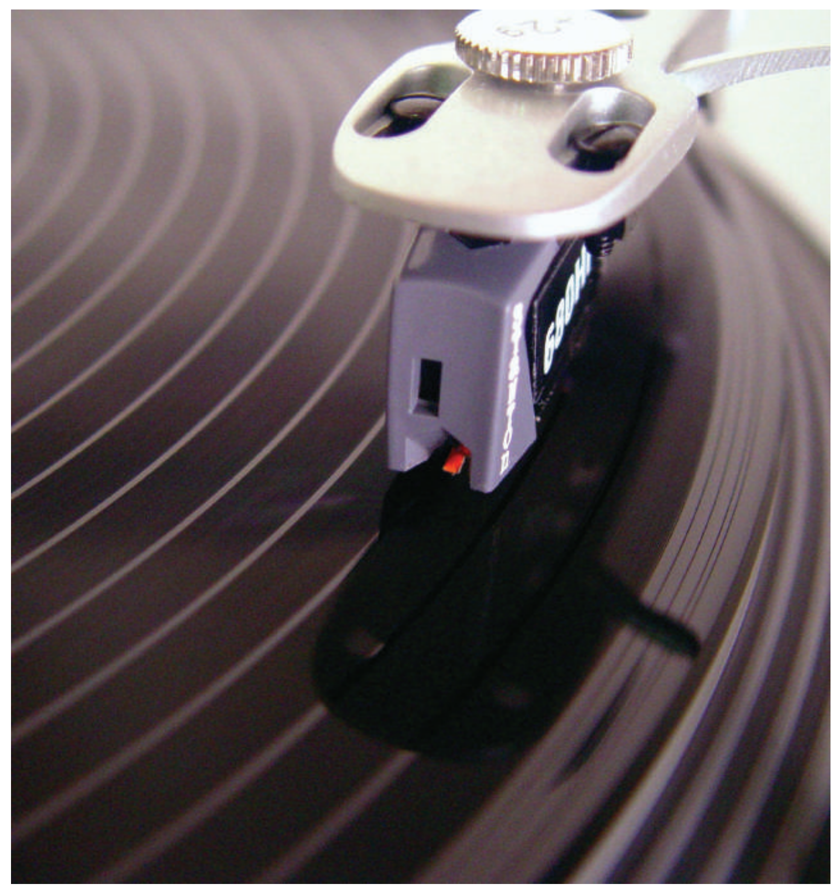
[4] Para los interesados, el valor exacto está dado por los llamados Números de Catalán.

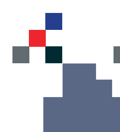
ción clásica, para grafos grandes) y permiten navegarlos eficientemente. Esto permite analizar subgrafos de la Web mucho mayores en memoria principal.

Un último ejemplo es el genoma humano, que contiene unas tres mil millones de bases (A, C, G, T). Si bien éste se puede representar en unos modestos 700 MB, en bioinformática se necesita mucho más que representarlos. Se necesita realizar complejos análisis para detectar repeticiones, similitudes, y muchas otras regularidades. Para ello se usa frecuentemente una estructura de datos conocida como árbol de sufijos. Si bien esta estructura permite responder eficientemente muchas preguntas complejas, una representación clásica requiere unos $20n$ bytes, ¡con lo cual necesitamos una memoria de 60 GB para representar un genoma! Sin embargo, en términos de Teoría de la Información, el árbol de sufijos es completamente redundante, pues no contiene nada que no se pueda derivar del texto mismo, y por ello debería ser posible, en principio, representarlo en tan poco espacio como el texto mismo, unos $2n$ bits. Hoy en día existen implementaciones de árboles de sufijos comprimidos que, si bien no se acercan tanto a este ideal, sí pueden almacenar un genoma en menos de 2 GB y simular todas las operaciones del árbol de sufijos en microsegundos.

Pero esto es un solo genoma. Hoy en día las grandes compañías están secuenciando miles de genomas por día. Pronto tendremos bases de datos bioinformáticas con miles y hasta millones de genomas. Esto, multiplicado por tres mil millones de bases, representa un desafío más allá de todas nuestras capacidades. Incluso pensando en 700 MB por genoma nos lleva a números imposibles. ¿Cómo haremos frente a estos desafíos? La Teoría de la Información nos da una pista. Los genomas de una misma especie se parecen mucho entre sí, en más de 99.9% en muchos casos. Es decir, una colección de genomas humanos se puede ver como una colección altamente repetitiva, donde las técnicas de Lempel y Ziv deberían tener mucho que ofrecernos. Y es así: estos compresores pueden obtener tasas de unas pocas centésimas de bit por símbolo cuando los aplicamos a este tipo de colecciones. Pero otra cosa son las estructuras de datos comprimidas. El estudio de estructuras como árboles de sufijos para colecciones altamente repetitivas está en sus comienzos, pero creo que es la clave para los desafíos que nos esperan en este campo.

Y no sólo la bioinformática promete inundarnos de datos en el siglo que comienza. Los datos astronómicos, los meteorológicos y ambientales, los de comportamiento social (clicks, tweets, compras online), los científicos, los geográficos, los multimediales, los de realidad aumentada, etc., se han convertido en esenciales para la ciencia, la tecnología, y los aspectos más básicos de la vida diaria, y no han hecho más que empezar. Se incrementarán rápidamente con la computación ubicua y con las interfaces directas con los órganos sensoriales o el cerebro. Esta avalancha de datos representará muy en breve desafíos gigantescos de almacenamiento y procesamiento. Estoy convencido de que la Teoría de la Información y la Algorítmica serán herramientas esenciales para enfrentar esos desafíos y ser capaces de dar el salto a una sociedad mucho más compleja que la de hoy. BITS





Planificación, preferencias y conocimiento: motivación y problemas abiertos

Jorge Baier

Profesor Asistente Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile. Doctor of Philosophy, University of Toronto; Ingeniero Civil Industrial y Magíster en Ciencias de la Ingeniería, Pontificia Universidad Católica de Chile; Áreas de interés: Planificación Automática, Búsqueda en Inteligencia Artificial, Representación y Razonamiento en Mundos Dinámicos.

jabaier@ing.puc.cl

La planificación —que es como suelo traducir la palabra en inglés *planning*— es una actividad que los seres humanos realizamos todos los días cuando pensamos qué se debe hacer para lograr cierto objetivo. Un ejemplo es la planificación de un viaje: si he decidido aceptar hoy dar una charla mañana a las 15:00 hrs. en Viña del Mar, ¿qué debo hacer? La respuesta a esta pregunta es probablemente un conjunto de acciones, algunas de las cuales tienen que ver con la preparación de la charla, otras tienen que ver con cómo llegar a Viña y otras con comunicar mi plan a otro agente (mi señora, por ejemplo).

Más específicamente, el problema implica componer acciones, las que consideramos entes productoras de un cambio en el mundo, de tal forma que al ejecutar esas acciones, se produzca un cambio deseado en éste. El problema es central en el desarrollo de la inteligencia artificial, dado que se considera a la planificación como una actividad que todo agente autónomo deberá en su momento realizar. Es de interés principal en esta área la planificación *independiente del dominio*, que significa que el programa que planifica debe ser capaz de procesar un problema en un dominio que nunca antes ha visto, ojalá, de forma tan hábil como lo hacemos las personas.

En su versión más sencilla —llamada planificación clásica *(classical planning)*— el problema se modela como una tupla *(I, A, G)*, donde *I* representa un estado inicial, *A* es un conjunto de acciones, y *G* es una condición que se debe cumplir sobre los estados objetivo, es decir, aquellos estados del mundo a los que nos interesa llegar. Cada acción *a* en *A* se modela como una tupla *(prec(a), eff(a))*, donde *prec(a)* representa la precondición de la acción, es decir, la condición que se debe cumplir antes de ejecutar la acción. Por otro lado *eff(a)* representa los efectos de la acción, es decir, aquellas condiciones que deben satisfacerse en el estado que resulta de ejecutar *a*. Dentro de *eff(a)* se distinguen los efectos positivos y los efectos negativos. Los efectos positivos son aquellos hechos que se harán verdaderos una vez que ejecutamos *a* (por ejemplo, al viajar de Santiago a Viña se hace verdadero que estoy en Viña). Los efectos negativos hablan de las condiciones que se hacen falsas una vez ejecutada la acción (en el ejemplo anterior, al viajar de Santiago a Viña se hace falso que estoy en Santiago).

En planificación clásica se supone que al ejecutar una acción en un estado del mundo *s*, el agente llega a otro estado del mundo *s*'. Más aún, se supone que este estado es único (el mundo es determinístico), y que se conoce exactamente cómo computar *s'* desde *s*. La condición *G* se puede pensar como definiendo un conjunto de estados del mundo a los que nos interesa llegar después de ejecutar una secuencia de acciones tomadas desde *A*.

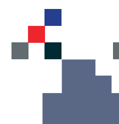
Así como lo definimos, el problema de planificación se puede entender como una búsqueda de un camino en un grafo. En efecto, lo que se quiere resolver es un problema de alcanzabilidad en el grafo determinado por el estado inicial, y por los estados generados por la aplicación sucesiva de acciones sobre él. Visto desde el punto de vista de la teoría de grafos, este problema se puede resolver usando el algoritmo de Dijkstra, publicado en 1959.

¿Qué hace entonces que este problema aún sea objeto de estudio? La respuesta a esta pregunta es larga y daré algunas ideas. Mientras la respondo, aprovecharé de describir temas que desarrollo en mi investigación. También espero dejar una lista de problemas que aún se consideran abiertos y que, creo, seguirán sin tener una respuesta satisfactoria por algún tiempo.

HEURÍSTICAS: UNA MANERA DE ATACAR ESTE DIFÍCIL PROBLEMA

El gran problema de usar un algoritmo como Dijkstra para planificación, es que simplemente no sirve desde el punto de vista práctico. El espacio de estados del grafo donde buscamos es demasiado grande en general. Desde el punto de vista teórico, alcanzabilidad es un problema NLOGSPACE-complete. Esa complejidad, sin embargo, está expresada en términos del tamaño del grafo, no del tamaño de nuestra tupla *(I, A, G)*. Lamentablemente, el grafo definido por tal tupla puede ser exponencial en el tamaño de ella: la complejidad de planificación clásica (de decidir si un plan existe) es realmente PSPACE-complete, lo que significa que no conocemos algoritmos eficientes para resolver estos problemas. Los mejores algoritmos que tenemos ejecutan, de hecho, en tiempo exponencial en el tamaño del problema.

Para verlo en términos un poco más prácticos, pensemos en el espacio de búsqueda del conocido y sencillo *mundo de bloques*, que consiste en una mesa llena de bloques, donde el agente puede tomar uno sólo a la vez y dejarlo sobre otro bloque o sobre la mesa. El objetivo es dejar los bloques en una configuración dada. Este sencillo problema, hasta hace unos diez años no era resuelto por ningún planificador automático de forma razonablemente rápida. Incluso hoy los mejores planificadores tienen problemas



para resolver ciertas configuraciones. ¿Cuál es el tamaño del espacio de estados de este problema? Si tenemos *n* bloques, está dado por:

$$\sum_{k=1}^{n} \frac{n!}{k!} \binom{n-1}{k-1}$$

con lo que concluimos que con 20 bloques, tenemos un espacio de alrededor de 3×10^{20} estados (agradezco a mi alumno Nicolás Rivera por deducir la fórmula). Por otro lado, el famoso puzle del cubo Rubik tiene alrededor de 4×10^{19} estados. ¿Diríamos que resolver problemas del mundo de bloques con 20 o más de estos, es más difícil que resolver un cubo Rubik?

Los seres humanos somos capaces de resolver muchos problemas con espacios de estados gigantes sin, aparentemente, requerir de un esfuerzo de cómputo demasiado grande. Una posible explicación es que tenemos la habilidad de distinguir sin mucho esfuerzo cuándo un estado *s* está más o menos cerca del objetivo, al mirar unas pocas características de *s*.

Pero, ¿cómo es que logramos que un computador, se dé cuenta que un problema es fácil, como lo hacemos nosotros? Con búsqueda ciega (por ejemplo, Dijkstra), resolver 20 bloques es *mucho* más difícil que el cubo Rubik. En planificación usamos algoritmos de búsqueda informados, que usan una función —llamada función heurística— para guiar la búsqueda. Más precisamente, la heurística es una función que estima la dificultad de llegar al objetivo desde un estado particular. En el año 2000 recién se desarrollaron los primeros planificadores automáticos que eran capaces de resolver el problema de planificación usando búsqueda heurística, donde la heurística usada es una que se computa *automáticamente*, es decir no es entregada por el usuario. Desde ese momento ha surgido gran interés por desarrollar esta técnica, especialmente porque estos planificadores demostraron ser notablemente mejores que los que existían en el pasado.

Resulta que una de las mejores heurísticas independientes del dominio es tan simple que se puede explicar con una sola oración: para estimar la dificultad de llegar a algún estado de *G* a partir de un estado *s* imaginamos otro problema en donde las acciones sólo tienen efectos positivos y luego resolvemos el problema desde *s* y retornamos el largo de la solución como la estimación. Afortunadamente, el problema sin efectos negativos, que es una relajación del problema original, se puede resolver en tiempo *polinomial* en el tamaño del problema original. Llamaremos a esta forma de relajar el problema la *relajación libre de efectos negativos* (RLEN).

Uno de los temas de mi investigación ha tenido que ver precisamente con esta problemática: sobre cómo construir estas funciones de manera automática. A pesar de que la relajación libre de efectos negativos funciona muy bien en muchos problemas, hay otros en los que conduce al planificador a tomar decisiones tan malas que hacen que un problema trivial para cualquier ser humano no pueda ser resuelto por los mejores planificadores actuales. Encontrar relajaciones mejores que la RLEN ha probado ser un problema muy difícil. Junto a mis colegas hemos propuesto una solución a éste [1]. Las heurísticas que propusimos permiten mejorar la RLEN en muchos casos, pero tienen la desventaja que son más lentas de computar, lo que hace que sean difíciles de aplicar.

HACIA MODELOS MÁS RICOS DE PLANIFICACIÓN

Tal como está descrita, la planificación clásica parece tener aplicaciones limitadas. A simple vista el mundo no parece determinístico (a menos que deseáramos modelar hasta las más mínimas interacciones físicas entre partículas). Tampoco es cierto que los seres humanos estemos interesados sólo en el objetivo, sin importar qué ejecutamos, porque todos tenemos preferencias sobre qué cosas queremos hacer, qué estados nos gustan o cuáles queremos evitar, etc. En mi ejemplo del viaje a Viña, junto con preferir minimizar el tiempo de viaje, podría preferir aprovechar el tiempo de viaje leyendo; esto hace difícil tomar una decisión sobre el tipo de vehículo a utilizar (bus versus mi auto).

Entonces, si tenemos un conjunto de preferencias, ¿cómo es posible construir planificadores automáticos que busquen soluciones preferidas en forma eficiente? Si el problema clásico es difícil, éste parece serlo aún más. En 2006 [2] propusimos un planificador automático que utilizaba heurísticas desarrolladas para planificación clásica para este problema. El planificador tuvo un desempeño aceptable en una competencia internacional en donde participó. Hasta el día de hoy, sin embargo, este problema es motivación para mi investigación. Por ejemplo, junto a mi alumno León Illanes, hemos descubierto que existen muchos problemas de estos, que los humanos consideramos “fáciles” y que son muy difíciles para los planificadores con preferencias actuales. El problema es que las heurísticas para planificación son aún muy malas en presencia de preferencias.

Otro tema que me motiva son los objetivos más complejos. ¿Qué pasa si mi objetivo no es solamente llegar a un estado, sino que me interesa que la trayectoria para llegar a ese objetivo tenga una cierta propiedad?¿Podemos hacer planificación eficiente para este caso? La respuesta a la que llegamos cuando investigamos esto es satisfactoria: resulta que cuando los objetivos son expresados usando lógica temporal lineal, es posible usar resultados de teoría de autómatas para reducir el problema a uno de planificación clásica [3]. El enfoque tiene un peor caso, muy malo, donde el resultado de la reducción es exponencial en el problema original. Sin embargo, para muchas situaciones prácticas este peor caso no se manifiesta.

Aún quedan problemas abiertos: no sabemos realmente si las heurísticas existentes son las mejores cuando se usan objetivos temporales. Actualmente no hay estudios

publicados que se refieran seriamente a este tema.

APLICACIONES NO ESTÁNDARES DE PLANIFICACIÓN

Planificación es un problema PSPACE-completo, lo que significa que podemos reducir todo problema en PSPACE a un problema de planificación. Es decir, podemos usar planificadores para problemas que no han sido necesariamente concebidos como problemas de planificación.

¿Hay algún problema interesante que se pueda resolver usando un planificador de forma más efectiva que usando otro "solver" específico? La respuesta también parece ser positiva. Junto con mis colegas mostramos que cierto tipo de problemas de diagnóstico dinámico (es decir, el problema de encontrar fallas en un sistema dinámico) también se puede resolver usando planificadores en forma efectiva [4]. Esto motivó incluso que propusiéramos un nuevo paradigma de planificación, para el cual aún no conocemos buenos planificadores [5].

INTEGRACIÓN Y ADQUISICIÓN DE CONOCIMIENTO PROCEDURAL

Imaginemos ahora que quiero implementar un agente (un robot o un programa autónomo) y quiero programarlo, pero no quiero hacer un programa gigante con miles de *if-then-else*, sino que quiero dar flexibilidad al agente. Es decir, estoy dispuesto a llenar al programa con constructos no-deterministas, que el agente, dependiendo de las condiciones "complete" durante la ejecución. En muchos casos, el problema de cómo completar estos hoyos no-determinísticos es algo que se puede reducir a planificación clásica [6] y, como consecuencia, es posible aprovechar el poder de los *solvers* actuales.

Claramente los seres humanos no planificamos desde cero cada vez que tenemos un problema de planificación. Yo ya sé cómo ir a la universidad todos los días y es poco lo que planifico todos los días. Incluso si deseo ir a otro lugar es posible que reutilice un pedazo de mi plan para resolver otro problema. En efecto, pareciera que el plan para llegar a la universidad parece como si fuera una especie de programa general, que yo mismo construí. Actualmente, ésta es toda un área de investigación que ha mostrado avances interesantes, pero aún parece estar lejos del rendimiento obtenido por enfoques más sencillos como los que construyen algún tipo de *porfolio* de planificadores.

MEJORES ALGORITMOS DE BÚSQUEDA

Los mejores planificadores que han surgido en los últimos años se han caracterizado no sólo por usar novedosas heurísticas, sino también por introducir importantes modificaciones a los algoritmos de búsqueda existentes. El conocido A*, un algoritmo de búsqueda que puede usar una heurística *h*, tiene muy mal rendimiento en planificación, usándose mejoras sobre éste, como por ejemplo, priorizar aquellas acciones que aparecen en el plan relajado obtenido de resolver la relajación LEN.

No está claro, sin embargo, si los algoritmos actuales son los mejores para el caso, por ejemplo, de preferencias. Cuando se planifica con preferencias el problema parece mucho más uno de optimización, pero en el que, además, sólo encontrar una solución factible (para qué decir la óptima) puede ser muy difícil. En estos casos, parece ser que la construcción de algoritmos incrementales, que retornen soluciones cada vez mejores es la mejor solución. Sin embargo, no está claro cómo aprovechar el esfuerzo realizado en las búsquedas pasadas para obtener buenas soluciones. El algoritmo que mejor rendimiento tiene hasta el momento es uno que descarta todo lo aprendido en el pasado [7], lo que no parece intuitivamente razonable. Cuando hay múltiples objetivos o preferencias, pareciera más razonable aprender qué pedazos del plan satisfacen ciertas preferencias. Estos pedazos se podrían reutilizar en futuras búsquedas. Este es otro tema que actualmente forma parte de mi foco en esta área. BITS

Referencias

[1] Jorge A. Baier, Adi Botea: Improving Planning Performance Using Low-Conflict Relaxed Plans. ICAPS 2009.

[2] Jorge A. Baier, Fahiem Bacchus, Sheila A. McIlraith: A heuristic search approach to planning with temporally extended preferences. Artif. Intell. 173(5-6): 593-618 (2009).

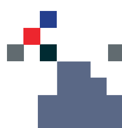
[3] Jorge A. Baier, Sheila A. McIlraith: Planning with First-Order Temporally Extended Goals using Heuristic Search. AAAI 2006: 788-795.

[4] Shirin Sohrabi, Jorge A. Baier, Sheila A. McIlraith: Diagnosis as Planning Revisited. KR 2010.

[5] Sammy Davis-Mendelow, Jorge A. Baier, and Sheila McIlraith: Assumption-Based Planning: Generating Plans and Explanations under Incomplete Knowledge. AAAI 2013. To appear.

[6] Jorge A. Baier, Christian Fritz, Sheila A. McIlraith: Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners. ICAPS 2007: 26-33.

[7] Silvia Richter, Jordan Tyler Thayer, Wheeler Ruml: The Joy of Forgetting: Faster Anytime Search via Restarting. ICAPS 2010: 137-144.



Tipos: garantías a medida

Éric Tanter
Profesor Asociado Departamento de Ciencias de la Computación, Universidad de Chile. Ph.D. Computer Science, Universidad de Nantes, Francia y Universidad de Chile (2004). Líneas de investigación: Desarrollo de Software, Lenguajes de Programación, Modularidad y Adaptabilidad del Software, Sistemas Concurrentes y Distribuidos, Ingeniería de Software.
etanter@dcc.uchile.cl

PROGRAMAS QUE AYUDAN A LOS PROGRAMADORES A PROGRAMAR

Desde que aparecieron los ensambladores, hasta hoy, el desarrollo de software se apoya de manera esencial en herramientas computacionales, cuyo objetivo es permitir al programador abstraerse de detalles de bajo nivel (como la ubicación exacta de los datos en memoria) para concentrarse en el desarrollo de soluciones expresadas en términos de conceptos inteligibles por humanos y fácilmente comunicables. El alto nivel de abstracción provisto por los lenguajes de programación, que incluye la posibilidad de construir nuevas abstracciones dedicadas (procedimientos, funciones, tipos de datos abstractos, clases, librerías, frameworks), es un habilitador crucial en el desarrollo de los sistemas altamente complejos sobre los cuales nuestra sociedad se construye.

Idealmente, uno quisiera tener, al mismo tiempo que está programando, la garantía de que el programa desarrollado es correcto. Correcto, en el sentido de que computa lo que se espera que compute, y que lo haga de la manera que uno espera (por ejemplo, respecto del uso de recursos). Sin embargo, la gran mayoría de las propiedades interesantes que uno quiere asegurar son indecidibles (partiendo con la propiedad aparentemente muy simple "¿este programa termina?"). La Figura 1 ilustra esta problemática y las distintas posibilidades a considerar.

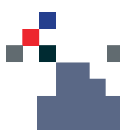
El campo de la verificación de programas es por ende un pozo sin fin, un eterno compromiso entre la expresividad requerida para lo que se quiere verificar, y la amenaza de la indecidibilidad. En la práctica, muchas veces uno quiere la garantía absoluta que todos los programas malos son rechazados. Esto significa que los sistemas de verificación son conservadores: sólo reportan que un programa cumple una cierta propiedad cuando lo pudieron demostrar, pero inevitablemente van a rechazar algunos programas que sí cumplen la propiedad (Figura 1c).

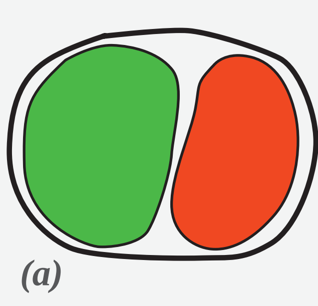
Mientras que las técnicas más avanzadas de verificación de software, como el análisis formal y la verificación de modelos, no han tenido mucha aceptación en la industria de software –excepto en áreas críticas donde la correctitud absoluta es una necesidad vital (automóviles, cohetes, etc.)– existe una forma liviana de métodos formales que cada programador usa día a día, muchas veces sin preguntarse mucho de qué se trata: los sistemas de tipos.

¿SISTEMAS DE TIPOS?

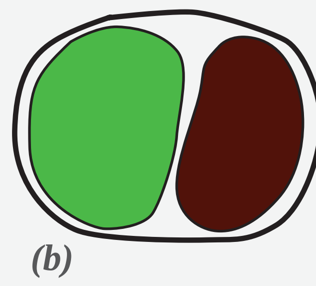
El objetivo de un sistema de tipos es asegurar estáticamente –es decir antes de la ejecución del programa– que ciertos errores nunca ocurren en tiempo de ejecución. ¿Cuáles son estos errores? Típicamente, se trata de errores "simples" como aplicar una operación primitiva a valores inadecuados, como lo es multiplicar dos valores que no son numéricos. Para operar, un sistema de tipos clasifica las expresiones de un programa según el tipo de valores que pueden producir. Luego, verifica que solamente se usan expresiones de tipos adecuados en lugares adecuados. Por ejemplo, si `e1` y `e2` son expresiones, la expresión `e1 + e2` es válida en tipos solamente si `e1` y `e2` son expresiones válidas de tipo numérico; de ser así, la expresión `e1 + e2` misma es válida, y tiene un tipo numérico. Esta clasificación de tipos se puede basar en anotaciones de tipo que el programador escribe explícitamente en el programa (como en C y Java), o pueden ser inferidas por el mismo sistema de tipos (como en ML y Haskell).

