Teoría de la computación

Germán Bordel

Departamento de Electricidad y Electrónica
Facultad de Ciencia y Tecnología, UPV/EHU
german.bordel@ehu.es

MSc BTC - UPV/EHU

Capítulos

PREFACIO
Teoría de la Computación

Teoría de la computación y Blockchain

Blockchain está basado en la generación de la confianza en que el registro de transacciones y las transmisiones de información están libres alteraciones, lo que se consigue mediante el uso de dos asimetrías:

- **Asimetría computacional**
  La Prueba de Trabajo (Proof of Work): la obtención del *nonce* supone un esfuerzo computacional tremendo (prueba y error) mientras que la comprobación de que un *nonce* dado es el valor correcto buscado es trivial.
- **Asimetría algorítmica**
  el uso de Criptografía de Clave Pública (Criptografía asimétrica) para firmar y asegurar los UTXOs (Unspent Transaction Outputs): son algoritmos en los que el nivel de esfuerzo para resolver un problema es muchísimo mayor que el necesario para comprobar que una solución lo es.

El uso de la asimetría algorítmica en Blockchain nos lleva interesarnos por la existencia de problemas inherentemente difíciles de resolver (o incluso por si los hay sin solución), y la relación de estos con aquellos cuyas soluciones son a su vez también más o menos difíciles de comprobar[a].

[a] También el asunto de si Bitcoin es Turing completo o no... ¿nos aclararemos?

Teoría de la computación

La Teoría de la Computación se ocupa de determinar /1/qué problemas se pueden resolver (o no), con el establecimiento de determinados /2/modelos de computación y en tal caso, /3/con qué eficiencia pueden resolverse, o en qué grado (por ejemplo, soluciones aproximadas versus precisas).

Esto nos lleva a dividirla en tres subáreas, yendo de lo más abstracto o teórico a lo más concreto:

/1/ Teoría de la **computabilidad**: qué problemas tienen solución.

/2/ Teoría de **autómatas y lenguajes formales**: un marco de modelización de la computación.

/3/ Teoría de la **complejidad computacional**: qué coste tiene resolver un problema.

El punto /3/ es la parte más práctica y más desarrollada del curso, impartida por L.J.R-F (CA- Complejidad de Algoritmos)
Ahora nos ocuparemos de /1/ y /2/ pero siguiendo en sentido inverso, yendo de este modo de lo más concreto a lo más abstracto.
Antes de esto, veremos porqué puede interesarnos esta teoría.

Teoría de la computabilidad /1/

La teoría de la computabilidad, también conocida como la teoría de la recursión, es una rama de la lógica matemática que estudia lo que podríamos considerar el entorno de la frontera entre lo computable y lo no computable, donde tenemos una infinidad de problemas abiertos, y la Máquina Universal de Turing es un resultado central y una herramienta principal.
Veremos únicamente una aproximación al entendimiento de la existencia de dicha frontera.

Teoría de Lenguajes Formales y Autómatas /2/

Los autómatas son el soporte de los modelos de computación, y están íntimamente ligados al lenguaje (a la lingüística).

Los distintos modelos de computación dan soporte teórico a las técnicas para afrontar problemas de cómputo de diversa complejidad, desde los triviales hasta los más complejos que pueden llegar a ser incluso intratables.

El conjunto de estos modelos tiene gran importancia a la hora de estructurar la disciplina de la computación, pero uno en concreto, la Máquina de Turing, es de una particular importancia porque determina la frontera entre lo computable y lo que no lo es. Es el soporte básico de la Teoría de la Computabilidad, que hemos considerado como la primera subárea de la computación.

Estudiaremos en primer lugar estos modelos generales, y posteriormente nos centraremos en el área de los "problemas difíciles", dentro de los cuales se encuentran los que presentan la asimetría utilizada en Blockchain.

Teoría de la Complejidad Computacional /3/

La complejidad computacional es la parte principal del curso, impartida por L.J.R-F

I - LENGUAJES Y AUTÓMATAS

Computar es procesar cadenas de símbolos

Cualquier problema, así como cualquier posible solución a un problema, se pueden transformar en una secuencia de símbolos.

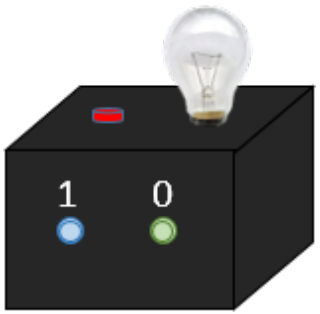
Sin ir a una rigurosa formulación matemática de la afirmación anterior, podemos intuir su validez con un pensamiento muy simple: no sabemos hacer otra cosa. Expresamos los problemas y sus soluciones por escrito, lo que no es otra cosa que expresarlos con secuencias de símbolos. Si nos ponemos rigurosos podríamos plantearnos la duda de si podemos encontrar un problema perfectamente caracterizable "de algún modo" pero imposible de describir con exactitud por escrito... ahí lo dejamos...

De hecho podemos codificar como una secuencia de símbolos (incluso en binario, con un alfabeto de sólo dos símbolos) los datos de entrada (p.ej. una lista de nombres a ordenar) y el resultado (la lista ordenada), de modo que nuestra computación es una función $f : \mathbb{N} \to \mathbb{N}$. [a]
Estas funciones, nuestros programas, también son texto, y por tanto números enteros, es decir, son "enumerables" lo que nos lleva a poder establecer siempre una relación de nuestra función f con otra $g : \mathbb{N} \to \{0, 1\}$, lo que nos lleva a hablar de "aceptores"...

[a] Pondremos un "1" como bit más significativo para dar sentido de número natural a todas las codificaciones que tengan ceros a la izquierda

Autómatas y Lenguajes Formales

Autómatas de estados finitos (AEFs)



Pensemos en una caja negra con un par de botones marcados como 0 y 1, otro que dice "reset", y una bombilla que lleva una etiqueta que dice "aceptado". No sabemos lo que hace, pero descubrimos que pulsando los botones en ocasiones se enciende la luz. Si asumimos que la máquina se encuentra en un *estado de equilibrio* cuando no pulsamos botones, y que ese estado interno *puede variar con las pulsaciones*, además de que el número de posibles estados será *finito*, la caja será un **Autómata de Estados Finitos**

En términos abstractos es:

Una colección finita de estados $\mathcal{Q}$ (los estados estables)
Un alfabeto finito Σ de símbolos (0 y 1 en nuestra caja negra)
Una función δ que para cada par (estado,símbolo) determina un nuevo estado.

(continúa...)

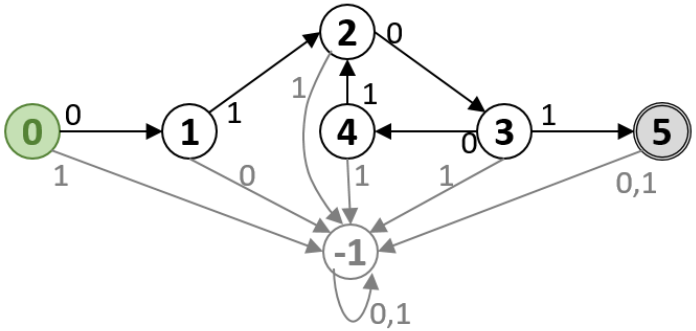
Dos tipos de estados son especiales:

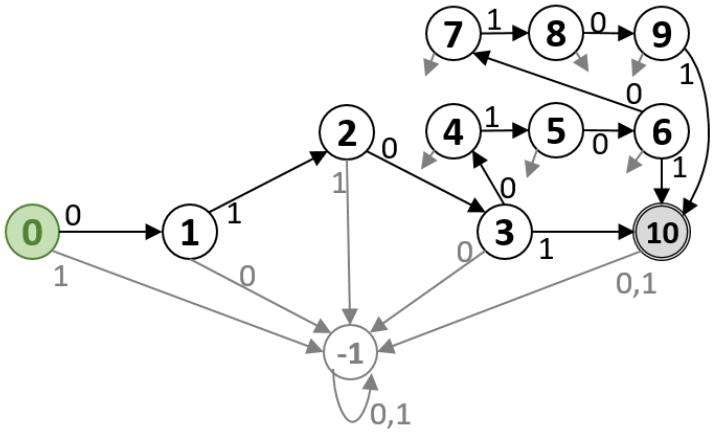
- Un estado inicial (al que llegamos con el botón "reset")
- Uno o varios estados finales, o de "aceptación" (que provocan el encendido de la bombilla).

Un AEF se dice que **"acepta"** toda secuencia de símbolos que lleva del estado inicial al final. El conjunto de secuencias aceptadas es el **"lenguaje"** modelado por el AEF, y denominamos **"palabras"** del lenguaje a dichas secuencias.

¿Podremos descubrir el lenguaje de nuestra máquina?

Dado que su número de estados es finito, parece razonable pensar que sí... no obstante, sólo probando combinaciones de entradas nunca podremos estar seguros. Supongamos que concluimos que el lenguaje es **01(001)*01**, donde * indica un número indefinido de repeticiones de la secuencia entre paréntesis. No podemos asegurar que después de 100 repeticiones el comportamiento no sea diferente. (continúa...)





Si disponemos de una cota máxima para el número de estados sí podremos determinar cual es el lenguaje de nuestra máquina

(si es p.ej. 6, bastará con comprobar las secuencias $01(001)^{\{0..5\}}01$.

Explorando sistemáticamente todas las posibilidades con dicha limitación, podríamos encontrar eventualmente otras secuencias (p.ej. (0110)*111), que al final nos llevarán determinar que el lenguaje es de la forma:

(para el ejemplo) $01(001)^*01 + (0110)^*111$

Como esta combinatoria es finita, determinaremos totalmente el lenguaje aceptado por nuestra máquina.

El lenguaje aceptado por un AEF es siempre una Expresión Regular, definida como:

- 0 y 1 son expresiones regulares.
- Si X e Y son expresiones regulares, su concatenación y su "suma" (X + Y) son expresiones regulares. (X+Y suele denotarse $X|Y$ en los lenguajes de programación)
- Si X es una expresión regular, también lo es su iteración X*.

  (los lenguajes de programación suelen proporcionar otros operadores que permiten hacer más concisas las expresiones. -aquí estamos definiendo-)

(continúa...)

Una expresión regular caracteriza con precisión el comportamiento de un autómata que la acepta, pero hay infinitos autómatas que determinan la misma expresión regular (podemos determinar el comportamiento de nuestra caja negra, pero no su estructura).

Los AEF son el modelo computacional más sencillo, y tienen muchas limitaciones (p.ej. no aceptan palíndromos).

Ejercicios:

- Construir el autómata para el lenguaje (ab)*(a+b)
- El lenguaje $\{a^x b^y c^z | y = x + z\}$ ¿puede especificarse como expresión regular?
- Argumentar (o demostrar) que no hay un AEF que acepte los palíndromos sobre un alfabeto dado.
- ¿Ser finito es condición suficiente para que un lenguaje sea regular?

Se pueden probar expresiones regulares online
(-Un texto para probar //incluso se puede obtener código para varios lenguajes-)

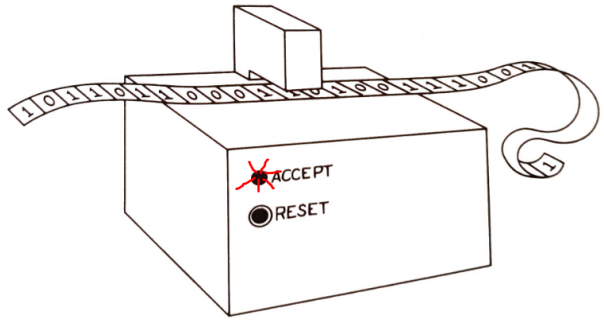
Para casa: Escribir rutinas (Python, Java,...) aceptoras (devolverán un valor booleano) de cadenas de caracteres que sean

- expresiones aritméticas con las 4 operaciones básicas (+-*/) entre enteros (p.ej. 12+4-234*89*7*2/34)
- números reales (p.ej. -73.001E-23)

Modelos computacionales

Se han teorizado muchos modelos de computación, pero todos resultan ser equivalentes a una de cuatro clases:

- Autómatas de estados finitos
- Autómatas con pila (pushdown automata)
- Autómatas linealmente acotados (linear bounded automata)
- Máquinas de Turing



Los cuatro modelos pueden ser vistos como variantes de la máquina de la figura, una generalización de la que hemos visto anteriormente.
La máquina encierra un mecanismo con estados estables, y responde (en el caso más general) al símbolo detectado en su cabezal con una re-escritura del mismo y un movimiento de una posición a derecha o izquierda[b]. Los símbolos en la cinta pertenecerán a un alfabeto finito.

