Universidad de Valladolid

Facultad de Ciencias

Trabajo de Fin de Grado

Grado en Matemáticas

Introducción a la Teoría de la Complejidad Computacional

Autor: Mario Morán Cañón

Tutor: Philippe T. Giménez

ÍNDICE GENERAL

INTRODUCCIÓN

Los matemáticos trabajamos con problemas. Trabajamos con ellos, y no simplemente los resolvemos, puesto que en la mayoría de las ocasiones al enfrentarnos a uno acabamos planteando muchos otros, y esta tarea es igualmente importante para el avance de las Matemáticas que la de resolverlos. Nuestros problemas son de muy diverso tipo y dificultad dependiendo del área y el contexto en que nos encontremos. Problemas sencillos muchas veces resueltos como los que nos planteaban en el colegio, o aún abiertos como los de la mayoría de investigaciones teóricas. E incluso algunos que a día de hoy no sabemos muy bien si están resueltos o no, véase la Conjetura *abc*. Problemas famosos por diversos motivos, desde su innegable utilidad práctica, como la que la resolución de las ecuaciones de Navier-Stokes tendría para la mecánica de fluidos, hasta el tiempo que se ha tardado en resolverlos, como es el caso del último teorema de Fermat. Problemas milenarios como los de la Geometría clásica, y problemas de áreas surgidas en el último siglo, que es la situación de la que nos ocupa: la Matemática Computacional.

La mayoría de las veces nuestros problemas consisten en encontrar estructuras que nos faciliten la resolución de otros problemas, y esto es, desde un punto de vista general, lo que abordamos en este texto: la definición de estructuras que nos permitan clasificar, precisamente, problemas. Esto plantea una dificultad, y es que los problemas pueden expresarse de muy diversas formas (desde preguntas con respuesta afirmativa o negativa hasta enunciados para los que se debe encontrar una prueba de su veracidad o falsedad), con lo que compararlos, en cualquier sentido, es una tarea casi imposible. Es por ello que necesitamos homogeneizar la forma de plantear y resolver los problemas, a fin de hacerlos comparables. Y esta homogeneización vino de la mano de la teoría de lenguajes formales, situada en la intersección de las Matemáticas, la Lógica y la Computación.

Pese a que dicha formalización de los problemas mediante lenguajes es la que nos va a interesar, no queremos ser presuntuosos y hacer creer al lector que esto fue lo que motivó, en el segundo tercio del siglo XX, el nacimiento de la Computación. Sin embargo, sí vuelve a estar relacionado con los problemas, pues lo que se perseguía (al igual que ocurre en la actualidad) era la resolución automática de los mismos. Una máquina (al menos las que conocemos hoy en día) no puede pensar, carece de creatividad, tan solo dispone de memoria (cada vez mayor) y la capacidad de ejecutar unas instrucciones que tiene previamente descritas. Es decir, se limita a almacenar información y aceptar órdenes adecuadamente expresadas. Entonces los únicos problemas que podrá resolver serán aquellos que tengan un proceso de resolución *mecánico*, es decir, consistente en un conjunto finito y ordenado de pasos sencillos, que formarán un *algoritmo*.

Ya las primeras computadoras automáticas hicieron patentes las ventajas que ofrecían sobre la resolución "a mano" de problemas para los cuales existe un algoritmo, especialmente los de tipo numérico, no solo en cuanto al tiempo de resolución sino también a la precisión, aún cuando están limitados a la aritmética finita. El producto más importante de la Computación a día de hoy, los ordenadores modernos, aventajan a los originales principalmente en memoria y velocidad, pero no en las tareas que pueden llevar a cabo, puesto que las operaciones básicas que están diseñados para realizar son esencialmente las mismas. Esta limitación en las operaciones provocó que tanto los planteamientos de los problemas resolubles algorítmicamente como los propios algoritmos fueran adaptados al lenguaje que marcaban las computadoras, lo que supone un acercamiento a la homogeneización que buscábamos.

No obstante, hay máquinas de computación de muy diversos tipos, y tanto las operaciones básicas que pueden realizar como la forma de expresarles las órdenes suelen variar de unas a otras, por lo que esa homogeneización no es tal. Surge entonces la necesidad de definir *modelos computacionales abstractos* que unifiquen las posibilidades de los diferentes modelos físicos, con lo que al reescribir los problemas y algoritmos en estos términos tenemos ya un soporte común que nos permite el estudio comparativo. Se produce entonces un fenómeno curioso: estamos adaptando la realidad a la abstracción y no al contrario, como solía suceder, lo que no hace sino reforzar el papel de las Matemáticas, pues solamente desde ellas pueden estudiarse los modelos abstractos.

En este punto tenemos resuelta la cuestión de la homogeneización de problemas para los que existe un algoritmo de resolución. Sin embargo no todos los problemas son de este tipo (si excluimos algoritmos que podemos considerar irrealizables, como por ejemplo la generación progresiva de todas las posibles concatenaciones de símbolos de un determinado alfabeto esperando obtener así en algún paso la demostración de un teorema), y a pesar de que la adaptación a un modelo abstracto puede no resultar interesante en esos casos con vistas a su resolución, sí lo es para nuestro objetivo de clasificación. De hecho, querremos que esa clasificación respete nuestra intuición y los considere como "más difíciles" (o al menos no "más fáciles") que aquellos para los que sí disponemos de un algoritmo realizable. Porque, por complicado que sea dicho algoritmo, para obtener la solución solo es necesario ser paciente y cuidadoso a la hora de seguir los pasos de que consta (algo que, como hemos indicado, incluso una máquina es capaz de hacer), mientras que para resolver uno de los otros problemas, por simple que sea, deberemos pensar, si bien es cierto que la costumbre nos hace ver como natural este proceso creativo, deformando en ocasiones la intuición de la que antes hablábamos al comparar algoritmos complicados y soluciones obtenidas "pensando". En vista de esto, la solución natural es extender los modelos abstractos de forma que engloben todos los problemas que se pueden resolver pensando (es decir, tanto aquellos para los que disponemos de un algoritmo como para los que no, ya que en realidad los algoritmos nos ahorran pensar porque ya lo hizo por nosotros su diseñador). Entra en juego la noción que podemos llamar *indeterminismo*, es decir, el modelo puede imitar el pensamiento humano. Evidentemente, todo ello desde un punto de vista abstracto que por el momento no es físicamente realizable.

Hemos logrado entonces la homogeneización de los problemas y sus resoluciones, con lo que ahora son comparables. La Teoría de la Complejidad Computacional es precisamente la rama de las Matemáticas Computacionales que se dedica al estudio y clasificación de los problemas según su dificultad. Ahora bien, ¿qué entendemos por dificultad? Es evidente que necesitamos una definición formal si queremos trabajar desde un punto de vista matemático, y esta definición debe ser lo más satisfactoria posible, en el sentido de que debe respetar nuestra intuición: si en la realidad tenemos un problema que la mayoría de nosotros diríamos que es más difícil que otro, entonces también debe serlo según la definición formal. Es decir, resolver la hipótesis de Riemann debe ser más difícil que sumar $3+4$.

Analicemos entonces lo que entendemos por un problema difícil. Si olvidamos el indeterminismo y asumimos que para todos los problemas existe un algoritmo que los resuelve (aunque sea irrealizable), tenemos que ver cuál es la diferencia entre un "buen" algoritmo y un algortimo irrealizable, pues ya hemos dicho que marcarán la dificultad de los respectivos problemas. En el ejemplo anterior, el algoritmo consistente en generar todas las posibles cadenas de símbolos de un alfabeto de forma progresiva (es decir, comenzamos con todas las posibles cadenas de longitud 1, luego construimos a partir de estas las de longitud 2 y así sucesivamente) y comprobar en cada paso si es la demostración de un teorema (o de un enunciado, en general, si asumimos que puede ser falso) es irrealizable principalmente por un motivo: el número de posibles cadenas aumenta exponencialmente con su longitud. Para la construcción una posibilidad es guardarlas todas, con lo que necesitamos una gran cantidad de memoria. Para comprobar si una determinada cadena es una demostración válida tendremos, como mínimo, que leerla, y dada la enorme cantidad de cadenas que hay, de longitud cada vez mayor, podríamos tardar una eternidad antes de lograr encontrar la buena. Nótese que en un alfabeto de 27 letras como el español, si una determinada demostración consta de 1000 símbolos (y sería bastante corta) tendríamos que generar, almacenar y leer al menos 27^{999} cadenas, que son tan solo las de longitud 999. Evidentemente ni el superordenador más potente que existe ahora mismo tiene la capacidad de procesar toda esta información en un tiempo razonable (ni aunque estableciésemos como unidad de tiempo razonable varios millones de veces la edad del universo). Es decir, el algoritmo requiere demasiados recursos.

Parece razonable asociar la noción de dificultad a la cantidad de recursos que precisa un problema para ser resuelto mediante un algoritmo. Evidentemente, un mismo problema puede resolverse de varias formas distintas, y no todas ellas tienen por qué consumir la misma cantidad de recursos (más bien al contrario). Entonces escogeremos en cada caso el algoritmo que menos recursos consuma, es decir, que sea más *eficiente*. Como ilustra el ejemplo, los recursos que más nos interesan son el espacio y el tiempo. En general el más valioso de los dos es el tiempo, basta con observar que el espacio es reutilizable, mientras que el tiempo, por desgracia, no. Incluso aunque no pudiéramos reutilizar el espacio (supongamos que escribimos con tinta indeleble en un papel), empleamos cierto tiempo en escribir y leer, por lo que, a no ser que el espacio sea muy limitado (lo cual no suele ocurrir en los ordenadores modernos) el tiempo será más preciado.