Una característica muy importante de los sistemas de tipos, y una razón decisiva de su amplia adopción en la industria, es la componibilidad: para poder verificar un componente solamente se necesita conocer los tipos de los componentes con los cuales interactúa, y no sus implementaciones. Esto diferencia

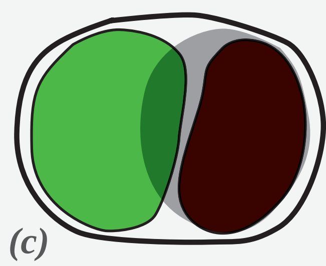




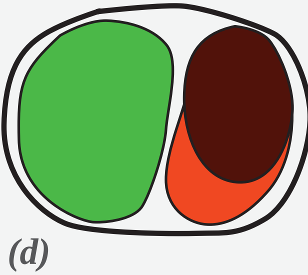
Dado una propiedad, algunos programas cumplen (verde), y otros no (rojo).



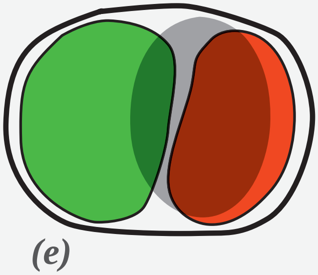
Mundo ideal: un verificador estático rechaza todos los programas rojos (correcto), y sólo aquellos (completo).



Aproximación correcta pero incompleta: el verificador rechaza todos los programas rojos, pero también rechaza algunos programas verdes.



Aproximación completa pero incorrecta: el verificador no rechaza ningún programa verde, pero acepta ciertos programas rojos.



Aproximación incompleta e incorrecta: el verificador no rechaza algunos programas verdes, y acepta ciertos programas rojos.

Figura 1

los sistemas de tipos de los análisis estáticos que requieren poder analizar todo el código fuente de un sistema a la vez. Por ejemplo, considerando una función, basta conocer el tipo de sus argumentos, y los tipos de las funciones que llama, para determinar si tiene un tipo de retorno bien definido y cuál es. Además, si el programador especificó el tipo de retorno esperado, se puede corroborar que la definición de la función cumple esta expectativa. En resumen, los tipos sirven como contratos que rigen las interacciones entre los componentes de un sistema, permitiendo el desarrollo independiente de los componentes y validando su integración.

¡COHERENCIA, POR FAVOR!

Una propiedad fundamental que se espera de un sistema de tipos, es que las predicciones de tipos sean coherentes con la ejecución del programa. Por ejemplo, si el sistema de tipos dice que un programa es válido y de tipo numérico, entonces ejecutarlo debería producir un valor de tipo numérico. Por más sorprendente que parezca, ciertos lenguajes carecen de esta propiedad. Por ejemplo, en C, el sistema de tipos no es coherente con la ejecución de los programas, lo que lleva a un estado extraño en el cual el sistema de tipos no asegura nada. Esto es porque el modelo de ejecución de C, de muy bajo nivel, no garantiza que se respeten las abstracciones establecidas al nivel del código fuente, y

verificadas por el sistema de tipos. La consecuencia de esto son los famosos "errores de segmentación" y otras joyas del "comportamiento no-definido" que todo programador C ha llegado a apreciar en su justo valor. Para volver a la Figura 1, el verificador de tipos de C corresponde al caso 1e: incompleto, e incorrecto.

Volviendo a lenguajes civilizados, un sistema de tipos puede ser visto como un sistema de demostración formal de que ciertos errores no ocurrirán en tiempo de ejecución. El costo a pagar por esta firme garantía es que el programador debe convencer al sistema de tipos que su programa cumple con las exigencias establecidas. Surgen nuevamente los monstruos de la indecidibilidad, y el necesario conservadurismo de la verificación de tipos. La consecuencia práctica es que en algunos casos, un programa válido será rechazado por el sistema de tipos (Figura 1c). Frente a este problema, hay dos puertas para avanzar.

PROMESAS DE TIPOS

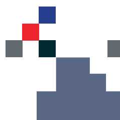
La primera consiste en tener un mecanismo con el cual el programador pueda indicarle al sistema que una expresión es de cierto tipo, aunque no lo pueda demostrar. En buenas cuentas, el programador le dice al verificador de tipos: "Confía en mí, te juro que está bien". Este mecanismo es una coerción, o *cast*, que no se verifica estáticamente. Por ejemplo, si el sistema de tipos rechaza `e1+e2` porque no logra convencerse de que `e2` realmente sea de tipo numérico, el programador puede insertar una coerción explícita: `e1+(<Num>e2)`. Obviamente, para proteger la integridad del programa, dicha coerción se tiene que traducir en una verificación dinámica –es decir en tiempo de ejecución– de que el valor de `e2` realmente sea de tipo numérico. De no ser así, se lanza un error que señala específicamente su causa. Es importante notar que dicho error es infinitamente mejor que un *crash* del programa o que un comportamiento arbitrario indefinido. Los peores errores son los que pasan desapercibidos y los que no dejan ninguna indicación de lo que pasó; eso lo saben muy bien los hackers que explotan los hoyos de seguridad de sistemas programados en C.

MEJORES TIPOS

La segunda puerta es mejorar el sistema de tipos, haciéndolo más expresivo, para así reducir el monto de programas válidos que son rechazados. Un ejemplo clásico de esta vía es la introducción en Java 1.5 de los "generics", es decir, del polimorfismo paramétrico, tal como se encuentra en los lenguajes ML y Haskell. Anteriormente, el sistema de tipos de Java sólo razonaba en base a polimorfismo de subtipos, es decir, el hecho de que un objeto es aceptable en un lugar donde se espera que tenga algún tipo A si entiende un conjunto de mensajes más amplio que el especificado por A. Dicho sistema de tipos era sin embargo, muy limitado al momento de usar colecciones de objetos, dado que no permite especificar el tipo de los elementos contenidos dentro de una colección. Por ejemplo, una lista es de tipo `List`, y no se sabe nada del tipo de sus elementos. Esto obligaba a los programadores a hacer uso de coerciones de manera muy frecuente, lo que a su vez, disminuyó mucho la relevancia misma del sistema de tipos, al dejar la puerta abierta a muchos errores en tiempo de ejecución. La introducción de los *generics* en Java resolvió este problema, permitiendo al programador especificar el tipo de los elementos contenidos en una colección, por ejemplo, declarando que una lista sólo puede contener valores numéricos. Pero esta mejora aumentó considerablemente la complejidad del sistema de tipos, dado que la interacción entre el polimorfismo de subtipos y el polimorfismo paramétrico no es trivial.

TIPOS DINÁMICOS

Frente a esta situación existe, en realidad, una tercera opción: la de optar por un lenguaje sin verificación estática de tipos. Dichos lenguajes, llamados lenguajes dinámicos, como lo son Python, Ruby y JavaScript, son muy atractivos por su simplicidad aparente. Al no tratar de asegurar nada estáticamente, los lenguajes dinámicos ahorran un costo conceptual grande a los programadores, permitiéndoles ejecutar cualquier programa sin permiso previo. Es crucial notar que estos lenguajes son seguros, a diferencia de C, y por ende no permiten comportamiento no-definido. Más bien, corresponden a lenguajes donde todas las verificaciones de tipos se hacen dinámicamente, similar al caso de coerciones presentado anteriormente. Por ejemplo, uno siempre puede ejecutar un programa que hace `e1+e2`. El ambiente de ejecución verificará dinámicamente que `e1` y `e2` producen un valor numérico antes de ejecutar la primitiva de adición. Esta verificación tiene un costo, y además puede traducirse en un error en tiempo de ejecución si es que `e1` o `e2` resulta no producir un valor numérico. Es por eso que hablé de "simplicidad aparente": al no verificar tipos estáticamente, una parte significativa del trabajo de depuración de programas en lenguajes dinámicos tiene que ver con repetir el trabajo que un sistema de tipos hubie-



se hecho automáticamente y exhaustivamente. Tener que escribir tests para todos los posibles casos de errores de tipos es contraproducente. Es, además, limitado, porque un conjunto de tests no es nada más que un conjunto de observaciones: no provee ninguna garantía de que ciertos errores no ocurrirán nunca.

TIPOS GRADUALES

Este último punto es la razón por la cual muchos sistemas parten siendo prototipados en un lenguaje dinámico, pero a medida que el programa va creciendo, la necesidad de tener fuertes garantías estáticas se hace sentir. ¿Qué se puede hacer en estos casos? En un principio, lo único que se puede hacer es portar el sistema a otro lenguaje, que sea tipado estáticamente. Esta conversión no es sin costo ni riesgo. Reconociendo esta situación, últimamente se ha dado énfasis a los sistemas de tipos "graduales", que permiten, dentro del mismo lenguaje, que conviva código tipado con código no tipado. El sistema de tipos hace su mejor esfuerzo para detectar incoherencias estáticamente y, cuando no puede, deja pasar, delegando la responsabilidad al ambiente de ejecución del programa. En buenas cuentas, un sistema de tipos graduales inserta coerciones automáticamente en todos los lugares donde tiene una duda. Esta mezcla parece ser la panacea, combinando las ventajas de ambos mundos en un entorno común.

No debería sorprenderles entonces que todos los últimos lenguajes industriales (Dart de Google, TypeScript y C# de Microsoft, ActionScript de Adobe, entre varios otros) cuentan con una forma de tipos graduales. Las bases teóricas de los tipos graduales son más bien recientes. Permitieron establecer resultados importantes, por ejemplo que no es correcto basarse en la noción de subtipos para introducir un tipo dinámico. También ayudaron a entender bien cuál es la garantía que se obtiene de un programa donde ciertas partes son tipadas y otras no: si bien un error de tipo puede producirse al ejecutar código estáticamente tipado, la causa de dicho error siempre será una entidad no tipada, que el sistema puede identificar. A la fecha, el área sigue siendo objeto de investigación muy activa para entender mejor los distintos compromisos posibles entre garantías fuertes, costos de ejecución, y complejidad para el programador. De hecho, cada uno de los lenguajes actuales con tipos graduales lo hace a su manera, distinta de los demás. A pesar de esta cacofonía ambiente, es de apostar que de aquí en adelante, ningún nuevo lenguaje pragmático podrá ignorar el permitir combinar verificación estática de tipos con verificación dinámica.

¿QUÉ TAN ÚTILES SON ESTOS TIPOS?

Una gran crítica a los sistemas de tipos tradicionales es que solamente verifican propiedades simples y que, por lo tanto, "no valen la pena". Es cierto que es útil saber que nunca se van a sumar peras con manzanas, pero más útil aún es saber que el resultado final es correcto, no solamente algún valor del tipo esperado. En los ojos de un sistema de tipos tradicional, 1 y 1000 son lo mismo (elementos de un tipo numérico), pero claramente al usuario final no le da lo mismo tener 1 o 1000 dólares en su cuenta. De manera similar, un programa que computa el resultado deseado no es muy atractivo si es que para lograrlo, hace un uso indebido de recursos o compromete la integridad del sistema. Mucho trabajo ha sido desarrollado en el área de tipos expresivos –o verificación basada en tipos, o programación certificada– donde los tipos se usan para especificar propiedades mucho más relevantes que la simple pertenencia a un conjunto de valores. En lo que queda de este artículo, quisiera mencionar brevemente tres enfoques para análisis precisos con tipos: sistemas de tipos subestructurales, sistemas de efectos y tipos dependientes.

TIPOS AVANZADOS I: SISTEMAS SUBESTRUCTURALES

En los sistemas de tipos tradicionales, la información de tipos que utiliza el sistema para analizar un pedazo de código goza de varias características llamadas "estructurales". Por ejemplo, si se sabe que la variable `a` es de tipo `Num` y que la variable `b` es de tipo `Bool`, no importa el orden de estas premisas. Tampoco interfiere con ellas el hecho de conocer más premisas, por ejemplo que otra variable `c` también está definida. Además, se puede usar el conocimiento de que `a` es de tipo `Num` tantas veces que se requiera (por ejemplo para poder tipar la expresión `a+a`, se usa el conocimiento relativo a la variable `a` dos veces). Los sistemas de tipos llamados subestructurales son sistemas en los cuales alguna(s) de estas características no se provee; por ejemplo, donde la información relativa a una variable se tiene que usar exactamente una vez. Estos sistemas permiten verificar propiedades interesantes, fuera del alcance de los sistemas tradicionales. Un ejemplo clásico es restringir las interfaces que

proveen acceso a los distintos recursos de un sistema, como archivos, *locks*, y memoria. Está claro que cada uno de estos recursos pasa por una serie de cambios de estado a lo largo de su vida: los archivos pueden ser abiertos o cerrados; los *locks* pueden ser tomados o no; y la memoria puede ser asignada o liberada. Los sistemas de tipos subestructurales proveen mecanismos para hacer un seguimiento estático coherente de estos cambios de estado, con el fin de prevenir la realización de operaciones en objetos que están en un estado inválido, como lo es leer o escribir en un archivo cerrado. Los sistemas de tipos tradicionales no pueden verificar estas propiedades y, por ende, las operaciones asociadas a recursos siempre pueden producir errores en tiempo de ejecución.

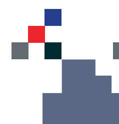
TIPOS AVANZADOS II: EFECTOS

La segunda familia de sistemas de tipos que aumenta considerablemente el rango de problemas controlables estáticamente es la de tipos de efectos. Como hemos visto, un sistema de tipos caracteriza el rango de posibles valores que puede resultar de la evaluación de una expresión, o sea, el "qué" se computa, pero no dice nada del "cómo" se obtiene el resultado. En un sistema de efectos, el tipo de una expresión también refleja los posibles efectos computacionales (acceso y escritura en memoria, entradas y salidas, etc.) que la evaluación de la expresión puede provocar. Un ejemplo seguramente muy familiar para los que han programado en Java, es el de las excepciones verificadas. En Java, el tipo de un método no solamente refleja el tipo de los argumentos y del valor retornado, sino también la posibilidad que el método lance una excepción, la cual produce una transferencia del flujo de control desde el que lanza la excepción al que la captura. Este es un ejemplo de un efecto de control. El sistema de tipos asegura en este caso que ninguna excepción declarada puede pasar desapercibida. Un área donde se usan mucho los tipos de efectos es para evitar los infames *data races* en programación concurrente: volviendo a nuestro programa estrella `e1+e2`, el conocer estáticamente que `e1` y `e2` sólo pueden leer o escribir (efectos de memoria) en regiones de memoria disjuntas, permite garantizar que es correcto ejecutar ambas expresiones en paralelo.

En forma relacionada, el saber que una expresión no tiene efectos computacionales (es decir, que es una expresión "pura") permite varias optimizaciones derivadas del razonamiento basado en ecuaciones, que conocemos de nuestros cursos de álgebra. Si una expresión `e1` aparece múltiples veces en un programa y es pura, entonces es totalmente correcto reemplazar `e1` por su valor v, porque `e1=v`. Así evitamos calcular `e1` varias veces. Para entender porqué esto no es cierto para cualquier expresión, considere que `e1` imprime en pantalla "ok". Claramente no es lo mismo para el usuario ver una vez "ok" que verlo varias veces. Lo mismo si `e1` retira 100 dólares de su cuenta. Un caso ejemplar de estas ideas es el lenguaje de programación Haskell. Es un lenguaje funcional en el cual todas las expresiones son puras: no existe ninguna forma nativa de hacer asignaciones, por ejemplo. Como todo lenguaje práctico tiene que permitir de alguna manera tener algún efecto sobre el mundo real, los diseñadores de Haskell tuvieron que ingeniar un mecanismo para soportar efectos dentro de este marco puro. Resulta que la respuesta vino del lado de las mónadas, estructuras componibles que representan computación. Sin entrar en detalles, en Haskell una función de tipo `Int→Int` es una función pura que, dado un entero, retorna un entero. Y nada más; el compilador puede hacer todas las optimizaciones que quiera cuando ve aplicaciones de dicha función. En cambio, si la función hace uso de entradas y salidas como imprimir en pantalla, esto se ve reflejado en su tipo, que pasa a ser `Int→IO Int`, donde `IO` es la mónada de los efectos de entradas y salidas. Existen mónadas para excepciones, continuaciones, estado compartido, no-determinismo, etc. Y obviamente un programador puede definir sus propios efectos, por ejemplo si quiere razonar sobre el flujo de valores en el programa, lo cual tiene aplicaciones en seguridad. Esta disciplina, por más estricta que sea, fomenta un estilo de programación donde se separa el manejo de efectos de las partes más computacionales de un sistema, lo que mejora su modularidad al mismo tiempo que permite más razonamiento estático, incluyendo más optimizaciones.

TIPOS AVANZADOS III: TIPOS DEPENDIENTES

Finalmente, es interesante considerar el caso de los tipos dependientes, que es el más extremo –y por ende también el más prometedor y desafiante– en muchos aspectos. Un tipo dependiente es un tipo (o más bien una familia de tipos) que depende de un valor. Por ejemplo, el tipo `Array[n]` es un tipo dependiente que representa los arreglos de tamaño `n`. En su forma más expresiva, los tipos dependientes son provistos por lenguajes que consideran los tipos como cualquier otro valor. Es decir que permiten definir funciones que producen tipos. ¿Qué se gana con



esto? La posibilidad de hablar de propiedades mucho más precisas. Consideremos el tipo de una función de ordenamiento de arreglos numéricos, `sort`. En un lenguaje tradicional lo único que podemos decir es que sort tiene el tipo `Array→Array` (o, si tenemos polimorfismo paramétrico, que tiene el tipo `Array[Num]→Array[Num]`. No podemos expresar al nivel del tipo de `sort` propiedades fundamentales para la corrección de `sort`. Por ejemplo, uno podría implementar `sort` como una función que retorna un arreglo arbitrario, y el sistema de tipos estaría satisfecho. Con un sistema de tipos dependiente, al dar a `sort` el tipo `Array[n]→Array[n]`, uno al menos garantiza que `sort` retorna un arreglo del mismo tamaño que el que recibió. Uno puede ir más lejos y especificar al nivel de tipos que el arreglo retornado tiene que ser una permutación del arreglo de entrada (o sea, que no se pueden inventar nuevos elementos), y ¡también que tiene que ser una permutación ordenada! Similarmente, el tipo de la función que concatena dos arreglos, `append`, es `Array[m]→Array[n]→Array[n+m]`, especificando que dado un arreglo de tamaño `n` y otro de tamaño `m`, el arreglo resultante es de tamaño `n+m`.

El lector atento y preocupado por los demonios de la indecidibilidad debería saltar de su silla en este momento: ¡estamos permitiendo que se haga computación (aquí `n+m`) al nivel de los tipos! Esto lleva dos preguntas no menores. Primero, ¿qué pasa con la terminación de la verificación de tipos? Ciertos lenguajes con tipos dependientes limitan la computación a nivel de tipos a dominios acotados, mientras otros permiten toda la expresividad de un lenguaje completo, abandonando la garantía de terminación de la verificación de tipos; otros lenguajes simplemente no permiten recursión generalizada en el lenguaje integrado. La segunda pregunta es cómo el sistema de tipos puede ser suficientemente poderoso para verificar estas propiedades, dado que las más interesantes son indecidibles. La respuesta es que no lo es, pero que permite al programador proveer una demostración de dicha propiedad, en conjunto con el programa. Este punto es quizás lo más fascinante, ya que explota una conexión muy profunda entre los cálculos computacionales y las lógicas formales (ver Figura 2). Un programa en un lenguaje de sistemas de tipos dependientes es entonces una mezcla de programas escritos por su contenido computacional (el trabajo que hacen) y programas escritos por su contenido “demostracional” (la demostración de que el programa computacional cumple alguna propiedad formulada como una proposición lógica al nivel de los tipos). El potencial de los tipos dependientes es el de crear programas certificados, desde su construcción. Diseñar e implementar lenguajes de programación con tipos dependientes que sean prácticos es un tema abierto, como lo es el explorar todo el espectro de aplicaciones de esta técnica.

EN CONCLUSIÓN

La verificación basada en tipos tiene un alto potencial de ser adoptada en la industria, por múltiples razones. Primero, no apunta a la verificación de propiedades globales de un sistema completo. Aún con tipos dependientes, nunca es necesario establecer un teorema sobre el sistema en su globalidad. Solamente se expresan propiedades locales. La conexión lógica entre estas propiedades locales y las propiedades globales que se desean no se verifica. La experiencia con otras técnicas de verificación formal muestra que apuntar a verificar dicho razonamiento global es un tremendo esfuerzo, fuera del alcance para la mayoría de la industria del software. Segundo, los programadores ya están acostumbrados a desarrollar en lenguajes con tipos estáticos. Si bien los sistemas de tipos que se exploraron aquí son semánticamente bastante más ricos que los tipos tradicionales, la metodología de programación es muy cercana. No es necesario aprender un nuevo lenguaje de modelamiento o una nueva herramienta de análisis. La integración fuerte entre programar, especificar y verificar, es un factor de adopción importante. Podemos prever un escenario de adopción progresiva de estas técnicas de verificación, con sistemas de tipos con una semántica paulatinamente más rica. De hecho, este escenario ya se hizo realidad cuando Java adoptó el polimorfismo paramétrico en su versión 1.5, y ahora con el lenguaje Scala –cuyo sistema de tipos es mucho más avanzado que el de Java– que está progresivamente reemplazando a Java en muchas aplicaciones (por ejemplo, Twitter y LinkedIn). Además, es posible diseñar sistemas de tipos muy expresivos que sean graduales, con tal de que no sea necesario demostrar todas las propiedades a la vez, sino que se puedan dejar varias a cargo de una verificación dinámica, e ir elaborando la verificación estática paso a paso.

Encontrar un sabio equilibrio entre la complejidad de la programación verificada y sus beneficios en términos de correctitud es un desafiante problema de Ingeniería de Software, que merece ser estudiado en profundidad. Dada la enorme dependencia de nuestra sociedad en los sistemas de software, es de esperar que la programación verificada se vuelva una práctica común, contribuyendo ampliamente a la correctitud del software que usamos día a día.

Si quiere saber más, dos excelentes libros de referencia son:

[1] Benjamin Pierce. Types and Programming Languages. MIT Press. 2002.

[2] Benjamin Pierce (editor). Advanced Topics in Types and Programming Languages. MIT Press. 2004.

Además, este artículo corto es muy claro e inspirador:

Tim Sheard, Aaron Stump, Stephanie Weirich. Language-based verification will change the world. Future of Software Engineering Research. ACM. 2010. BITS

La Correspondencia de Curry-Howard

La correspondencia de Curry-Howard establece un puente entre tipos y proposiciones lógicas, y programas y demostraciones de dichas proposiciones. Fue descubierta y refinada por los matemáticos Haskell Curry y William A. Howard.

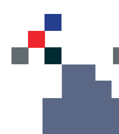
Consideremos la proposición lógica $(A \supset B) \wedge (B \supset C) \supset (A \supset C)$, que expresa la transitividad de la implicación lógica (si A implica B y B implica C entonces A implica C). Esta proposición corresponde al tipo $(A \rightarrow B) \times (B \rightarrow C) \rightarrow (A \rightarrow C)$. Lo único que hicimos es cambiar la notación de la implicación $\supset$ con la flecha $\rightarrow$ (tipo de funciones), y la notación de la conjunción $\wedge$ con el producto cartesiano $\times$. Este tipo es el de una función que toma dos argumentos, una función de tipo $A \rightarrow B$ y una de tipo $B \rightarrow C$ y retorna un función de tipo $A \rightarrow C$. ¿Existe un programa con este tipo? Si, el que retorna la función que compone ambas funciones recibidas como argumento:

```
f(g1:A→B, g2:B→C)(x:A) = g1 (g2 x)
```

En cambio, el hecho de que la propiedad $(A \supset B) \supset (A \supset C)$ no es cierta se ve reflejado en que no es posible definir una función pura con el tipo $(A \rightarrow B) \rightarrow (A \rightarrow C)$. Si lo duda, ¡inténtelo!

La correspondencia de Curry-Howard no se limita a la lógica proposicional de primer orden, conectando implicancia y funciones, pares y sumas con conjunción y disyunción. También conecta las cuantificaciones universales y existenciales con tipos dependientes, la lógica clásica con el manejo de continuaciones, la lógica modal con efectos, etc. Un buen punto de partida es la entrada Wikipedia: http://en.wikipedia.org/wiki/Curry-Howard_correspondence.

Figura 2



Teoría de Bases de Datos

Jorge Pérez
Profesor Asistente Departamento de Ciencias de la Computación, Universidad de Chile. Doctor en Ciencias de la Ingeniería (2011), Magíster en Ciencias de la Ingeniería (2004) e Ingeniero Civil en Computación (2003), Pontificia Universidad Católica de Chile. Líneas de Investigación: Bases de Datos - Datos Web, Lógica en Ciencia de la Computación.
jperez@dcc.uchile.cl

El área de Bases de Datos se preocupa de almacenar, consultar y actualizar grandes volúmenes de información. Las bases de datos están presentes hoy en innumerables aplicaciones y contextos, y muchos profesionales y autodidactas no necesariamente interesados en la computación, las usan a diario. Lo que muchos posiblemente desconocen es que el área de Bases de Datos posee una muy rica teoría que la soporta. De hecho, representan uno de los mejores ejemplos de la aplicación exitosa de teoría en la práctica. En este documento presentaremos de manera introductoria, algunos de los componentes teóricos de las bases de datos, los problemas y preguntas fundamentales que surgen, y cómo las respuestas y soluciones a estos problemas han implicado el desarrollo de un área que hoy es considerada una de las más importantes en Ciencia de la Computación. Nos centraremos en la teoría de las bases de datos relacionales, pero también incluiremos brevemente otros tipos de bases de datos.