[b] Nuestra máqina **AEF** será esta misma con la restricción de que **el movimiento se realiza siempre en el mismo sentido**. Esto es equivalente a introducir series de símbolos. Si la máquina escribe en la cinta no varía el lenguaje aceptado, pero en tal caso hablamos de una Máquina de Mealy.

La jerarquía de Chomsky

Cada uno de los cuatro tipos de autómata que hemos mencionado es capaz de aceptar un tipo de lenguaje, donde cada uno es más general que su predecesor

En 1956, Avram Noam Chomsky llegó, desde la lingüística, a esta clasificación de lenguajes de acuerdo con la potencia generadora de las gramáticas que los especifican:

Generalidad creciente $\rightarrow$

Modelo de computación	Autómatas de estados finitos	Autómatas con pila	Autómatas linealmente acotados	Máquinas de Turing
Clase de lenguaje	(LR) Lenguajes Regulares	(LIC) Lenguajes Independientes del Contexto	(LSC) Lenguajes Sensibles al Contexto	(LRE) Lenguajes Recursivamente Enumerables

Una gramática es $G = (V, T, P, S)$ con V cjto. de no terminales, T cjto. de terminales, P cjto de producciones, S símbolo de inicio.

ej. LR: identificadores (letra seguida por letras y números)
alfabeto: {letras}+{dígitos}
ID $\rightarrow$ letra **RESTO** (tantas reglas como letras)
RESTO $\rightarrow$ letraOdigito **RESTO** (tantas reglas como letras y digitos)

ej. LIC: sentencias... (p.ej. WHILE) [a]
WHILE $\rightarrow$ while (**COND**) **SENTENCIA**
COND $\rightarrow$ **BOOL** | **EXP OP EXP** | ...
SENTENCIA $\rightarrow$ ...

LR (tipo 3)	LIC (tipo 2)	LSC (tipo 1)	LRE (tipo 0)
$A \rightarrow a$ $A \rightarrow aB$	$A \rightarrow \alpha$	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$\gamma \rightarrow \alpha$
(ej. $L = \{a^*\}$)	(ej. $L = \{a^{*n} b^{*n}\}$)	(ej. $L = \{a^{*n} b^{*n} c^{*n}\}$)	(ej. cualquiera)

con a=terminal; A, B=no terminales; α, β, γ=cadenas de terminales y no terminales (α y β posiblemente vacías, y γ nunca vacía).

(La cadena vacía se representará con el símbolo terminal ϵ)

[a] Python usa *INDENT* y *DEDENT* para convertirse en Tipo 2 antes de ser "parseado"

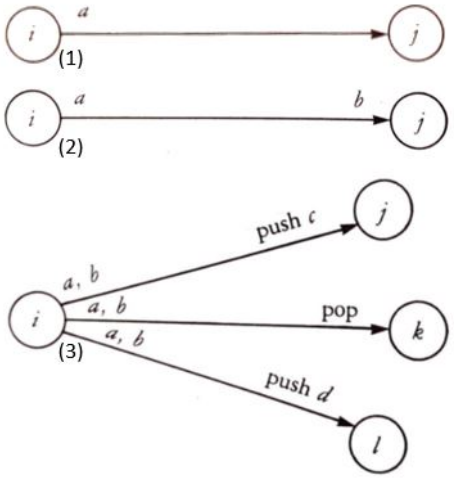
Volvamos sobre los ejercicios propuestos, pero ahora pensando en gramáticas.

Ejercicios:

- Construir las gramáticas por la derecha y por la izquierda para el lenguaje (ab)*(a+b)
- El lenguaje $\{a^x b^y c^z | y = x + z\}$ ¿puede especificarse con una gramática propia de un LIC?
- Construir la gramática que acepte los palíndromos sobre un alfabeto dado.

Autómatas con pila

Los Lenguajes Regulares son aceptados por Autómatas de Estados Finitos, como hemos visto. Sus transiciones son del tipo 1 en la figura (o de tipo 2 en el caso de las máquinas de Mealy).



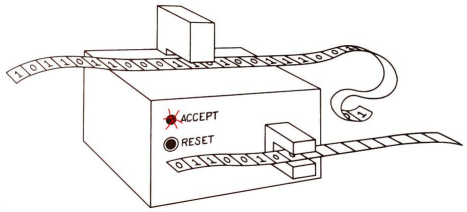
Los Lenguajes Independientes del Contexto son aceptados por Autómatas con Pila[a]. Sus transiciones son del tipo 3 en la figura. Hemos de fijarnos en dos circunstancias:

- las transiciones están marcadas por dos símbolos de salida del estado origen y por dos tipos de acciones en la llegada al destino (hablamos de ello a continuación).
- un mismo par de símbolos de salida del origen lleva a diferentes estados destino. No hay un destino determinado (hablamos de indeterminismo). Volveremos sobre esto más adelante.

(continúa...)

[a] La pila va a permitir lo que desde el lenguaje se denomina "dependencias anidadas".

El modelo de caja negra del Autómata Con Pila (en la figura) no encaja exactamente, de entrada, con el modelo general que hemos visto en el apartado "Modelos computacionales", donde la máquina sólo procesa una cinta con símbolos. En realidad este modelo es una representación de conveniencia: los lenguajes que acepta son también aceptados por máquinas más generales en las que volveremos al modelo original



Esta máquina lee la cinta moviéndose **siempre en la misma dirección** (como el AEF) y **manejando la cinta secundaria como una pila**[a], es decir con operaciones de lectura y escritura fijas: "pop" lee la cinta y retrocede, "push" avanza y escribe. En la situación inicial -de reset- esta cinta no contiene información (la secuencia a aceptar se encuentra en la cinta principal), y en la final -de aceptación- se encontrará en la misma posición e igualmente sin información[b]. Se trata de una "memoria de operación".

Obviamente esta máquina puede hacer todo lo que un AEF. Basta con que el manejo de la cinta auxiliar sea intrascendente (p.ej. hacer siempre push)

[a] Estructura de datos fundamental a todos los niveles en nuestros sistemas de computación reales. Es manejada incluso al más bajo nivel por los microprocesadores para posibilitar la existencia de las subrutinas

[b] Aunque esto no sea una condición necesaria a priori.

Autómatas Linealmente Acotados y Máquinas de Turing

Tanto los Autómatas Linealmente Acotados como las Máqinas de Turing comparten un mismo esquema de transición:

El funcionamiento es idéntico, y la diferencia estriba en que los primeros son versiones limitadas de la Máquina de Turing, el modelo computacional más general conocido. Su limitación consiste en que la longitud de la cinta es acotada por un valor K que determina que para un problema de longitud N se dispone de capacidad en la cinta para $N \times K$ símbolos. N posiciones serán ocupadas por el problema y $(N-1) \times k$ serán un "espacio de trabajo".

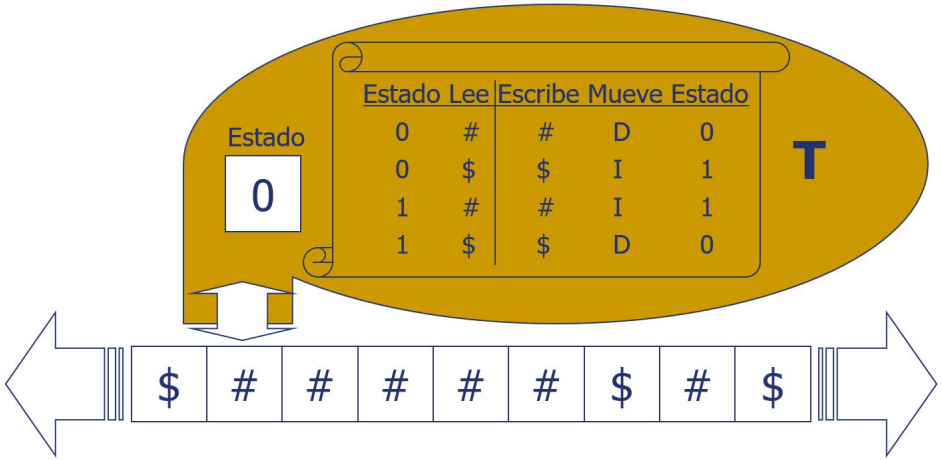
Que estas máquinas incluyen a los Autómatas a Pila no es tan sencillo de ver como que estos últimos incluyen a los AEF. La prueba pasa por ver que son capaces de simularlos usando su espacio de trabajo como pila[a,b].

Las máquinas de Turing trabajan frente a una cinta infinita.

(continúa...)

[a] De ahí que el modelo con una única cinta sea general.

[b] Démonos cuenta de que el uso de la pila esta acotado por la longitud del problema.



Una máquina de Turing puede definirse como una 7-tupla

$$M = (Q, \Sigma, \Gamma, s, b, F, \delta),$$

donde:

- Q es un conjunto finito de estados[a].
- $s \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados de aceptación (estadps finales que implican parada de la máquina).
- Σ es un conjunto finito de símbolos distinto del espacio en blanco, denominado alfabeto de máquina o de entrada[a].
- Γ es un conjunto finito de símbolos de cinta, denominado alfabeto de cinta ($\Sigma \subseteq \Gamma$).
- $b \in \Gamma$ (blanco) es el símbolo que ocupa las posiciones de la cinta sin información. Su cardinal es siempre infinito.
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ es la función (parcial) de transición, donde L indica izquierda y R derecha.

[b] Para un problema dado puede optarse por recodificaciones con diversos Σ, lo que afecta a la cardinalidad de Q. Como casos extremos, siempre puede usarse un Σ binario o un Q con sólo dos estados (este es un caso límite que exigiría infinitos símbolos, representando cada uno a una frase del lenguaje)

En la práctica...

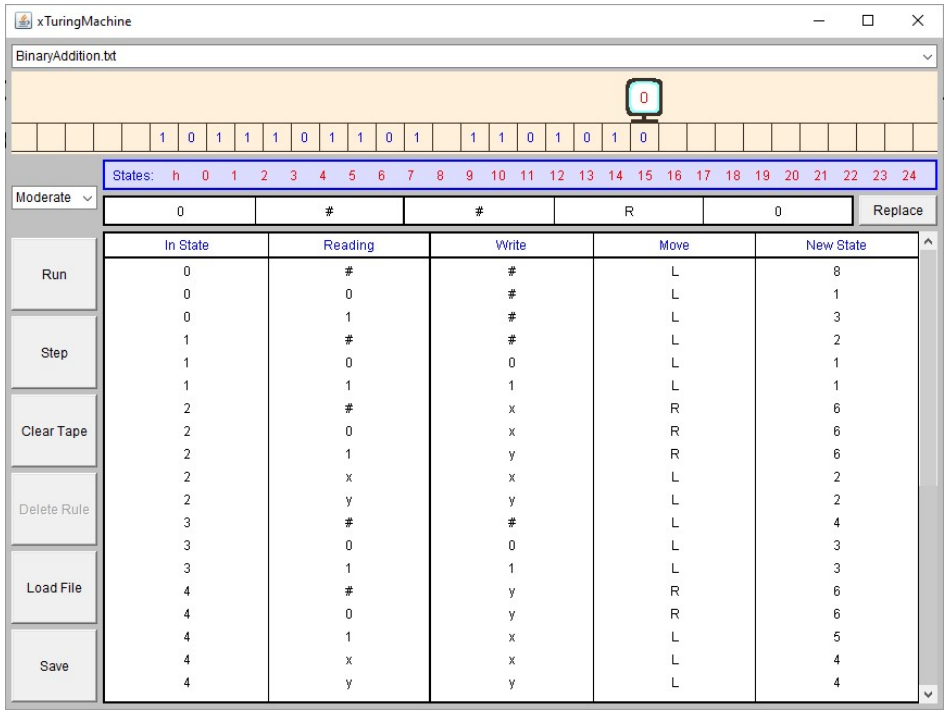
Los modelos de computación se utilizan en informática teórica como aceptores de secuencias de símbolos en el estudio de la pertenencia de diversos problemas a determinados lenguajes. Numerosos resultados teóricos permiten determinar si un lenguaje pertenece a alguno de los grupos vistos, aunque como hemos mencionado anteriormente hay aún muchas incógnitas sin resolver.