Esto se traslada a la Teoría de la Complejidad Computacional, con lo que surgen dos ramas plenamente relacionadas entre sí: la Complejidad de Tiempo y la de Espacio, cada una de las cuales estudia y clasifica los problemas en relación a los requerimientos del correspondiente recurso.

Si volvemos a permitir el indeterminismo, que nos evitará en general considerar los algoritmos irrealizables para todos aquellos problemas para los que no conocemos uno eficiente, tendremos que asegurarnos en todo caso de que la clasificación siga respetando la dificultad real: un solo paso indeterminista, por pequeño que sea, debe considerarse más difícil (es decir, que consume más recursos) que muchos pasos deterministas juntos. Una vez tenemos todos estos ingredientes, ya podemos empezar a analizar los problemas.

Aunque en esta exposición comenzamos motivando y planteando el objetivo de la Teoría de la Complejidad Computacional y hemos ido razonando los inconvenientes con que nos encontrábamos y aportando las correspondientes soluciones, el orden cronológico en que se desarrollaron estas ideas no fue exactamente este; de hecho muchos modelos computacionales fueron propuestos antes de la aparición de las primeras computadoras automáticas, y algunos son válidos para modelizar los ordenadores de hoy en día y los de un futuro a medio plazo (al menos hasta que se hagan más progresos en el campo de la computación cuántica), como iremos descubriendo a lo largo del texto. El lector interesado en una introducción histórica en el tema puede acudir a [21].

A pesar de que el título sea *Introducción a la Teoría de la Complejidad Computacional*, en esta memoria nos limitaremos a la Complejidad de Tiempo (por limitación de espacio, aunque resulte paradójico). No obstante, ambas teorías son en buena medida paralelas, y el capítulo 1 proporciona el marco de trabajo en que se desarrollan ambas: el modelo computacional abstracto de la máquina de Turing.

Los capítulos 2 y 3 están dedicados a introducir y manejar los primeros conceptos propios de la Complejidad Computacional, en especial las clases de complejidad, insistiendo en las más importantes (**P** y **NP**, que agrupan a los problemas resolubles en tiempo polinómico de forma determinista e indeterminista, respectivamente) y explicando por qué lo son. En el capítulo 4 se define una herramienta importante para el manejo abstracto de los problemas en términos de su complejidad: las reducciones; esto nos permite obtener problemas representativos de una determinada clase, en especial de la clase **NP**, a los que llamaremos problemas **NP**-completos.

Como colofón, en el capítulo 5 haremos un análisis detallado de la famosa conjetura de Cook, **¿P=NP?**, analizando sus implicaciones y las perspectivas sobre cuál de las respuestas (igualdad o desigualdad) es más probable. Este capítulo hará también las veces de conclusión, dado que en la conjetura de Cook intervienen, directa o indirectamente, la mayoría de los conceptos tratados en los capítulos precedentes; a reforzar ese carácter epilogal contribuye el hecho de que muchos de los comentarios se basen en preguntas abiertas y por lo tanto sean de carácter filosófico y especulativo.

Las principales referencias bibliográficas utilizadas en todos los capítulos son [3], [39] y [40], el resto de referencias puntuales se mencionan en el lugar correspondiente. Aunque

tenga un carácter divulgativo, sin apenas formalismos, animo a cualquier persona interesada no iniciada en la Teoría de la Complejidad Computacional (independientemente de su formación) a leer [20], pues proporciona una idea general de la relevancia de estas cuestiones, muy útil como punto de partida.

Puesto que se trata de una rama perteneciente a un área de las Matemáticas sin ninguna presencia en el Grado en Matemáticas de la Universidad de Valladolid, aunque transversal a todas las especialidades como iremos viendo, el texto pretende no solo aportar las definiciones y resultados principales, sino también hacer hincapié en las partes esenciales de la Computación y la propia Teoría de la Complejidad Computacional y dar a conocer el estilo de los resultados y las técnicas de demostración, bastante diferentes de las de otros campos. Todo ello acompañado de ejemplos para facilitar su comprensión, una interesante selección bibliográfica y escrito en un tono que esperemos que haga llevadera la lectura.

No quisiera terminar esta introducción sin agradecer el apoyo ofrecido por mi tutor, Philippe Giménez, quien se embarcó conmigo en este ambicioso y arriesgado proyecto sin apenas conocer el área, confiando en que pudiera sacarlo adelante, y que ha ido descubriéndola conmigo a lo largo de todo un año, proporcionándome materiales y consejo cuando ha podido, o pidiéndolo él mismo a otros cuando no. Y el primero de estos otros ha sido Luis Miguel Pardo, catedrático del Departamento de Matemáticas, Estadística y Computación de la Universidad de Cantabria, que en apenas unas horas nos dio su opinión afirmativa sobre la viabilidad del proyecto y nos proporcionó los primeros materiales, de su propia autoría, con los que me inicié en las Matemáticas Computacionales y la Computación, [24], [40] y [41], además de otras referencias bibliográficas. Muchas gracias por todo ello.

Quiero expresar también mi agradecimiento al Departamento de Álgebra, Análisis Matemático, Geometría y Topología que me ha permitido ser beneficiario de una Beca de Colaboración concedida por el Ministerio de Educación, Cultura y Deporte; además de a todos los profesores que me han dado clase durante estos cuatro años de Grado.

Por último, no quiero olvidar el apoyo de mis amigos y familia, sobre todo de mis padres, mi hermano y mis abuelos, que han sabido soportarme y ayudarme en los momentos difíciles.

CAPÍTULO 0

NOTACIÓN Y OTROS CONVENIOS

Antes de comenzar con la exposición teórica introduciremos una serie de convenios de notación y conceptos básicos que utilizaremos a lo largo del texto. Salvo que se indique lo contrario, se procurará utilizar los convenios notacionales más extendidos.

Convenios notacionales básicos

Denotaremos por $\mathbb{N} = \{0, 1, 2, 3, ...\}$ al conjunto de los números naturales, y por $\aleph_0$ a su cardinal. Si $x \in \mathbb{R}$, denotamos por $\lfloor x \rfloor$ a la parte entera (que aquí consideraremos igual a la función suelo) de x, es decir, al mayor entero n tal que $x \geq n$. Denotaremos por $\lceil x \rceil$ al menor entero n tal que $x \leq n$. Siempre que utilicemos un número real en un contexto en el que se precise un entero, se entenderá que nos referimos a $\lceil x \rceil$.

Salvo que se diga expresamente lo contrario, denotaremos las cadenas y vectores por letras minúsculas, por ejemplo u, salvo que pueda dar lugar a equívoco, en cuyo caso lo denotaremos por $\mathbf{u}$. Dado un vector o una cadena u, denotaremos por u_i a su i-ésima componente, y escribiremos $u = (u_1, ..., u_n)$ o bien, en el caso de cadenas y si no genera confusiones, $u = u_1 \cdots u_n$.

Al escribir $\log(x)$ nos referiremos al logaritmo de x en base 2.

Dado un conjunto E, denotaremos por $\sharp E$ a su cardinal, y por $\mathscr{P}(E)$ a su conjunto de partes. Para la función característica o indicatriz de un conjunto E utilizaremos la notación χ_E. Si $D \subseteq E$ denotamos por $\overline{D} = E \smallsetminus D$ al complementario de D en E.

Decimos que una condición $P(n)$ que depende de un número natural se verifica *para n suficientemente grande* si existe $n_0 \in \mathbb{N}$ tal que $P(n)$ se verifica para todo $n \geq n_0$.

Representación de objetos

Dado que el modelo computacional que vamos a considerar es el de la máquina de Turing, necesitaremos una forma de representar los objetos que queremos que estas traten, ya sea como argumentos de entrada, de salida, o como pasos intermedios en la computación. Dichos objetos (que pueden ser números, texto, expresiones lógicas, matrices, grafos e incluso las propias máquinas de Turing, entre muchos otros tipos) serán codificados por cadenas de símbolos sobre las que las máquinas de Turing podrán trabajar. A continuación precisaremos estos conceptos.

Definición 0.1 Sea Σ un conjunto finito que llamaremos *alfabeto* .

- Una *palabra* sobre Σ es una lista o cadena finita de símbolos de Σ. Podemos formalmente identificar las listas $x = x_1 \cdots x_n$ de símbolos de Σ ($x_i \in \Sigma, i = 1, ..., n$) con los elementos $(x_1, ..., x_n) \in \Sigma^n$. Denotaremos por $|x| = n$ a la *longitud* (o talla) de la palabra $x = x_1 \cdots x_n$.
- El conjunto de todas las palabras sobre el alfabeto Σ se denotará mediante Σ^*, y podemos identificarlo con la unión disjunta

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n .$$

- Σ^* es un semigrupo o monoide con la operación *concatenación* o *adjunción de palabras* (que denotaremos por $\cdot$, o, si no hay lugar a equívoco, simplemente por yuxtaposición) y la *palabra vacía* $\lambda \in \Sigma^0$ es el elemento neutro (y la única palabra de longitud 0). Denotaremos $x^0 = \lambda$ y $x^n = xx^{n-1}, \forall\, n \geq 1$.
- Los subconjuntos de Σ^* se denominan *lenguajes* .

Es sencillo comprobar que la longitud define un morfismo de semigrupos entre el anterior y $\mathbb{N}$, y que si Σ es un alfabeto finito, el conjunto Σ^* es infinito numerable, de donde se deduce que un lenguaje $L \subseteq \Sigma^*$ tendrá cardinal a lo sumo numerable, y que $\sharp\{L \subseteq \Sigma^*\} = \sharp\mathscr{P}(\Sigma^*) = 2^{\aleph_0}$. En particular, hay una cantidad infinita no numerable de lenguajes sobre un alfabeto finito.