Una base de datos relacional es un conjunto de *tablas* o *relaciones* (de ahí el nombre de "relacional"). Cada tabla o relación tiene un nombre, además de nombres para cada columna llamados atributos. Por ejemplo, la siguiente base de datos de bebedores tiene dos relaciones: **frecuenta** y **sirve**, la primera con atributos **bebedor** y **bar**, y la segunda con atributos **bar** y **cerveza**[1].

frecuenta	bebedor	bar
	Juan	Constitución
	Pablo	Liguria
	Marcelo	Loreto
	...	...

sirve	bar	cerveza
	Constitución	Delirium
	Liguria	Chimay
	Liguria	Kriek
	Loreto	Chimay
	Loreto	Delirium
	...	...

El uso habitual de una base de datos es el de consultar los datos. Por ejemplo, ser capaz de obtener desde la base de datos todos los nombres de bebedores que frecuentan bares que sirven Chimay y Kriek. En una base de datos de vuelos, podría ser obtener todos los pasajeros que deben conectar entre los vuelos 1380 y 960, o en una base de datos de estudiantes y cursos, decidir si existe un alumno tomando simultáneamente Cálculo y Computación.

LENGUAJES DE CONSULTA: LÓGICA EN ACCIÓN

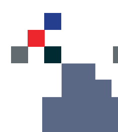
Una de las ventajas cruciales de las bases de datos relacionales son los llamados *lenguajes de consulta*. Tomemos por ejemplo el caso de la consulta:

C1: "Obtener los nombres de bebedores que frecuentan bares que sirven Kriek"

¿Cómo obtenemos esta información? Una posibilidad es usar un programa imperativo con la siguiente estrategia: avanzar una por una en las filas de la tabla **frecuenta** y para cada tupla **(bebedor, bar)** en esa tabla, buscar en la tabla **sirve** el nombre del bar y chequear que ese bar tiene datos para la cerveza Kriek. Si para cada posible consulta que necesitemos tuviéramos que diseñar una estrategia como la anterior, las bases de datos no serían tan usadas hoy en día.

En vez de diseñar un procedimiento para cada posible consulta, lo que se hace en una base de datos es expresar lo que queremos en un *lenguaje declarativo*, es decir, un lenguaje mucho más cercano a lo que esperamos obtener que al pro-

[1] El ejemplo de los bebedores, es un ejemplo clásico de la literatura de Bases de Datos.



cedimiento necesario para obtenerlo. El lenguaje paradigmático es la Lógica de Predicados de Primer Orden. La anterior consulta puede simplemente expresarse como:

C1: $\exists$ ***Y(frecuenta (X,Y) AND sirve (Y, "Kriek"))***

Ésta es una fórmula que esencialmente obtiene todos los X (bebedores) para los que existe un valor Y tal que (X,Y) es una fila en la tabla **frecuenta** y (Y,"Kriek") es una fila en la tabla **sirve**. Similarmente se puede hacer fácilmente una consulta, por ejemplo, para obtener los bebedores que frecuentan bares que sirven al menos dos cervezas:

C2: $\exists$ ***Y*** $\exists$ ***U*** $\exists$ ***V***
(frecuenta (X,Y) AND sirve (Y, U) AND sirve (Y,V) AND U≠V)

Edgard F. Codd en 1970 [1] demostró que toda fórmula de Lógica de Primer Orden podía traducirse en una expresión de lo que él llamó Álgebra Relacional, un álgebra muy simple de operaciones sobre tablas. Así por ejemplo, la consulta C1 se puede escribir como:

C1': $\pi_{bebedor} (\sigma_{cerveza} = \text{"Kriek"} (frecuenta \bowtie sirve))$

El álgebra relacional se basa en un par de operaciones básicas: proyección (π) usada para quedarse con alguna de las columnas de una relación (en el ejemplo, con la columna correspondiente al bebedor), *selección* (**σ**) usada para seleccionar las filas que cumplen con cierta condición (en el ejemplo las filas tales que el atributo cerveza es "Kriek"), y producto o join (⋈) que se usa para combinar dos tablas que comparten un mismo atributo (en el ejemplo, las tablas frecuenta y sirve combinadas por su atributo común bar). Adicionalmente cuenta con las operaciones de conjuntos, unión (U) y diferencia (-), y una operación auxiliar para cambiar (dentro de una expresión) el nombre de una tabla. Codd demostró también que toda expresión del álgebra relacional podía traducirse en una fórmula de lógica de primer orden y por lo tanto el álgebra y la lógica definían exactamente las mismas consultas. El resultado de Codd es crucial porque para responder cualquier consulta en la lógica, sólo se necesitan procedimientos que implemente cada una de las operaciones básicas del álgebra. Dado que estas operaciones pueden implementarse por separado, ellas pueden optimizarse de manera específica. Esto hace que hoy las consultas a una base de datos relacional puedan responderse eficientemente.

El trabajo de Codd significó el inicio de lo que hoy conocemos como las bases de datos relacionales y es en gran parte el responsable de que las bases de datos sean tan ampliamente usadas hoy. Por un lado la lógica nos da un lenguaje simple donde expresar lo que queremos obtener. Por otro, el álgebra nos permite implementar estas consultas de manera muy eficiente. En el momento que Codd propuso su modelo basado en relaciones, álgebra y lógica, habían otros sistemas de bases de datos pero estos eran *ad hoc*, difíciles de usar y con poco soporte teórico. Por su trabajo de 1970, Codd recibió en 1981 el "ACM Turing Award", considerado el premio Nobel de computación. Murió en 2003 a la edad de 79 años. Cabe destacar que tanto el álgebra relacional como la lógica de primer orden están hoy plasmadas en el lenguaje SQL, el estándar para consultar bases de datos relacionales.

EXPRESIVIDAD VERSUS COMPLEJIDAD

El uso de lógica en el corazón de las bases de datos nos permite estudiarlas teóricamente y responder formalmente preguntas muy interesantes. Por ejemplo ¿qué tipo de consultas son las que puede responder una base de datos relacional? Suponga que se tiene una base de datos de conexiones de vuelos directos, por ejemplo, que hay un vuelo directo entre Santiago y Nueva York, y otro entre Nueva York y Londres, etc. Suponga ahora que se quiere responder la siguiente consulta:

C3: "¿Es posible volar desde Santiago a Moscú?"

La consulta implica que no nos importa tener escalas, sólo si es que será posible completar el viaje. Ciertamente la información necesaria para responder **C3** se encuentra en la base de datos, pero se puede demostrar formalmente que esta consulta no se puede *expresar* como una fórmula de la lógica de primer orden y, por lo tanto, (por el resultado de Codd) no existe una consulta en el álgebra relacional que permita responder **C3** [2]. Esto implica que por ejemplo, una consulta de SQL tampoco podrá usarse para responder **C3**. Esencialmente, la lógica de primer orden no se puede utilizar para consultar en general por caminos de *largo arbitrario.* Y esto se extrapola a diferentes escenarios. Por ejemplo, suponga que tiene una base de datos que mantiene las dependencias de cursos en una carrera universitaria, es decir, qué curso es requisito de qué otro. Naturalmente se necesita que este conjunto de dependencias no contenga ciclos ya que implicaría que

los estudiantes no podrían completar la carrera. Es decir queremos impedir que exista una secuencia de cursos A1, A2, ..., Ak, tal que A1 es requisito de A2, A2 es requisito de A3, etc. y que finalmente Ak sea requisito de A1. Pues bien, se puede demostrar que no existe una fórmula en lógica de primer orden que me permita chequear si hay un ciclo en las dependencias de cursos; el álgebra relacional no tiene el poder expresivo suficiente para determinar la existencia de ciclos en los datos. No es difícil notar que los ciclos y los caminos de largo arbitrario están íntimamente relacionados.

Lo profundo del resultado es indicarnos las limitantes del álgebra en cuanto a su poder para responder consultas. Pero la teoría también nos ayuda a determinar qué es lo que le falta a la lógica para ser capaz de responder **C3**. En este caso lo que le falta a la lógica es recursión. No es importante la definición formal para efectos de este artículo, pero lo interesante es que dado que **C3** es una consulta natural que nos gustaría responder en un sistema de bases de datos, necesitamos agregar poder expresivo a nuestro lenguaje. En la práctica esta observación teórica se tradujo en una adición al estándar inicial de SQL (el estándar SQL 3 incluye la posibilidad de hacer recursión).

A priori suena simple y hasta atractivo lo de agregar expresividad a un lenguaje, pero ¿no estaremos perdiendo algo? La respuesta es sí, lo que se pierde es *eficiencia:* cada nueva funcionalidad agregada a un lenguaje implica la implementación de una nueva operación la que puede ser más costosa de ejecutar en un computador. Esto nos lleva al tema de complejidad.

En complejidad computacional se intenta determinar qué tantos recursos computacionales se necesitarán para resolver un problema particular. Generalmente se considera el tiempo como el más importante de estos recursos. Para el caso de las bases de datos el problema es, dada una consulta fija sobre una base de datos de tamaño N ¿cuánto tiempo tardará en ejecutar la consulta como función de N? Si ahora consideramos todas las posibles consultas en un lenguaje específico y tomamos el máximo de estas funciones obtenemos la *complejidad del lenguaje*.

Se puede demostrar que la complejidad de la lógica de primer orden es sumamente baja: la complejidad está en la clase AC^0 [3] que contiene problemas masivamente paralelizables. Esta clase está propiamente contenida en la clase de problemas solubles en tiempo polinomial, PTIME. En cambio, DATALOG que es el lenguaje estándar que incluye recursión, es completo para PTIME y, por lo tanto, considerablemente más complejo de evaluar que la lógica de primer orden. A pesar de ser más complejo que la lógica de primer orden, DATALOG sigue teniendo un procedimiento de evaluación eficiente (de hecho, de tiempo polinomial) por lo que se puede utilizar en la práctica. Sin embargo uno podría agregar mucha expresividad a un lenguaje y sacrificar la complejidad a un nivel que lo vuelva impráctico. Un ejemplo es la lógica de segundo orden que permite expresar consultas sumamente complejas. Considere nuevamente la base de datos de vuelos directos, y suponga que además se cuenta con una tabla **países** de nombres de países por los que un viajero quiere pasar. La lógica de segundo orden permite expresar consultas como la siguiente:

C4: "¿Existe un itinerario de vuelo que pase exactamente una vez por cada país en la tabla países?"

Usar lógica de segundo orden es poco razonable ya que su complejidad es más alta que la de los problemas NP-Completos y por lo tanto será poco usable en la práctica ya que la ejecución de una consulta tardará un tiempo demasiado alto.

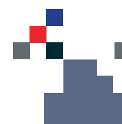
En general se pueden utilizar diversos lenguajes de consulta en bases de datos relacionales y siempre existirá este "tira y afloja" entre expresividad y complejidad. El ideal es llegar a un equilibrio entre ambas medidas; encontrar un lenguaje suficientemente expresivo para las aplicaciones que quiero modelar y a la vez suficientemente eficiente y simple de ejecutar para que las consultas se puedan responder en la práctica. La lógica de primer orden es un muy buen ejemplo en el caso de las bases de datos relacionales.

MÁS ALLÁ DE SÓLO CONSULTAR

Hay muchos otros problemas en teoría de bases de datos más allá de sólo consultar. Uno de ellos es el problema de equivalencia de consultas: dadas dos consultas distintas en álgebra relacional (o lógica de primer orden), ¿será el caso que para todas las posibles bases de datos las consultas entregan exactamente el mismo resultado? Alguien podría pensar que esto no puede ocurrir si las consultas son distintas, pero considere la siguiente consulta:

$$\textbf{C5: } \pi_{bebedor}\,(\text{frecuenta} \bowtie (\sigma_{cerveza\,=\,\text{"Kriek"}}\ \text{sirve}))$$

No es difícil notar que, sea cual sea la base de datos donde la ejecutemos, las consultas **C1'** y **C5** entregarán siempre los mismos resultados. También se puede argumentar que la consulta **C5** se puede



ejecutar más eficientemente que **C1'**. Si un sistema de bases de datos pudiera entonces establecer la equivalencia entre consultas, podría notar que da lo mismo responder **C1'** o **C5,** y si alguien consulta por **C1'**, el sistema podría en vez ejecutar **C5**, optimizando el tiempo de respuesta. Note que establecer la equivalencia entre consultas no parece ser un problema trivial para nada. De hecho ¿cómo chequeamos en general que dos consultas entregan el mismo resultado para todas las posibles bases de datos? La mala noticia es que el problema de equivalencia para el álgebra relacional es *indecidible:* **no existe un algoritmo que, dadas dos consultas del álgebra relacional, determine si éstas son equivalentes.** Una noticia negativa de parte de la teoría para la práctica.

Lo típico que se hace en investigación teórica en computación una vez que se determina que un problema no puede ser resuelto algorítmicamente, es buscar restricciones bajo las cuales el problema se vuelve decidible. Una interesante restricción del álgebra relacional es el fragmento que considera expresiones donde sólo las operaciones de proyección (π), selección (σ) y join ($\bowtie$) son permitidas.

Este fragmento se llama *fragmento conjuntivo* del álgebra relacional. El fragmento conjuntivo contiene la mayor parte de las consultas de la forma SELECT-FROM-WHERE de SQL y, por lo tanto, tiene gran interés práctico. Se puede demostrar [4] que para este fragmento el problema de determinar equivalencia de consultas es NP-Completo. Si bien el problema no tiene una solución eficiente (NP-Completo significa que será difícil de resolver en la práctica), al menos sabemos que el problema es decidible. Otro problema de interés teórico que ha tenido implicaciones prácticas es el de *responder consultas usando vistas* [5]. Suponga que un sistema de bases de datos tiene un conjunto de consultas **V1, V2, V3,...** etc., *precomputadas*, es decir, antes las consultó y el resultado de cada una está almacenado. A este conjunto de consultas se les llama vistas. Cuando llega una nueva consulta, digamos **Q**, al sistema de base de datos, se podría antes de ejecutar **Q** desde cero intentar computarla como una combinación de las vistas **V1, V2, V3,...,** y así ahorrar considerable tiempo. Una solución a este problema permitiría a un sistema de bases datos mantener un caché de consultas más solicitadas, o suficientemente generales, de manera tal que puedan ser utilizadas para responder diversas otras consultas. En [5], los autores plantearon este problema y propusieron un algoritmo para el caso en que las vistas son expresadas en el fragmento conjuntivo del álgebra relacional.

El problema de responder consultas usando vistas se ha usado (tanto su teoría como sus algoritmos) en diversos otros problemas de interés práctico, como por ejemplo, el de integración de datos. En un sistema de integración de datos, ya no es sólo una base de datos la que se quiere consultar sino varias posiblemente distribuidas. El usuario no tiene acceso a cada una de las bases de datos por separado sino más bien tiene acceso a una única interfaz global. Se puede pensar por ejemplo en un sistema de bibliotecas distribuidas en distintas universidades. Cada universidad almacena información de libros de manera independiente y no necesariamente usando los mismos esquemas (tablas, nombres de tablas y atributos). Para simplificar el acceso a todas estas posibles bases de datos, se le presenta al usuario una base de datos virtual, o sea nombres y atributos de tablas pero sin datos. Cada vez que el usuario hace una consulta sobre la base de datos virtual, el sistema traduce estas consultas en diversas consultas sobre las distintas bases de datos de bibliotecas, en donde se computan las respuestas y luego se entrega una visión unificada al usuario. Una forma de modelar y resolver este problema, es pensar que cada posible base de datos distribuida es una vista precomputada de la base de datos global (virtual). El sistema entonces no tiene acceso a los datos de la base de datos global y cada nueva consulta debe ser respondida usando sólo la información de las vistas, que en este caso corresponderían a cada una de las bases de datos de bibliotecas independientes [6, 7].

NUEVOS MODELOS DE DATOS

Hemos hecho una revisión de algunos problemas teóricos importantes en bases de datos relacionales. Obviamente la lista visitada está lejos de ser exhaustiva. Una buena referencia para el lector interesado puede ser [8, 9]. Muchos de estos problemas teóricos que han sido estudiados por décadas, hoy deben ser revisitados ante el surgimiento de nuevos modelos de datos principalmente motivados por el manejo e intercambio de información en la Web.

Uno de estos modelos es XML, que ha demostrado ser una interesante fuente de desafíos teóricos donde los resultados para bases de datos relacionales no siempre se pueden adaptar. Otro modelo muy en boga hoy es el modelo de datos de grafos. En este modelo, los datos están representados como puntos y las relaciones entre ellos como links. Un típico ejemplo son las redes sociales tipo Facebook o Twitter. En estas nuevas bases de datos, los datos atómicos son los identificadores de objetos, sean estos usuarios, empresas, ciudades, etc., y las relaciones entre ellos vienen dadas por las relaciones de

amistad (Facebook), por qué usuario sigue a qué otro (Twitter), etc. La misma Web es un espacio gigante de almacenamiento de datos estructurado como grafo. En estos nuevos modelos de datos, las consultas que se quiere responder son principalmente consultas de navegación, por ejemplo, saber cuántos usuarios se pueden alcanzar desde uno en particular siguiendo las "relaciones de amistad". El problema de complejidad se vuelve aún más desafiante dado el gran volumen de datos que deben ser consultados y nuestra discusión acerca de la relación entre expresividad y complejidad toma un nuevo color. En estas aplicaciones de datos a gran escala no tenemos otra opción si no sacrificar expresividad si queremos obtener respuestas en tiempos razonables. Los mismos problemas de equivalencia, responder consultas usando vistas, o integración de datos toman también un nuevo aire.

Hay mucho por hacer hoy en el estudio teórico de Bases de Datos y como ha sido la tónica en los últimos cuarenta años, es de esperar que esta investigación teórica pueda ser exitosamente traspasada a la práctica. BITS

Referencias

[1] E. F. Codd. A relational model of data for large shared data banks. Communications of the ACM 13 (6): 377, 1970.

[2] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In POPL 1979, pages 110-117.

[3] L. Libkin. Elements of Finite Model Theory. Springer 2004.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In STOC 1977, pages 77-90.

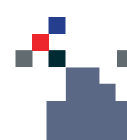
[5] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava. Answering Queries Using Views. In PODS 1995, pages 95-104.

[6] J. D. Ullman. Information Integration Using Logical Views. Theor. Comput. Sci. 239(2): 189-210, 2000.

[7] M. Lenzerini. Data Integration: A Theoretical Perspective. In PODS 2002, pages 233-246.

[8] D. Maier. The Theory of Relational Databases. Computer Science Press, 1983.

[9] S. Abiteboul, R. Hull, V. Vianu. Foundations of Databases. Addison-Wesley, 1995.



Indexación y Compresión de Datos para motores de búsqueda

Diego Arroyuelo

Académico Departamento de Informática, Universidad Técnica Federico Santa María. Doctor en Ciencias Mención Computación, Universidad de Chile; Licenciado en Ciencias de la Computación, Universidad Nacional de San Luis, Argentina. Líneas de Investigación: Estructuras de Datos y Algoritmos, Compresión y Búsqueda en Texto, Estructuras de Datos Comprimidas, Índices Comprimidos para Recuperación de la Información.

darroyue@inf.utfsm.cl

Probablemente muy pocos usuarios de un motor de búsqueda Web se han detenido a pensar, respecto de los recursos y tecnologías involucrados en la generación de la página de resultados de su consulta. La misma debe ser generada en unas pocas décimas de segundo y debe contener diez resultados relevantes para el usuario, ojalá los más relevantes. Además, debajo del título y la URL de cada uno de los resultados encontrados, la página debe mostrar pequeñas porciones del documento ─conocidas como *snippets*─ que contienen las palabras usadas en la consulta. Para complicar más la situación, usualmente decenas de miles de consultas arriban por segundo a un motor de búsqueda (aproximadamente 38 mil consultas por segundo para Google en 2012, lo que equivale a 2.4 millones por minuto, o 3.3 mil millones por día [1, 2]). Además, estas búsquedas deben realizarse sobre varias decenas de miles de millones de páginas web [3]. En este escenario, tanto la Indexación como la Compresión de Datos son indispensables para el funcionamiento de los motores de búsqueda actuales. Esto impacta en los tiempos de respuesta a las consultas, en el ahorro de espacio de almacenamiento, en la consecuente reducción del uso de espacio físico, en la reducción de los tiempos de transferencia de datos y en el ahorro de energía usada [14].

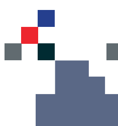
En el presente artículo mostraré los avances más recientes y los desafíos más importantes en el área de Indexación y Compresión de Datos en motores de búsqueda, particularmente aquellos en los que he estado involucrado con mi investigación. Intentaré dar pistas que permitan comprender cómo los motores de búsqueda actuales son capaces de manejar grandes volúmenes de datos y la carga de trabajo antes mencionada. Todos los resultados experimentales aquí mostrados, han sido obtenidos dentro del marco de mi grupo de investigación y validados con los resultados similares en el estado del arte.

Dada una base de datos de documentos de texto (las páginas web), denotamos con v al vocabulario con todas las palabras distintas que aparecen en dichos documentos. Para facilitar su manipulación, a cada documento se le asigna un identificador (o docID), que es un número entero que lo identifica. Los usuarios presentan consultas q a la base de datos, las que constan de una o más palabras. El objetivo es encontrar documentos relacionados con esas palabras. Existen dos variantes principales de consultas: las consultas tipo AND (que buscan los documentos que contengan a todas las palabras de q) y las consultas tipo OR (que buscan los documentos que contengan *al menos* una de las palabras de q). Para mostrar los resultados en orden de relevancia para el usuario, el motor de búsqueda debe hacer un ranking de los mismos. Ésta es una etapa muy importante para la búsqueda. Sin embargo, en este artículo no discutiremos aspectos relacionados con el ranking de resultados.

LA INDEXACIÓN DE BASES DE DATOS DE DOCUMENTOS

Los documentos de la base de datos se conocen de antemano a las búsquedas, por lo que se pueden indexar, esto es, construir una estructura de datos que acelere las respuestas a las consultas. Las estructuras de datos clásicas en este caso son los índices invertidos [9, 13, 22, 30]. Por cada palabra p del vocabulario, el índice invertido tiene una lista invertida v que almacena los docIDs de los documentos que contienen a p. Esto es similar al índice que podemos encontrar al final de la mayoría de los libros técnicos. Por cada docID en $L(p)$, también se almacena información útil para el ranking de resultados, como la cantidad de ocurrencias de p en cada uno de los documentos.

El grueso de las listas invertidas usualmente se almacena en memoria secundaria, aunque algunas listas (generalmente las más consultadas) se almacenan en memoria principal, en lo que se llama el caché de listas invertidas. Muchos motores de búsqueda incluso precalculan los resultados de las consultas más frecuen-



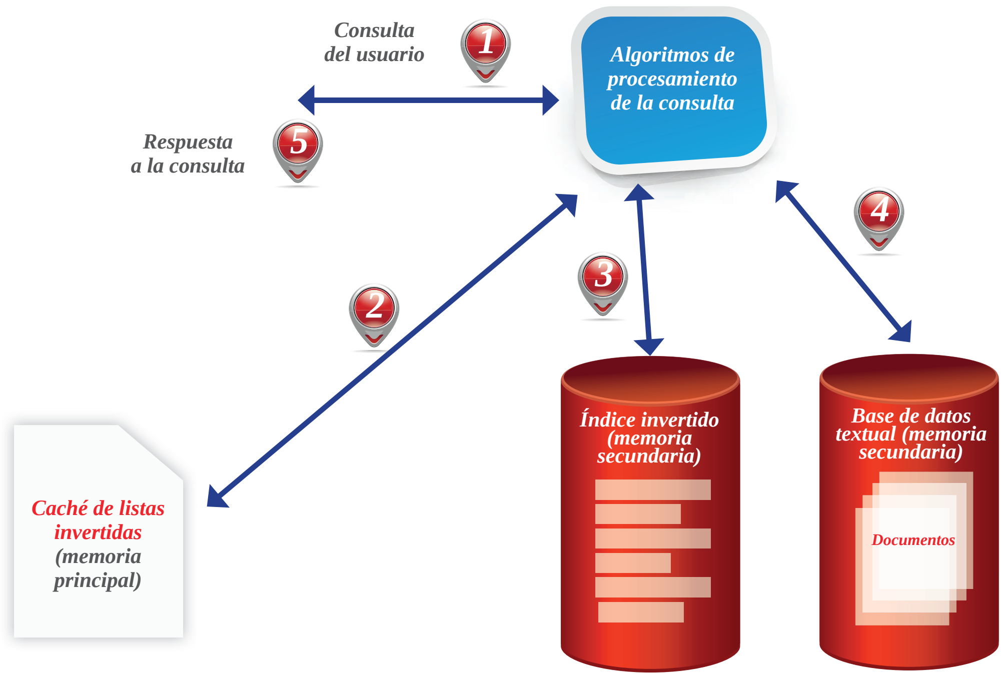


Figura 1 • Pasos para responder una consulta en un motor de búsqueda.

tes y las almacenan en caché. El motor de búsqueda también debe almacenar la base de datos de documentos en memoria secundaria. Ésta es necesaria tanto para la generación de los snippets como para mostrar la versión "en caché" de la página, entre otros usos. La Figura 1 resume el proceso de búsqueda, con los siguientes pasos: (1) La consulta llega al procesador de consultas; (2) el procesador intenta resolver la consulta usando las listas almacenadas en caché; (3) si el caché no contiene la lista invertida de alguna de las palabras de la consulta, ésta debe recuperarse desde la memoria secundaria; (4) después de usar el índice para encontrar las diez mejores páginas para la respuesta (pueden ser más), éstas deben recuperarse desde la memoria secundaria para generar los snippets; (5) la página de respuesta es enviada al usuario.

A continuación estudiaremos cómo se logra realizar el proceso de la Figura 1 de manera eficiente (en menos de un segundo el usuario debe tener su respuesta), describiendo los principales desafíos que se presentan.