En muchas ocasiones se usan las Máquinas de Turing más como autómatas transductores que como aceptores, es decir, como mecanismos para recibir una secuencia de entrada y producir una de salida. En este sentido, una Máquina de Turing puede codificar en su estructura la mecánica de la solución de determinados problemas, de modo que enfrentada a su enunciado en la cinta obtengamos un resultado en la misma una vez se detenga.

En este sentido de transducción, el modelo de computación dado por la Máquina de Turing, que mencionamos como el modelo más general posible, supone que para cualquier problema (decidible) podemos "fabricar" una MT que lo resuelva (dada una entrada en la cinta, nos proporciona una salida en la misma). No hay que confundir esto con la Máquina Universal de Turing de la que hablamos más adelante.

Ejercicios

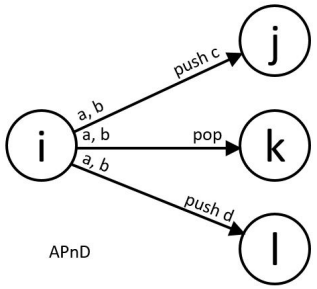
- Resolver algunos problemas simples usando el simulador de máquina de Turing (en la web del curso).

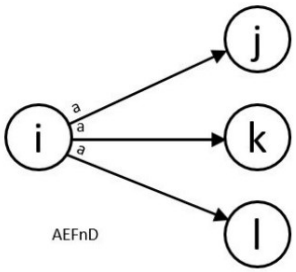


In State	Reading	Write	Move	New State
0	#	#	L	8
0	0	#	L	1
0	1	#	L	3
1	#	#	L	2
1	0	0	L	1
1	1	1	L	1
2	#	x	R	6
2	0	x	R	6
2	1	y	R	6
2	x	x	L	2
2	y	y	L	2
3	#	#	L	4
3	0	0	L	3
3	1	1	L	3
4	#	y	R	6
4	0	y	R	6
4	1	x	L	5
4	x	x	L	4
4	y	y	L	4

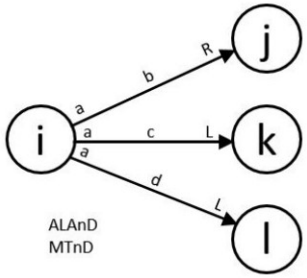
Indeterminismo

Al hablar del Autómata con Pila, mencionamos la posibilidad de que de un estado se pueda transitar con la misma combinación de símbolos (a, b) a varios estados diferenciados y con distintas acciones sobre la pila.





En realidad, podemos decir esto mismo para todos los autómatas vistos (pero no lo mencionamos con los AEFs sino con los Autómatas con pila, por lo que veremos a continuación.)



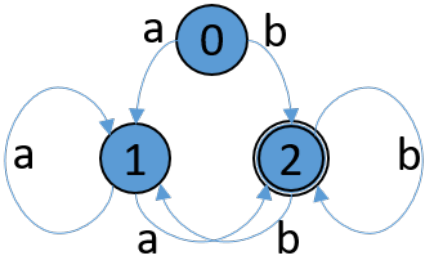
(continúa...)

No mencionamos el indeterminismo con los Autómatas de Estados Finitos porque para todo autómata indeterminista hay uno determinista equivalente.

- Ejercicio: Comprobar que todo AEF no determinista puede tener un equivalente determinista.

Podemos probarlo de un modo sencillo resolviendo un caso particular y determinando que el mecanismo establecido es general.

Puede hacerse con un caso simple como el de la figura.



- Ejercicio: ¿Cual es la expresión regular del lenguaje aceptado por el autómata?.

(continúa...)

Indeterminismo y Autómatas con Pila
No sucede lo mismo con los autómatas con pila. Los lenguajes reconocidos por autómatas a pila deterministas son un subconjunto de los de contexto libre, y se denominan consecuentemente "lenguajes de contexto libre deterministas".

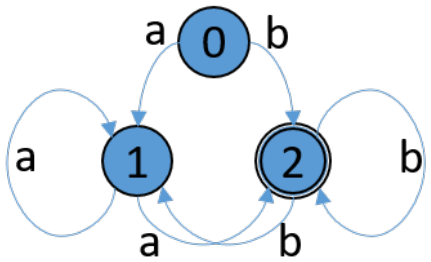
Indeterminismo y Autómatas Linealmente Acotados
En el caso de los autómatas linealmente acotados desconocemos si tenemos la posibilidad de obtener un equivalente determinista de todo modelo indeterminista, es decir, **no sabemos si unos y otros definen el mismo lenguaje**.

Indeterminismo y Máquinas de Turing
En el caso de las Máquinas de Turing sí que tenemos también la posibilidad de obtener un equivalente determinista de todo modelo indeterminista (con paciencia). Es por ello que estas máquinas, el modelo más general, se consideran deterministas.

(continúa...)

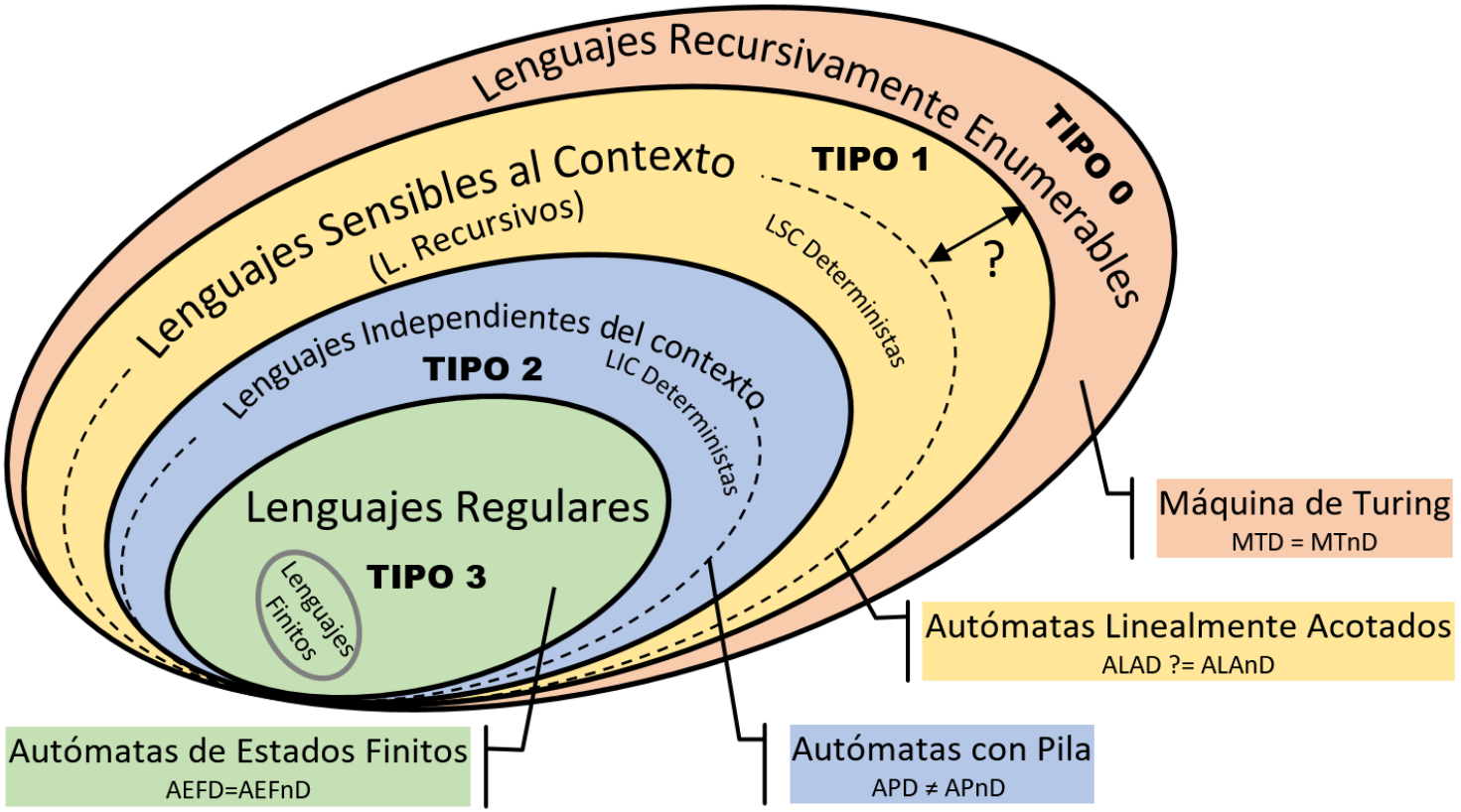
¿Cómo "funciona" un autómata indeterminista?

Cuando el autómata es indeterminista "se asume" que para secuencias del lenguaje toma siempre las transiciones que llevan al estado final de aceptación, mientras que para secuencias fuera del lenguaje dará igual las elecciones que realice, puesto que nunca llegará a un estado final de aceptación. (El llamado "algoritmo con suerte" -lucky algorithm-)



analizar el funcionamiento con "ab" y "aaba"

En resumen...



Tipo 3: Expresiones regulares: ascensores, máquinas de vending, semáforos, (PLCs en general), ...
Tipo 2: Lenguajes de programación (if (...) then ... else ...); en su mayor parte los lenguajes naturales.
Tipo 1: Lenguajes naturales en determinadas ocasiones ("Las raices cuadradas de 25, 16 y 9 son 5, 4 y 3, respectivamenete")
Tipo 0: Cualquier lenguaje estructurado por frases (los enumerables que pueden generarse recursivamente y no se pueden definir con reglas con contexto).

El Problema de la Parada

Los autómatas y lenguajes formales, son el soporte teórico que permite asociar los distintos problemas computacionales con distintas clases de dificultad y las herramientas adecuadas para resolverlos (p.ej. los "parsers" para procesar textos que se ajustan a gramáticas independientes de contexto: programas en lenguajes informáticos, documentos HTML, o XML, ...)

Conforme pasamos de los casos más simples hacia los más complejos los problemas son más difíciles de especificar y de tratar, disponiendo de menos estrategias y herramientas, hasta llegar al punto, con los Lenguajes Recursivamente Enumerables, en que podemos encontrar "cualquier cosa".

¿Cualquier cosa? ¿Y podremos resolver "cualquier cosa"?

Hasta que Kurt Gödel dijo lo contrario, en 1930, pensábamos que sí. Pero el caso es que hay problemas que no tienen solución.

Que esto es así, lo reafirmó Alan Turing con su máquina en 1936, estudiando el problema que denominamos "Problema de la Parada"

(continúa...)

El enunciado del Problema de la Parada es el siguiente:

¿Podemos determinar a priori si una maquina de Turing cualquiera[a] terminará por resolver un problema cualquiera que le presentemos?

Enunciados equivalentes:

- ¿Parará una MT cualquiera al ponerla en marcha frente a la cinta vacía?
- ¿Resolverá una MT cualquiera al menos un problema?
- ¿Representa una MT cualquiera una aplicación de N en N?
- Dadas cualesquiera dos MTs, ¿realizan la misma función?

Otros enunciados equivalentes "en el mundo real"

- ¿Una ejecución de un programa cualquiera se quedará sin memoria antes de terminar?
- ¿Cuánto debemos esperar para estar seguros de que un programa cualquiera no terminará dando una solución?

(continúa...)

[a] Constantemente decimos "cualquiera". No se trata de resolver para una máquina concreta, sino de tener el mecanismo capaz de analizar cualquier máquina y dar respuesta a la pregunta.