Definiremos también una serie de operaciones elementales con lenguajes:

- *Unión de lenguajes:* Dados $L_1, L_2 \subseteq \Sigma^*$ definimos

$$L_1 \cup L_2 := \{x \in \Sigma^* : x \in L_1 \text{ ó } x \in L_2\} .$$

- *Concatenación de lenguajes:* Dados $L_1, L_2 \subseteq \Sigma^*$ definimos su concatenación:

$$L_1 \cdot L_2 := \{x_1 \cdot x_2 \in \Sigma^* : x_1 \in L_1, x_2 \in L_2\} .$$

Se puede encontrar más información sobre los lenguajes y sus operaciones en [37, sección 1.5], [24, secciones 1.1 y 1.2], [14, sección 2.7] o [34, secciones 1.7 y 1.8]. Un libro avanzado sobre el tema es [44].

El siguiente concepto es clave en el tratamiento automático de la información, pues permite "traducir" los objetos con los que nosotros trabajamos de forma que puedan ser manipulados por, en el caso que nos ocupa, máquinas de Turing, que normalmente utilizarán el alfabeto binario, $\Sigma = \{0,1\}$.

Definición 0.2 *Codificar* un conjunto finito A (que podemos interpretar como un alfabeto) en un alfabeto código Σ es dar una aplicación inyectiva $c : A \longrightarrow \Sigma^*$. Llamaremos codificación de un elemento $a \in A$ a su imagen $c(a) \in \Sigma^*$.

Una vez tenemos una forma de codificar un alfabeto en otro, disponemos en la literatura especializada de múltiples procedimientos para representar objetos matemáticos como cadenas de elementos de un alfabeto dado, de forma que el proceso sea reversible y podamos volver a obtener el objeto a partir de su representación. La codificación de alfabetos nos aporta una cierta independencia del alfabeto en la mayoría de situaciones que se plantean a lo largo del texto.

Por ejemplo, un entero puede representarse en el alfabeto $\Sigma = \{0,1\}$ por su expresión binaria (9 se representa como 1001), y un grafo no orientado según su *matriz de adyacencia* (con tantas filas y columnas como vértices tenga el grafo, y con un 1 en la posición (i, j) si hay un arista entre los vértices i y j, tras una numeración de los mismos, o un 0 en caso contrario).

Dado un objeto x, denotaremos por $\llcorner x \lrcorner$ a alguna representación suya que no especificaremos, aunque en muchas ocasiones omitiremos los símbolos $\llcorner \lrcorner$, escribiendo solamente x, siempre que dicha omisión no induzca a confusiones.

La idea de la representación nos permite hablar de computar una función $f : \mathscr{A} \longrightarrow \mathscr{B}$ tal que $\mathscr{A}, \mathscr{B} \nsubseteq \Sigma^*$ donde Σ es el alfabeto que estamos utilizando, pues la identificamos implícitamente con la correspondiente función $\tilde{f} : \Sigma^* \longrightarrow \Sigma^*$ tal que $\tilde{f}(\llcorner x \lrcorner) = \llcorner f(x) \lrcorner$.

Funciones booleanas y problemas decisionales

Las funciones y fórmulas definidas a continuación nos proporcionarán numerosos ejemplos a lo largo del texto.

Definición 0.3 Funciones, asignaciones y fórmulas booleanas:

- Una *función booleana* es una función cuyas variables (que llamaremos *variables booleanas*) toman valores en el conjunto $\{VERDADERO, FALSO\}$, que identificaremos con $\{0,1\}$, y que devuelve un valor en el mismo conjunto. Todas las funciones booleanas pueden definirse en términos de las tres funciones booleanas *conjunción* $\wedge$, *disyunción* $\vee$ y *negación* $\neg$, definidas según la tabla de verdad siguiente:

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

- Llamaremos *asignación booleana* o simplemente *asignación* a un elemento de $\{0,1\}^n$ (donde n es el número de variables distintas que intervienen en una función booleana) sobre el que evaluaremos la función. Diremos que la asignación satisface la función, o que la hace cierta, si al evaluarla en ella obtenemos 1 (o $VERDADERO$) como resultado.
- Una *fórmula booleana* es una fórmula con variables booleanas utilizando los operadores $\wedge, \vee$ y $\neg$.

Las funciones booleanas son un caso particular de funciones con imagen en el conjunto $\{0,1\}$. Llamaremos *problema decisional* al problema de determinar el lenguaje $L_f = \{x \in \{0,1\}^* : f(x) = 1\}$, donde f es una de las funciones anteriores. Identificaremos el problema computacional de, dado x, calcular $f(x)$, con el de decidir el lenguaje L_f, es decir, dado x, decidir si $x \in L_f$.

Una fórmula booleana φ decimos que es *satisfactible* si existe alguna asignación de variables $z \in \{0,1\}^n$ donde n es el número de variables de la fórmula, tal que $\varphi(z) = 1$. En otro caso decimos que no es satisfactible.

Definición 0.4 Una *fórmula booleana está en forma CNF* (del inglés *Conjunctive Normal Form*) si es de la forma

$$\bigwedge_i \left(\bigvee_j v_{i_j} \right)$$

donde cada v_{i_j} es una variable u_l o su negación (que denotaremos por $\overline{u_l}$). Los términos v_{i_j} reciben el nombre de *literales* y los términos $\left(\bigvee_j v_{i_j}\right)$ el de *cláusulas*. Una fórmula CNF se dice que es kCNF si cada cláusula contiene a lo sumo k literales.

El siguiente lema asegura que cualquier función booleana puede expresarse mediante una fórmula CNF de "tamaño" exponencial:

Lema 0.5 Dada una función booleana $f : \{0,1\}^l \to \{0,1\}$, existe una fórmula CNF φ en l variables de tamaño $l2^l$ tal que $\varphi(u) = f(u)$ para todo $u \in \{0,1\}^l$, donde por tamaño de una fórmula CNF entendemos es número de símbolos $\wedge$ y $\vee$ que contiene.

Esquema de la demostración: Para cada $v \in \{0,1\}^l$ se puede probar que existe (y de hecho se puede construir de forma relativamente sencilla) una cláusula $C_v(z_1, z_2, ..., z_l)$ en l variables tal que $C_v(v) = 0$ y $C_v(u) = 1$ para todo $u \in \{0,1\}^l$, $u \neq v$. Sea φ la conjunción de todas las cláusulas C_v con v tal que $f(v) = 0$,

$$\varphi = \bigwedge_{v : f(v)=0} C_v(z_1, z_2, ..., z_l) \, .$$

Nótese que φ tiene tamaño a lo sumo $l2^l$. Para todo $u \in \{0,1\}^l$ tal que $f(u) = 0$ se cumple que $C_u(u) = 0$ y por lo tanto $\varphi(u) = 0$. Si $f(u) = 1$ entonces $C_v(u) = 1$ para todo $v \in \{0,1\}^l$ tal que $f(v) = 0$ y por lo tanto $\varphi(u) = 1$, con lo que tenemos que para todo $u \in \{0,1\}^l$, $\varphi(u) = f(u)$. □

EJEMPLO 0.6 (Expresando la igualdad de cadenas)

La fórmula $(x_1 \vee \overline{y_1}) \wedge (\overline{x_1} \vee y_1)$ está en forma CNF y solo es satisfecha por aquellos valores de x_1 e y_1 que son iguales. Entonces la fórmula

$$(x_1 \vee \overline{y_1}) \wedge (\overline{x_1} \vee y_1) \wedge \cdots \wedge (x_n \vee \overline{y_n}) \wedge (\overline{x_n} \vee y_n)$$

es satisfecha si y solo si cada x_i tiene asignado el mismo valor que y_i.
Entonces aunque "=" no sea un operador booleano básico como $\wedge$ o $\vee$, podemos utilizarlo como abreviatura, puesto que la fórmula $\phi_1 = \phi_2$ es equivalente (en el sentido de que tiene las mismas asignaciones que la satisfacen) a $(\phi_1 \vee \overline{\phi_2}) \wedge (\overline{\phi_1} \vee \phi_2)$.

Notación de Landau

La siguiente es una notación para la comparación asintótica de funciones, debida en los casos de la *o* y la *O* al matemático alemán Edmund Landau. Permite establecer las funciones cota superior, inferior y ajustada asintóticas. La notación de las cotas superiores asintóticas, *o* y *O*, ya introducida en la asignatura de Cálculo Infinitesimal, será probablemente conocida por el lector, mientras que las de las cotas inferiores asintóticas y las cotas ajustadas asintóticas se obtienen a partir de las anteriores, aportando en algunos casos mayor claridad a la exposición.

Aunque esta notación es válida en un contexto más amplio, nos limitaremos a presentarla para el caso de funciones de $\mathbb{N}$ en $\mathbb{N}$, que serán las que trataremos en el texto.

Definición 0.7 Sean $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ dos funciones.

- *Cota superior asintótica.* Decimos que $f = O(g)$ si existe una constante $M \geq 0$ tal que $f(n) \leq M \cdot g(n)$ para todo n suficientemente grande. Decimos que $f = o(g)$ si para todo $\varepsilon > 0$ se cumple que $f(n) < \varepsilon \cdot g(n)$ para todo n suficientemente grande.
- *Cota inferior asintótica* . Decimos que $f = \Omega(g)$ si $g = O(f)$. Decimos que $f = \omega(g)$ si $g = o(f)$.
- *Cota ajustada asintótica* . Decimos que $f = \Theta(g)$ si $f = O(g)$ y $g = O(f)$.