EL DESAFÍO DE COMPRIMIR LOS ÍNDICES DE BÚSQUEDA

Las listas invertidas requieren usualmente una gran cantidad de espacio de almacenamiento. Por ejemplo, las listas invertidas de docIDs para la reconocida colección de documentos TREC GOV2 requieren aproximadamente 23 GB, si usamos un entero de 32 bits por cada docID (en total son 5.750 millones de docIDs en todo el índice). Dicha colección contiene páginas del dominio .gov de Estados Unidos, con 25 millones de documentos y un total de 426 GB. A raíz de esto, es común que las listas invertidas se mantengan en el disco, como se mencionó anteriormente. Dada una consulta q, las listas invertidas de cada palabra de la consulta se transfieren a memoria principal para ser procesadas. Sin embargo, la baja latencia de acceso a la memoria secundaria puede influir negativamente en la latencia de una consulta.

Una solución a este problema es comprimir las listas invertidas [13, 30]. Esto permite emplear de mejor manera el espacio disponible para el caché de listas. La compresión también produce una importante mejora en el tiempo de transferencia de las listas desde el disco, ya que se transfieren menos bytes (véase, por ejemplo, el estudio en [13, sección 6.3.6]). En ambos casos, la compresión impacta favorablemente en el tiempo de respuesta a los usuarios.

La compresión de datos sin pérdida (es decir, aquella en la que al descomprimir se obtienen exactamente los mismos datos originales), en general funciona detectando regularidades en los datos a comprimir. Por ejemplo, la compresión de textos generalmente se basa en detectar las regularidades del lenguaje escrito y así reducir el espacio de la codificación. En el caso de las listas invertidas, la compresión se logra usando lo que se conoce como *gap encoding:* los docIDs se almacenan ordenados de manera creciente en las listas invertidas. Luego, se almacena el primer docID en la lista de manera absoluta (es decir, tal cual es). El resto de los elementos de la lista, en cambio, se almacenan como la diferencia entre docIDs consecutivos en la lista (esas diferencias se conocen como gaps). Como la lista original está ordenada por docIDs crecientes, el resultado es una lista de números (gaps) enteros positivos. Posteriormente, las listas se dividen en bloques de tamaño constante (por ejemplo, 128 docIDs por bloque) y se usa un compresor de números enteros sobre los gaps [5, 13, 14, 16, 19, 23, 29, 30, 33].

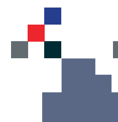
Dichos compresores codifican un número entero en una cantidad de bits proporcional al valor del entero: mientras menor es su valor, menos bits se usan. Dado que almacenamos los gaps en lugar de los docIDs, normalmente se obtienen distribuciones con muchos valores pequeños, logrando comprimir las listas. La división en bloques de las listas se hace para no descomprimir toda la lista en tiempo de búsqueda, sino sólo los bloques que contienen docIDs relevantes para una consulta. Los compresores más efectivos son aquellos que permiten una alta velocidad de descompresión ─dado que esto impacta en el tiempo de respuesta─ sumado a una tasa de compresión razonable, aunque no necesariamente óptima. Aquí se destacan los métodos VByte [29], S9 [5] y PForDelta [33].

Para dar una idea de la compresión que puede alcanzarse en la práctica, el índice invertido comprimido para TREC GOV2 usa entre 6,20 GB (PForDelta) y 7,35 GB (VByte). En promedio esto es entre 8,84 bits y 10,48 bits por docID, menor que los 32 bits originales. Respecto de la velocidad de descompresión de las listas, se obtienen entre 446 millones y 1.010 millones de docIDs por segundo. Una consulta tipo AND puede responderse en promedio en 12 a 14 milisegundos por consulta.

Algunos trabajos recientes [31] muestran cómo optimizar los resultados del párrafo precedente. Esto se logra generando gaps mucho más pequeños en las listas invertidas, mediante la asignación cuidadosa de docIDs a los documentos de la colección [10, 11, 15, 25, 26, 27]. Una de las técnicas más simples y efectivas es la de ordenar las páginas lexicográficamente por sus URLs, y luego asignar los docIDs siguiendo ese orden. De esta manera, las páginas que corresponden a un mismo sitio (y, por tanto, muy probablemente usan un vocabulario similar) van a tener docIDs consecutivos, o en su defecto muy similares. Esto se refleja en las listas invertidas, generando gaps más pequeños comparados con una asignación aleatoria de los docIDs, como la usada en los resultados del párrafo anterior. Para la colección TREC GOV2 enumerada de acuerdo a URLs, el espacio usado por el índice invertido es entre 3,69 GB (5,25 bits por docID) y 6,57 GB (9,35 bits por docID). La velocidad de descompresión es prácticamente la misma que la del párrafo anterior. El tiempo de respuesta para consultas tipo AND es ahora de entre 6 y 7 milisegundos por consulta, reduciendo a la mitad los tiempos del párrafo anterior (docIDs asignados aleatoriamente). Esta mejora se produce porque al asignar los docIDs cuidadosamente, menos bloques de la lista deben ser descomprimidos en tiempo de búsqueda [27, 31]. Nótese que este resultado permite duplicar la capacidad de respuesta, a la vez que permite usar menos espacio.

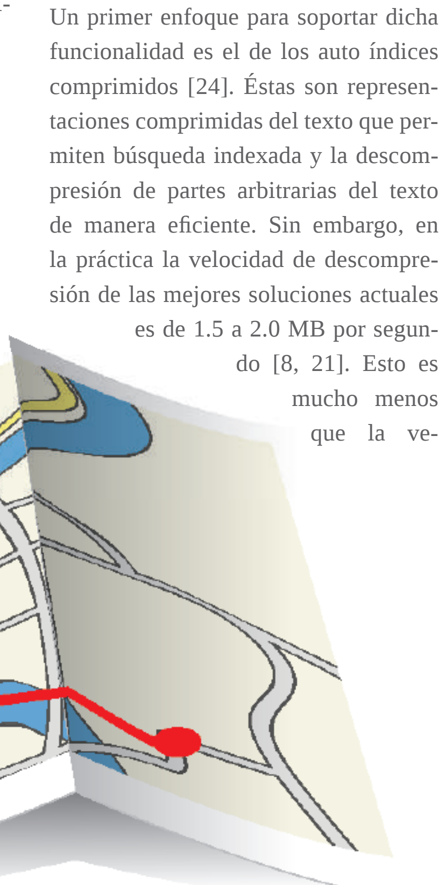
Dados estos resultados, vale la pena cuestionarse si existe una mejor manera de representar las listas invertidas cuando los docIDs han sido asignados cuidadosamente. Después de todo, los motores de búsqueda actuales trabajan casi exclusivamente en memoria principal [14], lo que hace que este tipo de mejoras sean importantes. Una alternativa puede ser usar métodos de compresión que logran buenas tasas de compresión, como Interpolative Encoding [23]. Sin embargo, este método no es eficiente al descomprimir las listas invertidas, lo que afectaría el tiempo de respuesta. Afortunadamente, hemos podido mostrar que existen mejores representaciones para las listas invertidas en esos casos [7], basándonos en la siguiente observación: el índice invertido para TREC GOV2 ordenada por URLs contiene aproximadamente 60% de gaps con valor 1 (comparado con un 10% de gaps con valor 1 para la colección ordenada aleatoriamente). Pero si el 60% de los gaps tienen valor 1, es probable que estos se agrupen en posiciones consecutivas de las listas invertidas. Llamamos *runs* a esos agrupamientos de 1s consecutivos. Los resultados empíricos en [7] muestran que esto es cierto en la práctica.

En nuestro trabajo [7] mostramos que en estos casos es mejor usar compresión *run-length* de los runs: en lugar de codificar los 1s en un run de mane-



ra explícita (como en los trabajos previos), es más eficiente representarlos implícitamente indicando la presencia del run (con algún tipo de marcador), seguido por la longitud del run. De esta manera se representan runs muy largos usando muy pocos bits. La compresión run-length es muy conocida [19] y ha sido usada con éxito en diversas áreas, como la compresión de imágenes [30]. Sin embargo, no había sido usada en el escenario planteado, aunque una técnica similar ya había sido empleada para comprimir grafos de la Web [12]. Sin embargo, dicha codificación no puede ser empleada directamente en nuestro caso. En cambio, tuvimos que adaptar los principales métodos de compresión de números enteros para soportar compresión tipo run-length. A pesar de que encontrar una asignación de docIDs que produzca la cantidad óptima de runs es un problema NP-hard [20], las enumeraciones cuidadosas de documentos producen runs de suficiente buena calidad.

Usando nuestros compresores adaptados para compresión run-length, el índice invertido para TREC GOV2 ahora requiere entre 3,31 GB (4,71 bits por docID) y 3,65 GB (5,20 bits por docID). Esto es una mejora de un 10% respecto del índice con docIDs asignados por URLs (y un 49% de mejora respecto del índice con documentos enumerados aleatoriamente). Estos métodos descomprimen entre 472 millones y 1.351 millones de docIDs por segundo (una mejora de hasta 34% respecto a los índices ordenados por URL). Esta mejora se explica porque los runs de 1s no se descomprimen de manera explícita, lo cual es útil para muchas aplicaciones. Finalmente, el tiempo de búsqueda es similar (y, en algunos casos, levemente superior) al obtenido para orden por URL, es decir 6 a 7 milisegundos por consulta. Aquí nuevamente el algoritmo que procesa la consulta no descomprime los runs de manera explícita.



EL DESAFÍO DE COMPRIMIR LA WEB TEXTUAL

Para poder generar snippets, los motores de búsqueda deben almacenar en sus servidores el texto de las páginas web. Obviamente, esto usa mucho espacio, por lo que debe comprimirse. Además, para responder a una consulta se deben extraer los diez mejores documentos encontrados y generar sus snippets. El principal problema aquí es cómo comprimir textos de gran tamaño y soportar la descompresión de páginas arbitrarias eficientemente. Conviene recordar que la compresión se basa en detectar regularidades en los datos. Cuando el volumen de datos es muy grande, esto no es una tarea fácil [18].

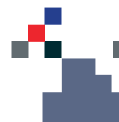
Un primer enfoque para soportar dicha funcionalidad es el de los auto índices comprimidos [24]. Éstas son representaciones comprimidas del texto que permiten búsqueda indexada y la descompresión de partes arbitrarias del texto de manera eficiente. Sin embargo, en la práctica la velocidad de descompresión de las mejores soluciones actuales es de 1.5 a 2.0 MB por segundo [8, 21]. Esto es mucho menos que la ve-

locidad de descompresión lograda por compresores estándar como `snappy` o LZ4, que llega a ser de 1.0 a 1.5 GB por segundo [4], o incluso la velocidad de descompresión de `gzip` que llega a ser de cientos de MBs por segundo. Por esta razón los motores de búsqueda prefieren usar compresores estándar, en particular de la familia Lempel-Ziv de 1977 [32], como todos los mencionados anteriormente. Una solución ineficiente es concatenar las páginas web formando un único texto, y luego usar un compresor estándar. Sin embargo, para extraer una página web debemos descomprimir todo el texto, algo absurdo en el caso de la Web. Otra posible solución es comprimir las páginas por separado. Pero esto afecta notablemente la tasa de compresión, ya que las regularidades entre páginas no son comprimidas.

Una solución bastante aceptada en la literatura [18, 6] corresponde a un punto intermedio entre los enfoques anteriores. Se concatenan las páginas web, pero esta vez formando bloques de un tamaño máximo definido (generalmente entre 50 KB y 1 MB). Luego, cada bloque es comprimido por separado. Para obtener una página web sólo se debe descomprimir el bloque que la contiene, lo cual es eficiente en tiempo de búsqueda. Además, se logra una buena tasa de compresión, porque las regularidades entre las páginas que conforman un bloque son eventualmente detectadas por un compresor. Los resultados de Ferragina y Manzini [18] muestran que la Web textual puede comprimirse a un 5% de su tamaño original usando compresores estándar. Esto fue muy comentado en su momento, ya que es bastante menos que las tasas de compresión de 20% a 25% observadas en documentos convencionales. La razón es que el gran volumen de texto de la Web contiene muchas repeticiones y regularidades que pueden aprovecharse para mejorar la compresión. Sin embargo, lograr esa tasa de compresión tiene su precio: nuestro trabajo [6] mostró que, en el escenario de un motor de búsqueda, extraer las diez primeras páginas de respuesta a una consulta usando dicho enfoque toma 25 milisegundos por consulta. Esto es muy costoso y afecta al tiempo total de respuesta.

Otra solución efectiva es la de Turpin et al. [28], donde proponen enumerar el vocabulario de la colección de acuerdo a la frecuencia con la que las palabras aparecen en las páginas web: la palabra más frecuente recibe un identificador de palabra 0, la segunda palabra más frecuente recibe un identificador 1, y así siguiendo. Luego, las palabras de cada página son reemplazadas por su correspondiente identificador, obteniendo una secuencia de números enteros. Dado que las palabras más frecuentes tienen identificadores con menor valor, se utiliza un método de compresión de números enteros (tal como los usados para comprimir listas invertidas) sobre estas secuencias, obteniendo compresión. Sin embargo, las tasas de compresión alcanzadas son menores a las logradas con compresores estándar: 29% para TREC GOV2. Esto tiene una explicación teórica: este tipo de compresión basada sólo en la frecuencia de las palabras se conoce como compresión de orden 0. Los compresores estándar, por otra parte, alcanzan compresión de orden superior: son capaces de detectar los contextos en que aparecen las palabras, por lo tanto cuentan con mayor información que les permite una mejor compresión. Para una consulta, los 10 mejores resultados pueden descomprimirse en sólo 0.2 milisegundos, lo cual no afecta el tiempo total de una consulta [6]. La practicidad de este método ha hecho que algunos motores de búsqueda lo adopten para almacenar sus colecciones de documentos.

Una manera efectiva de mejorar la tasa de compresión alcanzada con Turpin et al. es usar una compresión de dos fases [6]. Es importante notar que el método de Turpin et al. comprime el texto modificando la manera en que codifica las palabras del mismo. Esto quiere decir que la estructura original del texto (y sus potenciales regularidades) se mantienen y, por ende, es posible aplicar un método de compresión estándar sobre él [17]. La compresión de dos fases entonces significa: en la primera fase se comprime con Turpin et al., mientras que en la segunda fase se aplica un compresor estándar al resultado de la primera fase. En particular, en [6] los mejores compromisos entre tasa de compresión y velocidad de descompresión se lograron usando el compresor `snappy` en la segunda fase. Dicho compresor es de los más rápidos para descomprimir, pero tiene tasas de compresión del 40% - 50%, lo cual es poco eficiente [4]. Las malas tasas de compresión se deben a que `snappy` puede detectar regularidades dentro de una ventana muy pequeña del texto (aproximadamente 32 KB). Pero si en la primera etapa el texto se comprime con Turpin et al., esto significa que artificialmente estamos permitiendo que más regularidades quepan dentro de la ventana de compresión. Esto ha mostrado ser muy efectivo, ya que se alcanzan tasas de compresión del 12% con una velocidad de descompresión de 4 milisegundos por consulta, e incluso tasas de compresión del 16% con una velocidad de descompresión de 0.5 milisegundos por consulta [6]. Esto explica en parte cómo un motor de búsqueda puede manejar grandes volúmenes de texto, y a la vez responder consultas y mostrar snippets en pocas décimas de segundo.



CONCLUSIONES Y TRABAJOS FUTUROS

La indexación y la compresión de datos son vitales para el funcionamiento de los motores de búsqueda actuales. Los resultados mostrados permiten comprender (a grandes rasgos) cómo hacen los motores de búsqueda para responder consultas de manera muy rápida sobre grandes volúmenes de texto. Nuestro trabajo apunta a mejorar la eficiencia general de un motor de búsqueda. Nuestros principales resultados muestran mejores compromisos entre espacio usado y tiempo requerido para soportar las búsquedas [6, 7]. En particular, buscamos hacer más eficientes los pasos (2), (3) y (4) de la Figura 1. Un tema de investigación que estamos desarrollando (y que no discutimos aquí) es cómo agrupar los documentos en bloques. Por un lado, quisiéramos que documentos sintácticamente similares pertenezcan al mismo bloque, para comprimirlos mejor. Por otro, quisiéramos que los documentos que van a ser parte de la respuesta a una consulta también estén almacenados en el mismo bloque, para reducir la cantidad de bloques que se descomprimen y mejorar el tiempo de extracción. Respecto del procesamiento de la consulta en la Figura 1, estamos estudiando la manera de usar la representación run-length para mejorar algorítmicamente el proceso. La idea es considerar cada lista invertida como un conjunto de intervalos. Estos conjuntos deben intersectarse (consultas tipo AND) o unirse (consultas tipo OR). Algunos resultados preliminares indican que el tiempo de las consultas tipo OR puede reducirse a casi la mitad del tiempo original. Sin embargo, estamos hablando de consultas OR exhaustivas y sin ranking. Es necesario estudiar cómo agregar ranking sin afectar este resultado.

AGRADECIMIENTOS

Este trabajo es financiado por el Proyecto Fondecyt 11121556 de Iniciación en la Investigación. El grupo de investigación está conformado por Senén González (estudiante de Magíster, DCC, Universidad de Chile), Mauricio Oyarzún (estudiante de Doctorado, Universidad de Santiago de Chile) y Víctor Sepúlveda (Yahoo! Labs Santiago). BITS

Referencias

[1] http://en.wikipedia.org/wiki/Google_Search

[2] http://searchengineland.com/google-search-press-129925

[3] http://www.worldwidewebsize.com/

[4] http://mattmahoney.net/dc/text.html

[5] V. N. Anh y A. Moffat. Inverted index compression using word-aligned binary codes. Information Retrieval, 8(1):151–166, 2005.

[6] D. Arroyuelo, S. González, M. Marín, M. Oyarzún y T. Suel. To index or not to index: Time-space trade-offs in search engines with positional ranking functions. In Proc. ACM SIGIR, pp. 255-264. 2012.

[7] D. Arroyuelo, S. González, M. Oyarzún y V. Sepúlveda. Document Identifier Reassignment and Run-Length-Compressed Inverted Indexes for Improved Search Performance. En proc. ACM SIGIR, páginas 173-182, 2013.

[8] D. Arroyuelo y G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. ACM J. of Experimental Algorithmics, Volume 15, article 1.5, 2010.

[9] R. Baeza-Yates y B. Ribeiro-Neto. modern information retrieval: The concepts and technology behind search, Second edition. Pearson Education Ltd. 2011.

[10] R. Blanco y A. Barreiro. Document identifier reassignment through dimensionality reduction. In Proc. ECIR, pp. 375–387, 2005.

[11] D. Blandford y G. Blelloch. Index compression through document reordering. In Proc. DCC, pp. 342–351, 2002.

[12] P. Boldi y S. Vigna: The webgraph framework I: compression techniques. In Proc. WWW, pp. 595-602, 2004.

[13] S. Büttcher, C. Clarke y G. Cormack. Information Retrieval: Implementing and evaluating search engines. The MIT Press. 2010.

[14] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In Proc. WSDM, pp. 1, 2009.

[15] S. Ding, J. Attenberg y T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In Proc. WWW, pp. 311–320, 2010.

[16] P. Elias. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 21(2):194–203, 1975.

[17] A. Fariña, G. Navarro y J. Paramá. Boosting text compression with word-based statistical encoding. The Computer Journal, 55(1):111-131, 2012.

[18] P. Ferragina y G. Manzini. On compressing the textual web. In Proc. WSDM, pp. 391–400, 2010.

[19] S. Golomb. Run-length encoding. IEEE Transactions on Information Theory, 12(3):399–401, 1966.

[20] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar y S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In Proc. VLDB, pp. 13–23, 2004.

[21] S. Kreft y G. Navarro. On Compressing and indexing repetitive sequences. Theoretical Computer Science 483:115-133, 2013.

[22] C. Manning, P. Raghavan y H. Schütze. Introduction to information retrieval, Cambridge University Press. 2008.

[23] A. Moffat y L. Stuiver. Binary interpolative coding for effective index compression. Information Retrieval, 3(1):25–47, 2000.

[24] G. Navarro y V. Mäkinen: Compressed full-text indexes. ACM Computing Surveys, 39(1), 2007.

[25] W.-Y. Shieh, T.-F. Chen, J. Shann, y C.-P. Chung. Inverted file compression through document identifier reassignment. Information Processing and Management, 39(1):117–131, 2003.

[26] F. Silvestri. Sorting out the document identifier assignment problem. In Proc. ECIR, pp. 101–112, 2007.

[27] F. Silvestri, S. Orlando y R. Perego. Assigning identifiers to documents to enhance the clustering property of full-text indexes. In Proc. ACM SIGIR, pp. 305–312, 2004.

[28] A. Turpin, Y. Tsegay, D. Hawking, H. Williams: Fast generation of result snippets in web search. In Proc. SIGIR, pp. 127–134, 2007.

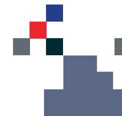
[29] H. Williams y J. Zobel. Compressing integers for fast file access. The Computer Journal, 42(3):193–201, 1999.

[30] I. Witten, A. Moffat y T. Bell. Managing gigabytes: Compressing and indexing documents and images, Second Edition. Morgan Kaufmann, 1999.

[31] H. Yan, S. Ding y T. Suel. Inverted index compression and query processing with optimized document ordering. In Proc. WWW, pp. 401–410, 2009.

[32] J. Ziv y A. Lempel: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3): 337-343, 1977.

[33] M. Zukowski, S. Héman, N. Nes y P. Boncz. Super-scalar RAM-CPU cache compression. In Proc. ICDE, pp. 59, 2006.



Algoritmos, Estructuras de Datos y mousse de chocolate

Jérémy Barbay
Profesor Asistente Departamento de Ciencias de la Computación, Universidad de Chile. Docteur en Informatique de l'université de Paris XI (2002).
Líneas de investigación: Diseño y Análisis de Algoritmos, Estructuras de Datos.
jbarbay@dcc.uchile.cl

PREPAREMOS UN MOUSSE DE CHOCOLATE

Una receta especifica el número de personas que se puede alimentar con ella (el output), la lista de ingredientes (la entrada) y la lista de instrucciones (el proceso en sí).

Una receta es al humano lo que los programas o algoritmos son para los computadores: se necesitan parámetros, al igual que el número n de porciones que se desea producir; incluye algunas operaciones condicionales *("IF (…) THEN (…)") y* operaciones lógicas; especifica procesos paralelos y secuenciales (e.g. "hacer en paralelo"); y se refiere a otras recetas, como por ejemplo la del "Baño María" (Figura 2).

Mousse de chocolate (para n porciones)

Implementos:
- Un recipiente para el "Baño María".
- Un recipiente, con tapa para la mezcla.
- IF el recipiente más pequeño para el "Baño María":
  - No tiene tapa o
  - No se puede poner en el refrigerador,
- THEN
  - Otro recipiente con tapa para poner la mezcla en el refrigerador.
  - Espátula.

Ingredientes:
- $n+1$ huevos,
- n cucharadas de azúcar,
- n* *25 gramos* de chocolate negro.

Preparación:
- Hacer en paralelo:
  - Poner el chocolate a derretir a "Baño María".
  - Preparar la clara de los huevos:
    - Separar la yema de la clara de los huevos.
    - IF la yema cae sobre la clara,
    - THEN sacar con la cáscara de huevo.
    - Batir las claras hasta que estén firmes.
- Cuando el chocolate esté líquido:
- Sacar el recipiente del agua hirviendo.
- Lentamente agregar el azúcar y la yema de los huevos, y revolver enérgicamente.
- Agregar las claras batidas, incorporando rápidamente.
  - Comenzar con una cuchara.
  - Mezclar rápidamente hasta que la preparación esté uniforme.
  - Doblar la cantidad cada vez que se agregue más.
- Dejar enfriar al menos dos horas en el refrigerador.
- Servir acompañado de una fruta ácida o galletas.

Figura 1

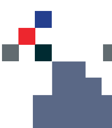
Baño María

- Derretir x a Baño María.
  - Implementos:
    - Dos recipientes redondos para poner uno dentro de otro, con una diferencia de diámetro de 4 centímetros.
    - Una espátula o una cuchara de madera.
- **Ingredientes:**
  - x para derretir
  - Agua
- **Preparación:**
  - Llenar el recipiente más grande hasta 1/3 de agua, de tal forma que el agua y el recipiente más pequeño (vacío) completen el recipiente más grande.
  - Hervir el agua.
  - Poner x en el recipiente más pequeño y luego poner en el agua hirviendo.
  - Disminuya la llama para que el agua siga hirviendo, sin que se escape del recipiente.
  - Revuelva x con frecuencia en el recipiente más grande.
  - Sacar cuando esté suficientemente líquido.

Figura 2

¿CUÁNTO TIEMPO DEMORA PREPARAR UN MOUSSE DE CHOCOLATE?

Esta pregunta es sobre el producto, no sobre la receta. Dado que el mousse de chocolate se sirve frío y que la receta menciona dos horas de tiempo de espera en el refrigerador, probablemente podemos deducir que cualquier receta de mousse



de chocolate en la que se use solamente el refrigerador para enfriar el producto, requiere de dos horas de preparación, pero eso sólo da una *cota inferior en el tiempo* requerido, y no es una información tan útil para alguien que desee saber con cuánto tiempo de anticipación debe empezar a cocinar antes de comer.

De manera similar, en construcción algorítmica se distingue entre la complejidad de un algoritmo particular (e.g. la receta de mousse de chocolate de la Figura 1) y la complejidad del problema en sí mismo (e.g. hacer un mousse de chocolate con cualquier otra receta). La evaluación de la segunda, por lo general, depende de los supuestos sobre los recursos disponibles (e.g. en nuestro ejemplo suponiendo que el mousse se enfría mediante un refrigerador, en lugar de una herramienta más sofisticada).