Sean

- m el número código de la maquina de una Turing T cualquiera[a].
- n el número código de un problema cualquiera para la máquina T

Hagamos la hipótesis de que existe H, una máquina capaz de vaticinar si (m, n) termina, es decir, H resuelve el problema de la parada.

Construyamos una MT K que admita un "problema" p, mediante la composición por concatenación de $K1, K2, K3$, tres MTs tales que:

- $K1$ duplica p
- $K2$ resuelve si hay parada para (p, p) (es la máquina H)
- $K3$ actúa inversamente al resultado de $K2$ (si $K2$ determina que (p,p) para, $K3$ entra en un ciclo infinito, y si $K2$ determina lo contrario, $K3$ va a un estado final)

Si damos a K su propio número código k llegaremos a una contradicción. Lo único que puede "fallar" es la hipótesis de que H exista, luego el Problema de la Parada es un problema sin solución[b].
Turing habla de números computables y no computables

[a] Todo el desarrollo teórico de Turing es aritmético (como lo era el de Gödel) ... un problema es un número, cosa que es trivial una vez que hemos aceptado que todo problema es expresable como secuencia de símbolos.

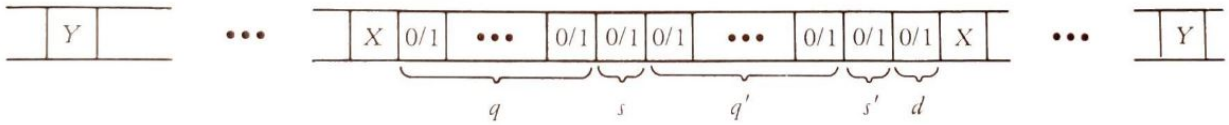
[b] Podemos ver esto sobre la MT o, alternativamente, como rutinas de código en un lenguaje de programación cualquiera.

La Máquina Universal de Turing

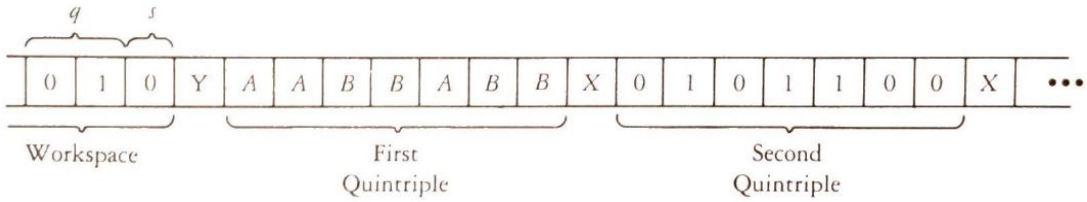
La Máquina Universal de Turing

La Máquina Universal de Turing es una máquina de Turing cuya capacidad consiste en simular cualquier máquina de Turing.

- Trabaja con la cinta dividida en dos secciones que contienen:
  - La entrada "problema" que debe resolver la máquina simulada.
  - La descripción de la máquina a simular
- Su tabla expresa dos fases de funcionamiento
  - Dado el estado y símbolo de entrada, localizar la quíntupla correspondiente en la descripción de la máquina.
  - Ejecutar en función de la descripción, sobre la cinta problema.



Reglas de una MT con alfabeto binario en la cinta de una MUT



La máquina anterior en funcionamiento.

Completitud de Turing

El término "completitud de Turing" es aplicable a cualquier sistema de manipulación de datos que puede ser utilizado para simular una máquina cualquiera de Turing. Es aplicable por tanto a conjuntos de instrucciones en procesadores, lenguajes de programación, etc. En el caso de los sistemas físicos de cómputo suele hablarse de sistemas "Turing Equivalentes" (son sistemas equivalentes aquellos que pueden simularse uno a otro).

Hay una condición que no pueden cumplir las computadoras reales por el mero hecho de ser "reales": su almacenamiento no puede ser infinito. En realidad sólo tendremos máquinas "linealmente acotadas completas", si bien el valor de K puede ser suficientemente elevado para permitirnos la licencia de hablar de Completitud de Turing.

En realidad es un término teórico de aplicación poco habitual en sistemas reales puesto que nuestras máquinas de cómputo y sistemas de programación son ampliamente Turing equivalentes o completos. Basta, por ejemplo, con disponer de una instrucción de salto condicional en un conjunto de instrucciones y la capacidad de interaccionar libremente con la memoria[a]. (Volveremos, eventualmente, un poco sobre esto en el capítulo último)

(continúa...)

[a] Como curiosidad, el "Juego de la Vida de Conway" es Turing Completo (https://playgameoflife.com/)

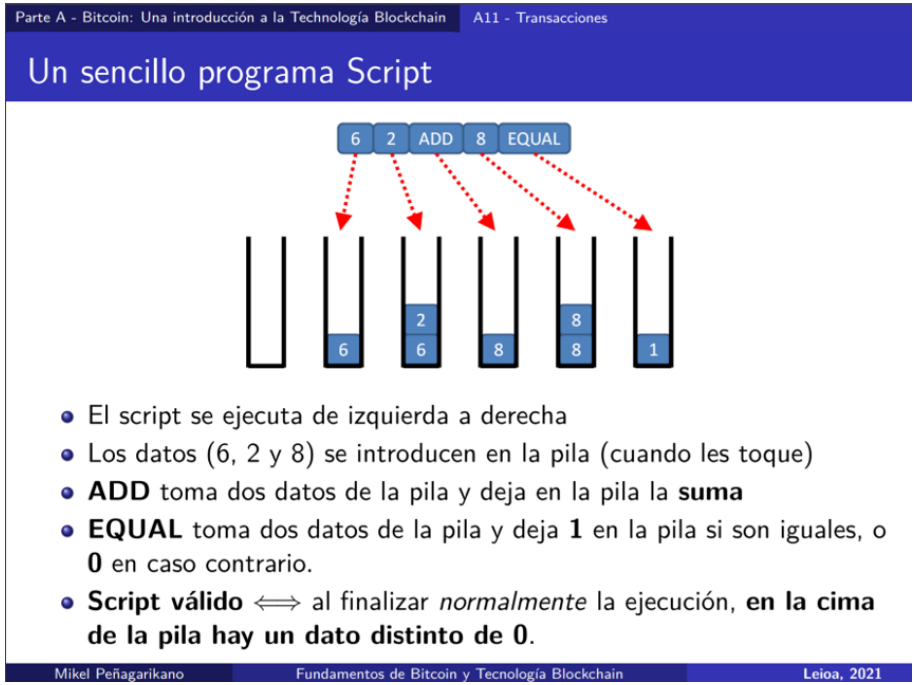
Completitud de Turing en Bitcoin

Al parecer hay cierta controversia sobre si el lenguaje Script de Bitcoin es Turing completo o no.

En la medida en que está construido por diseño como un mecanismo de ejecución sin ciclos (se procesa en una dirección única), que trabaja frente a una memoria de almacenamiento gestionada como una pila, estamos en condición de afirmar que no es Turing Completo. Ni siquiera será un lenguaje dependiente de contexto

Esta funcionalidad limitada sería un mecanismo de seguridad preventiva al no permitir programar rutinas sofisticadas que den pie a la búsqueda de debilidades. Y es un punto en que otras iniciativas basadas en Blockchain pueden ofrecer más flexibilidad, como es el caso de Ethereum, que es programable mediante lenguajes muy similares a los de propósito general y por tanto Turing Completos.

(continúa...)



Estamos un poco "lejos" de la definición formal de autómata con pila, pero a falta de una demostración en toda regla, podemos "intuir" que esto tiene una relación muy directa. . .
. . . pero podemos hacernos una idea:

Notación: estado(cinta, pila)-¿(acción, estado) con #N=número cualquiera (pero en concreto N)

S0(#6,*) → (push(#6),S0);
S0(#2,*) → (push(#2),S0);
S0(ADD,#2) → (pop(),Sadd#2);
Sadd#2(#8,#6) → (pop(),S#8&#8);
S#8&#8(EQUAL,*) → (push(1),SF);

La imagen de arriba se corresponde con un procesamiento capaz de hacer 2 pop y un push en un paso, que ya no es propio de un Autómata a Pila pero, como vemos, es equivalente si expandimos los estados.

II - PROBLEMAS DIFÍCILES (HARD)

Problemas tratables e intratables

Los problemas se clasifican como más o menos difíciles en función del mejor algoritmo que conozcamos para resolverlos.

En consecuencia, la clasificación no es definitiva mientras no exista una prueba formal que demuestre la no existencia de un algoritmo mejor que el mejor algoritmo disponible.

Tomamos como métrica su coste computacional (el del mejor algoritmo), es decir, siempre estamos hablando del crecimiento de la dificultad con el crecimiento del volumen de los datos de entrada. Podemos enfrentarnos a problemas considerados "muy costosos" y resolverlos sin problema si el tamaño de la entrada es suficientemente pequeño.

Independientemente de que podamos "afinar" en una clasificación de niveles de dificultad más detallada, todo lo que sigue tiene como criterio básico considerar como problemas tratables los de coste polinómico (o inferior) e intratables los de mayores costes (básicamente $O(c^n) c > 1$, y $O(n!)$).

Un caso paradigmático

Una compañía concreta de logística opera 91.000 camiones en EEUU. Cada día, cada camión sale de un almacén y reparte paquetes por numerosas empresas y viviendas de particulares. Al parecer un mapeo de rutas para reducir giros a la izquierda supuso una reducción de 466.000 millas en año y medio, casi 200.000 litros menos de combustible y 506 toneladas métricas menos de dióxido de carbono arrojadas a la atmósfera. Grandes beneficios por una simple planificación de las rutas.

- Sería más eficaz planear directamente las rutas más cortas: un camión repartirá paquetes en N puntos, que con el almacén suman N+1; una tabla de (N+1)x(N+1) conteniendo todas las distancias entre puntos puede servir para encontrar el circuito más corto que resuelva el reparto . No es difícil escribir el programa que pruebe todos los posibles circuitos y seleccione el más corto. Pero... **¡es difícil ejecutarlo!**
- El problema es que el número de rutas es inmenso: $N!$. El camión parte del almacén y pude ir a N puntos, de ahí a N-1, con lo que hemos de probar $Nx(N-1)$ posibilidades, y así acabamos finalmente con $N!$. El número de caminos crece con N más que exponencialmente[a]. Los camiones de la compañía reparten una media de 170 paquetes por viaje, aunque repartan varios en un mismo punto, podemos pensar en una estimación mínima de 20 paradas, lo que supone 20! posibles circuitos, o lo que es lo mismo: 2.432.902.008.176.640.000 posibilidades a considerar. Evaluando 10^{12} posibilidades por segundo se tardaría 28 días en resolver el problema (y tendría un coste en computación mucho mayor que el ahorro a conseguir en el reparto)

[a] La verdad es que este problema no es $O(n!)$, puede resolverse en $O(c^n)$ mediante programación dinámica. Es aplicable el Principio de Optimalidad de Bellman. Un buen ejemplo para distinguir entre complejidad del problema y complejidad del algoritmo.

El problema anterior se conoce como **el problema del viajante** (TSP-traveling-salesman problem)[a], y no se conoce ninguna solución de coste polinómico por mucho que elevemos la potencia del mismo ($> O(n^c)\ \forall c$).

Y lo que es peor... no se trata de un problema aislado... hay miles de este tipo (en lógica, grafos, aritmética, planificación,...).[b]

Si esto es frustrante, más lo es saber que estos problemas comparten una característica: si se encuentra un algoritmo de complejidad $O(n^c)$ para resolver uno de ellos, tiene que haber algoritmos con la misma complejidad para todos.