Un estudio más profundo sobre esta notación aparece en [30].

CAPÍTULO 1

UNAS PINCELADAS SOBRE COMPUTACIÓN

Nadie habrá dejado de observar que con frecuencia el suelo se pliega de manera tal que una parte sube en ángulo recto con el plano del suelo, y luego la parte siguiente se coloca paralela a este plano, para dar paso a una nueva perpendicular, conducta que se repite en espiral o en línea quebrada hasta alturas sumamente variables. Agachándose y poniendo la mano izquierda en una de las partes verticales, y la derecha en la horizontal correspondiente, se está en posesión momentánea de un peldaño o escalón. Cada uno de estos peldaños, formados como se ve por dos elementos, se sitúa un tanto más arriba y adelante que el anterior, principio que da sentido a la escalera, ya que cualquiera otra combinación producirá formas quizá más bellas o pintorescas, pero incapaces de trasladar de una planta baja a un primer piso.

Las escaleras se suben de frente, pues hacia atrás o de costado resultan particularmente incómodas. La actitud natural consiste en mantenerse de pie, los brazos colgando sin esfuerzo, la cabeza erguida aunque no tanto que los ojos dejen de ver los peldaños inmediatamente superiores al que se pisa, y respirando lenta y regularmente. Para subir una escalera se comienza por levantar esa parte del cuerpo situada a la derecha abajo, envuelta casi siempre en cuero o gamuza, y que salvo excepciones cabe exactamente en el escalón. Puesta en el primer peldaño dicha parte, que para abreviar llamaremos pie, se recoge la parte equivalente de la izquierda (también llamada pie, pero que no ha de confundirse con el pie antes citado), y llevándola a la altura del pie, se le hace seguir hasta colocarla en el segundo peldaño, con lo cual en éste descansará el pie, y en el primero descansará el pie. (Los primeros peldaños son siempre los más difíciles, hasta adquirir la coordinación necesaria. La coincidencia de nombre entre el pie y el pie hace difícil la explicación. Cuídese especialmente de no levantar al mismo tiempo el pie y el pie).

Llegando en esta forma al segundo peldaño, basta repetir alternadamente los movimientos hasta encontrarse con el final de la escalera. Se sale de ella fácilmente, con un ligero golpe de talón que la fija en su sitio, del que no se moverá hasta el momento del descenso.

Julio Cortázar, Instrucciones para subir una escalera, *Historias de cronopios y de famas*, 1962.

Nuestro objetivo es estudiar la complejidad de ciertos problemas que son susceptibles de ser resueltos mediante computación, es decir, para los cuales hay un algoritmo (conjunto ordenado y autocontenido de operaciones que resuelven un problema en una cantidad finita de tiempo y espacio) que los resuelve. El texto de Julio Cortázar que aparece sobre estas líneas es un ejemplo pintoresco de algoritmo, que resuelve el problema (de índole no matemática) de subir una escalera. Como ya vimos en la introducción, la noción de computación está indisolublemente ligada a la de modelo computacional, que necesitaremos conocer de forma precisa a la hora de diseñar un algoritmo.

La tarea de describir matemáticamente un modelo computacional puede resultar ardua, pero una vez lo hemos logrado se nos plantea un primer problema: ¿Podemos resolver con este modelo todos los problemas que podemos resolver con otros modelos? Evidente-

mente no parece muy práctico desarrollar un modelo computacional para cada problema que se nos plantea, y mucho menos construirlo físicamente en caso de que fuera necesario. Afortunadamente, la respuesta a esta pregunta es afirmativa, y podremos encontrar un modelo matemático (varios, de hecho) relativamente simple, la *máquina de Turing*, capaz de simular cualquier otro modelo computacional.

Pero podemos ir más lejos, y ya que pretendemos estudiar complejidad de los problemas, preguntarnos si la eficiencia de los algoritmos que resuelven un mismo problema en distintos modelos es similar, o por el contrario existe algún modelo en el que es "mucho más eficiente" resolverlo (precisaremos esto más adelante). Entramos en el terreno de la *eficiencia computacional*. De nuevo la respuesta parece ser afirmativa, según la *tesis de Church* o de *Church-Turing*.

En base a esto, comenzaremos enfatizando la diferencia entre computabilidad y complejidad computacional (sección 1.1) y continuaremos nombrando algunos de los modelos computacionales más relevantes (sección 1.2), para finalizar con una descripción y un estudio pormenorizado del modelo de la máquina de Turing (secciones 1.3, 1.4, 1.5 y 1.6).

Como referencias bibliográficas, además de [3, capítulo 1], destacan [40, sección 2], [13, capítulo 6] y [34, capítulos 4 y 5].

1.1 COMPUTABILIDAD FRENTE A COMPLEJIDAD COMPUTACIONAL

Una vez hemos fijado un modelo computacional, una de las primeras preguntas que nos planteamos es ¿qué puede ser computado con nuestro modelo? O mejor aún, ¿qué no puede ser computado? En la introducción del capítulo ya adelantamos que podemos considerar que todos los modelos computacionales razonables y realizables son equivalentes en este sentido. Entonces las preguntas anteriores se transforman en: ¿son todos los problemas computables por un modelo determinado? La intuición y la experiencia nos sugieren que la respuesta es un no rotundo, pero podemos dar una demostración formal, utilizando un argumento de cardinalidad.

Definición 1.1

1. Decimos que un *problema es computable* cuando existe un algoritmo que lo resuelve, lo cual lleva implícito que lo hace en tiempo y espacio finitos. En caso contrario, decimos que el problema es *indecidible* o *incomputable* .

2. Decimos que un *lenguaje es computable* (respectivamente indecidible) si lo es su correspondiente problema decisional.

Lema 1.2 Existen (infinitos) problemas indecidibles.

Demostración. Fijaremos como modelo computacional la máquina de Turing, como haremos en el resto del texto en virtud de la tesis de Church (que veremos en la sección 1.2). La máquina de Turing que resuelve un problema se identifica con el algoritmo.

Según anunciamos en el capítulo preliminar (profundizaremos sobre esto en la sección 1.4), las máquinas de Turing pueden representarse como cadenas finitas sobre un alfabeto, y razonamos que hay una cantidad infinita numerable de cadenas sobre un alfabeto.

Consideremos ahora los problemas decisionales. Cada lenguaje define un único problema decisional, y de nuevo en el capítulo preliminar vimos que existe una cantidad infinita no numerable de lenguajes sobre un alfabeto.

Entonces, por cardinalidad, no es posible asociar a cada problema decisional una máquina de Turing que lo resuelva (o, dicho de otra forma, un algoritmo), y tendremos solo una infinidad numerable de problemas decisionales computables, con lo que nos resta una infinidad no numerable de problemas de este tipo indecidibles. □

Esta es una demostración no constructiva. Tendremos que esperar a la subsección 1.4.3 para ver ejemplos de lenguajes indecidibles.

Centrémonos ahora en los problemas computables. Ya sabemos que hay un algoritmo que los resuelve, luego podremos tener la solución en tiempo y espacio finitos. Pero *finito* no implica *pequeño*. Estamos entrando en el terreno de la *complejidad*. A continuación veremos dos ejemplos que nos ayudarán a comprenderla mejor.

EJEMPLO 1.3 *(Multiplicación)*

Consideremos el problema de multiplicar dos enteros n y m. Rápidamente se nos ocurren dos formas obvias de hacerlo: sumando n m veces (o viceversa), o aplicando el algoritmo clásico que se enseña en la escuela. Así, se tiene que $12 \times 11 = 12+12+12+12+12+12+12+12+12+12+12 = 132$ utilizando el método de la suma repetida, y el mismo resultado se obtiene con el algoritmo clásico:

$$\begin{array}{cccc} & & 1 & 2 \\ \times & & 1 & 1 \\ \hline & & 1 & 2 \\ & 1 & 2 & \\ \hline & 1 & 3 & 2 \end{array}$$

Probablemente el algoritmo de la suma repetida parezca más simple, pero sin duda diríamos que el algoritmo clásico es *mejor*.
Para multiplicar 12×11 utilizando el algoritmo de la suma repetida tenemos que hacer 10 sumas, mientras que con el algoritmo clásico realizamos dos multiplicaciones por un número de una cifra y una suma (si es que multiplicar por 1 merece recibir tal nombre).

El análisis de la eficiencia de los algoritmos se hace estudiando cómo aumenta el número de operaciones básicas (tales como sumas, multiplicaciones y, normalmente, divisiones de números de una sola cifra) al aumentar el tamaño de la entrada (en este caso el número de dígitos de los números que queremos multiplicar).
Entonces, el número de operaciones básicas al multiplicar dos números de n dígitos es como mínimo $n \cdot 10^{n-1}$ en el caso del algoritmo de la suma repetida, y como máximo $2 \cdot n^2$ en el caso del algoritmo clásico. Es decir, una calculadora de bolsillo, utilizando el algoritmo clásico, calcularía en menos tiempo el producto de dos enteros de 11 cifras que un supercomputador que empleara el algoritmo de la suma repetida, y aumentando ligeramente el tamaño de los números incluso una persona con papel y lápiz superaría al supercomputador. Esto pone de manifiesto que la mejora de los algoritmos es mucho más importante que el aumento de la velocidad de procesamiento de los ordenadores a la hora de resolver problemas de forma rápida.
Como curiosidad, la transformada rápida de Fourier permite multiplicar dos números de n cifras utilizando $O(nlog(n)log(log(n)))$ operaciones, para más detalles ver [3, subsección 10.6.1].