¿CUÁNTO TIEMPO TOMA SEGUIR ESTA RECETA?

La interrogación es un poco vaga: ¿es preguntar (1) cuánto tiempo antes de la cena debemos empezar a cocinar, o (2) cuánto tiempo vamos a estar activamente siguiendo la receta, sin poder enfocarnos seriamente en otra cosa? ¡Ninguna pregunta es trivial de contestar! Además, el tiempo necesario para ejecutar una receta depende de muchos factores (e.g. podríamos estar derramando una gran cantidad de yema en la clara de los huevos, y toma bastante tiempo recuperarla por cada huevo). En la cocina, por lo general estamos contentos con una aproximación del tiempo necesario, con arreglo a suposiciones generales: suponiendo que la cantidad es razonable (*$4 \leq n \leq 8$ porciones)*; fundir el chocolate será el tiempo dominante en la parte paralela de la receta, tomando aproximadamente 15 minutos; la mezcla tardará aproximadamente 15 minutos más, mientras que dejar que se enfríe en el refrigerador tomará, como se especifica, 2 horas. Esto entrega respuesta a las dos interpretaciones de la pregunta: deberíamos empezar a cocinar 2,5 horas antes de la cena, y estaremos ocupados por 30 minutos.

Del mismo modo, el análisis de las construcciones algorítmicas requiere definir formalmente la propiedad que sea analizada, tal como la cantidad de *tiempo necesario para preprocesar los datos*, versus la cantidad de *tiempo* esperada por un usuario, *antes de recibir una respuesta a su consulta*. Cuando se trata de construcciones algorítmicas, también hacemos uso de aproximaciones al calcular el tiempo de ejecución (en particular, al *identificar el tiempo dominante de un proceso*); "supuestos generales" (lo cual puede ser problemático: en particular, es mucho más arriesgado suponer que n será pequeño, ya que la gente está mucho más dispuesta a que el computador trabaje, a que ellos mismos lo hagan); y diversas medidas del costo de la ejecución (e.g. el tiempo activo y el tiempo de espera de la receta). El contexto general de la investigación en análisis de algoritmos es *identificar formalmente los problemas* que los computadores debieran resolver; *definir nuevos algoritmos* para la solución de esos problemas; y *analizar los algoritmos* de una manera que *ayude a decidir* qué algoritmos se deben utilizar en cada caso particular.

¿CUÁNTAS HERRAMIENTAS SE NECESITAN?

Si va a compartir la cocina con otras personas, o si preparará varias recetas al mismo tiempo, o si su cocina está mal equipada; debe comprobar con cuidado que tiene todo el equipo necesario. Para hacer su tarea más fácil, la receta debe incluir el equipo que se necesita para ejecutarla.

Del mismo modo, las construcciones algorítmicas tienen *requisitos de espacio*, por una razón muy parecida: el usuario podría compartir la máquina con otros, o ésta podría estar ejecutando varios programas al mismo tiempo, o ejecutarse en un dispositivo móvil con recursos limitados.

¿CUÁNTA ENERGÍA SE NECESITA?

Necesitamos energía para derretir el chocolate (quemador de gas), para batir la clara de los huevos (batidora eléctrica), para mezclar todo (con la mano) y para enfriar el producto (refrigerador). Muy a menudo el tiempo es el recurso crítico, pero en algún caso especial la energía también podría serlo (por ejemplo, si está en un pueblo de África Central, donde no hay refrigerador ni electricidad).

Del mismo modo, los recursos energéticos de las construcciones algorítmicas son críticos cuando se destinan a *equipos integrados o nanorobots*, donde la energía es un recurso tan crítico como el tiempo; y los teraservidores masivamente paralelos de Google o Amazon,

en los que consumo de energía es un recurso aún más crítico.

¿SE PUEDE PREPARAR EN EL FONDO DEL MAR? ¿O EN LA LUNA?

La temperatura de ebullición del agua se ve afectada por la presión: en un entorno en el que hierve a una temperatura más baja, no se derrite el chocolate. Batir la clara de los huevos es una acción mecánica que es afectada por la gravedad; en un entorno con menos gravedad los huevos batidos serán más livianos y el resultado de la receta distinta.

Del mismo modo, el rendimiento de las construcciones algorítmicas se ve afectado por condiciones externas. *Computadores enviados al espacio* tienen mayores tasas de error debido al aumento de las radiaciones en comparación con la superficie de la tierra, lo que se corrige a través de la autoestabilización de algoritmos. Más cerca de la tierra, en *equipos con varios tipos diferentes de memoria* (e.g. una barata, grande y lenta; una rápida y pequeña), los algoritmos están diseñados para optimizar el uso de aquellas.

¿SE PUEDE USAR LA RECETA PARA ALIMENTAR A UN MILLÓN DE PERSONAS?

La receta se expresa en función del número n de las porciones, que generalmente se supone es más bien pequeño, alias cuatro a ocho, ¡aunque sólo sea por el tamaño limitado de los recipientes en los cuales se preparan los ingredientes y el producto! Para preparar más porciones de mousse de chocolate, uno puede utilizar cuencos más grandes, repetir la receta varias veces, ejecutar la misma receta varias veces en paralelo, o una combinación de estas soluciones (e.g. usar recipientes tan grandes como razonable sea y repetir la receta). En particular, si se tarda t horas y e energía en preparar n porciones, la preparación de $2n$ porciones puede ser optimizada para tardar menos de $2t$ horas y gastar menos de $2e$ de energía (e.g. no lavar los recipientes entre dos recetas ahorra tiempo).

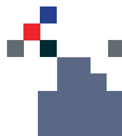
Del mismo modo, el comportamiento de un algoritmo en entradas de tamaño creciente es objeto de muchos estudios. En algunos casos, el tiempo de ejecución de un algoritmo aumenta más lentamente que el tamaño de su entrada (e.g. *buscar en un arreglo ordenado* de tamaño n=128 a través de comparaciones requiere $1+log_2 n=8$ comparaciones, mientras que *buscar en una matriz ordenada* de tamaño $n=2*128=256$ toma $1+log_2 n=9<2*8$ comparaciones), y, a veces, crece más rápido (e.g. *clasificar n=128 elementos* a través de comparaciones requiere $n\ log_2 n=128*7$ comparaciones, mientras que clasificar $n=128*2=256$ elementos a través de comparaciones requieren $log_2 n=256*8>2*128*7$.

¿PUEDES HACERME UNA RECETA DETALLADA CON FOTOS?

Eso no tomará mucho más tiempo que una simple ejecución de la receta: sólo mantener la cámara cerca y publicar las fotos en orden.

De manera similar, los algoritmos diseñados para producir una salida particular (e.g. ordenar un arreglo) implícitamente describen un objeto a través de su proceso (e.g. codificar una permutación), y es poco el trabajo adicional requerido para almacenar ese objeto (e.g. anotando el resultado de cada comparación realizada por el algoritmo de ordenamiento). En algunos casos particulares (e.g. *merge sort*), esta codificación ha resultado ser útil (e.g. estructuras de datos comprimidas para permutaciones, más pequeñas y más rápidas que las previamente conocidas). BITS

Mousse de chocolate casero.



Demostraciones de Nula Divulgación: o cómo convencer a alguien que mi sudoku tiene solución, sin revelarla

Alejandro Hevia
Profesor Asistente Departamento de Ciencias de la Computación, Universidad de Chile. Ph.D. Computer Science, University of California, San Diego (2006); Ingeniero Civil en Computación, Universidad de Chile (1998). Director del Grupo de Respuesta a Incidentes de Seguridad Computacional, CLCERT. Líneas de Investigación: Criptografía Aplicada, Seguridad Computacional.
ahevia@dcc.uchile.cl

En este artículo presentaremos un concepto aparentemente paradójico pero extremadamente poderoso surgido del corazón de la computación teórica: el de las demostraciones de Nula Divulgación. Mostraremos cómo aplicarlo a los juegos de sudoku, lo que analizaremos está lejos de ser un mero juego. Veremos además cómo la Nula Divulgación permite el surgimiento de un amplio espectro de aplicaciones modernas para preservar la privacidad de las personas, en particular, aplicaciones que demuestran convincentemente la posesión de un dato secreto sin revelarlo.

LA HISTORIA

Como buenos fanáticos del sudoku (ver Figura 1), Alicia y Roberto rutinariamente intercambian puzles de sudoku para resolver. En una oportunidad, Alicia encuentra un puzle sudoku fascinante, el cual decide compartir con Roberto y pedirle que lo resuelva. Roberto, quien no quiere perder el tiempo, sospecha que el puzle no tiene solución por lo que desafía a Alicia a demostrarle que sí la tiene. Claramente, Alicia podría darle la respuesta a Roberto, quien la comprobaría y listo. Funciona, pero ¡le quitaría toda la diversión a resolver el puzle! ¿Cómo podemos hacerlo mejor? ¿Cómo puede Alicia convencer a Roberto que el puzle tiene solución sin revelársela? Para resolver este problema, debemos primero discutir el concepto de demostración[1].

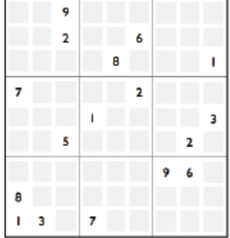
TEOREMAS Y DEMOSTRACIONES

¿Qué exactamente le pide Roberto a Alicia al solicitarle una demostración? Una demostración matemática es típicamente una lista de pasos lógicos, algo que Alicia puede escribir y enviar a Roberto. Tal experiencia está tan arraigada en la comunidad matemática que rara vez nos preguntamos qué hay detrás. Al examinarlo en su manifestación más simple, la demostración de un teorema en matemáticas en realidad está formada por dos componentes: un teorema o aseveración lógica a demostrar (digamos x) y la demostración misma (un texto, digamos w). En nuestro caso, llamemos Z al conjunto de todos los puzles sudoku que tienen solución. Dado un puzle x, si Alicia puede dar la demostración w = "la solución del puzle por filas es 5, 8, ..., 2", entonces Roberto puede convencerse que $x \in Z$ simplemente ejecutando un procedimiento que chequee que la solución w es válida, un proceso razonablemente rápido.

Así, no es difícil convencerse de que una demostración matemática cualquiera, tal como la hemos descrito, puede verse como un proceso: dado un teorema x, el demostrador (Alicia) le envía un sólo mensaje –la demostración w– al verificador (Roberto), quien lo revisa en forma rápida e independiente, para emitir una decisión: la demostración es válida o no. En Teoría de la Computación, el conjunto de todos los problemas para los cuales pueden darse demostraciones cortas, eficientemente chequeables como la anterior, se denomina NP.

Paréntesis: P versus NP

Es interesante notar que la clase NP de problemas recién descrita coincide con una clase importantísima de problemas usados en la práctica: aquellos donde verificar una solución del problema puede hacerse en forma eficiente, aún si no sabemos cómo encontrar una solución. Estos problemas aparecen por todos lados en ingeniería, especialmente al resolver problemas de optimización. Por ejemplo, el problema del Camino Hamiltoneano: dado x = "Un grafo G de n nodos", determinar si existe en G un camino que pase por todos los nodos exactamente una vez[2]. Claramente, si nos



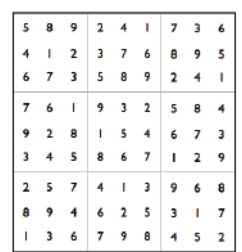
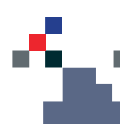


Figura 1 • El Sudoku es un tipo de puzle que consiste en completar una matriz de 9 filas y 9 columnas con números del 1 al 9. El juego parte con la matriz sólo parcialmente llena (unas 18 a 20 celdas, como se ve en el tablero de la izquierda) y el problema consiste en completar las celdas restantes con valores tales que toda fila y toda columna contenga números distintos, así como cada una de las 9 submatrices disjuntas de tamaño 3x3 que particionan la matriz (como se ve en el tablero de la derecha).

[1] Este artículo está basado en una presentación de Mike Rosulek [6].

[2] Recordar que un grafo es simplemente un conjunto de nodos unidos por aristas. Un ejemplo clásico de grafo es un mapa de las ciudades y carreteras de un país: las ciudades son los nodos y las carreteras conectando las ciudades son las aristas. En dicho caso, un Camino Hamiltoneano es una ruta para visitar todas las ciudades pasando una y sólo una vez por cada ciudad.



dan como "pista" un camino (llamémosle w), podemos contestar rápidamente simplemente verificando si el camino dado pasa por cada nodo una y sólo una vez. Sin embargo, nadie sabe en la actualidad cómo encontrar eficientemente un camino para un grafo arbitrario. Como comparación, existe la clase de problemas (llamada *P*) que contiene a todos aquellos que pueden resolverse eficientemente sin necesidad de "pistas": por ejemplo, determinar si hay un camino entre dos nodos cualesquiera en un grafo puede hacerse rápidamente (con un algoritmo del tipo greedy, ver [1]). ¿Cómo se comparan P y NP? ¿Encontrar una solución es tan fácil como verificar si está correcta? Pareciera que no. Sin embargo, en estricto rigor, ¡no lo sabemos! Esta pregunta, la relación **P versus NP**, es probablemente el problema abierto más famoso de la Teoría de la Computación. De hecho, actualmente hay un premio, del Clay Mathematics Institute, de 1 millón de dólares para el primero que lo resuelva.

Existe una subclase de problema dentro de los problemas en NP denominados problemas NP-Completos. Estos problemas representan en algún sentido los problemas más difíciles de resolver en NP. Un ejemplo de éstos es justamente el problema del Camino Hamiltoneano. Los problemas NP-Completos tienen la propiedad que, si uno puede encontrar la solución de uno de ellos eficientemente, entonces podemos encontrar la solución de todos los problemas en NP eficientemente. Tal resultado, denominado el Teorema de Cook-Levin, es teóricamente impresionante, pues caracteriza los problemas más difíciles en la clase. Sin embargo, también es útil en la práctica. En la demostración de dicho teorema se entrega una manera explícita (la cual llamamos t) de transformar una instancia x de un problema L cualquiera en NP en una instancia $x'=t(x)$ del problema NP-Completo. Usando esta *función* podemos, por ejemplo, transformar el problema de demostrar que un texto cifrado C dado (calculado usando un sistema de clave pública) contiene un mensaje M igual a 0 ó 1, a demostrar que un cierto grafo $G = t(G)$ contiene un Camino Hamiltoneano. Esto es, si podemos demostrar que G tiene un camino, entonces estaremos demostrando que C codifica un mensaje igual a 0 ó 1. Quizás suene muy teórico, pero poder traducir la solución de un problema cualquiera (en NP) a encontrar la solución de un problema NP-Completo es extremadamente útil, como veremos al final.

DEMOSTRACIONES INTERACTIVAS: GENERALIZANDO LAS DEMOSTRACIONES MATEMÁTICAS

Volvamos al problema de Alicia. A priori, no se ve claro cómo podría ella demostrar que el sudoku tiene solución, sin revelar dicha solución; menos aún si Roberto debe enviarle un sólo mensaje. Aquí llega la interacción a nuestro auxilio.

Todo aquel que haya tenido que entender una demostración matemática se ha dado cuenta que no siempre es fácil convencerse de una demostración simplemente leyéndola. Mucho mejor es poder hacer preguntas a quien nos presenta la demostración, lo cual típicamente facilita comprenderla. Por lo mismo, ¿no sería "más fácil" si ahora le permitimos a Alicia y Roberto conversar? Esto es, ¿ayudaría si le permitimos a Alicia demostrar interactivamente a Roberto que su teorema es cierto? En esta conversación, Alicia y Roberto intercambian mensajes hasta que finalmente Roberto se detiene y decide si "le cree" a Alicia o no. Este proceso se denomina una demostración interactiva y veremos que efectivamente ayuda. En nuestro ejemplo del sudoku, Alicia y Roberto podrían usar la estrategia que llamaremos **Chequear-Fila-Luego-Columna:** Roberto podría pedirle a Alicia que le muestre una cierta columna específica (escogida por él) de la solución del sudoku, verificando que sólo contenga números distintos. Luego, Roberto podría repetir este proceso para otra fila y, si Alicia responde exitosamente a ambas preguntas, Roberto decide creerle. ¿Piensa usted que Roberto debiera estar convencido?

Para responder, necesitamos una definición un poco más precisa de lo que es una demostración interactiva. Un protocolo o sistema de demostración interactiva para un problema L es un par de algoritmos, uno para el demostrador (P por *Prover* en inglés) y otro para el verificador (V), los cuales para poder capturar la idea de una demostración, deben satisfacer dos propiedades: correctitud y robustez. Un protocolo es correcto si para toda aseveración x verdadera (esto es, $x \in Z$), P siempre o con alta probabilidad convence a V. Un protocolo es robusto si ningún demostrador, aún uno malicioso que ejecuta un programa distinto a P, puede convencer a un V honesto que una cierta aseveración falsa, $x \notin L$, excepto con una probabilidad minúscula.

Claramente, el protocolo **Mostrar-La-Solución** descrito al comienzo, donde Alicia simplemente revela la solución a Roberto, satisface las dos condiciones: una Alicia (en el rol de P), honesta siempre, puede convencer a un Roberto (en el rol de V) honesto si es que el puzle sudoku exhibido tiene una solución; por otro lado, si el puzle no tiene solución, Alicia no tiene ninguna posibilidad de exhibir una solución que convenza a Roberto.

Por otro lado, el protocolo **Chequear-Fila-Luego-Columna** anterior no satisface la definición pues una Alicia (en el rol de P) que no siga las reglas puede engañar a Roberto (en el rol de V). Esto porque, aun si el sudoku no tiene solución, en general no es demasiado difícil completar una fila arbitraria y luego otra columna arbitraria para que tengan cada una sólo números distintos. Más adelante mostraremos no sólo cómo corregir este problema, sino cómo evitar que Alicia tenga que mostrar parte o toda la solución a Roberto.

TIRANDO MONEDAS

Un aspecto fundamental de las demostraciones interactivas es su utilización de la aleatoriedad, lo cual es mejor graficado a través de una historia. El famoso matemá-

tico Ronald Fisher [2] contaba la historia de Muriel, una dama de sociedad, quien le aseguraba que el orden en la combinación de los ingredientes en la preparación de un té con leche afectaba su sabor. Según Muriel, un té vertido sobre leche sabe distinto a leche vertida sobre té. Más aún, Muriel le aseguraba poder distinguirlos. Intrigado, Ronald prepara el siguiente experimento para determinar si creerle a Muriel o no.

1) Primero, en privado y lejos de la vista de Muriel, el matemático tira una moneda. Si es cara pone primero té y luego leche en una taza. Si es sello, lo hace al revés.
2) Luego, le entrega la taza a Muriel, quien la prueba e intenta adivinar.
3) Ronald y Muriel repiten el proceso anterior n veces.
4) Si Muriel acierta todas las n veces, Ronald acepta y cree que Muriel tiene la habilidad señalada.

Claramente, si no hay diferencia en los distintos tipos de té con leche, Muriel debiera adivinar una iteración con probabilidad 1/2. Por otro lado, si Muriel puede adivinar debiera responder correctamente siempre (o casi siempre), lo cual nos dice que el protocolo es correcto.

¿Por qué, de acertar Muriel todas las veces esta demostración, debiera convencer a Ronald? Es interesante notar que la decisión de poner primero leche o té en la taza es aleatoria e independiente de Muriel, lo cual es fundamental para argumentar la robustez del protocolo: la probabilidad de Muriel de adivinar las n veces sin tener la habilidad es minúscula (esto es, a lo más 2^{-n}). Luego, el protocolo es efectivamente una demostración interactiva. Notemos un aspecto interesante aquí: luego de ser convencido, Ronald no puede convencer a otros *ni transmitir su convencimiento*. Todo lo que tiene Ronald ahora son sus notas (por ejemplo, "en la primera iteración vertí leche y luego té, Muriel adivinó. En la segunda, vertí té y luego leche, Muriel adivinó..." etc.), las cuales no convencen a nadie pues pudieran haber sido falsificadas. Esto es algo contraintuitivo; convencer a alguien no es lo mismo que revelar información "útil", un punto que desarrollamos en detalle más adelante.

Paréntesis: ¿cuán grande es la clase de los problemas demostrables interactivamente? Sorprendentemente, el simple cambio de agregar interactividad a una demostración tiene un efecto inmenso en el tipo de problemas que podemos demostrar. Es posible mostrar que *IP*, la clase de todos los problemas que admiten demostraciones interactivas donde el verificador (Alicia) es eficiente, contiene no sólo a la clase NP sino a problemas mucho más difíciles de resolver. Por ejemplo, contiene problemas que requieren tiempo exponencial. En la práctica esto significa que, por ejemplo, hay problemas que no sabemos cómo verificar una solución eficientemente, pero para los cuales ¡*sí* podemos demostrar que existe una solución! Un ejemplo de ello es el problema de demostrar que dos grafos del mismo número de nodos NO son isomorfos, esto es, demostrar que uno de ellos no es simplemente una permutación de los nodos del otro. No es ni siquiera claro qué *solución o pista* uno pudiera recibir para poder resolver el problema. De hecho, se desconoce si este problema está en NP, se conjetura que no.

NULA DIVULGACIÓN

Al demostrar una aseveración, una persona puede o no estar revelando información. En el caso de los puzles de sudoku, Alicia está revelando parte o toda la solución a Roberto, lo cual a ambos les gustaría evitar. Más generalmente, existen muchos escenarios donde queremos poder demostrar algo, un teorema (una propiedad de un cierto objeto) sin revelar el *cómo* podemos demostrarlo. Por ejemplo, al conectarse remotamente a un servidor, un usuario debe autentificarse remotamente, usualmente vía *demostrar* que conoce su contraseña, ¿y cómo lo hace? Simplemente la revela al servidor. Lamentablemente, esto puede causar que la contraseña se filtre y afecte la seguridad del servidor.

¿Puede un usuario demostrar que conoce una contraseña sin revelarla? ¿Puede Alicia demostrar que su puzle tiene solución sin revelarla?

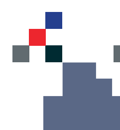
Información y aprender

Lo que buscamos es un protocolo de demostración interactiva que permita demostrar una aseveración sin revelar "demasiada información", de hecho, sin "revelar ninguna otra información más allá de que la propiedad demostrada se tiene". Pero, ¿qué significa revelar información? Una manera de responder esto consiste en preguntarse qué puede hacer un verificador (por ejemplo Roberto), después de interactuar con el demostrador (por ejemplo Alicia) en la demostración de una aseveración x. Lo que sea que Roberto ha aprendido sobre x puede verse como una función f sobre x, la cual Roberto no podía calcular antes de la demostración, pero sí puede después de ella (ver Figura 2). Claramente, la única diferencia entre antes y después son los mensajes que recibió de Alicia durante la demostración. En el protocolo **Mostrar-la-Solución** el verificador Roberto puede calcular la función $f(x)$= "la solución del puzle x" puesto que de hecho recibe tal información de Alicia durante la demostración.

Si algo es calculable por mí solo, es que ya lo sabía

Ahora bien, si Roberto puede generar por sí solo una transcripción de todos los mensajes que recibiría en una demostración con Alicia *pero sin interactuar con ella*, entonces podemos decir que Roberto no ha aprendido nada. ¿Por qué? Si Roberto puede calcular por sí

Figura 2 • Una interacción entre Alicia y Roberto donde éste último aprende algo.



solo dichos mensajes, entonces tiene todo lo necesario para calcular $f(x)$, pero ahora sin haber interactuado con Alicia. Luego, la interacción con Alicia no aporta nada extra para calcular $f(x)$, lo cual significa que Roberto no aprende nada nuevo. Conceptualmente, un protocolo de demostración interactiva (esto es, un par de algoritmos para el demostrador P y para el verificador V) es de *Nula Divulgación* (o *Zero-Knowledge* en Inglés) si existe una manera, un algoritmo, S, para producir una transcripción de los mensajes intercambiados entre P y V en una conversación, pero donde:

- El algoritmo S nunca conversa con el demostrador P, sólo con V, y
- La transcripción "luce igual", esto es, *indistinguible*, de una transcripción real de la conversación entre el verificador V y el demostrador P.

Como *bonus*, notemos que un protocolo de Nula Divulgación no permite al verificador V convencer a un tercero de lo demostrado por el demostrador P, puesto que la transcripción de la conversación misma con P –el único registro de la interacción con P– puede ser falsificada por el mismo V y, por ende, no puede significar evidencia alguna para un tercero (ver Figura 3).

Un primer ejemplo

Un ejemplo simple de protocolo de Nula Divulgación es el protocolo descrito anteriormente, diseñado por Ronald Fisher para permitir a Muriel demostrar su supuesta habilidad para diferenciar los tipos de té. La conversación entre Ronald y Muriel está contenida en el registro del matemático donde indica, para cada iteración, la moneda tirada (cara o sello), y si Muriel adivinó o no. Claramente, dicha tabla puede ser generada sin nunca interactuar con Muriel. No sólo Ronald no aprende nada, sino que un tercero no puede ser convencido de las supuestas habilidades de Muriel si la única evidencia de Ronald al respecto es sólo una tabla que cualquiera podría generar por sí solo. Ahora bien, este protocolo es algo artificial pues el demostrador necesita una habilidad algo extraña, la cual no sabemos si existe. Necesitamos un mejor ejemplo, y qué mejor que un protocolo para demostrar que un sudoku tiene solución. Este protocolo necesita el equivalente a una caja fuerte pero en el mundo digital.