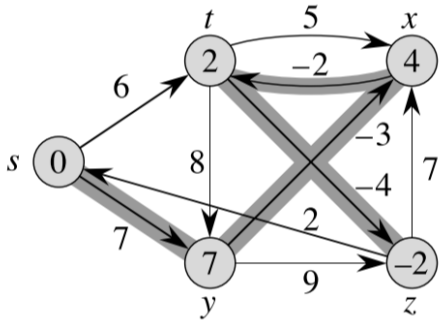
Nadie ha encontrado una solución polinómica a ninguno de ellos, pero tampoco se ha demostrado que no exista. Muchos de estos problemas son conocidos desde hace tiempo, pero es desde los años 70 que surge el concepto que los relaciona y comienza una preocupación especial por ellos, efectivamente orientada a encontrar soluciones polinómicas o a demostrar la imposibilidad de encontrar tales soluciones.

(continúa...)

[a] En concreto hemos planteado la versión TSP-OPT: la búsqueda de la solución óptima al problema del viajante.

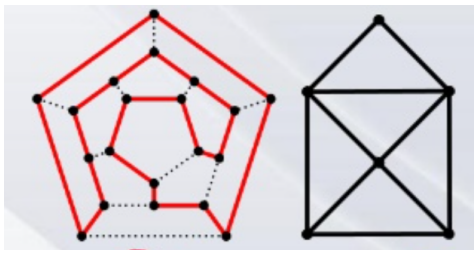
[b] Un conjunto del que no desvelamos por ahora cómo lo denominamos.

Y si decíamos que era "inquietante" saber que son muchos los problemas de este tipo, no lo es menos saber que muchos de ellos se parecen bastante a otros de coste polinómico.



El camino más corto ente dos puntos de un grafo dirigido se obtiene muy eficientemente con el algoritmo de Bellman-Ford ($\Theta(nm)$ para n vértices y m lados). Sin embargo la simple obtención del camino más largo (acíclcico) es NP-Completo (incluso saber si existe un camino de longitud $> L$).

Saber si existe un tour Euleriano en un grafo no dirigido (recorrer todos sus lados pasando una sola vez por cada uno) es trivial (es necesario y suficiente que todos sus vértices tengan grado par), Sin embargo encontrar un ciclo Hamiltoniano (pasar por todos sus vértices una sola vez) no tiene solución polinómica -y es una especie de simplificación de nuestro problema del viajante-



Así como el problema de la ordenación, el problema del camino más corto es un buen ejemplo de problema con muchas soluciones con diferentes costes.

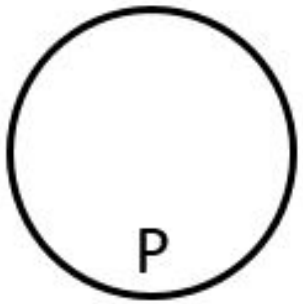
Clases P y NP, y NP-completo

Clase P

Un problema es de **clase P** si puede ser resuelto en tiempo polinómico o menor. Bajo el punto de vista que vamos desarrollar en este capítulo, eso significa ser un problema "tratable".

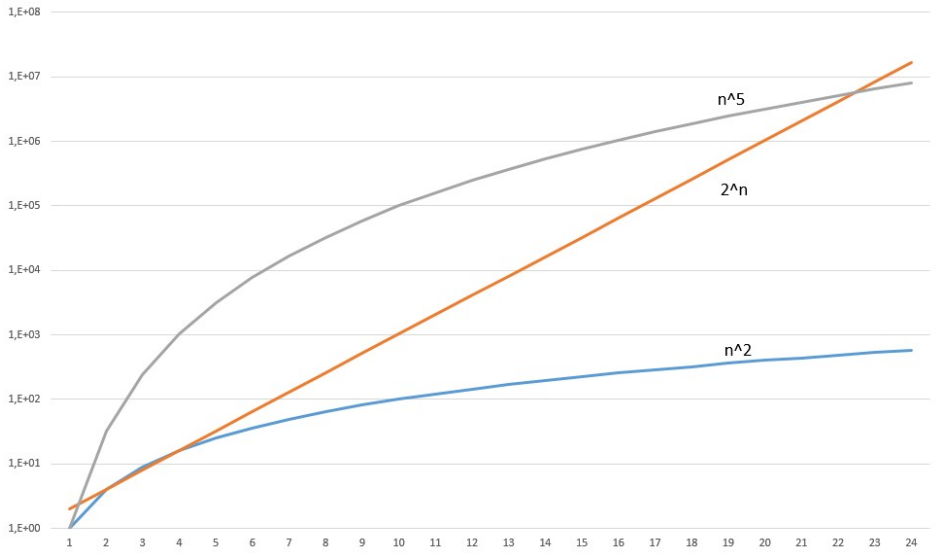
Un problema que requiere una solución de coste, p. ej., $\Theta(n^{100})$ no deja de ser de clase P por mucho que se trate de un coste elevado (1 googol para $n = 10$). Su coste, para valores grandes de n, siempre será menor que el asociado a otras funciones de coste de mayor crecimiento. A la hora de la verdad, no es normal encontrar costes superiores a $O(n^5)$ (por poner una potencia razonable), y cuando algo así sucede suele ser cuestión de tiempo que alguien encuentre una solución que rebaje ese exponente.

Ejemplos evidentes de costes polinómicos de diversos grados: suma, producto, de vectores, matrices.



Una reflexión sobre la clasificación que comenzamos a ver y el término "tratable".

Ojo, que estamos hablando de "Complejidad Computacional" tal y como se define, y que "tiene la relación que tiene" con la dificultad real.



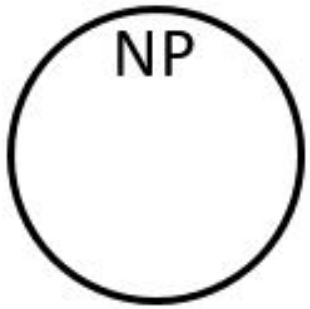
Quizás siempre sea mejor un algoritmo polinómico que uno exponencial, pero un algoritmo polinómico puede ser bastante "intratable" en sentido "real".

Clase NP

Un problema es de **clase NP** si puede ser resuelto en tiempo polinómico mediante un mecanismo no determinista[a]. NP es "**Nondeterministic Polinomial**".

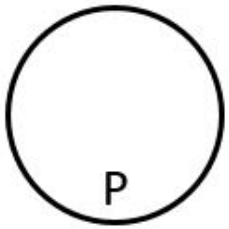
Esto es equivalente a decir que un problema es NP si sus soluciones son verificables en tiempo polinómico por un mecanismo no determinista. Esto es fácil de entender si sabemos que la solución de un problema de modo indeterminista supone seguir todos los caminos posibles en cada paso y seleccionar la primera alternativa que llegue a la solución; es "acertar" en cada paso y por tanto similar a comprobar una solución.

Se denomina "**certificado**" a una solución propuesta del problema, y para que éste sea NP, el certificado debe poder verificarse en tiempo polinómico respecto al tamaño de la entrada y del mismo certificado.

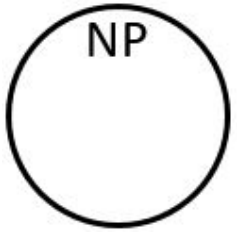


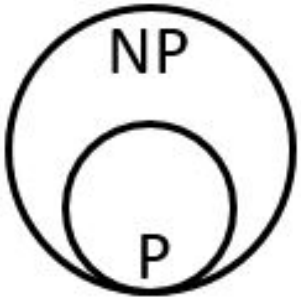
[a] Así que... lo desvelamos ahora,... en el apartado anterior debimos haber dicho que un problema es P si puede ser resuelto en tiempo polinómico mediante un mecanismo determinista (Y quien dice un mecanismo dice una Máquina de Turing).

Clase NP



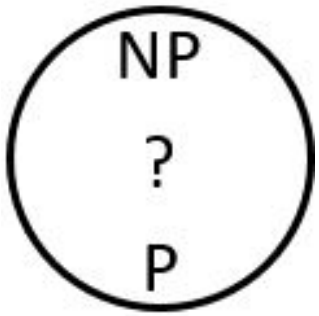
P y NP son dos conjuntos definidos independientemente.





Obviamente si un problema es de clase P, lo será también de clase NP: no puede ser más costosa la verificación de un certificado que su obtención. Osea que todos los problemas P son NP.

El caso contrario no tiene porqué ser cierto de entrada, pero tampoco podemos asegurar que no lo sea. Este es uno de los problemas más importantes de la ciencia de la computación: **el problema ¿$P = NP$?** (volvemos enseguida a él para justificarlo)

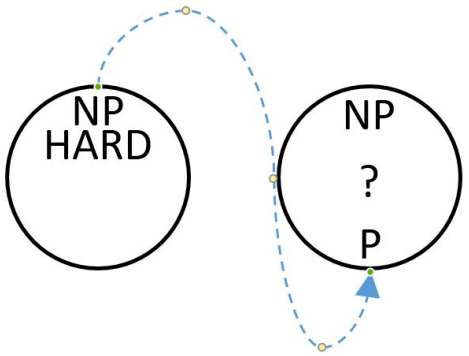


Clase NP-Hard

Definimos un conjunto más de problemas: los NP-HARD.
Son aquellos que cumplen con lo siguiente:

- Si existiera un algoritmo de coste polinomial para este problema, entonces podría convertirse[a] todo problema en NP en este problema, y de este modo todo NP sería resoluble en tiempo P respondiendo afirmativamente al problema $¿P = NP?$[b]

Estos problemas son al menos tan difíciles de resolver como los más difíciles en NP.



[a] lo que se conoce como "reducción"

[b] La situación contraria, que ningún problema NP-Hard tuviera solución polinómica, también resolvería el problema $¿P = NP?$ concluyendo que $P \neq NP$.

Clase NP-Completa

Los problemas de la clase **NP-Completa** son los que cumplen dos condiciones:

- Ser NP.
- Ser NP-HARD

Los problemas de la clase **NP-Completa**, al ser NP-Hard, son, como hemos dicho, al menos tan difíciles como los más difíciles de la clase NP. Forman una clase de equivalencia... son todos ellos reducibles de unos a otros.

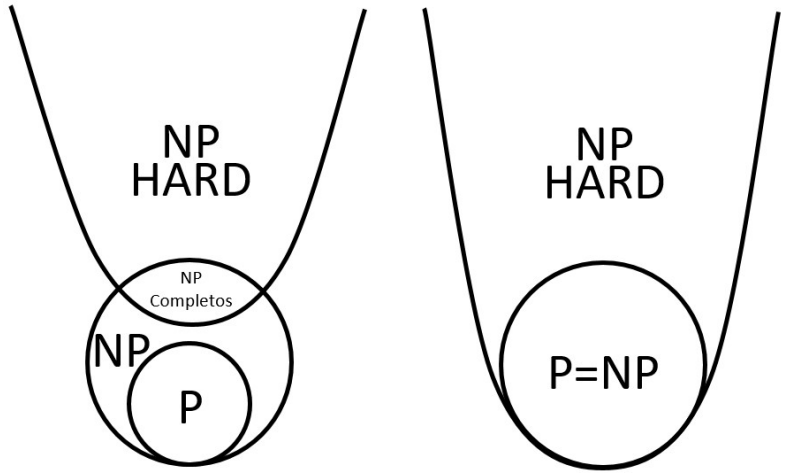
Este conjunto es particularmente interesante porque siendo todos sus elementos equivalentes, incluye a un buen número de problemas muy estudiados en la literatura (p.ej. los que hemos planteado al comienzo del capítulo: el Problema del Viajante[a], el camino más largo entre vértices de un grafo, y los ciclos hamiltonianos)[b]

[a] no en su versión de optimización, sino en el denominado de "búsqueda" -búsqueda de un ciclo de coste menor a k-, o el correspondiente de decisión.

[b] Lista de problemas en esta categoría

El problema ¿P=NP?

El problema ¿P=NP? es uno de los "7 problemas del milenio" establecidos por el Instituto Clay de Matemáticas ($1M por la solución de cualquiera de ellos[a])



Parece lo más razonable que $P \neq NP$, pero al no estar demostrado no podemos afirmar que nunca se "descubra" un algoritmo concreto para un problema concreto, que desencadene una revolución, puesto que no pocos sistemas tecnológicos están basados en problemas NP o NP-Hard.