EJEMPLO 1.4 *(El problema del viajante o Traveling Salesman Problem)*

Consideremos ahora un viajante que debe visitar las 48 capitales de provincia de la España peninsular, recorriendo la menor distancia posible para ahorrar costes. Para simplificar el problema, supongamos que puede viajar en helicóptero entre dos capitales cualesquiera (y por lo tanto viaja en línea recta). Para asegurarnos de obtener siempre la solución al problema (es decir, el trayecto más corto), un algoritmo consistiría en elegir el orden en que visitará cada una de las 48 ciudades, y luego sumar las distancias entre cada una y la siguiente. En este caso tenemos 48 posibilidades para la primera ciudad, 47 para la segunda... Es decir, tenemos un total de 48! = 12413915529536072670862289047373375038521486354677700000000000 posibles itinerarios. Suponiendo que un ordenador pudiera procesar una ruta en el tiempo que tardaría la luz en recorrer en el vacío la distancia equivalente al diámetro de un átomo de hidrógeno ($3{,}3 \cdot 10^{-19} s$), tardaría 10 cuadrillones de veces la edad actual del universo en procesarlas todas. En efecto, el algoritmo terminará en un tiempo finito, pero probablemente el viante no tenga tanta paciencia.
En el problema del viajante tradicional se suele pedir que la ciudad de partida sea la misma que la de destino, reduciendo a 47! las posibilidaddes (aún muy por encima de las posibilidades de cualquier ordenador).

Estos dos ejemplos hacen referencia a la *eficiencia* a la hora de resolver un problema, pero de formas distintas. En el ejemplo de la multiplicación estamos comparando dos algoritmos que resuelven el mismo problema, y tras un análisis concluimos que el algoritmo clásico es más eficiente. De estas cuestiones se ocupa la *complejidad algorítmica* , dedicada al estudio pormenorizado de la eficiencia de los algoritmos.

En el sentido de la complejidad algorítmica, ya hemos visto que el algoritmo de búsqueda exhaustiva que soluciona el problema del viajante no es nada eficiente. Aunque ahora

no estamos en condiciones de probarlo (tendremos que esperar al capítulo 4), pese a que podemos encontrar un algoritmo mucho mejor que este, no hay grandes esperanzas de conseguir un algoritmo eficiente (lo cual no significa que el viajante tenga que renunciar a su viaje, como veremos en la sección 4.5). No es simplemente un problema del algoritmo, sino que esta ineficiencia es inherente al problema. Este es el objeto de estudio de la *Complejidad Computacional* , que se dedica a la clasificación de los *problemas* (y no de los algoritmos) según su "dificultad"(en el sentido de la eficiencia con que pueden ser resueltos). Así, desde el punto de vista de la Complejidad Computacional, el problema de la multiplicación es "más sencillo"que el del viajante. Precisaremos todos estos conceptos a lo largo del texto.

Evidentemente, la complejidad algorítmica y la computacional están íntimamente relacionadas, pues la dificultad de un problema (siempre en el sentido anterior) viene marcada por la eficiencia del mejor algoritmo que lo resuelve. Probablemente el lector esté más familiarizado con la complejidad algorítmica, y no tanto con la *Teoría de la complejidad computacional*, que es a la que nos dedicaremos.

1.2 DIVERSOS MODELOS COMPUTACIONALES: LA TESIS DE CHURCH

En esta sección hablaremos sucintamente sobre algunos de los modelos computacionales más conocidos, aparte de la máquina de Turing, para después presentar la tesis de Church, que conjetura la equivalencia de todos ellos.

La primera noción similar a la de funciones computables fue introducida por K. Gödel en su famosa tesis [23] en la que demuestra la incompletitud de la Teoría Elemental de Números. Se trataba de la clase de funciones que hoy conocemos como funciones primitivas recursivas, pero que él llamó “rekursiv”. Gödel introdujo la noción de computabilidad efectiva en una conferencia en Princeton en 1934, a la que asistió S.C. Kleene, estudiante de doctorado tutelado por A. Church. Gödel advirtió que era preciso admitir formas más generales de recursión, pues sus funciones “rekursiv” no cubrían todo el espectro de funciones computables, con lo que definió la clase de las funciones generales recursivas, naciendo la noción de algoritmo. Referencias más detalladas sobre las funciones generales recursivas son [7, capítulo 4], [14, capítulo 13] y [37, capítulo 10].

Church, ayudado por Kleene (ver [29]), desarolló la noción de λ-cálculo, alternativa a las funciones generales recursivas también basada en la recursividad que proporciona otra noción de algoritmo. Church pueba en [8] que coincide con la de Gödel.

Otro modelo computacional son los sistemas de Post, desarrollados por Emil Leon Post y que de nuevo son equivalentes a la máquina de Turing. En [35] y [14, capítulo 14] se puede encontrar la definición precisa.

Desde la introducción hemos hecho hincapié en el modelo subyacente en toda referen-

cia a un proceso computacional, y ya hemos adelantado que la tesis de Church solventaría el problema de la dependencia de dicho modelo. Para estos de los modelos que acabamos de citar, y también para todos aquellos que no hemos mencionado, se ha probado su equivalencia con todos los demás al poco tiempo de definirlos (ya sea directamente o por transitividad), muchas veces por sus propios creadores, y especialmente con uno de los modelos, la máquina de Turing (que definiremos de forma precisa en la sección 1.3).

En su artículo [8], Alonzo Church hace referencia a la noción intuitiva de *función efectivamente calculable* sobre enteros no negativos, que identifica con las funciones recursivas. Esta es una primera forma débil de la tesis de Church, que él concibió no como un teorema, puesto que el concepto "efectivamente calculable" no tenía una definición matemática precisa, sino como la formalización de dicha intuición, que ya existía previamente. Un enunciado más general es el siguiente:

Tesis de Church: Una función computable en algún modelo computacional físicamente realizable es computable en cualquier otro de tales modelos.

De nuevo, esta afirmación no tiene carácter matemático, pues no se puede formalizar la noción de modelo computacional físicamente realizable, y por lo tanto no es susceptible de ser probada, aunque sí podría ser refutada (algo poco verosímil, según los expertos). Para indagaciones más profundas sobre el carácter de la tesis de Church, ver [43].

Esta tesis soluciona el problema de la dependencia del modelo computacional. Ahora bien, aunque Manuel Blum dio una formalización axiomática de la teoría de la complejidad (casi) independiente del modelo elegido en [4], por comodidad es habitual fijar un modelo y trabajar sobre él. Dada su simplicidad definitoria, el modelo matemático universalmente elegido como patrón es el de la máquina de Turing, que de hecho ha inspirado la construcción de los ordenadores físicos. Puesto que Turing también concibió un análogo a la tesis de Church en términos de las máquinas de Turing, al siguiente caso particular de la anterior tesis se le conoce como tesis de Church-Turing , y es al que se suele hacer referencia en la literatura.

Tesis de Church-Turing: Todo modelo computacional físicamente realizable puede ser simulado por una máquina de Turing.

Cuando utilizamos el término *simular* nos referimos a que toda función computable por un modelo computacional físicamente realizable puede ser computada por una máquina de Turing, esto es, que la máquina de Turing puede *simular* el funcionamiento del otro modelo.

Una versión más fuerte de la tesis de Church-Turing, y más interesante desde el punto de vista de la Teoría de la Complejidad Computacional, es la siguiente:

Tesis de Church-Turing (versión fuerte): Todo modelo computacional físicamente realizable puede ser simulado por una máquina de Turing con sobrecoste polinómico.

Al decir que *puede ser simulado con sobrecoste polinómico* nos referimos a que t pasos de cálculo en el modelo inicial pueden ser simulados con t^c pasos de la máquina de Turing (en la subsección 1.3.1 precisaremos la noción de paso de cálculo).

Esta versión fuerte de la tesis de Church-Turing no es tan aceptada como la otra, principalmente debido al modelo de *computación cuántica*, que se ha demostrado que aventaja en determinados aspectos a la máquina de Turing (como en la factorización de enteros, con el algoritmo de Shor). Sin embargo, no está claro que los ordenadores cuánticos sean físicamente realizables.

1.3 LA MÁQUINA DE TURING

Ha llegado el momento de definir de forma precisa el modelo de la máquina de Turing, sobre el que trabajaremos en adelante. Este modelo fue definido por Alan Turing en su artículo [45], y él mismo se encargó, en sucesivas conferencias, de introducir ciertas modificaciones. Aunque se puede pensar en la máquina de Turing como un ordenador moderno simplificado, especialmente en el caso determinista, surgió (y en muchas ocasiones será preferible considerarla en este sentido) como un modo de formalizar los algortimos, para poder razonar matemáticamente sobre ellos. De hecho, uno de los motivos de su sencillez es que modeliza la idea de una persona haciendo cálculos provista de lápiz y un borrador. De esta forma, cada máquina resuelve un problema, representa un algoritmo.

1.3.1 La máquina determinista

Comenzaremos introduciendo la definción formal, que posteriormente explicaremos para hacer patente la sencillez a la que nos referíamos antes.