Cajas fuertes digitales

En el mundo físico, una manera de enviar un número sin revelarlo es meter un papel con el número escrito en una caja fuerte, la cual es cerrada y luego enviada por encomienda, sin adjuntar la llave. En el mundo digital, su análogo se denomina un compromiso o *commitment* (en inglés). Podemos crear y abrir commitments. Dado un valor v, para crear un commitment c con v adentro, calculamos $c = commitment(v\,;\,r)$ donde r es un valor aleatorio. Luego, el commitment c se envía al receptor. Con ello, quien envía sabe el valor v y conoce la llave r, mientras que el receptor sólo ve la caja "por fuera", esto es c, lo cual no revela v. Sólo quien envía puede abrir un commitment, esto es, convencer a alguien que el valor almacenado en c es v. En el mundo físico, esto se haría simplemente enviando al receptor la llave de la caja fuerte por correo. En el mundo digital, el valor r juega el rol de la clave y es lo que se envía. Los commitments son diseñados con dos propiedades análogas a su contraparte física: (1) el receptor de un commitment c no puede saber qué valor "contiene" el commitment, y (2) una vez recibido el commitment, quien lo envió no puede convencer al receptor que el valor almacenado es distinto a v, esto es, no puede abrirlo a un valor distinto al almacenado originalmente en él. Claramente el receptor de la caja fuerte al no poder abrir la caja no sabe qué valor contiene, y quien la envío no puede ya cambiar el valor almacenado dentro de la caja fuerte una vez que dicha caja está en posesión del receptor. Un esquema de encriptación de clave pública puede usarse como commitment si quien envía el commitment es quien genera el par de claves pública y privada, siempre que sea posible para el receptor verificar que la clave pública fue generada correctamente. Para más detalles y ejemplos, ver [5].

DEMOSTRACIÓN DE NULA DIVULGACIÓN PARA SUDOKU

Mostraremos ahora cómo Alicia usando commitments puede demostrar a Roberto que su sudoku tiene solución sin revelarla[3]. En el siguiente protocolo, Alicia actúa como el demostrador (P) y Roberto como el verificador (V). Ambos conocen el puzle sudoku sin resolver (llamémosle x):

1.- Alicia: escoge una permutación secreta al azar π de{1,...,9} (por ejemplo, π (1)=, π (2)=3, etc.), la cual aplica a cada uno de los valores en las celdas de la solución.

Figura 3 • En Nula Divulgación, cualquiera sea lo que Roberto puede calcular después de ver la transcripción de la conversación con Alicia, lo puede calcular sin conversar con Alicia.

[3] Este protocolo fue propuesto originalmente en [3], donde se puede ver más detalles y un análisis más formal.

Ello produce una matriz del mismo tamaño, llamémosla x', con todos los valores re-etiquetados: donde había un 1, ahora hay un 5, donde había un 2 hay ahora un 3, etc. (ver Figura 4a). Alicia entonces crea commitments para cada una de las celdas de la nueva tabla x', formando una tabla de 9x9 con 81 commitments, la cual llamamos *C*. Finalmente, Alicia envía *C* a Roberto.

2.- Roberto: desafía a Alicia, escogiendo al azar una y sólo una de las siguientes opciones:

(a) Revelar fila: en esta opción, Roberto escoge una fila al azar y le pide a Alicia que le revele los valores (abra los commitments) para toda esa fila.
(b) Revelar columna: en esta opción, Roberto escoge una columna y le pide a Alicia que le revele los valores para toda esa fila.
(c) Revelar submatriz: en esta opción, Roberto escoge al azar una de las 9 submatrices de 3x3 del tablero, y le pide a Alicia revelarla.
(d) Revelar posiciones iniciales: en esta opción, Roberto pide a Alicia revelar los valores correspondientes a todas las posiciones "llenas" en el sudoku original *x*.

3.- Alicia: revela lo solicitado por Roberto en el paso anterior.

4.- Roberto: revisa que los valores revelados por Alicia sean consistentes con la solución de un sudoku: si es una fila, columna o submatriz, todos sus valores deben ser distintos, del 1 al 9. Si Alicia debe revelar los valores para las posiciones "llenas" del sudoku original, entonces Roberto debe verificar que los valores revelados sean consistentes con una *permutación* de los valores reales del sudoku inicial *x*. Por ejemplo, si donde había un 1 en *x* ahora hay un 5, Roberto debe chequear que en toda otra celda donde había un 1 ahora hay un 5. También debe chequear que los valores revelados para todas las celdas que tenían valores distintos en *x* siguen siendo distintos (ver Figura 4b).

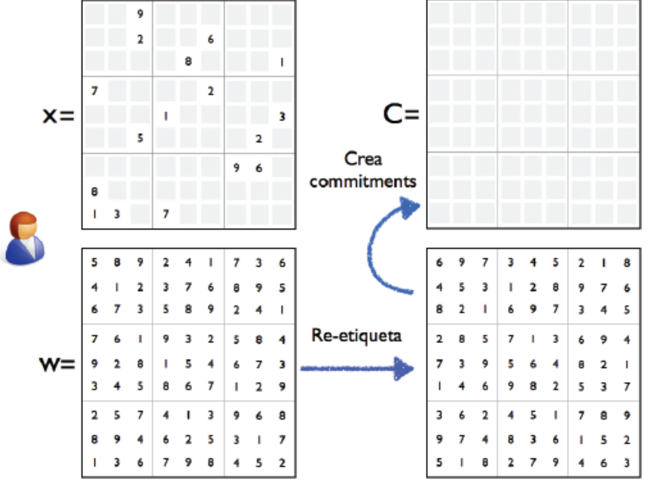


Figura 4a • Protocolo de Nula Divulgación para sudoku, paso 1.

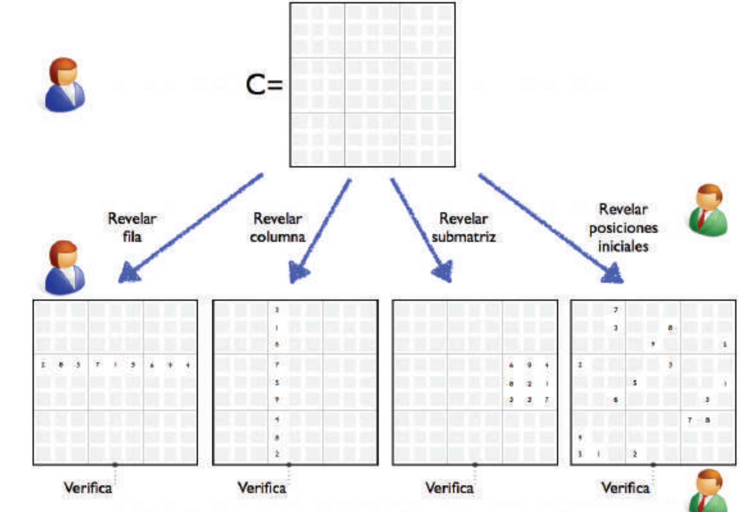
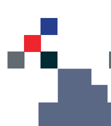


Figura 4b • Protocolo de Nula Divulgación para sudoku, pasos 2, 3 y 4.

5.- Roberto y Alicia: repiten todos los pasos anteriores *n* veces.

Primero notemos que hay un total de 28 posibles preguntas que Roberto puede hacer en el paso 2. Esto porque puede preguntar una fila específica (opción (a), 9 posibilidades, una por fila) o columna específica (opción (b), 9 posibilidades) o submatriz específica (9 posibilidades) o simplemente preguntar por los valores en las posiciones originales (1 posibilidad).

Si Alicia conoce una solución, puede almacenarla en forma permutada en los commitments, y siempre contestar exitosamente cualquier pregunta de Roberto en el paso 3. Así, luego de iterar *n* veces, Roberto queda convencido. Pero, ¿y si Alicia está engañando a Roberto y simplemente no existe una solución para el sudoku? Es fácil ver que ninguna selección de valores para los commitments puede lograr convencer ante todas y cada una de las 28 posibles preguntas de Roberto. Supongamos Alicia intenta demostrar a Roberto que un *x* sin solución realmente la tiene. Entonces, lo mejor que puede hacer es intentar adivinar la pregunta que Roberto le hará en el paso 2 y preparar commitments con valores que "funcionen" para dicha opción (por ejemplo si apuesta que Roberto escogerá



la opción (b), esto es preguntará por los valores de una columna específica, Alicia podría poner valores distintos en todas las columnas). Sin embargo, claramente esta estrategia no puede funcionar para una o más de las otras opciones y Alicia será descubierta (no podrá responder) si le preguntan por ellas. Una manera de verlo es la siguiente: si no hay solución, no importa qué valores Alicia ponga en los commitments, hay al menos una de las 28 preguntas específicas que requiere revelar valores de los commitments que son inconsistentes con una solución de sudoku válida. Con probabilidad al menos 1/28, Roberto escogerá dicha pregunta exponiendo a Alicia como mentirosa. Esto implica que, en principio, con probabilidad al menos 27/28 (aproximadamente 96.4%) Alicia podría contestar a Roberto correctamente en el paso 3. Pero, ¿no es eso es muy alto? Sí, pero es sólo para una iteración. Basta recordar que los pasos 1 a 3 se realizan varias veces, n veces, en forma independiente, usando permutaciones y commitments nuevos cada vez. Así, la probabilidad que Alicia engañe a Roberto en todas las n iteraciones es a lo más $(27/28)^n \approx (1/2)^{0{,}05n}$. Un cálculo simple nos muestra que si $n = 2500$ esta probabilidad se convierte en aproximadamente $2^{-125} \approx 1/10^{38}$, esto es, 0,...(37 ceros)...2. ¡Esta probabilidad es bajísima! De hecho, es menor que la probabilidad de ganarse cinco veces seguidas el premio mayor del Loto en Chile[4]. Si Alicia puede hacer esto, ¡sería millonaria y poco incentivo tendría para engañar a Roberto!

Generalizaciones

Recientemente [7] fue demostrado que el problema de decidir si una instancia de sudoku tiene solución es NP-Completo. En teoría, esto nos permite demostrar en Nula Divulgación que cualquier problema L en NP tiene solución, en dos pasos: (1) transformamos L en una instancia de sudoku, y (2) demostramos en Nula Divulgación que existe una solución para este sudoku usando el protocolo anterior. En otras palabras, para demostrar que un problema cualquiera (en NP) tiene solución sin revelarla, nos basta transformarlo matemáticamente en una instancia de sudoku, y luego jugar el rol de Alicia más arriba. ¡Quién dijo que la teoría era aburrida!

DEMOSTRACIÓN DE CONOCIMIENTO

Usando un protocolo de Nula Divulgación, Alicia puede entonces demostrarle a Roberto que el sudoku en cuestión tiene solución. ¿Puede entonces Roberto concluir que Alicia sabe la solución? No en general. Conceptualmente, una demostración de Nula Divulgación permite demostrar que una solución existe, pero nada más. Es posible que el demostrador pudiera realizarla sin necesariamente conocer una solución[5]. Sin embargo, hay muchas situaciones en las cuales esto no es suficiente, lo que importa es si el demostrador conoce una *solución* al problema. Por ejemplo, cuando un servidor desea autentificar a una persona, lo importante es verificar que ésta conoce la contraseña almacenada en el servidor. Al servidor le interesa verificar si el usuario la sabe, no si existe (pues siempre existe). En nuestro caso, quizás Roberto sólo desea resolver sudokus que Alicia ha resuelto, por lo que le gustaría tener certeza que ella sabe la solución. Efectivamente, el protocolo de la sección anterior sí garantiza que Alicia sabe la solución, lo cual demostraremos a continuación. Primero necesitamos preguntarnos qué significa “saber”.

¿Cómo podemos definir matemáticamente que Alicia “conoce” o “sabe” la solución? Pese a que hemos llamado Alicia al demostrador, típicamente quienes realizan las demostraciones son programas, no personas. ¿Qué significa entonces que un programa *conozca* un valor o que una máquina *sabe* algo?

Interrogando al demostrador

La respuesta a esta interrogante es simple pues evita contestar la pregunta respecto a qué es conocer algo. En vez de eso, decimos que una demostración interactiva garantiza que el demostrador “sabe” una solución w para un problema x dado, si podemos “extraer” la solución del demostrador, simplemente haciéndole las preguntas adecuadas. En otras palabras, si un demostrador P^* convence a V con alta probabilidad, entonces existe un algoritmo extractor E que calcula el valor w sólo a partir de interactuar con P^*. Pero, ¿qué impide que cualquier verificador pueda usar el algoritmo E y robarle la solución al demostrador P? Después de todo, queremos un protocolo que sea una demostración de conocimiento y –al mismo tiempo– de Nula Divulgación. La respuesta a esta aparente contradicción es clara si pensamos qué significa exactamente tener un demostrador que nos convence. Un demostrador exitoso P^* es simplemente un programa ejecutado sobre una entrada x, el cual puede usar una cierta aleatoriedad (lanzar monedas o bits aleatorios que usa durante su ejecución). El algoritmo E puede entonces invocar o ejecutar a P^* varias veces sobre la misma entrada x y sobre la misma aleatoriedad r. Eso es algo que ningún verificador típicamente puede hacer en una ejecución real. Un demostrador no permitirá al verificador dictar la aleatoriedad a usar por el demostrador; tampoco ejecutará el mismo protocolo más de una vez con el mismo verificador.

[4] En el juego de Loto en Chile, el premio mayor lo obtiene quien acierta a los números en seis bolitas seleccionadas de un total de 41. La probabilidad de adivinar la combinación ganadora es de 1/4.496.388.

[5] Esto es en general para un protocolo de Nula Divulgación arbitrario, y no es el caso para el protocolo anterior para sudoku, el cual sí garantiza que Alicia sabe la solución, como veremos más adelante.

Protocolo de Nula Divulgación y demostración de conocimiento para sudoku

Veremos ahora que el protocolo dado en la sección anterior es también una demostración de conocimiento. El extractor E en cuestión es simplemente un programa que actúa como Roberto pero que en el paso 2 pregunta primero por la opción (a), digamos la fila 1, y luego de recibir una respuesta correcta, *ejecuta P de nuevo, desde el comienzo con la misma aleatoriedad*, pero ahora le pregunta en el paso 2 por la fila 2 en la opción (a) (notemos que los commitments son iguales pues la aleatoriedad es la misma.) El algoritmo E puede repetir este proceso de "ejecutar de nuevo" varias veces, recuperando las respuestas para todas las opciones posibles: puede preguntar no sólo por todas las filas, sino por todas las columnas, todas las submatrices y todos los valores en las posiciones originales "llenas" del sudoku. Al obtener las respuestas correctas a *todas*[6] las preguntas posibles de Roberto en el paso 2, el extractor E puede deducir cuál fue la permutación usada por el demostrador y, a partir de los valores revelados de todas las filas (o columnas) obtener la solución w del sudoku original x. Esto nos demuestra que, si Alicia puede convencer a Roberto usando el protocolo de Nula Divulgación descrito anteriormente, entonces Alicia debe saber la solución. De hecho, incluso si ella no lo supiera (por ejemplo, si para convencer a Roberto usara algún algoritmo extraño, ultra sofisticado, que no necesitase la solución para funcionar), entonces ¡ella misma podría ejecutar E y "extraer de sí misma" la solución!

APLICACIÓN A VOTACIÓN ELECTRÓNICA

Concluimos el artículo comentando que las aplicaciones de los protocolos de Nula Divulgación son inmensas y variadas. Una en particular es su uso en votación electrónica. Efectivamente, es posible diseñar sistemas de votación electrónica en los cuales el voto de una persona es encriptado usando un esquema de encriptación de clave pública. Supongamos que la votación es de tipo plebiscito: el voto puede sólo ser 1 (a favor) o 0 (en contra). En estos sistemas el esquema de encriptación usado tiene una propiedad muy útil, es *homomórfico:* la multiplicación de todos los textos cifrados permite obtener un texto cifrado con la suma de los votos. Luego, para saber si se aprueba o no el plebiscito basta desencriptar el texto cifrado así calculado. Si el valor obtenido es un número mayor que la mitad de los votos, entonces el resultado final es a favor; si no, es en contra. Fácil, ¿no?

Hay un detalle crítico, sin embargo, para que el sistema anterior funcione. Si un votante malicioso puede encriptar un valor distinto a 0 ó 1 en su voto, digamos 100 entonces, su voto afectará el total en forma inapropiada: el efecto de su voto es equivalente a 100 votos a favor, algo claramente inaceptable. Para tener la certeza que el voto encriptado es sólo 0 ó 1 *sin revelar el voto específico* podemos utilizar demostraciones de Nula Divulgación. En principio, el problema que queremos resolver es demostrar que una cierta encriptación tiene como solución (su desencriptación) es 0 ó 1 y nada más, algo que se puede demostrar está en NP. Dado que el problema de determinar si un sudoku tiene solución es NP-Completo podemos transformar el problema anterior en una instancia de sudoku, demostrar en Nula Divulgación que conocemos una solución al puzle, y ello inmediatamente nos daría una demostración para el problema original, esto es, que el voto dado es válido. Más aún, dado que dicho protocolo es una demostración de conocimiento, entonces podemos tener la certeza que el votante sabe su voto, y por ende, es suyo (y no es simplemente una copia del texto cifrado usado por otra persona). Vale notar que, por cierto, ésta no es la única manera de dar un protocolo de Nula Divulgación para este problema, hay otros más eficientes e incluso no interactivos (por ejemplo [4]), pero ése es tema de otro artículo.

Es difícil pensar en una aplicación más práctica y con mayor impacto social que la votación electrónica. ¿Quién hubiera pensando que la Teoría de la Computación nos daría herramientas para fundar nuestra democracia en el mundo digital? BITS

Referencias

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. Introduction to Algorithms (3. ed.). MIT Press, 2009.

[2] R.A. Fisher. Mathematics of a lady tasting tea. In J. R. Newman, editor, The World of Mathematics, Statistics and the Design of Experiments, volume 3, pages 1512 - 1521. Simon & Schuster, New York, 1956.

[3] Ronen Gradwohl, Moni Naor, Benny Pinkas, and Guy N. Rothblum. Cryptographic and physical zero-knowledge proof systems for solutions of sudoku puzzles. Theory Comput. Syst., 44(2):245 - 268, 2009.

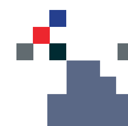
[4] J. Groth. Non-interactive zero-knowledge arguments for voting. In proceedings of ACNS 05, LNCS series, pages 467 - 482. Springer-Verlag, 2005.

[5] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography. Chapman & Hall/CRC Cryptography and Network Security. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[6] Mike Rosulek. Zero-knowledge proofs, with applications to sudoku and where's waldo? Presentación educacional, University of Montana, 2008.

[7] T. Seta T. Yato. Complexity and completeness of finding another solution and its application to puzzles. IEICE Trans. Fundam. Electron. Commun. Computer Science, E86-A(5):1052 - 1060, 2003.

[6] En estricto rigor, no es necesario obtener las respuestas para todas las preguntas, basta preguntar por todas las filas, o todas las columnas, o todas las submatrices para obtener todos los valores de la matriz.



Problemas de satisfacción de restricciones

Miguel Romero
Estudiante de Doctorado en Ciencias de la Computación, Universidad de Chile. Ingeniero Civil Matemático y Magíster en Ciencias mención Computación, Universidad de Chile. Áreas de interés: Bases de Datos, Lógica para la Computación.
mromero@dcc.uchile.cl

Un problema de *satisfacción de restricciones* (o CSP, por sus siglas en inglés), consiste en buscar una asignación de valores para un conjunto de variables que respete ciertas restricciones. Estos tipos de problemas son fundamentales en Ciencia de la Computación, ya que permiten modelar problemas de distintas áreas, como Inteligencia Artificial, Bases de Datos y Teoría de Grafos, por nombrar algunas.

En los años setenta, partiendo con el trabajo de Montanari [30], la comunidad de Inteligencia Artificial comenzó una intensa investigación en esta clase de problemas. El área de CSP ocupa un lugar prominente en Inteligencia Artificial y ha sido fundamental en el estudio de satisfacibilidad booleana, *scheduling*, razonamiento temporal y visión computacional, entre otros [11, 18, 29]. En general, es NP-Completo resolver un CSP, por lo que los esfuerzos se han enfocado en buscar casos tratables y heurísticas para la búsqueda de soluciones [11, 18].

Por otra parte, a partir de los influyentes trabajos de Feder, Kolaitis y Vardi [15, 27, 28] a finales de los noventa, se inició un estudio sistemático de CSP, desde un punto de vista teórico. En este enfoque, hay un énfasis en comprender la diferencia entre casos tratables y no tratables, utilizando herramientas de complejidad computacional y NP-completitud. La pregunta básica que ha guiado esta línea de investigación ha sido la siguiente:

¿Qué tipos de CSP son tratables y qué tipos no lo son?

Durante el último tiempo, ha habido avances importantes y se han establecido conexiones interesantes con otras áreas como Bases de Datos, Lógica, Álgebra Universal y Complejidad computacional. Es importante recalcar la relevancia de este enfoque. Primero, nos ayuda a entender conceptualmente qué es lo que hace a un CSP admitir un algoritmo eficiente. Segundo, en términos prácticos, conocer los límites entre lo tratable y lo intratable, nos ayuda a desarrollar mejores algoritmos y heurísticas.

En este artículo daremos una breve mirada a los aspectos teóricos de satisfacción de restricciones. Introduciremos el concepto de CSP y revisaremos brevemente los avances más importantes en los últimos quince años.

FORMULANDO UN CSP

Pensemos en el problema de 2-coloración: nuestro input es un grafo G y queremos saber si cada nodo puede ser pintado con un color entre {Azul, Rojo}, de manera que nodos adyacentes reciben colores diferentes. Podemos formular este problema como un CSP de la siguiente manera:

1. Nuestras variables son los nodos de G.
2. El espacio de valores posibles para las variables es {Azul, Rojo}.
3. Por cada arco (x,y), tenemos una restricción indicando que x e y reciben valores distintos.

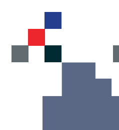
Como podemos ver, cada restricción determina los valores que pueden tomar cierto conjunto de variables. La manera estándar de escribir una restricción, es indicar primero las k variables que se ven afectadas por ella y, luego, una relación k-aria, indicando los posibles valores que pueden tomar estas variables. En el caso de 2-coloración, por cada arco (x,y), tendremos una restricción de la forma ((x,y), Neq), donde Neq indica los pares de colores distintos:

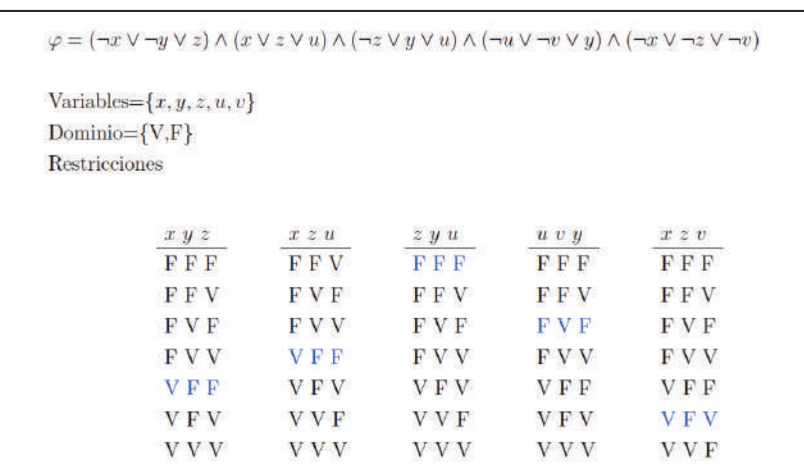
Neq={(Azul, Rojo), (Rojo, Azul)}

Luego, cada asignación de valores a las variables que respeta las restricciones, efectivamente corresponde a un 2-coloreo.

Consideremos ahora el famoso problema 3-SAT: Nuestro input es una fórmula proposicional en 3CNF (conjunción de cláusulas con 3 literales, ver ejemplo en Figura 1) y debemos decidir si es satisfacible. Nuestras variables son simplemente las variables de la fórmula, los valores posibles son {V,F}, y cada cláusula genera una restricción. Por ejemplo, si tomamos la primera cláusula $(\neg x \vee \neg y \vee z)$ de la fórmula φ en la Figura 1, entonces tendremos la restricción ((*x,y,z*), R), donde R indica los posibles valores que hacen verdadera esta cláusula, es decir, todos los triples excepto (V,V,F), como muestra la primera restricción de la Figura.

Nuevamente es claro que una asignación que respeta todas las restricciones, corresponde a una asignación que hace verdadera a la fórmula. A modo de ejemplo, consideremos la asignación x=V, y=F, z=F, u=F, v=V, para la fórmula φ de la Figura 1. Notemos que la primera restricción afecta a (*x, y,* z). Si reemplazamos cada variable por el valor asignado,





$\varphi = (\neg x \vee \neg y \vee z) \wedge (x \vee z \vee u) \wedge (\neg z \vee y \vee u) \wedge (\neg u \vee \neg v \vee y) \wedge (\neg x \vee \neg z \vee \neg v)$

Variables=$\{x, y, z, u, v\}$
Dominio={V,F}
Restricciones

$x\ y\ z$	$x\ z\ u$	$z\ y\ u$	$u\ v\ y$	$x\ z\ v$
F F F	F F V	F F F	F F F	F F F
F F V	F V F	F F V	F F V	F F V
F V F	F V V	F V F	F V F	F V F
F V V	V F F	F V V	F V V	F V V
V F F	V F V	V F V	V F F	V F F
V F V	V V F	V V F	V F V	V F V
V V V	V V V	V V V	V V V	V V F

Figura 1 • Formulación de 3-SAT como un CSP.

obtenemos la tupla (V,F,F), la cual es una tupla permitida, como muestra el color azul en la Figura. Lo mismo sucede con todas las restricciones, lo que implica que nuestra asignación es una solución. Es fácil verificar además que esta asignación efectivamente satisface a la fórmula.