Otras clases

Is Elliptic Curve Discrete Logarithm Problem NP-Hard or NP-Complete

Ask Question

Asked 1 year, 8 months ago Active 1 year, 8 months ago Viewed 215 times

2

I have trouble classifying Elliptic Curve Discrete Logarithm Problem as NP-Hard or NP-Complete. Where does ECDLP belong? Any brief comprehensive answer is encouraged. Thanks.

elliptic-curves cryptography np-complete discrete-logarithms

1

share cite improve this question follow

edited Mar 13 '19 at 22:56
Henno Brandsma
181k ● 6 ■ 71 ▲ 185

asked Mar 13 '19 at 7:15
alyssaeliyah
203 ■ 2 ▲ 9

I don't think anyone knows.... – Angina Seng Mar 13 '19 at 7:19

@LordSharktheUnknown Why, please enlighten me? – alyssaeliyah Mar 13 '19 at 7:27

2 It's unknown. You'll be famous if you can prove it to be NP hard or NP complete... – Henno Brandsma Mar 13 '19 at 16:46

@Why is it unknown? Why is it not NP-Hard? – alyssaeliyah Mar 14 '19 at 2:10

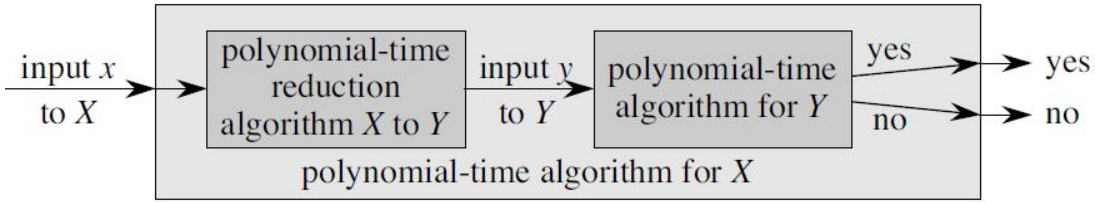
Because there is no proof that it is or is not. There are many open problems in this area. Cryptography is based on ignorance. – Henno Brandsma Mar 15 '19 at 18:19

El "Problema de la Curva Elíptica del Logaritmo Discreto" es un problema fuera de las clases conocidas.

Problemas de decisión y reducciones

Cuando hablamos de clases P y NP, se hace en términos de "problemas de decisión". La salida es un único bit: "si" o "no". p.ej ¿tiene el grafo un recorrido de Euler?, o ¿tiene el grafo un ciclo hamiltoniano?.

Pero nuestros problemas son más frecuentemente de optimización, de búsqueda de la mejor solución.
A menudo podemos enlazar unos problemas con otros. Por ejemplo: para saber cual es la longitud del camino más corto en un grafo, podemos plantear el problema ¿Contiene el grafo un camino más corto que k? que sirve para ir acotando k (k=1,2,4,8,16,32,24,20,22,23) y obtener la solución al problema original.



Si aplicamos una "reducción polinomial" del problema X a un problema polinómico Y, el resultado es también polinómico. El algoritmo de reducción tardará un tiempo $O(n^c)$ y su salida no puede ser más larga que el tiempo que tarda, de modo que si el coste de Y es $O(m^d)$ la resultante es $O((n^c)^d) = O(n^{cd})$

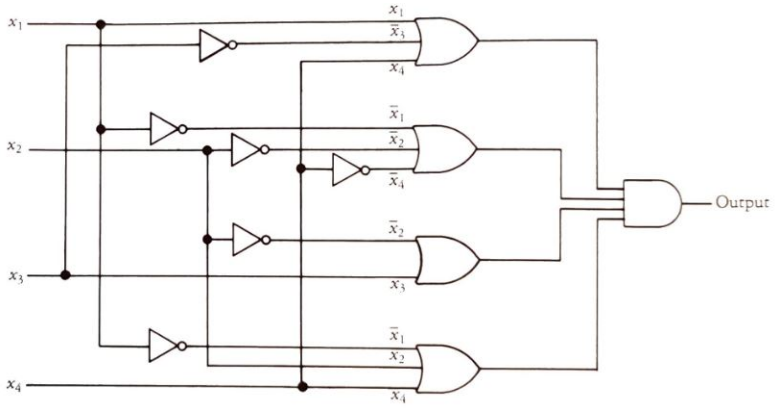
(continúa...)

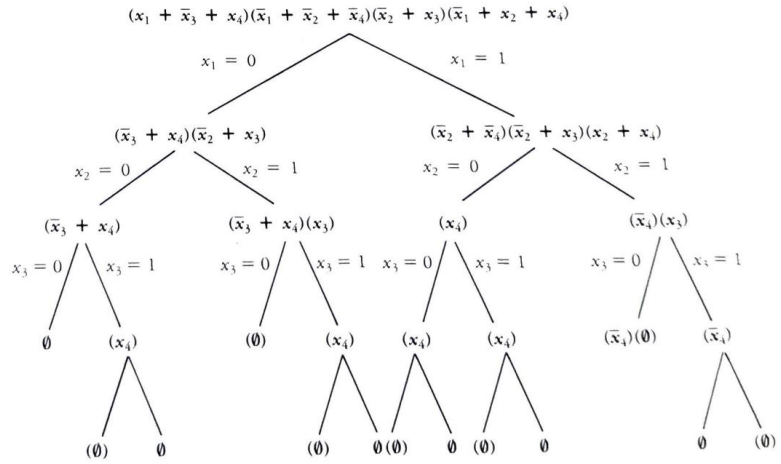
Las reducciones pueden hacerse relacionando unos problemas con otros muy diversos modos, pero habitualmente se toma un problema como origen de reducción a otros muchos.
A continuación nos centramos en el problema de la satisfacibilidad...

El problema de la "satisfacibilidad" (SAT)

Este es un claro representante de los problemas NP-Complejos. Consiste en determinar si una expresión lógica (en formato FNC-Forma Normal Conjuntiva, para uniformizar) se puede satisfacer o no. Es decir, si existe alguna combinación para los valores de sus variables que den "cierto" como resultado.
En 1960 Martin Davis y Hilary Putnam desarrollaron un algoritmo para reducir el coste, sin dejar de ser NP. En 1962 Davis-Putnam-Logemann-Lovelandes desarrollaron un segundo algoritmo usando backtracking. En 1971, Stephen Cook determinó que el problema es NP-Completo, inaugurando de ese modo la categoría.

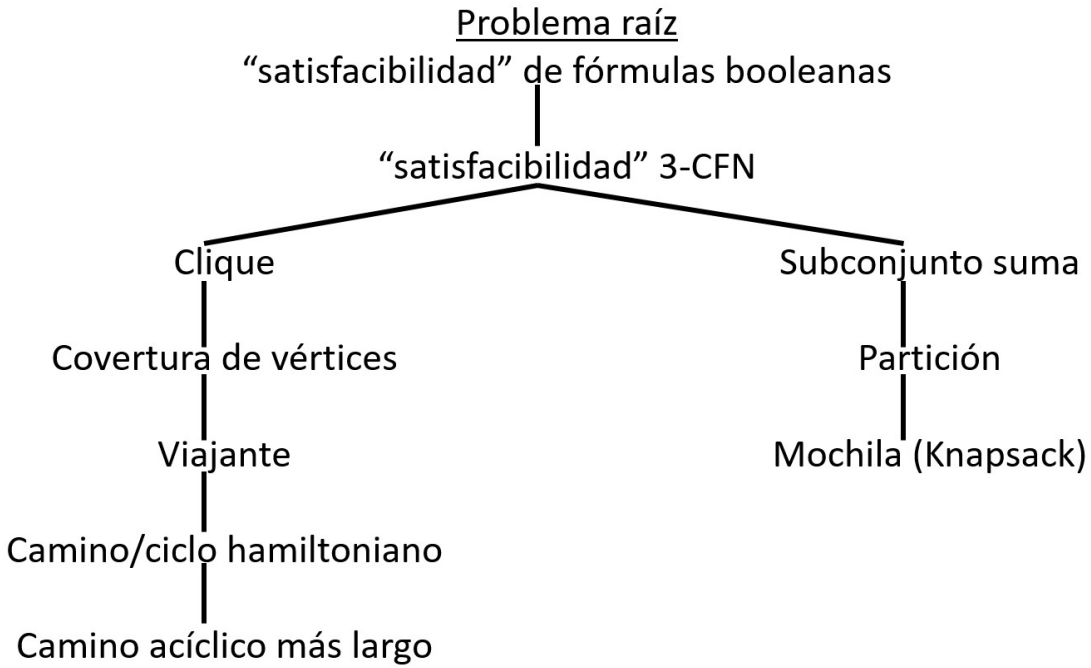
$$(x_1 + \overline{x_3} + x_4)(\overline{x_1} + \overline{x_2} + \overline{x_4})(\overline{x_2} + x_3)(\overline{x_1} + x_2 + x_4)$$





El ejemplo es 3-SAT. Curiosamente (o no tanto) 2-SAT es un problema P.
Notación alternativa: $(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Reducciones desde SAT

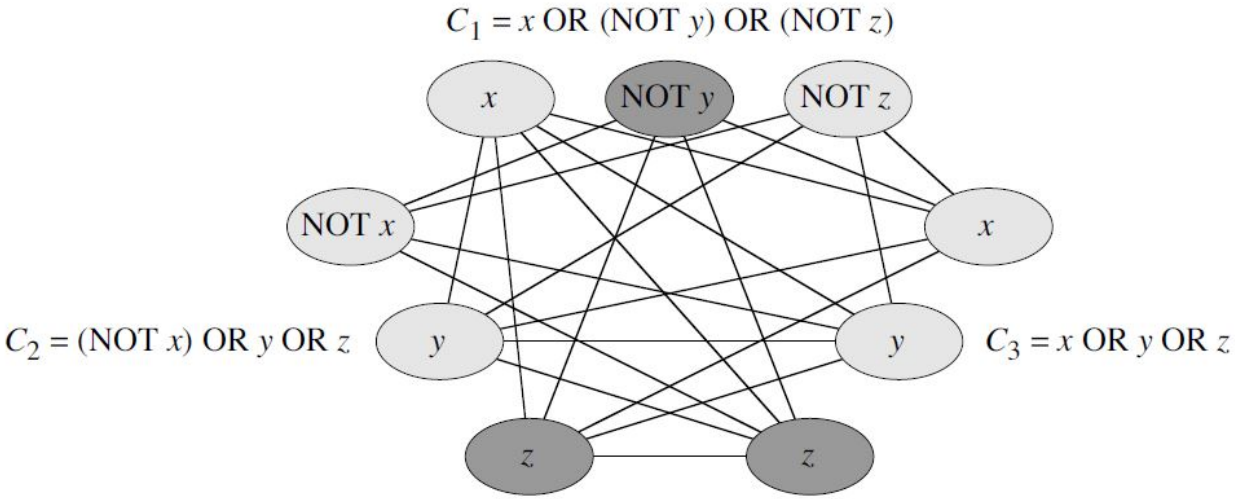


De SAT a Clique

$$(x + \overline{y} + \overline{z})(\overline{x} + y + z)(x + y + z)$$

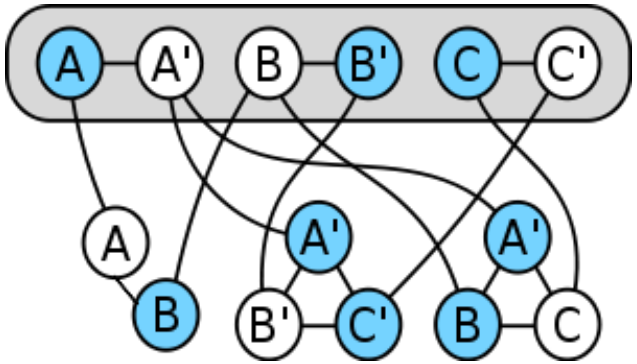
Se añade una arista entre v_i y v_j si se cumplen dos condiciones:

- pertenecen a diferentes clausulas
- sus literales no son negación uno de otro.