Definición 1.5 *(Máquina de Turing determinista)* Una máquina de Turing determinista con una sola *cinta de entrada* (que denotaremos por CE), en la que autorizamos solo lectura, y con k *cintas de trabajo* (que denotaremos por CT_i, $1 \leq i \leq k$) es un quíntuplo $M := (\Sigma, Q, q_0, F, \delta)$ donde:

- Σ es un conjunto finito (al que llamaremos *alfabeto de la máquina*), que suponemos que contiene dos símbolos auxiliares: $\triangleright$, al que llamaremos *cursor*, y $\square$, el *símbolo blanco*. Generalmente al referirnos al alfabeto de la máquina no consideraremos estos dos símbolos auxiliares.
- Q es un conjunto finito (al que llamaremos *espacio de estados*).
- $q_0 \in Q$ es el *estado inicial*.
- $F \subseteq Q$ es el conjunto de *estados finales*.
- $\delta : (Q \setminus F) \times \Sigma^{k+1} \longrightarrow Q \times \Sigma^k \times \{-1, 0, 1\}^{k+1}$ es una aplicación definida sobre un subconjunto de $(Q \setminus F) \times \Sigma^{k+1}$, llamada *función de transición*, que cumple que si $\delta(q, u_0, ..., u_k) = (p, v_1, ..., v_k, n_0, ..., n_k)$, con $q, p \in Q$, $u_0, ..., u_k, v_1, ..., v_k \in \Sigma$, $n_0, ..., n_k \in \{-1, 0, 1\}$, entonces:

- Si $u_i = \triangleright$ entonces $v_i = \triangleright$ y $n_i = 1$, $i \in \{1,...,k\}$. Si $u_0 = \triangleright$ entonces $n_0 = 1$.
- Para todo $u_i \neq \triangleright$ se tiene que $v_i \neq \triangleright$, $i \in \{1,...,k\}$.

Para entender la acción dinámica de una máquina de Turing introduciremos el concepto de sistema de transición asociado.

Definición 1.6 Dada una máquina de Turing $M := (\Sigma, Q, q_0, F, \delta)$ con una cinta de entrada y k cintas de trabajo, consideramos el grafo orientado cuyo conjunto de vértices es C_M y cuyo conjunto de aristas orientadas es $\rightarrow_M$, $(C_M, \rightarrow_M)$, que denominaremos *sistema de transición* .

Los elementos de C_M se denominan *configuraciones* y representan la imagen de la máquina en un instante determinado. Se tiene que $C_M \subseteq Q \times (\Sigma^*)^{k+1} \times \mathbb{N}^{k+1}$. Un elemento $C := (q, \mathbf{x}, \mathbf{y}_1, ..., \mathbf{y}_k, n_0, ..., n_k) \in Q \times (\Sigma^*)^{k+1} \times \mathbb{N}^{k+1}$ es una configuración (es decir, $C \in C_M$) si, y solo si, verifica:

- $q \in Q$, es un estado (el estado de la configuración).
- $\mathbf{x} := x_1 \cdots x_n \in \Sigma^*$ (por comodidad en el manejo de las cadenas, denotamos por n a la primera posición de $\mathbf{x}$ distinta de $\square$ a partir de la cual todos los símbolos son el símbolo blanco).
- $\mathbf{y}_i := y_{i,1} \cdots y_{i,s_i} \in \Sigma^*$ para cada $i = 1, ..., k$ (denotamos por s_i a la primera posición de $\mathbf{y}_i$ distinta de $\square$ a partir de la cual todos los símbolos son el símbolo blanco).
- $n_0, n_1, ..., n_k \in \mathbb{N}$ son las posiciones de los cabezales de la unidad de control en las diferentes cintas, $0 \leq n_0 \leq n+1$ para la cinta de entrada y $0 \leq n_i \leq s_i + 1$, $i = 1, ..., k$ para las k cintas de trabajo (en todas las cintas la posición 0 estará ocupada por el cursor).

Si $C = (q_p, \mathbf{x}, \mathbf{y}_1, ..., \mathbf{y}_k, n_0, ..., n_k)$ es una configuración con $q_p \in F$, la llamaremos *configuración final.*

Podemos representar gráficamente una configuración de una máquina de Turing para aclarar conceptos en lo que se conoce como modelo gráfico de una máquina de Turing.

Se divide cada una de las cintas (la cinta de entrada y las k cintas de trabajo) en celdas que pueden contener un símbolo del alfabeto Σ (o alguno de los símbolos auxiliares). Cada cinta tiene adosado un cabezal que se puede desplazar por ella, todos los cuales están asociados a una unidad de control capaz de almacenar un estado (es decir, una cantidad finita de información). La configuración C anterior viene representada por la figura 1.1.

No toda la información de la máquina de Turing será utilizada simultáneamente, sino que en cada paso de cálculo sólo se utiliza la celda de cada cinta marcada por el cabezal correspondiente de la unidad de control (n_i representa la posición señalada por el cabezal en la cinta i). El cursor $\triangleright$ indica el principio de cada cinta.

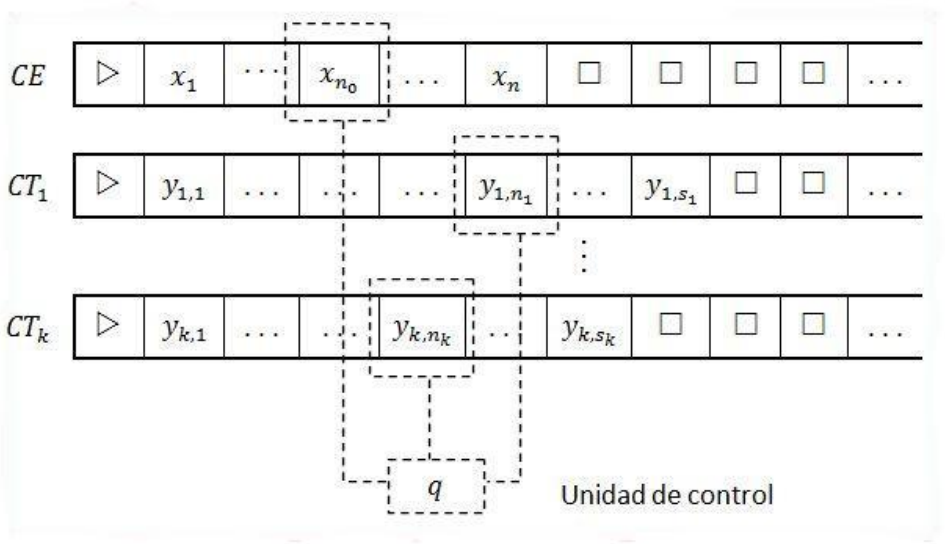

Figura 1.1: Representación gráfica de una máquina de Turing con una cinta de entrada y k cintas de trabajo en la configuración C.

Al comenzar una computación mediante una máquina de Turing, se introduce en la cinta de entrada el argumento de entrada de la computación (será una cadena, y colocaremos un símbolo en cada celda de dicha cinta, rellenando el resto de celdas con □) precedida del cursor ▷ (que por lo tanto ocupará la primera posición de la cinta). En las cintas de trabajo, se escribe en la primera posición el cursor y se rellena el resto de la cinta con el símbolo blanco. Nótese que las cintas son infinitas, pero en cada una tenemos siempre una cantidad finita de información.

Cuando una máquina de Turing se encuentra en una configuración como la anterior efectúa un *paso de cálculo* , que podemos dividir en cinco etapas: PARADA, LECTURA, TRANSICIÓN, ESCRITURA y MOVIMIENTOS.

1. PARADA. Se verifica la *condición de parada*, marcada por los estados finales. Si se cumple dicha condición de parada (es decir, el estado es uno de los estados finales), se devuelve el contenido de la última cinta de trabajo (por lo que en ocasiones también la llamaremos *cinta de salida*). El pseudocódigo que define esta etapa es el siguiente (para una máquina de Turing con k cintas de trabajo, actuando sobre la configuración C anterior):

```
si q ∉ F entonces
    LECTURA;
si no
    DEVOLVER contenido de CT_k;
fin si;
```

2. LECTURA. En esta fase se recuperan los contenidos de las celdas de cada cinta señaladas por los cabezales de la unidad de control.

   `contenido`$:=(q, x_{n_0}, y_{1,n_1}, ..., y_{k,n_k}) \in Q \times \Sigma^{k+1}$;
   TRANSICIÓN;

3. TRANSICIÓN. Se aplica la función de transición a la información leída en el paso anterior.

   `transición`$:=\delta$(`contenido`$) = (q', w_1, ..., w_k, \varepsilon_0, ..., \varepsilon_k) \in Q \times \Sigma^k \times \{-1,0,1\}^{k+1}$;
   ESCRITURA;

4. ESCRITURA. Esta etapa consta de dos partes: en la primera se cambia el contenido de la unidad de control al nuevo estado $q' \in Q$; en la segunda se sobreescribe el contenido de cada celda de las cintas de trabajo a la que apuntaba el cabezal por el nuevo símbolo correspondiente.

   $q:=q'$;
   para i **desde** 1 **hasta** k **hacer**
   $\quad y_{i,n_i}:=w_i$;
   fin para;
   MOVIMIENTOS;

5. MOVIMIENTOS. Se trata de mover los cabezales de la unidad de control de cada cinta conforme a lo indicado en la lista de movimientos $(\varepsilon_0, ..., \varepsilon_k) \in \{-1,0,1\}^{k+1}$, donde -1 significa mover una posición a la izquierda, 0 significa mantener la misma posición y 1 mover una posición a la derecha.

   para i **desde** 0 **hasta** k **hacer**
   $\quad n_i:=n_i+\varepsilon_i$;
   fin para;
   PARADA;

El resultado de un paso de cálculo sobre la configuración $C := (q, \mathbf{x}, \mathbf{y}_1, ..., \mathbf{y}_k, n_0, ..., n_k) \in Q \times (\Sigma^*)^{k+1} \times \mathbb{N}^{k+1}$, si $q \notin F$, es otra configuración $C' := (q', \mathbf{x}, \mathbf{y'}_1, ..., \mathbf{y'}_k, n'_0, ..., n'_k) \in Q \times (\Sigma^*)^{k+1} \times \mathbb{N}^{k+1}$ donde:

- q' es el nuevo estado.
- x, la entrada, no ha sido modificada.
- Si $y_i = (y_{i,1}, ..., y_{i,n_i}, ..., y_{i,s_i})$, entonces $y'_i = (y_{i,1}, ..., y_{i,n_{i-1}}, w_i, y_{i,n_{i+1}}, ..., y_{i,s_i})$, $i = 1, ..., k$. Es decir, reemplazamos el contenido de la celda a la que apuntaba el cabezal por el nuevo símbolo, sin modificar el resto.
- $n'_i := n_i + \varepsilon_i$, $i = 0, ..., k$, los cabezales han sido desplazados si procedía.