Ahora estamos en condiciones de dar una formulación más general. Una instancia de un CSP queda definida por tres componentes:

1. Un conjunto finito de variables V.
2. Un dominio finito de valores D.
3. Un conjunto finito de restricciones C. Cada restricción es de la forma $((x_1, x_2, \ldots, x_k), R)$, donde $\{x_1, x_2, \ldots, x_k\}$ son variables en V y R es una relación k-aria sobre D. Como ya mencionamos, cada restricción determina los posibles valores que pueden tomar ciertas variables.

En un CSP nuestra instancia es un triple (V,D,C) y el objetivo es decidir si existe una *solución*, es decir, una asignación h: $V \rightarrow D$ de valores a las variables, que respeta todas las restricciones. Esto último significa que para cada restricción $((x_1, x_2, \ldots, x_k), R)$ en C, tenemos que $(h(x_1), h(x_2), \ldots, h(x_k))$ está en la relación R.

Observemos que en el caso de 2-coloración, nuestras instancias tienen sólo restricciones binarias y éstas sólo usan la relación Neq, mientras que en 3-SAT las restricciones son ternarias y usamos varios tipos de relaciones dependiendo de la forma de la cláusula. Por tanto, 2-coloración y 3-SAT son distintos tipos de CSP, en el sentido que las instancias tienen distinta estructura. Más aún, sus complejidades son radicalmente diferentes: mientras 2-coloración se puede resolver eficientemente mediante un algoritmo *greedy*, 3-SAT es NP-Completo [2].

Lo anterior ilustra el hecho de que satisfacción de restricciones es NP-Completo en general, pero en algunos casos, dependiendo de la estructura de las instancias, podemos encontrar una solución de manera eficiente.

CSP y homomorfismos

Uno de los principales aportes conceptuales del trabajo de Feder y Vardi [15] fue formular un CSP como un problema de homomorfismos entre estructuras. Esta formulación dio origen a un amplio estudio teórico y permitió establecer conexiones con otras áreas más abstractas. A continuación definimos la noción de estructura y homomorfismo.

Un *esquema* es un conjunto de símbolos de relación $R_1, \ldots, R_p$. Cada símbolo Ri tiene una aridad $k_i>0$ asociada. Una *estructura* A sobre el esquema $R_1, \ldots, R_p$, es básicamente un conjunto de relaciones $R_1^A, \ldots, R_p^A$ (de la aridad correspondiente), sobre cierto dominio finito dom(A). Es decir, la estructura A interpreta los símbolos de relación en dom(A).

Por ejemplo, cada grafo dirigido puede ser visto como una estructura, donde el dominio son los nodos y tenemos una relación binaria indicando cada arco. En el caso de un grafo no dirigido o grafo a secas, esta relación será simétrica. En este contexto, el esquema es un símbolo binario E. Similarmente, cada fórmula en 3CNF puede ser vista como una estructura. Notemos que (debido a que la disyunción conmuta) podemos asumir sin pérdida de generalidad que sólo tenemos cuatro tipos de cláusulas: $(x \vee y \vee z)$, $(\neg x \vee y \vee z)$, $(\neg x \vee \neg y \vee z)$ y $(\neg x \vee \neg y \vee \neg z)$. Por tanto, podemos considerar un esquema $T_1, \ldots, T_4$, en donde cada símbolo es ternario y cada uno representa cierto tipo de cláusula. De esta manera, el dominio de la estructura son las variables de la fórmula y tenemos 4 relaciones $T_1, \ldots, T_4$, las cuales indican los triples de variables que conforman cada tipo de cláusula. En la Figura 2, la estructura A^φ representa a φ.

Un *homomorfismo* entre dos estructuras (con el mismo esquema) A y B es un mapeo de dom(A) a dom(B), el cual preserva las relaciones de A. Formalmente, si el esquema es $R_1, \ldots, R_p$, entonces h: dom(A) → dom(B) es un homomorfismo, si para cada $1 \leq i \leq p$ y cada tupla $(t_1, \ldots, t_k)$ en R_i^A, se tiene que $(h(t_1), \ldots, h(t_k))$ está en R_i^B. Si existe homomorfismo de A a B, diremos que A es homomorfo a B.

Como ya podemos imaginar, existe una fuerte simbiosis entre homomorfismos y CSPs. De alguna forma, la característica de "preservar las relaciones" que posee un homomorfismo se alinea con "respetar las restricciones" en una solución de un CSP. Para ilustrar esto, consideremos el esquema E (,.) para representar grafos y la estructura K_2 con dominio {Azul, Rojo} y relación E^{K2}={(Azul, Rojo),

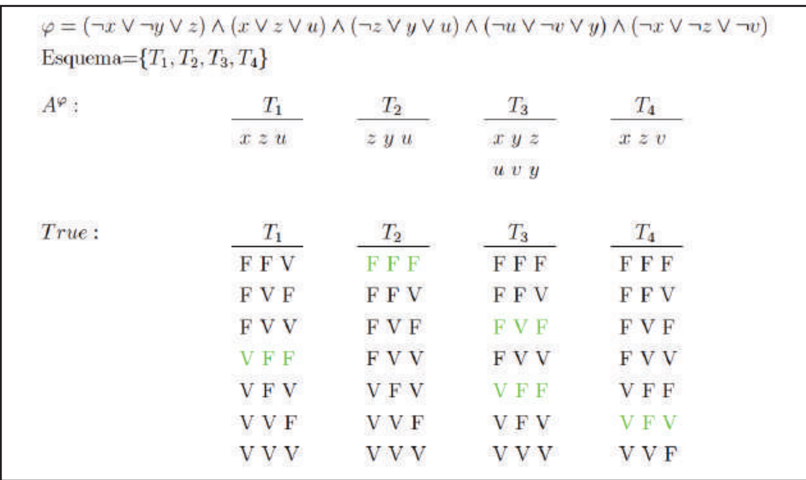


Figura 2 • El problema 3-SAT como existencia de homomorfismos.

(Rojo, Azul)}. Entonces es claro que un grafo G tiene un 2-coloreo si y sólo si G (visto como estructura) es homomorfo a K_2.

Este fenómeno no sólo se restringe a problemas de coloración, sino que es inherente a cualquier CSP. Para ver esto, consideremos una instancia genérica I = (V,D,C). Construiremos dos estructuras A(I) y B(I) como sigue:

1. En nuestro esquema tenemos un símbolo de relación por cada relación que aparece en alguna restricción de C.
2. El dominio de A(I) es el conjunto de variables V. Si tenemos una restricción $((x_1,\ldots, x_k), R)$ en C, entonces la tupla $(x_1,\ldots, x_k)$ está en $R^{A(I)}$.
3. El dominio de B(I) es el conjunto de valores D. Si R aparece en alguna restricción, entonces $R^{B(I)}$ es precisamente R.

Intuitivamente, A(I) codifica las variables y la interacción de los conjuntos de variables que aparecen restringidos. La estructura B(I) codifica los posibles valores que pueden tomar las variables restringidas. Luego, cada mapeo de A(I) a B(I) que respete las relaciones de A(I) (es decir, un homomorfismo), respeta a su vez las restricciones en I, y viceversa. Dicho de otra manera, existe un homomorfismo de A(I) a B(I) si y sólo si I tiene una solución.

Para ilustrar la construcción, consideremos el problema 3-SAT. En este caso, el esquema consta de cuatro relaciones ternarias T_1 ,…,T_4 , representando cada tipo de cláusulas, como vimos anteriormente. Para una fórmula φ, la estructura A(I) es la representación de la fórmula A^φ y B(I) es la estructura True, como muestra la Figura 2.

No es difícil ver que una asignación satisface φ si es un homomorfismo de A^φ a True. Por ejemplo, si consideramos nuevamente la asignación x=V, y=F, z=F, u=F, v=V, la cual hace verdadera a φ, podemos ver que define un homomorfismo. En efecto, la tupla (x,z,u), en la relación T_1 , es mapeada a la tupla (V,F,F), la cual pertenece a T_1 en True, como podemos apreciar de color verde en la figura. Las demás tuplas en A^φ también son respetadas, por tanto tenemos un homomorfismo.

Lo anterior nos dice que podemos identificar cualquier CSP con un problema de existencia de homomorfismo entre estructuras. Por tanto, de ahora en adelante pensaremos que toda instancia a un CSP es un par de estructuras A y B.

Tipos de CSPs

Para cada CSP, asumiremos que tenemos un esquema fijo a priori, y que las instancias están definidas sobre este esquema (por ejemplo, el esquema de los grafos o de las fórmulas). Cada tipo de CSP queda determinado por la forma de sus instancias, es decir la forma de las estructuras A y B.

Para formalizar esto, supongamos que tenemos dos conjuntos de estructuras C y D, potencialmente infinitos, sobre cierto esquema. Denotaremos por CSP(C,D) al CSP cuyas instancias A y B cumplen que A está en C y B está en D. Por ejemplo, 2-coloración es precisamente CSP(Grafos, $\{K_2\}$), donde Grafos es el conjunto de todos los grafos. A su vez, 3-SAT es equivalente a CSP(Fórmulas, {True}), donde Fórmulas son todas las estructuras que representan fórmulas en 3CNF.

Ahora podemos formular la pregunta fundamental que ha motivado todo este estudio:

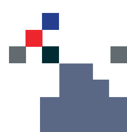
¿Para qué conjuntos C y D, CSP(C, D) admite un algoritmo polinomial?

En la literatura, a este tipo de conjuntos C y D se les llama “islas tratables”. Como podemos imaginar, ésta es una pregunta bastante ambiciosa y aún no se tiene respuesta para ella. Debido a esto, la investigación se ha enfocado en casos particulares [7, 21].

Un caso interesante, el cual estudiaremos en las siguientes secciones, es tomar D como el conjunto de todas las estructuras. Es decir, para un conjunto C, estudiaremos el problema CSP(C,-), en donde las instancias A y B, cumplen que A está en C, y no hay restricción sobre B. A continuación, abordaremos la siguiente pregunta: ¿para qué conjuntos C, CSP(C,-) admite un algoritmo eficiente? Como veremos más adelante, esta interrogante tiene importantes aplicaciones en teoría de Bases de Datos.

CASOS TRATABLES: ÁRBOLES

Gran parte de los CSPs tratables se basan



en una simple idea algorítmica. Supongamos que tenemos dos grafos dirigidos T y B. Asumamos además que T tiene forma de árbol, como vemos en el ejemplo de la Figura 3. Queremos saber si existe un homomorfismo de T a B.

Para esto, construiremos para cada nodo t de T, un conjunto B_t de nodos de B. Intuitivamente, cada B_t indicará los posibles valores que puede tomar la variable t, en una solución parcial acotada al subárbol "debajo" de t. Más precisamente, cada B_t contiene exactamente los nodos b de B tal que existe un homomorfismo de T_t (el subárbol de T enraizado en t) a B, el cual mapea t a b. Si somos capaces de computar estos conjuntos, nuestro problema está resuelto: que exista homomorfismo de T a B es equivalente a que B_r sea no vacío, donde r es la raíz de T.

La observación clave es que, debido a que T tiene forma de árbol, es posible calcular los conjuntos B_t's eficientemente. Para esto, comenzamos por las hojas y vamos computando recursivamente B_t hasta llegar a la raíz de la siguiente manera:

1. Para cada hoja h de T, B_h contiene todos los nodos de B.
2. Si t es un nodo interno y $t_1,\ldots, t_r$ son sus hijos, entonces para determinar si un nodo b está en B_t, basta ver que existan nodos b1,…,br contenidos en $B_{t1},\ldots,B_{tr}$, respectivamente, tal que para cada $1\leq i\leq r$, hay un arco entre b y b_i, en la misma dirección que el arco entre t y t_i (y así respetar este arco o restricción). Por ejemplo, para el nodo t de la Figura 3, un nodo b estará en B_t, si existen b_1, b_2, b_3, cada uno en B_{t1}, B_{t2}, B_{t3}, respectivamente, tal que los arcos (b,b_1), (b_2,b) y (b,b_3) están en B.

Observemos que cada vez que visitamos un nodo interno t y verificamos si b está en B_t, nos basta visitar cada nodo en $B_{t1},\ldots,B_{tr}$ una sola vez, por tanto podemos computar Bt en a lo más $O(|B|^2 H_t)$, donde H_t es la cantidad de hijos de t. Lo anterior nos dice que nuestro algoritmo toma tiempo a lo más $O(|B|^2|T|)$. Luego, la clase Trees, de todos los grafos dirigidos con forma de árbol, es una isla tratable.

Esta idea básica ha sido generalizada a estructuras arbitrarias y más aún a estructuras que tienen una forma "parecida" a un árbol, utilizando el concepto de *treewidth* [10, 17]. El treewidth de una estructura mide qué tanto se parece ésta a un árbol: mientras más pequeño el treewidth, más parecida a un árbol. Por ejemplo, todos los árboles tienen treewidth 1.

Para finalizar, enunciamos el resultado que describe las islas tratables definidas por la noción de treewidth:

Teorema [10, 17]. Sea k≥1 y C un conjunto de estructuras. Si cada estructura en C tiene treewidth a lo más k, entonces CSP(C,-) se puede resolver en tiempo polinomial.

Por tanto, toda clase C con "treewidth acotado" es una isla tratable. Un caso particular es la clase Trees, descrita previamente.

Juegos de fichas

El resultado anterior, que involucra la noción de treewidth, puede ser explicado conectando CSP con otros conceptos como definibilidad en Datalog, test de k-consistencia o lógicas con variables finitas [12]. Otra forma interesante de entender estos resultados es mediante ciertos tipos de juegos combinatoriales llamados *juegos de fichas* [12, 26, 28]. Como veremos en la siguiente sección, estos juegos nos permitirán definir nuevas islas tratables.

Para empezar, definiremos un juego sobre dos grafos dirigidos A y B. En este juego, hay dos jugadores: *Spoiler* y *Duplicator*. Una posición del juego es un par (a,b) en dom(A)×dom(B). En el primer round, Spoiler escoge un elemento a_0 de A y Duplicator responde con un elemento b_0 en B. Después de esto, la posición del juego es (a_0, b_0). En los rounds subsiguientes, si la posición actual es (a,b), entonces la siguiente posición (a',b') queda determinada como sigue: Spoiler escoge un vecino a' de a y Duplicator res-

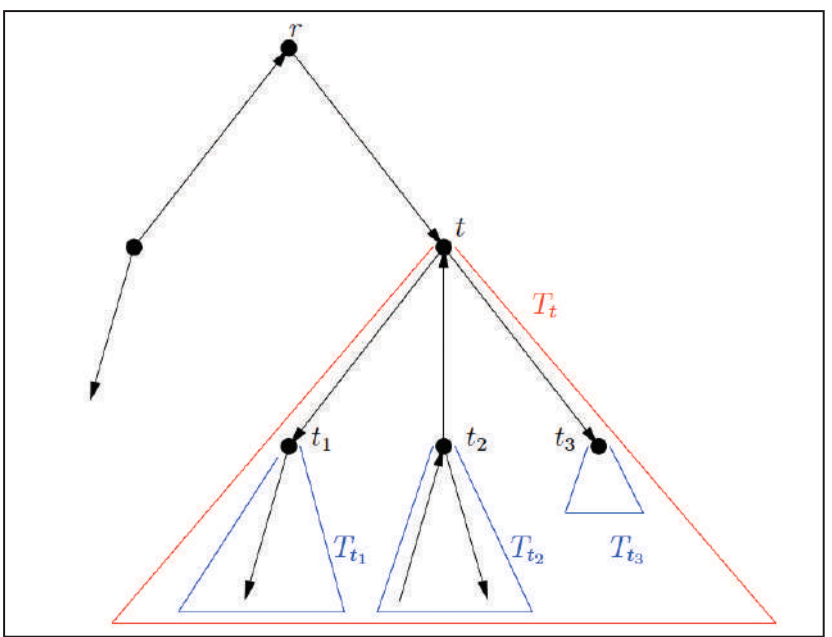


Figura 3 • El árbol T y algunos de sus subárboles.

ponde con un vecino b' de b, de manera que la dirección del arco entre a y a', es la misma que la del arco entre b y b'. Por ejemplo, si (a',a) está en A, entonces (b',b) debe estar en B. Si Duplicator tiene una estrategia que le permite jugar para siempre con Spoiler, sin importar como juegue este último, entonces decimos que Duplicator gana este juego sobre A y B.

Supongamos por un momento que A es un árbol, como el de la Figura 3. No es difícil ver que, si existe un homomorfismo h de A a B, entonces Duplicator gana este juego sobre A y B (de hecho, esto es cierto incluso para un A arbitrario). En efecto, en cada round, si Spoiler juega un elemento a, entonces Duplicator responde con h(a). Como h respeta la dirección de los arcos, esto siempre será una jugada válida para Duplicator. Por otra parte, supongamos que Duplicator gana este juego sobre A y B. Tampoco es difícil ver que esto implica la existencia de un homomorfismo de A a B. Intuitivamente, debemos "simular" una partida de este juego y hacer que Spoiler juegue desde la raíz hacia las hojas (escogiendo un nodo y luego todos sus hijos, continuando recursivamente). Las respuestas dadas por Duplicator en esta partida (cuando éste juega con su estrategia ganadora) constituirán el homomorfismo. Como A es un árbol no hay conflictos en los valores asignados y el homomorfismo queda bien definido. Por tanto, cuando A es un árbol, verificar si Duplicator gana este juego entre A y B es un método correcto y completo para existencia de homomorfismos [12].

Como podemos sospechar, la definición original de este juego, introducido por Vardi y Kolaitis en 1990 [26], llamado el *juego existencial de k fichas*, es más general y aplica para estructuras arbitrarias. El resultado anterior aplica para la definición original de la siguiente manera: si A tiene treewidth a lo más k, entonces decidir acaso Duplicator gana el juego existencial con k+1 fichas sobre A y B es un método correcto y completo para existencia de homomorfismos.

Pero aún falta lo más importante: ¿cómo decidimos si Duplicator puede ganar el juego con k fichas? Afortunadamente, esto se puede hacer en tiempo polinomial en A y B (k está fijo) [12, 25]. Por tanto, esto nos da una explicación alternativa del hecho de que toda clase de treewidth acotado es una isla tratable.

Más allá de treewidth acotado

Una pregunta natural que surge es la siguiente: ¿existen islas tratables más generales que treewidth acotado? ¿Es treewidth acotado la noción óptima? Como veremos a continuación, podemos definir islas tratables aún más generales.

Diremos que A y A' son *equivalentes* si existe un homomorfismo de A a A' y de A' a A. Notemos que si A y A' son equivalentes, entonces, para cualquier estructura B, A es homomorfo a B si y sólo si A' es homomorfo a B (ya que los homomorfismos componen). Luego, A y A' son realmente "indistinguibles" para cualquier CSP.

Pensemos por un momento en grafos dirigidos. Supongamos que queremos decidir si A es homomorfo a B. Asumamos, además, que A es equivalente a un árbol T. Es claro que si tuviéramos disponible T junto con la entrada, podríamos verificar eficientemente si A es homomorfo a B, ya que bastaría verificar acaso T es homomorfo a B. Ahora bien, ¿qué sucede si T no está disponible con la entrada? ¿Es posible aún resolver el problema eficientemente?

A primera vista esto es imposible, ya que no hay manera obvia de computar tal árbol T [12]. Sorprendentemente, utilizando juegos de fichas aún es posible resolver el problema eficientemente sin necesidad de computar el árbol equivalente T. De igual forma, esto extiende a treewidth acotado. Por tanto, si A es equivalente a una estructura de treewidth a lo más k, decidir si Duplicator gana el juego de k+1 fichas sigue siendo correcto y completo para existencia de homomorfismos.

Esto implica que podemos expandir nuestras islas tratables de treewidth acotado a treewidth acotado "módulo equivalencia", como queda enunciado a continuación:

Teorema [12]. Sea k≥1 y C un conjunto de estructuras. Si cada estructura en C es equivalente a una estructura de treewidth a lo más k, entonces CSP(C,-) se puede resolver en tiempo polinomial.

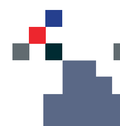
Es importante destacar que, para cualquier k≥1, existen estructuras de treewidth arbitrariamente grande que son equivalentes a estructuras de treewidth a lo más k. Por tanto, estas nuevas islas tratables no sólo contienen a las anteriores, sino que son una generalización interesante y no trivial.

La frontera entre lo tratable y lo intratable

El concepto de treewidth acotado genera islas tratables. Como ya vimos, esto se puede extender a treewidth acotado módulo equivalencia. Pero, ¿podemos ir más allá? ¿Es treewidth acotado módulo equivalencia el límite entre lo tratable y lo intratable?

Sorprendentemente, Grohe en el 2003 dio una respuesta afirmativa a esta pregunta [20]. En un resultado matemáticamente profundo, demostró que si asumimos ciertos supuestos de complejidad parametrizada [16] (supuestos que se creen ciertos, al estilo P≠NP), entonces una clase de estructuras C es una isla tratable si y sólo si C tiene treewidth acotado módulo equivalencia, es decir, existe k≥1 tal que cada estructura en C es equivalente a otra que tiene treewidth a lo más k.

Por tanto, lo que hace que CSP(C,-) admita un algoritmo polinomial es esencialmente que C tenga treewidth acotado módulo equivalencia.



APLICACIONES EN BASES DE DATOS

Calcular el join natural de un conjunto de tablas está en el corazón de todo sistema de manejo de base de datos relacionales. Como este problema es intratable en general [1], definir tipos de joins que admitan evaluación eficiente es un problema fundamental en Computación.

Consideremos por un momento el problema 3-SAT y el ejemplo de la Figura 1. Para computar las soluciones de φ debemos buscar asignaciones a las variables {x,y,z,u,v} que respeten cada restricción. Si observamos la figura, la conexión con base de datos relacionales es clara: podemos pensar las restricciones, como un conjunto de tablas relacionales sobre los atributos {x,y,z,u,v}. Es fácil ver que el conjunto de soluciones para φ es precisamente el join natural de estas 5 tablas.

Esta relación fue observada en 1977 por Chandra y Merlin [8], quienes mostraron que evaluar una consulta J del álgebra relacional que sólo utiliza join natural, sobre una base de datos D es equivalente a verificar existencia de homomorfismos entre un par de estructuras A y B. Más aún, la estructura A es esencialmente la consulta J codificada de manera natural como una estructura. Por tanto, las siguientes dos preguntas son equivalentes:

1. ¿Qué tipos de joins admiten evaluación eficiente?
2. ¿Qué clases C, hacen que CSP(C,-) sea tratable?

A partir de las secciones anteriores, podemos deducir que los tipos de joins relacionales que admiten evaluación eficiente son precisamente los joins con treewidth acotado módulo equivalencia.

Esto muestra una de muchas aplicaciones en Bases de Datos. Más aún, esta transferencia de herramientas no es unidireccional: muchas técnicas de Bases de Datos son utilizadas en satisfacción de restricciones, lo cual ha generado una interesante simbiosis entre estas dos áreas [1, 8, 12, 19, 27].

LA CONJETURA DE LA DICOTOMÍA DE CSPs

No todos los problemas de satisfacción de restricciones tienen la forma CSP(C,-), para un conjunto C de estructuras. Pensemos en 2-coloración: este problema corresponde a CSP(-,{K_2}), por tanto la estructura "de la mano derecha" está fija y es K_2. En este caso, nuestra instancia es una estructura (grafo) G y debemos determinar si es homomorfo a K_2. Lo mismo sucede con 3-SAT, ya que corresponde a CSP(-,{True}).

Por tanto, mucho problemas interesantes tienen la forma CSP(-,{B}), para una estructura fija B, lo cual ha generado mucho interés en este tipo de CSPs [3, 7, 23, 31]. De hecho, los trabajos más profundos matemáticamente acerca de la complejidad de CSP tienen que ver con esta clase de problemas. El motor detrás de este estudio es una conjetura hecha por primera vez en 1993 por Feder y Vardi [15]:

Conjetura (Dicotomía CSP). Para toda estructura B, el problema CSP(-,{B}), o bien puede ser resuelto en tiempo polinomial, o bien es NP-Completo.

A primera vista, puede sonar que esta conjetura es obviamente cierta. Notemos que cada CSP está en NP. Sin embargo, la clase NP probablemente no satisface esta dicotomía, ya que existen problemas que se cree no están en P ni son NP-Completos, como es el caso de Isomorfismos de Grafos y de Factorización [2]. Esto se ve reforzado por el teorema de Ladner [2], que dice que si P≠NP, entonces existen problemas que no están en P y que no son NP-Completos. Luego, nada impide que algunos de estos problemas "intermedios" puedan ser formulado como un CSP de la forma CSP(-,{B}). Por tanto, no hay ninguna razón a priori para pensar que la dicotomía CSP es cierta.

Esta conjetura sigue abierta hasta hoy. Sin embargo, muchos casos importantes han sido resueltos. Algunos de estos casos han sido la dicotomía de Schaefer [31], quien demostró la conjetura para estructuras B con dos elementos, y el trabajo de Hell y Nesetril [23], quienes demostraron la dicotomía para el caso en que B es un grafo no dirigido. En particular, demostraron que CSP(-,{B}) es tratable si B es bipartito, y NP-Completo en caso contrario.

En los últimos años, ha tomado fuerza una línea de ataque a la dicotomía CSP que utiliza herramientas de álgebra universal [3, 4, 6, 7], la cual ha logrado importantes avances. En [4], Bulatov demostró la conjetura para estructuras B con 3 elementos y en [6], para una clase interesante de estructuras llamadas *conservativas*. Además, en [3], Bulatov, Krokhin y Jeavons enriquecieron la conjetura de la dicotomía CSP, conjeturando el borde exacto que separa los casos tratables de los intratables.