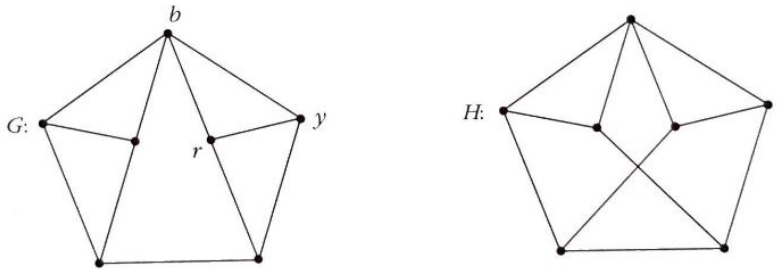
De SAT a Cobertura de vértices

$$(A + B)(\overline{A} + \overline{B} + \overline{C})(\overline{A} + B + C)$$



Ejemplo inverso: De 3-color mapping a SAT

Uno se puede colorear con tres colores y otro no.



Procedimiento para convertirlos en SAT:

Para cada vertice v_i incluir $(r_i + g_i + b_i)$

Para cada arista (v_i, v_j) incluir $(\overline{r_i} + \overline{r_j})(\overline{g_i} + \overline{g_j})(\overline{b_i} + \overline{b_j})$

Para N vértices y M aristas tendremos una expresión con $N + 3xM$ clausulas en $3xN$ variables (normalmente larga, pero no dificil de resolver por ello)

III - LOS LIMITES DEL CONOCIMIENTO

Recapitulando

Nos interesan los problemas asimétricos... difíciles de resolver pero con soluciones (certificados) fáciles de verificar. Lo que nos llevo a hablar de las clases P, NP, NP-Completo y NP-Hard, puesto que es en el terreno de los NP donde se encuentran esos algoritmos asimétricos.

Previamente hablamos de modelos computacionales como el marco desde el que entender correctamente la mencionada clasificación (con la ejecución de modelos deterministas y no deterministas).

Hablando de los diferentes conjuntos mencionamos el problema "¿P=NP?" (o mejor la conjetura "$P \neq NP$")

En cuanto a los modelos, nos encontramos con el "problema de la parada" expresado con Máquinas de Turing.

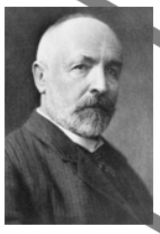
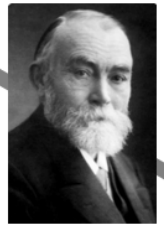
Estos problemas tienen su base en la dificultad de formalizar el conocimiento (¿P=NP?), y no sólo eso,... en la imposibilidad de hacerlo hasta el extremo (Problema de la Parada)

Ahora vamos a ver el origen y fundamento de estas dificultades.



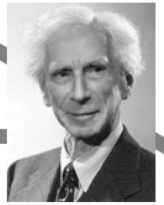
Aristóteles (384ac-322ac)
Euclides (325ac-265ac)


al-Khwārizmī
(780-850)



Frege (1848-1925)


Whitehead
(1861-1947)

Cantor
(1845-1918)



Russell (1872-1970)


Zermelo
(1871-1953)


Fraenkel
(1891-1965)


Dedekind
(1831-1916)

Hilbert
(1862-1943)


Peano
(1858-1932)


Gödel (1906-1978)


Church (1903-1995)


Noether
(1882-1935)


Herbrand
(1908-1931)


Kleene
(1909-1994)


Turing
(1912-1954)

Verdad y No verdad. Aristóteles I.

Dos pilares del conocimiento son la lógica aristotélica (proposicional), y el método axiomático.

Los trabajos de Aristóteles constituyen la primera investigación sistemática sobre los principios del razonamiento válido[a].

Estableció la **Lógica proposicional** que perduró hasta prácticamente el siglo XIX. Su elemento central sería el "Silogismo", como argumento deductivo que permite partir de premisas para llegar a conclusiones. Un Silogismo afirmaría una conclusión a partir de dos premisas.

Tanto premisas como conclusiones se formulan como "Juicios" (proposiciones). Los Juicios son oraciones con dos términos: un sujeto y un predicado. Un sujeto ha de ser un término "singular" (Sócrates, Platón), y un predicado puede ser un término Singular o "Universal" (mortal, hombre).

[a] Tratando de establecer lo que es "verdad": "Decir de lo que no es que es, o de lo que es que no es, es falso, y decir de lo que es que es, o de lo que no es que no es, es verdadero" (de su libro Metafísica)

Verdad y No verdad. Aristóteles II.

En concreto, su primera teoría sistemática de inferencia válida trataba los Silogismos Categóricos... los establecidos para juicios categóricos, que podrían ser de cuatro tipos: Todo S es P, Ningún S es P, Algunos S son P, Algunos S no son P.

Lo que hace Aristóteles a este respecto es estudiar toda la combinatoria de silogismos sobre juicios categóricos (4) con la posible distribución de los términos de la conclusión en las premisas, lo que llamó "Figuras" (3: AB,BC,AC; AB,AC,BC; AB,CB,AC). $4^3 \times 3 = 192$ silogismos.

Ejemplo (1^{era} figura A=griegos, B=hombres, C=mortales; juicios de tipo 1, 1, 1)

1. Todos los griegos son hombres.
2. Todos los hombres son mortales.
3. Por lo tanto, todos los griegos son mortales.

Otras muchas contribuciones son de menor impacto en el pensamiento lógico-filosófico-matemático (clasificación de tipos de predicados: categorías; clasificó 13 tipos de falacias,...)

Mencionaremos no obstante sus tres axiomas: no contradicción, identidad y tercio excluso (en notación formal $\neg(A \wedge \neg A)$, $A \leftrightarrow A$, $A \vee \neg A$)

Euclides y el método axiomático

Al parecer Euclides fue un compilador del conocimiento matemático de la época (aritmética, geometría y álgebra), pero destaca bajo nuestro punto de vista por su formalización de la geometría en base a 5 postulados (axiomas)

1. Dados dos puntos se puede trazar una recta que los une.
2. Cualquier segmento puede prolongarse de manera continua en cualquier sentido.
3. Se puede trazar una circunferencia con centro en cualquier punto y de cualquier radio.
4. Todos los ángulos rectos son congruentes (iguales tras una isometría).
5. Si una recta corta a otras dos formando, a un mismo lado de la secante, dos ángulos internos agudos, esas dos rectas prolongadas indefinidamente se cortan del lado en el que están dichos ángulos[a]. (reformulado como axioma de las paralelas)

En un conjunto de axiomas debe darse lo que se denomina "independencia" entre ellos. Es decir, que ninguno sea deducible de los demás siguiendo las reglas de manipulación establecidas. Este concepto es central en todo lo que veremos a continuación, y volveremos sobre él hacia algo más adelante.

[a] Este axioma, menos obvio que los demás, intentó deducirse de los otros 4 durante mucho tiempo. Finalmente se concluyó su independencia, y la posibilidad de ser sustituido por otros dando lugar a geometrías diferentes (no euclididanas): la elíptica (Riemman), donde no ha paralelas, y la hiperbólica (Lobachevsky), que presenta más de una paralela.

Cantor, Hilbert

CANTOR

Desarrolla la teoría de conjuntos, sobre la que Russell establece posteriormente el desarrollo de la lógica (y descubre su paradoja).
La teoría avanza con Zermelo al desarrollar su sistema de axiomas para evitar las paradojas. Este campo (desarrollo de la lógica con base en la teoría de conjuntos) sigue siendo la base de desarrollos fundamentales... la hipótesis del continuo... los métodos de forcing...

HILBERT

En 1920 propuso de forma explícita un "proyecto de investigación" (lo que en matemáticas suele denominarse "programa") en metamatemática, que acabó siendo conocido como programa de Hilbert. Quería que la matemática fuese formulada sobre unas bases sólidas y completamente lógicas. Creía que, en principio, esto podía lograrse, mostrando que:

> "toda la matemática se sigue de un sistema finito de axiomas escogidos correctamente; y se puede probar que tal sistema axiomático es consistente"

Este es el número 2 de los "problemas de Hilbert" .

Frege

En 1879, Frege publicó su revolucionaria obra titulada Ideografía o Escritura de conceptos (Begriffsschrift), en la que sentó las bases de la lógica matemática moderna, iniciando una nueva era en esta disciplina. Su objetivo era mostrar que las matemáticas surgen de la lógica.

La lógica había estado fundamentada hasta entonces en el cálculo proposicional Aristotélico, en base al cual la verdad o falsedad de una oración se establecía por medio de una serie de operadores lógicos (negación, conjunción, disyunción,...) que permitían desarrollar silogismos.

En su Ideografía, Frege hace el primer tratamiento sistemático de la lógica proposicional. Define una base axiomática, que pretende permitir que todas las leyes de la lógica sean deducidas.

Dentro de este esfuerzo, su contribución más destacada es la "**teoría de la cuantificación**"dando lugar a la **lógica de predicados** axiomáticos[a]. Con base en ésta, Frege fue capaz de expandir el ámbito de los operadores lógicos para poder estudiar la validez lógica de expresiones que contuvieran afirmaciones como todo, nada, algunos, cualquiera, etc. Las variables cuantificadas son ubicuas hoy en día en matemáticas y lógica.

[a] Es lo que conocemos como lógica de primer orden (y da lugar lugar posteriormente a la lógica de segundo orden y las "lógicas de alto nivel")

Sobre independencia en lógica matemática I

Uno de los propósitos declarados de Frege era aislar los principios de inferencia genuinamente lógicos, de modo que en la representación adecuada de la prueba matemática, uno en ningún momento apelaría a la ïntuición". Si había un elemento intuitivo, debía aislarse y representarse por separado como un axioma: a partir de ahí, la prueba debía ser puramente lógica y sin lagunas (el "logicismo"[a]).

Podemos ver esto usando la Geometria de incidencia (Hilbert)

- Los términos usados en la axiomática pueden "sugerirnos algo", pero debemos abstraernos de ello, son indefinidos: linea y punto
- Lo mismo diremos de las relaciones, son indefinidas: incidencia

Sólo nos interesa el comportamiento que viene definido por los axiomas:

1. para cualesquiera dos puntos distintos existe una única recta con la que inciden ambos
2. para cualquier linea existen al menos dos puntos que inciden en ella
3. existen 3 puntos diferentes con la propiedad de que ninguna recta incide con los tres

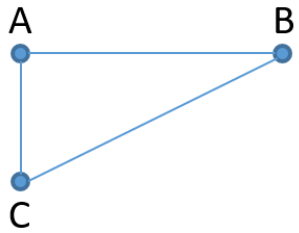
[a] Oxford dict: Postura filosófica que defiende la importancia de la lógica de los razonamientos en detrimento de su aspecto sociológico. (iniciada por Leibnitz)

Sobre independencia en lógica matemática II

Este sistema axiomático puede aplicarse sobre modelos reales.

UNA Interpretación:

Sólo hay tres puntos A, B, C; sólo hay tres lineas: $\overline{AB}$, $\overline{BC}$ y $\overline{AC}$; la incidencia se interpreta como "P incide con L"significa que P es un punto en L.



Si repasamos los axiomas vemos que se cumplen.
Esto es un modelo de la geometría de incidencia (también lo es la geometría del plano)

OTRA Interpretación:
Punto es 2, 3, 5, 7; Linea es 6, 10, 14, 15, 21, 35; la incidencia se interpreta como: "P indice con L "significa que P es un factor de L.
Si repasamos los axiomas vemos que se cumplen.

Sobre independencia en lógica matemática III

La deducción en este sistema axiomático se hará en abstracto y posteriormente puede transmitirse a los modelos

Planteémos un teorema:
Definición: dos lineas son paralelas si no hay un punto incidente con las dos.
Teorema: existen lineas paralelas... ¿Cierto o falso? ¿hay una prueba usando los axiomas 1,2,3?

Si probamos en la primera interpretación veremos que no es cierto
Si probamos en la segunda si que las hay (p.ej. 6 y 35), así que aquí si es cierto

Si, trasladado a dos modelos, obtenemos resultados contradictorios, tenemos que no se trata de un teorema sino de un enunciado independiente del sistema axiomático, y este -o su contrario- podría ser añadido como un nuevo axioma, definiendo un corpus distinto.[a]

[a] Basar las matemáticas en la lógica será por tanto encontrar uno que no lleve a contradicciones y que no deje teoremas sin probar.