A continuación veremos dos ejemplos de máquinas de Turing.

EJEMPLO 1.7

La siguiente máquina de Turing $M = (\Sigma, Q, q_0, F, \delta)$ con una cinta de entrada y tres cintas de trabajo hace sumas y multiplicaciones en $\mathbb{F}_2$.

- El alfabeto de la máquina es $\Sigma = \{0,1\} \cup \{+,\times\} \cup \{\triangleright, \square\} \cup \{E\}$. El símbolo E será devuelto en caso de error en la computación (lo que ocurrirá solo si la entrada no es de la forma esperada).
- El espacio de estados es $Q = \{q_0, s, m, q_p\}$, asociaremos s y m con la operación suma y multiplicación, respectivamente.
- q_0 será el estado inicial.
- $F = \{q_p\}$, un único estado final (o de parada).
- La función de transición es la siguiente:

$$
\begin{array}{rccll}
\delta: & (Q \smallsetminus F) \times \Sigma^4 & \longrightarrow & Q \times \Sigma^3 \times \{-1,0,1\}^4 & \\
& (q_0, \triangleright, \triangleright, \triangleright, \triangleright) & \longmapsto & (q_0, \triangleright, \triangleright, \triangleright, 1,1,1,1) & \\
& (q_0, a, \square, \square, \square) & \longmapsto & (q_0, a, \square, \square, 1,0,0,0) & \forall\, a \in \{0,1\} \\
& (q_0, +, a, \square, \square) & \longmapsto & (s, a, \square, \square, 1,0,0,0) & \forall\, a \in \{0,1\} \\
& (q_0, \times, a, \square, \square) & \longmapsto & (m, a, \square, \square, 1,0,0,0) & \forall\, a \in \{0,1\} \\
& (e, b, a, \square, \square) & \longmapsto & (e, a, b, \square, 1,0,0,0) & \forall\, a,b \in \{0,1\},\ \forall\, e \in \{s,m\} \\
& (e, \square, a, b, \square) & \longmapsto & (q_p, a, b, a \circledast b, 0,0,0,0) & \forall\, a,b \in \{0,1\},\ \forall\, e \in \{s,m\}
\end{array}
$$

donde $a \circledast b$ es el elemento que aparece en la tabla, de las dos siguientes, correspondiente al estado e (que puede ser s o m) en la intersección de la fila correspondiente al símbolo a y la columna del símbolo b:

s	0	1
0	0	1
1	1	0

m	0	1
0	0	0
1	0	1

Para todos los argumentos de δ que no estén recogidos en la tabla anterior, teniendo en cuenta que CT_3 siempre estará "vacía" (pues al empezar está llena de símbolos blancos, salvo el cursor, y solo se rellena con los pasos correctos de la función de transición que acabamos de ver), y que el primer paso de cálculo siempre se realizará de forma correcta, la función se define por $\delta(q,a,b,c,\square) = (q_p, b, c, E, 0,0,0,0)$.

EJEMPLO 1.8

Construiremos ahora una máquina de Turing que, dada una cadena de 0 y 1, decida si es un palíndromo (es decir, si se lee igual de izquierda a derecha que de derecha a izquierda).
Consideremos el lenguaje $\mathsf{PAL} = \{x \in \{0,1\}^* : x \text{ es un palíndromo}\}$. La siguiente máquina de Turing $M = (\Sigma, Q, q_0, F, \delta)$ con una cinta de entrada y dos de trabajo decide el lenguaje PAL.

- El alfabeto de la máquina es $\Sigma = \{0,1\} \cup \{\triangleright, \square\}$.
- El espacio de estados es $Q = \{q_0, q_{copiar}, q_{izq}, q_{compr}, q_p\}$.
- q_0 es el estado inicial.
- $F = \{q_p\}$.
- La función de transición viene dada de la forma siguiente:

$$
\begin{array}{rccll}
\delta: & (Q \setminus F) \times \Sigma^3 & \longrightarrow & Q \times \Sigma^2 \times \{-1,0,1\}^3 & \\
& (q_0, \triangleright, \triangleright, \triangleright) & \longmapsto & (q_{copiar}, \triangleright, \triangleright, 1, 1, 1) & \\
& (q_{copiar}, a, \square, \square) & \longmapsto & (q_{copiar}, a, \square, 1, 1, 0) & \forall a \in \{0,1\} \\
& (q_{copiar}, \square, \square, \square) & \longmapsto & (q_{izq}, \square, \square, -1, 0, 0) & \\
& (q_{izq}, a, \square, \square) & \longmapsto & (q_{izq}, \square, \square, -1, 0, 0) & \forall a \in \{0,1\} \\
& (q_{izq}, \triangleright, \square, \square) & \longmapsto & (q_{compr}, \square, \square, 1, -1, 0) & \\
& (q_{compr}, a, a, \square) & \longmapsto & (q_{compr}, a, \square, 1, -1, 0) & \forall a \in \{0,1\} \\
& (q_{compr}, a, b, \square) & \longmapsto & (q_p, b, 0, 0, 0, 0) & \forall a, b \in \{0,1\},\ a \neq b \\
& (q_{compr}, \square, \triangleright, \square) & \longmapsto & (q_p, \triangleright, 1, 0, 0, 0) &
\end{array}
$$

Especificar de forma completa una máquina de Turing es tedioso y no siempre aclaratorio (al igual que sucede con las *Instrucciones para subir una escalera* con las que abrimos el capítulo, resulta poco práctico describir detalladamente acciones evidentes). Por ello en adelante daremos descripciones de su funcionamiento, ahorrándonos explicitar la función de transición. De esta forma, el funcionamiento de la máquina anterior se puede describir en cuatro pasos:

1. Recorrer de izquierda a derecha la cinta de entrada copiando su contenido en la primera cinta de trabajo.
2. Volver al principio de la cinta de entrada, recorriéndola de derecha a izquierda, manteniendo el cabezal de la primera cinta de trabajo al final.
3. Recorrer la cinta de entrada de izquierda a derecha y la primera cinta de trabajo de derecha a izquierda, comprobando en cada paso que los símbolos coincidan. Si en algún paso no coinciden, escribir 0 en la segunda cinta de trabajo y parar.
4. Escribir 1 en la segunda cinta de trabajo y parar.

Desde el punto de vista de la complejidad (algorítmica), esta máquina decide si una cadena está en PAL recorriéndola tres veces, es decir, si llamamos n a la longitud de la cadena de entrada, la máquina termina su computación en $3n$ pasos.

Cabe destacar que las máquinas que Turing describió por primera vez en [45] eran máquinas deterministas (él las llamó *automatic machines*) con una sola cinta de trabajo infinita en ambas direcciones (que el propio Turing compara con un papel, reforzando la idea que adelantábamos al principio de la sección de que pretendía formalizar los cálculos de una persona con lápiz y papel). En la sección 2.2 se prueba la equivalencia de ambas definiciones no solo en el sentido de que una máquina puede simular a la otra, sino también desde el punto de vista de la complejidad algorítmica (es decir, el sobrecoste de dicha simulación es aceptable).

1.3.2 Terminología básica y funciones computables

A continuación introduciremos algunas definiciones básicas, así como la noción de computabilidad asociada al modelo de la máquina de Turing.

Definición 1.9 Dadas dos configuraciones C y C' de una máquina de Turing M, escribiremos:

- $C \to_M C'$ para denotar que C' se obtiene de C en un solo paso de cálculo.
- $C \vdash_M C'$ para denotar que C' se obtiene de C en un número finito de pasos de cálculo (es decir, es alcanzable por un camino finito que parte de C dentro del grafo del sistema de transición asociado a M).

Definición 1.10 Sea M una máquina de Turing de alfabeto Σ (sin considerar los símbolos auxiliares), sea $x \in \Sigma^*$ una palabra en dicho alfabeto.

- Llamaremos *configuración inicial* sobre x a la configuración

$$I(x) := (q_0, \triangleright x \square \ldots, \triangleright \square \ldots, \ldots, \triangleright \square \ldots, 0, \ldots, 0) \in C_M,$$

  es decir, aquella en la que la cinta de entrada contiene la palabra x, todas las cintas de trabajo están "vacías" y los cabezales de la unidad de control en todas las cintas están en la posición del cursor.

- Se denomina *resultado* de M sobre la entrada x, y se representa por $\mathrm{Res}_M(x) \in \Sigma^*$ al contenido de la última cinta de trabajo de M una vez alcanzada la condición de parada.