CONCLUSIONES

Los problemas de satisfacción de restricciones ocupan un lugar privilegiado en Ciencia de la Computación y son de alto interés tanto teórico como práctico. El estudio teórico de la complejidad de CSPs ha sido muy fructífero, no sólo porque nos permite entender a fondo este tipo de problemas, sino porque en el camino se han desarrollado potentes herramientas teóricas, que son interesantes por sí mismas, y se han establecido fuertes conexiones con otras áreas de interés.

En este artículo hemos cubierto una pequeña fracción del tema. Por ejemplo,

hemos asumido que nuestro esquema está fijo a priori, no obstante muchos problemas interesantes requieren que el esquema sea parte de las instancias. En este contexto, es posible encontrar islas tratables más generales que treewidth acotado módulo equivalencia, basadas en los conceptos de *hypertree width* [19], *coverwidth* [9] y *fractional hypertree width* [22]. Se ha conjeturado en [21], que fractional hypertree width acotado módulo equivalencia es la noción óptima, sin embargo, esto permanece abierto hasta hoy. Otros trabajos relevantes incluyen: el estudio de CSP como un problema de optimización (satisfacer la mayor cantidad de restricciones posibles) y el respectivo análisis de aproximabilidad que esto conlleva [14, 24], y el estudio de la complejidad de contar las soluciones de un CSP [5, 13]. Por ejemplo, es interesante destacar que la dicotomía CSP ha sido resuelta en su versión con respecto a complejidad de conteo [5]. BITS

Referencias

[1] S. Abiteboul, R. Hull, V. Vianu. Foundations of Databases. Addison-Wesley, 1995.

[2] S. Arora, B. Barak. Complexity Theory: A Modern Approach, Cambridge University Press, Cambridge, UK, 2009.

[3] A. Bulatov, P. Jeavons, A. Krokhin. Classifying the Complexity of Constraints Using Finite Algebras. SIAM J. Comput. 34(3): 720-742 (2005).

[4] A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. J. ACM 53(1): 66-120 (2006).

[5] A. Bulatov. The Complexity of the Counting Constraint Satisfaction Problem. ICALP (1) 2008: 646-661.

[6] A. Bulatov. Complexity of conservative constraint satisfaction problems. ACM Trans. Comput. Log. 12(4): 24 (2011).

[7] A. Bulatov. On the CSP Dichotomy Conjecture. CSR 2011: 331-344.

[8] A. Chandra, P. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. STOC 1977: 77-90.

[9] H. Chen, V. Dalmau. Beyond Hypertree Width: Decomposition Methods Without Decompositions. CP 2005: 167-181.

[10] R. Dechter, J. Pearl. Tree Clustering for Constraint Networks. Artif. Intell. 38(3): 353-366 (1989).

[11] R. Dechter. Constraint processing. Morgan Kauman, 2003.

[12] V. Dalmau, Ph. Kolaitis, M. Vardi. Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics. CP 2002: 310-326.

[13] V. Dalmau, P. Jonsson. The complexity of counting homomorphisms seen from the other side. Theor. Comput. Sci. 329(1-3): 315-323 (2004).

[14] V. Deineko, P. Jonsson, M. Klasson, A. Krokhin. The approximability of MAX CSP with fixed-value constraints. J. ACM 55(4) (2008).

[15] T. Feder, M. Vardi. Monotone monadic SNP and constraint satisfaction. STOC 1993: 612-622.

[16] J. Flum, M. Grohe. Parameterized Complexity Theory. Springer-Verlag, 2006.

[17] E. Freuder. Complexity of K-Tree Structured Constraint Satisfaction Problems. AAAI 1990: 4-9.

[18] E. Freuder, A. Mackworth. Constraint-based reasoning. MIT Press, 1994.

[19] G. Gottlob, N. Leone, F. Scarcello. Hypertree Decompositions and Tractable Queries. J. Comput. Syst. Sci. 64(3): 579-627 (2002).

[20] M. Grohe. The Complexity of Homomorphism and Constraint Satisfaction Problems Seen from the Other Side. FOCS 2003: 552-561.

[21] M. Grohe. The Structure of Tractable Constraint Satisfaction Problems. MFCS 2006: 58-72.

[22] M. Grohe, D. Marx. Constraint solving via fractional edge covers. SODA 2006: 289-298.

[23] P. Hell, J. Nesetril. On the complexity of H-coloring. J. Comb. Theory, Ser. B 48(1): 92-110 (1990).

[24] P. Jonsson, M. Klasson, A. Krokhin. The Approximability of Three-valued MAX CSP. SIAM J. Comput. 35(6): 1329-1349 (2006).

[25] Ph. Kolaitis, J. Panttaja. On the Complexity of Existential Pebble Games. CSL 2003: 314-329.

[26] Ph. Kolaitis, M. Vardi. On the Expressive Power of Datalog: Tools and a Case Study. PODS 1990: 61-71.

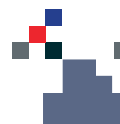
[27] Ph. Kolaitis, M. Vardi. Conjunctive-Query Containment and Constraint Satisfaction. PODS 1998: 205-213.

[28] Ph. Kolaitis, M. Vardi. A Game-Theoretic Approach to Constraint Satisfaction. AAAI/IAAI 2000: 175-181.

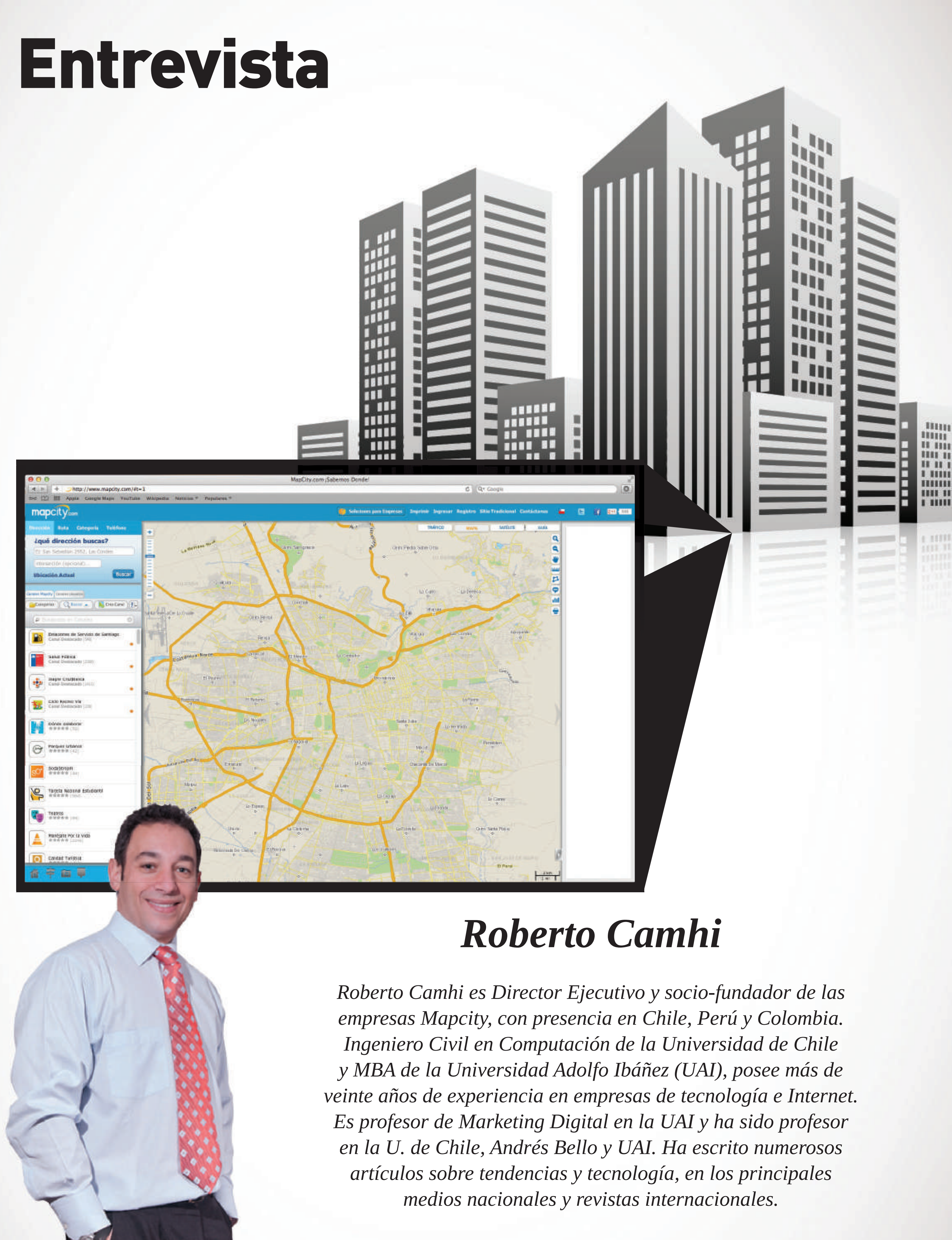
[29] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. AI Magazine 13(1): 32-44 (1992).

[30] U. Montanari. Networks of Constraints. Fundamental Properties and Application to Picture Processing. Information Science, 7:95-132, 1974.

[31] T. Schaefer. The Complexity of Satisfiability Problems. STOC 1978: 216-226.



Entrevista



Roberto Camhi

Roberto Camhi es Director Ejecutivo y socio-fundador de las empresas Mapcity, con presencia en Chile, Perú y Colombia. Ingeniero Civil en Computación de la Universidad de Chile y MBA de la Universidad Adolfo Ibáñez (UAI), posee más de veinte años de experiencia en empresas de tecnología e Internet. Es profesor de Marketing Digital en la UAI y ha sido profesor en la U. de Chile, Andrés Bello y UAI. Ha escrito numerosos artículos sobre tendencias y tecnología, en los principales medios nacionales y revistas internacionales.

Por: Benjamín Bustos y Claudio Gutiérrez

1 • ¿Qué es lo que más valoras hoy de la formación del DCC y cómo ha influenciado en tu carrera profesional?
El DCC me entregó (y lo sigue haciendo) una formación muy sólida desde el punto de vista técnico, con fuerte orientación a la resolución de problemas. En la Escuela de Ingeniería y en particular en el DCC, se nos enseñó a pensar lógicamente, de modo de poder enfrentar los problemas cotidianos de la misma forma, lo que me ha sido de gran utilidad en mi vida.

2 • Por el contrario, ¿qué le agregarías a la formación del DCC, que pueda ayudar a formar ingenieros en Computación mejor capacitados para los requerimientos actuales?
Hoy en día, no sólo basta con tener las competencias técnicas necesarias para enfrentar un proyecto. Lo que más se valora y requiere en las empresas es la capacidad de gerenciar, liderar y entender los problemas de negocio de los clientes, para dar soluciones a ellos. Si no se es capaz de analizar el problema del cliente, innovar y crear soluciones para ellos, un ingeniero no será nunca completo y quedará relegado exclusivamente al ámbito tecnológico.

3 • Eres fundador del exitoso portal Mapcity.com. ¿Cómo enfrentan la competencia de sistemas similares como Google Maps y otros? ¿Cómo ha sido posible mantener un servicio independiente de los grandes jugadores en este terreno? ¿Es viable eso?
Mucha gente ve a Mapcity como un portal de Internet de búsqueda de direcciones, pero la realidad es que eso es sólo la cara más visible. Mapcity es una compañía con presencia en Chile, Perú y Colombia, cuya función es ayudar a grandes empresas a tomar mejores decisiones de negocio basados en información territorial. Resolvemos el problema del "Dónde". Para ello contamos con mucha información demográfica, de personas, comercio y empresas, las que combinamos con herramientas de análisis para hacer el *delivery* de las soluciones. El portal es muy importante, de hecho recibe cerca de tres millones de consultas al mes, pero genera menos del 5% de los ingresos de la empresa.

Hoy Mapcity provee los mapas de Google Maps y además es representante en Chile y Perú de ellos, por lo que la relación con Google es muy amistosa y de colaboración.

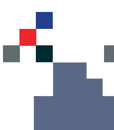
4 • ¿Utilizan bases de datos espaciales para guardar y consultar la información geográfica?
Así es. Las bases de datos espaciales son la fuente o insumo básico para nuestros procesos, las que se deben mantener actualizadas permanentemente.

5 • ¿Qué tan difícil sería agregarle al sitio consultas del tipo "dónde está el restaurante de sushi más cercano a mi posición actual" o "dónde está la sucursal del banco X más cercana"?
Ese tipo de consultas se pueden responder actualmente, para lo cual existen los canales temáticos, que son más de 160. Basta con posicionarse en el mapa, encender el canal que se está buscando y se encontrarán en la cercanía los elementos buscados, como bancos, cajeros, farmacias y un sinfín de posibilidades.

6 • ¿Qué infraestructura (hardware/software) se requiere para mantener el sitio funcionando?
Tenemos nuestros propios servidores en un data center en Chile, para los servicios locales, y en Estados Unidos para los demás países. Para ello utilizamos ocho servidores Blade balanceados y ocho raqueables.

7 • ¿Cómo se actualiza la información geográfica?¿Cada cuánto tiempo?
Toda la información se actualiza permanentemente y de acuerdo a necesidades de la empresa, por ejemplo para satisfacer los requerimientos de un determinado proyecto. En el caso de los mapas, liberamos nuevas versiones dos veces al año y la actualización se realiza a partir de imágenes satelitales y terreno.

8 • ¿Cuáles son los desafíos técnicos que se vienen en el área en el futuro cercano?
Seguir innovando y diferenciarnos de nuestra competencia. El tema no es tecnológico, es de modelo de negocio. En eso estamos trabajando ahora, para desarrollar nuevos productos y siempre tener nuestra ventaja. BITS



Estudiantes y académicos de la UC Temuco junto con gente de empresas.

UC Temuco: la investigación como elemento que fortalece la docencia. Una mirada desde la región

Oriel Herrera
Director Escuela de Ingeniería Informática, Universidad Católica de Temuco. Doctor en Ciencias de la Ingeniería, área Ciencia de la Computación y Magíster en Ciencias de la Ingeniería Pontificia Universidad Católica de Chile. Ingeniero Civil Industrial mención Informática, Universidad de la Frontera, Temuco.
oherrera@inf.uct.cl

Marcos Lévano
Profesor Asistente Escuela de Ingeniería Informática, Universidad Católica de Temuco. Magíster en Ingeniería Informática Universidad de Santiago de Chile. Ingeniero Informático Universidad Nacional de Trujillo.
mlevano@inf.uct.cl

La investigación y la docencia en Computación en una universidad regional, como lo es la Universidad Católica de Temuco (UC Temuco), tienen sus singularidades que pueden representar e interpretar el sentir y la realidad de sus símiles en otras locaciones.

Si nos centramos primeramente en el plano docente, el capital humano estudiantil lo conforma un grupo importante de estudiantes con bajo puntaje de ingreso, pero muchos de ellos con capacidades cognitivas altas, que simplemente no han tenido la oportunidad de desarrollarlas dadas las falencias del sistema educacional chileno que en parte margina a estudiantes provenientes de los quintiles sociales más bajos. Es por ello, que debemos hacernos cargo de la brecha de entrada que traen estos estudiantes, lo que conlleva desafíos mayores en comparación a las universidades que concentran los puntajes de ingreso más altos.

Uno de estos desafíos guarda relación con los aspectos metodológicos de cómo se aborda la docencia. Es por ello que la Universidad implementa a partir de 2007 un nuevo modelo educativo, que se adopta en la Facultad de Ingeniería a partir de 2010 [1]. Este nuevo modelo se centra en una formación basada en competencias. De este modo se definen competencias específicas y competencias genéricas que se incorporan a los distintos cursos de la malla curricular (ahora llamada itinerario formativo).

Ha sido muy interesante y satisfactorio poder trabajar en cada curso, aparte de las competencias específicas, las competencias genéricas asignadas, las que deben ser evidenciadas (aprobadas) por los estudiantes. Así, por ejemplo, a una asignatura le puede corresponder abordar la competencia genérica de creatividad e innovación, por lo que el profesor debe incluir en sus actividades elementos que permitan trabajar y desarrollar esta competencia.

LA INVESTIGACIÓN EN EL CURRÍCULO

Dado el escenario anterior, dentro de las múltiples iniciativas, se ha explorado la inserción de la investigación dentro del currículo de la carrera de Ingeniería Civil en Informática. El propósito es que las investigaciones tributen al desarrollo de las competencias específicas y genéricas.

Para ello, al final de cada año del itinerario formativo, se incluye una asignatura llamada Taller de Integración, donde los estudiantes deben integrar las competencias desarrolladas durante el año correspondiente. De este modo, cada segundo semestre, todos los estudiantes se encuentran cursando esta asignatura, con la exigencia y nivel de complejidad acorde a su avance en el currículo.

Dentro de este Taller los estudiantes deben desarrollar un proyecto grupal. Es aquí donde entran en juego las diversas investigaciones que realiza la Escuela.

Cada uno de los proyectos de investigación puede ser dividido en distintos aspectos, de modo que los proyectos de integración de los estudiantes consisten en tomar y desarrollar algunos de estos aspectos (ver Figura 1). Así por ejemplo, el Proyecto 1 de Integración de tercer año,

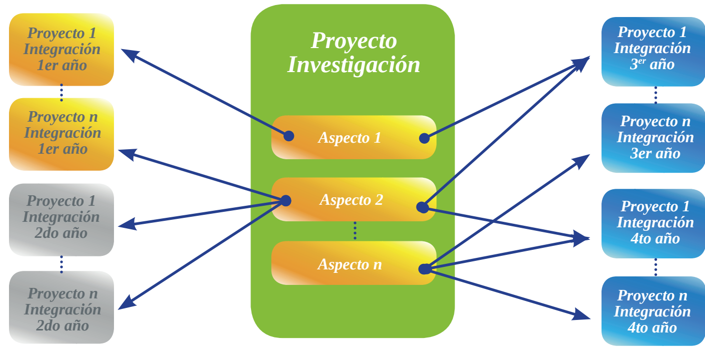
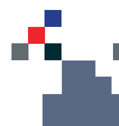


Figura 1 • Proyectos de Investigación divididos en distintos aspectos que son abordados por proyectos de estudiantes en el curso de Taller de Integración de cada nivel.



incluirá los aspectos 1 y 2 de un proyecto de investigación. Del mismo modo, el aspecto 2 de un proyecto de investigación puede estar tributando a cinco proyectos de integración de los estudiantes.

Los proyectos de investigación que han fortalecido la docencia son variados, abarcando desde proyectos con financiamiento externo, pasando por proyectos internos y hasta ideas incipientes de proyectos en formación que pasarán a formulación formal.

Algunos proyectos

Presentamos a continuación algunos de los proyectos de investigación y su aporte al desarrollo de la docencia.

Proyecto de investigación: Manipulación de modelos 3D de sondajes mineros y su visualización gráfica.
Profesor responsable: Alberto Caro.

Proyecto 1 • estudiantes

Gráfica de archivo de datos de sondaje mediante técnica de triangulación (sólido 3D), y su manipulación espacial mediante el uso de Wiimote. Dado un archivo CSV de sondaje con los datos espaciales, éste es leído desde un programa Python, el cual genera el modelo 3D que se traspasa a OpenGL para su posterior manipulación con el dispositivo Wiimote.

Áreas: Programación, Procesamiento de Imágenes, Microcontroladores, Interfaces Bluetooth, Física, Cálculo Vectorial.
Nivel: tercer año.

Proyecto 2 • estudiantes

Uso de Kinect para detección de gestos simples que permitan controlar un robot Lego NXT. El robot realiza diversos desplazamientos en base a los movimientos de manos detectados a través de un dispositivo Kinect.

Áreas: Programación, Visión por Computador, Robótica
Nivel: segundo año.

Proyecto 3 • estudiantes

Reconstrucción 3D de un sólido en revolución mediante visualización por Kinect. Un objeto se ubica sobre una plataforma que gira, controlada por un robot Lego NXT. Mediante la Kinect se le realiza un escaneo y, a través de un software en Python, las matrices recibidas son procesadas por el CAD/Script Blender, el cual reconstruye el objeto en 3D.

Áreas: Programación, Visión por Computador, Robótica, Procesamiento de Imágenes
Nivel: primer año.

Proyecto de investigación: Modelo de estudio virtual bajo plataforma Linux.
Profesor responsable: Alejandro Mellado.

Proyecto 1 • estudiantes

Modificación de WebCam Studio para personalización de entorno de control de televisión.

Áreas: Programación, eficiencia de algoritmos, Procesamiento de imágenes.
Nivel: segundo año.

Proyecto de investigación: Interpretación de datos provenientes de microarreglos mediante modelos flexibles de redes neuronales.
Profesor responsable: Marcos Lévano.

Proyecto 1 • estudiantes

Redes neuronales artificiales y aplicaciones.
Interpretación de datos provenientes de microarreglos mediante modelos flexibles de redes neuronales artificiales, liderado por el profesor Lévano que tuvo como propósito interpretar patrones en datos provenientes de expresiones de genomas. Mediante un proceso comparativo de modelos computacionales flexibles de máquinas de aprendizaje que imitan al cerebro humano, se descubre información útil para los expertos en el análisis de genes en microarreglos. Entre los modelos estudiados están mapas de Kohonen y las redes de Durbin. El proyecto vincula hitos que conciernen a distintas aplicaciones y desarrollos de algoritmos en reconocimiento automático.

Áreas: Inteligencia Artificial, Reconocimiento de Patrones.
Nivel: tercer año.

Proyecto 2 • estudiantes

Modelos de máquinas de aprendizaje para análisis ROC con foco en medir la capacidad de detección de patrones que dan las máquinas artificiales frente al problema de agrupamiento de datos. Se hacen comparaciones de eficiencia de máquinas. La importancia de esas técnicas en análisis de datos promete grandes aportes a la biología, la agricultura, datos masivos de periódicos, análisis de mercado entre otras, permitiendo las investigaciones de nuevos patrones de interés que inciden en cambios significativos para la toma de decisiones.

Áreas: Inteligencia Artificial, Reconocimiento de Patrones, Simulación.
Nivel: tercer año.

Proyecto de investigación: Diseño de redes de comunicación resilientes y sin infraestructura para apoyar asistencia a desastres de gran escala.
Profesores Responsables: Roberto Aldunate y Oriel Herrera.

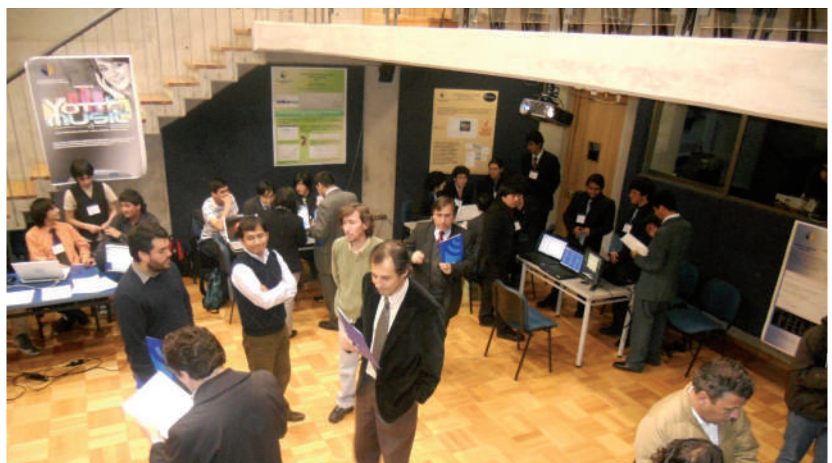
Proyecto 1 • estudiantes

Construcción de un prototipo de sistema distribuido de vehículos autónomos que colectivamente se orientan a mantener conectividad de la MANET formada por rescatistas.

Áreas: Comunicación de Datos, Robótica, Sensores
Nivel: cuarto año.

Desafíos futuros

Existen otros proyectos en desarrollo con potencial de vinculación a la docencia, que tienen un componente adicional que es la interdisciplinariedad, principalmente con el área de Recursos Naturales (agronomía, forestal, medio ambiente, acuicultura).



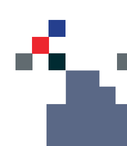
Los proyectos anteriores y los que vendrán potenciarán una acción en construcción, que es la generación de incubadoras de tesistas y estudiantes en general, para el desarrollo de aplicaciones. Recientemente con la incorporación de un nuevo Doctor se complementará la investigación y la docencia con el área de cómputo de alto desempeño y modelos de series de tiempo, robusteciendo y ampliando el espectro de áreas para postulación a proyectos.

La motivación de ser parte del desarrollo de proyectos, permite en el estudiante cubrir el logro de las competencias genéricas y específicas, acelerando el proceso de formación y disminuyendo rápidamente las brechas de formación académica con que llegan. BITS

Referencias

[1] Universidad Católica de Temuco (2008). Modelo educativo UCTEMUCO: principios y lineamientos, [en línea]. Ed. 1, Temuco: UCT, 2008. [ref. julio 2011] .
Web: http://www.uctemuco.cl/docencia/modelo-educativo/docs/modelo_educativo.pdf

Estudiantes exponiendo sus proyectos a invitados externos.



www.dcc.uchile.cl

Crea, desafía y transforma el mundo a través de la Computación.

Únete a nuestros programas de:

- Doctorado en Ciencias de la Computación
- Magíster en Ciencias de la Computación
- Magíster en Tecnologías de la Información
- Diplomas de Postítulo

Síguenos en:

https://twitter.com/dccuchile
https://www.facebook.com/DCC.UdeChile

www.dcc.uchile.cl/revista
revista@dcc.uchile.cl