Russell

Bertrand Russell es una pieza central en lo que estamos contando:

- Sacó a la luz los trabajos de Fregue, que eran prácticamente desconocidos ya que habían sido criticados fuertemente por los grandes matemáticos de la época.
- Aportó 3 volúmenes dedicados a axiomatizar la ciencia (tenía previsto un cuarto para la geometría, pero confesó estar exhausto y haber sufrido un deterioro en sus facultades mentales con sus anteriores trabajos): los "Principia Mathematica"[a].
- Para ello utilizó la lógica axiomatizada de Frege sobre la teoría de conjuntos (paralizando elproyecto de Fregue para axiomatizar la aritmética).
- Encontró una importante debilidad en la teoría de conjuntos (la paradoja de Russell), y apuntó un mecanismo para evitarla (la Teoría de Tipos[b])
- Interesado en el concepto de Número, estudió a fondo el simbolismo de Peano y su conjunto de axiomas para la aritmética, descubriendo que Fregue había llegado a definiciones equivalentes e integrándolo y mejorándolo en su obra[c]

[a] En clara referencia a los "Philosophiæ naturalis principia mathematica" de Newton

[b] Es adelantarnos, pero Alonzo Church encontró una debilidad semejante en su Cálculo Lambda y y reslvió de modo semejante con el Cálculo Lambda Tipado

[c] La actual definición de cero es la conocida como de Frege-Russell.

La paradoja de Russell

En el desarrollo de su obra, Russell analizó determinados aspectos desarrollados por Cantor sobre la cardinalidad de los conjuntos, lo que le llevó al descubrimiento de una clase muy interesante: la clase de todas las clases. Esta clase contiene dos tipos de clases: aquellas clases que se contienen a sí mismas, y aquellas que no. Fue así como llegó a demostrar que resultaba una contradicción: Y es un miembro de Y, si y solo si, Y no es un miembro de Y.

Fijémonos en el conjunto de todos los conjuntos que no incluyen a sí mismos:

$$E = \{x : x \text{ es un conjunto y } x \notin x\}$$

y preguntémonos si $E \in E$. Pueden pasar dos cosas:

1. $E \in E$) entonces E es un conjunto y por la definición de E, $E \notin E$!! Falso
2. $E \notin E$) entonces E es un conjunto y $E \notin E$, es decir $E \in E$!! Falso

Tenemos que tanto una opción como la contraria son falsas.

Ante la Paradoja de Russell se consideró que era preciso limitar las posibilidades en la teoría de conjuntos y de ese modo evitar el problema. Esto fue abordado por Zermelo y Fraenkel estableciendo el conjunto de 10 axiomas[a] que fundamentan hoy en día la teoría de conjuntos (con uno añadido, el axioma de elección[b])

[a] Podemos decir que 6 son evidentes y 4 no lo son tanto.

[b] El axioma de elección también se debe a Zermelo, pero se menciona por separado porque, aunque es normalmente admitido, se pone en cuestión en ocasiones.

El teorema de Gödel

En 1930 Kurt Gödel intentó demostrar que el Cálculo de Predicados era "completo" (que se podría probar cualquier fórmula de modo mecánico), y terminó por demostrar lo contrario. Hay sentencias ciertas que no pueden ser probadas (la veracidad no puede probarse paso a paso pero sí de "otros" modo).

El método de Gödel consistía en expresar toda sentencia en el cálculo de predicados mediante un número (el número de Gödel). EL proceso consta de tres etapas:

1. Establecer axiomas para el cálculo de predicados y una regla de inferencia por la cual se puedan deducir nuevas fórmulas de las antiguas.
2. Establecer axiomas para la aritmética en términos del cálculo de predicados
3. Definir una numeración para cada fórmula.

El teorema de Gödel -axiomas del calculo de predicados

Los axiomas para el cálculo de predicados en lenguaje implicativo quedan como sigue:

1. $\forall y_i(F \to (G \to F))$
2. $\forall y_i(F \to (G \to H)) \to ((F \to G) \to (F \to H))$
3. $\forall y_i((\neg F \to \neg G) \to ((\neg F \to G) \to F))$
4. $\forall y_i(\forall x(F \to G) \to (F \to \forall xG))$ (sin que x pueda ser libre en F)
5. $\forall y_i((F \to G) \to (\forall y_i F \to \forall y_i G))$
6. $\forall y_i(\forall xF(x) \to F(y))$ (sin que y sea cuantificada al ser sustituida)

donde F, G, H son "fórmulas", $\forall y_i$ se refiere a un grupo indeterminado de variables cualificadas universalmente, $\to$ es la implicación y $\neg$ indica negación.

A esto añadimos la regla de inferencia : Si F y $F \to G$, entonces G

El teorema de Gödel - axiomas de la aritmética

Definimos la aritmética siguiendo a Peano

1. $\forall x \quad \neg(0 = sx)$
2. $\forall x, y \quad (sx = sy) \rightarrow (x = y)$
3. $\forall x \quad x + 0 = x$
4. $\forall x, y \quad x + sy = s(x + y)$
5. $\forall x, y \quad x \times sy = x \times y + x$
6. $\forall x \quad x \times 0 = 0$

faltaría definir la igualdad

1. $\forall x \quad x = x$
2. $\forall x, y, z \quad (x = y) \rightarrow ((x = z) \rightarrow (y = z))$
3. $\forall x, y \quad (x = y) \rightarrow ((A(x, x) \rightarrow A(x, y))$ (donde A es cualquier fórmula de dos variables)

Aquí nuestra regla de inducción es: $(P(0)\&\forall x(P(x) \rightarrow P(sx))) \rightarrow \forall x P(x)$

El teorema de Gödel - esquema de numeración

Se asigna un valor numérico consecutivo a cada símbolo, y se utilizan para codificar la sentencia siendo exponentes de una secuencia de primos consecutivos.

Símbolo	Código	Simbolo	Código	Simbolo	Código
0	1	(	6	$\neg$	11
s	2	)	7	&	12
$+$	3	,	8	$\exists$	13
$\times$	4	x	9	$\forall$	14
$=$	5	1	10	$\rightarrow$	15

ejemplo: $x_1 + sx_{11} = s(x_1 + x_{11})$

$$2^9\,3^{10}\,5^3\,7^2\,11^9\,13^{10}\,17^{10}\,19^5\,23^2\,29^6\,31^9\,37^{10}\,41^3\,43^9\,47^{10}\,53^{10}\,59^7$$

Los número s de Gödel tienden a ser enormes, pero son computables.

El teorema de Gödel - preparación

Consideremos el siguiente predicado:

Prueba(x,y,z)

que viene a decir que "x es el número de Gödel de una prueba X de la fórmula Y de una variable (y número de Gödel y), a la que se da el valor entero z.
Este predicado no es más que un "alias" de un (largo) predicado con tres variables libres: x, y, z (en la notación de Gödel x_1, x_{11}, x_{111})
Esta expresión conlleva una serie de acciones que incluyen:

- dado un entero decodificarlo a la cadena de caracteres de la que es número de Gödel
- dada una cadena de caracteres, determinar si es una expresión válida como fórmula en el cálculo de predicados.
- dada una secuencia de fórmulas, determinar si es una prueba de la última en la secuencia.

El teorema de Gödel - EL TEOREMA

Usaremos el predicado anterior para definir este otro[a] (sea G, con número de Gödel g):

$$\neg\exists x Prueba(x, y, y)$$

Estamos negando la existencia de una prueba de la fórmula Y cuando asignamos su propio número de Gödel a su variable libre.

El Teorema de Gödel prueba el predicado anterior mediante su aplicación a sí mismo:

El predicado ($G \equiv$) $\neg\exists x Prueba(x, g, g)$ es cierto, pero no puede probarse en el sistema aritmético formal.

Prueba:

Supongamos que G se puede probar, y que p sea el número de Gödel de su prueba. Entonces Prueba(p,g,g) es cierto. Pero esto contradice que $\neg\exists x Prueba(x, g, g)$, lo que no nos deja otra conclusión que la de que no existe tal prueba P.

La fórmula $\neg\exists x Prueba(x, g, g)$ es efectivamente cierta, puesto que acabamos de establecer lo que esta diciendo: que no tiene prueba.

[a] Una demostración en la que un predicado con dos variables recibe el mismo argumento en las dos es lo que se denomina "prueba por diagonalización", y aparece frecuentemente en lógica matemática y en teoría de conjuntos infinitos. Fue usado por primera vez por Cantor para demostrar que los números reales no son contables.

Cálculo Efectivo

Aun cuando haya predicados no probables, la capacidad de realizar pruebas mecánicamente (cuando sea posible) es de interés, y en ese sentido se realizaron diversos intentos.

El primero pudo ser el modelo de Herbrand-Gödel de las "funciones recursivas generales"[a]. Es uno de los primeros resultados **teoría de la demostración (o de la prueba)**, estableciendo un nexo entre cuantificación y lógica de primer orden. Esto fue formalizado posteriormente por Stephen Kleene.

En 1936, Alonzo Church, con este mismo objetivo definió lo que llamó "calculabilidad efectiva": cualquier proceso llevado a cabo por pasos discretos según reglas bien definidas. Estableció lo que llamó "Cálculo λ" convencido de haber establecido un mecanismo capaz de llevar a cabo todo aquello que fuese "efectivamente calculable".

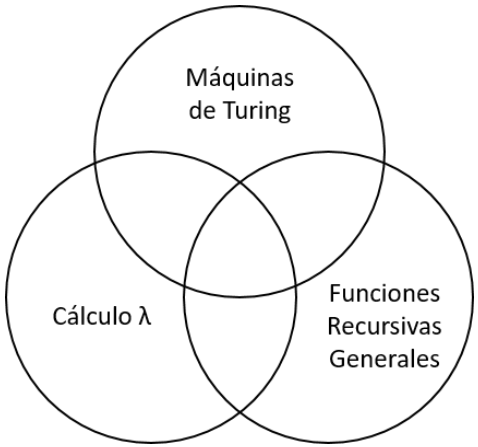
En menos de un año apareció otro mecanismo con la misma pretensión: la Máquina de Turing.

[a] Jackes Herbrand murió con 23 años en accidente de montaña en Los Alpes, poco después de formular un par de teoremas en teoría de números y lógica, que llevan su nombre y están en el origen del modelo de funciones recursivas.

La conjetura de Church-Turing

Teniendo en cuenta que los tres mecanismos eran independientes podrían presentar diferentes capacidades. No obstante Church demostró que su Cálculo λ coincidía en capacidades con el modelo de funciones recursivas, y posteriormente Turing hizo lo mismo respecto a su Máquina y el Cálculo λ

Kleene demostró el resto de relaciones entre los tres mecanismos fortaleciendo la "Conjetura Church-Turing", la existencia del concepto "computable", los tres métodos serían capaces de ajustarse a dicho concepto



Consecuencias en el mundo de la programación

Clasificación general de lenguajes

Máquina de Turing

Alan Turing

1938

Lenguajes

Cálculo Lambda

Sin olvidar las funciones recursivas de Herbrand-Gödel"

Alonzo Church

Imperativos
- Fortran
- Cobol
- Pascal
- C
- ...

O/B objetos
- Object Pascal
- C++
- Javascript
- Java
- Python
- ...

Frameworks
- Ruby on Rails
- ...

Funcionales
- Lisp
- Scheme
- ML
- Hope
- CLOS
- Ocaml
- ...
- Haskell
- Clojure

Lógicos
- Prolog
- ...

- Javascript
- Scala
- Java 8


```
let rec long = function
                |[] -> 0
                |x::xs -> 1 + long xs;;

let rec ordenar = function
                |[] -> []
                |x::xs -> insertar x (ordenar xs)
      and insertar e = function
                |[] -> [e]
                |x::xs -> if x > e
                          then e::x::xs
                          else x::(insertar e xs);
```

Ejemplo OCaml

Prolog (1970 - 1997 (v.4)) es un lenguaje lógico-declarativo, pero esa es "otra historia"

"la otra historia" .