- Se llama *conjunto de parada* de la máquina M, y se denota por $P(M)$, al conjunto de las entradas para las que dicha máquina termina su computación (es decir, alcanza un estado final):

$$P(M) := \left\{ x \in \Sigma^* : I(x) \vdash_M C \text{ y el estado de } C \text{ es } q \in F \right\}.$$

La siguiente terminología es propia de máquinas de Turing que resuelven problemas decisionales, es decir, aquellas cuyo resultado está en el conjunto $\{0,1\}$ (que significan aceptar y rechazar, respectivamente). Se trata de un tipo muy importante de problemas que trataremos constantemente (pues tenderemos a reescribir todos los problemas en estos términos, cuando sea posible), hasta el punto de considerar que, salvo que se indique expresamente otra cosa, las máquinas consideradas serán de esta clase, omitiendo mencionarlo en general.

Definición 1.11 Sea M una máquina de Turing de alfabeto Σ que resuelve un problema decisional y termina su ejecución en todas las entradas. Asumiremos que el resultado de la máquina es o bien 1, o bien 0.

- Una palabra $x \in \Sigma^*$ se dice que es *aceptada* por M si $\text{Res}_M(x) = 1$. En otro caso se dice que es *rechazada*.
- El conjunto de las palabras aceptadas por M se denomina *lenguaje aceptado por M*, y se representa por $L(M) := \{x \in \Sigma^* : \text{Res}_M(x) = 1\}$.

Las siguientes definiciones terminan de establecer la noción de computabilidad para las máquinas de Turing.

Definición 1.12 Un lenguaje $L \subseteq \Sigma^*$ se llama *recursivamente enumerable* si es el lenguaje aceptado por alguna máquina de Turing (es decir, si existe una máquina de Turing M con alfabeto Σ tal que $L = L(M)$). Se llama *lenguaje recursivo* si tanto él como su complementario $\Sigma^* \setminus L$ son recursivamente enumerables. Los lenguajes recursivamente enumerables pero no recursivos se denominan *indecidibles*.

Definición 1.13 Una función $f : D(f) \subseteq \Sigma^* \longrightarrow \Sigma^*$ es una *función computable* (por una máquina de Turing) si existe una máquina de Turing M con alfabeto Σ que cumpla $\text{Res}_M = f : D(f) \subseteq \Sigma^* \longrightarrow \Sigma^*$.

1.3.3 La máquina indeterminista

La introducción del indeterminismo en las máquinas de Turing nos permitirá aumentar considerablemente su potencia computacional, resolviendo de forma inmediata tareas que aparentemente resultan muy costosas para una máquina determinista. La discusión sobre si ese aumento de la potencia computacional es una mera apariencia o es real, conocida bajo el nombre de conjetura de Cook, es la cuestión abierta más importante de la Teoría de la Complejidad Computacional; la analizaremos con detalle en el capítulo 5.

Definición 1.14 Una *máquina de Turing indeterminista* con una sola cinta de entrada (que denotaremos por CE), en la que autorizamos solo lectura, y con k cintas de trabajo (que denotaremos por CT_i, $1 \leq i \leq k$) es un quíntuplo $M := (\Sigma, Q, q_0, F, \delta)$ donde Σ, Q, q_0 y F son como en la máquina determinista y la función de transición $\delta : (Q \setminus F) \times \Sigma^{k+1} \longrightarrow Q \times \Sigma^k \times \{-1,0,1\}^{k+1}$ es una correspondencia que también cumple que si $\delta(q, u_0, ..., u_k) = (p, v_1, ..., v_k, n_0, ..., n_k)$, con $q, p \in Q$, $u_0, ..., u_k, v_1, ..., v_k \in \Sigma$, $n_0, ..., n_k \in \{-1,0,1\}$, entonces:

- Si $u_i = \triangleright$ entonces $v_i = \triangleright$ y $n_i = 1$, $i \in \{1, ..., k\}$. Si $u_0 = \triangleright$ entonces $n_0 = 1$.
- Para todo $u_i \neq \triangleright$ se tiene que $v_i \neq \triangleright$, $i \in \{1, ..., k\}$.

La terminología de la sección 1.3.2, junto con la representación de la máquina mediante el sistema de transición, es aplicable también a máquinas de Turing indeterministas (de hecho, puesto que toda máquina determinista es un caso particular de máquina indeterminista, se podrían definir todas esas nociones en términos de máquinas indeterministas).

La diferencia fundamental con la máquina determinista reside en que aquella seguía una serie de pasos determinados de forma única por la entrada, mientras que en esta en un paso de cálculo concreto pueden realizarse diferentes operaciones que pueden dar lugar a distintos resultados. Es decir, en la máquina *determinista* la entrada *determina* la computación (y por ende el resultado), mientras que en la indeterminista una misma entrada puede admitir diferentes resultados.

En términos del grafo del sistema de transición, para una configuración inicial dada el subgrafo que representa los posibles pasos de cálculo de la máquina determinista hasta obtener el resultado es un camino (lineal), mientras que para la máquina indeterminista toma la forma de un árbol (cuyas hojas son todos los posibles resultados para la entrada).

Diremos que $y \in \Sigma^*$ es un resultado de la máquina indetermista M con entrada $x \in \Sigma^*$ si es el contenido de la última cinta de trabajo de alguna configuración final C_f tal que $I(x) \vdash_M C_f$.

El siguiente teorema asegura que las definiciones 1.12 y 1.13 son consistentes independientemente de que las máquinas sean deterministas o indeterministas (de ahí que no hayamos indicado en las mismas el tipo de determinismo).

Teorema 1.15 Si una máquina de Turing indeterminista decide un lenguaje (o computa una función), entonces existe una máquina de Turing determinista que decide el mismo lenguaje (o computa la misma función), y se verifica también el recíproco.

Demostración. Es obvio que si una máquina determinista decide un lenguaje o computa una función existe una máquina indeterminista que también lo hace, ella misma, pues una máquina determinista es un caso particular de una indeterminista.

Ahora queremos demostrar que se cumple el recíproco. Sea M una máquina indeterminista $(\Sigma, Q, q_0, F, \delta)$ con una sola cinta de lectura-escritura que actúa como cinta de entrada y de trabajo (el teorema 2.6, aunque enunciado en términos de máquinas deterministas, es también válido para el caso indeterminista con la misma demostración, y nos permite asumir esto; aunque sea un resultado presentado posteriormente no incurrimos en ningún fallo de lógica). Pretendemos definir una máquina determinista M_D que simule el funcionamiento de M.

La función de transición es una correspondencia $\delta : (Q \smallsetminus F) \times \Sigma \longrightarrow Q \times \Sigma \times \{-1, 0, 1\}$. Sea k el máximo número de valores distintos que toma la función de transición para cualquier par $(q, \sigma) \in (Q \smallsetminus F) \times \Sigma$ (será siempre una cantidad finita). Para cada uno de dichos pares numeraremos esos (a lo sumo k) valores distintos que toma δ.

La máquina determinista M_D será una máquina con una cinta de entrada y dos cintas de trabajo, con un alfabeto con k símbolos adicionales $\eta_1,...,\eta_k$, que servirán para indicar los valores de la función de transición δ. La segunda cinta de trabajo simulará la cinta de la máquina M, mientras que en la primera cinta de trabajo generaremos en cada etapa una cadena de elementos de $\{\eta_1,...,\eta_k\}$ siguiendo el orden de longitud creciente y dentro de las cadenas de la misma longitud el orden lexicográfico (esto es, en la primera etapa escribe la cadena η_1, en la segunda η_2,..., en la k-ésima η_k, en la $k+1$-ésima $\eta_1\eta_1$, a continuación $\eta_1\eta_2$ y así sucesivamente). También se amplía el conjunto de estados, para guardar tanto los estados de la máquina M como los propios que necesita M_D (que no detallamos, pero que marcan la etapa en la que se encuentra y la acción que debe realizar dentro de cada etapa, como describiremos a continuación).

En cada etapa trabajaremos sobre la entrada x simulando el funcionamiento de M siguiendo las indicaciones de la primera cinta de trabajo. Más concretamente, en la etapa r-ésima ($r>0$) la máquina M_D actúa de la forma siguiente:

1. M_D borra el contenido de la segunda cinta de trabajo (sobreescribe las posiciones utilizadas de la etapa anterior con $\square$) y copia en ella la entrada x de la cinta de entrada.

2. M_D escribe en la primera cinta de trabajo la r-ésima cadena según el orden descrito anteriormente, $\eta_{i_{r_1}}\cdots\eta_{i_{r_m}} \in \{\eta_1,...,\eta_k\}$, sobreescribiendo la cadena de la etapa anterior.

3. M_D simula el funcionamiento de M en la segunda cinta de trabajo, deteniéndose en a lo sumo m pasos de cálculo. En el paso j-ésimo, M_D lee el símbolo $\eta_{i_{r_j}}$ de la primera cinta de trabajo y busca en la numeración de la función de transición δ para el estado y el símbolo de la segunda cinta de trabajo que marque el cabezal de la unidad de control el resultado correspondiente a $\eta_{i_{r_j}}$, si existe (puede ser que para esos valores del estado y el símbolo el número de valores que toma δ sea menor que el marcado por $\eta_{i_{r_j}}$, en cuyo caso se pasa a la siguiente etapa); en ese caso se aplica dicho resultado a la segunda cinta de trabajo.

4. Si la simulación de M finaliza en esos m pasos de cálculo, también lo hace la de M_D (es decir, alcanza su propio estado final). En otro caso continúa con la siguiente etapa.

Es claro que si M no alcanza un estado final en la entrada x tampoco M_D lo hace. Por el contrario, si M alcanza dicho estado final para x es porque existe una cadena finita $\eta_{i_1}\cdots\eta_{i_m} \in \{\eta_1,...,\eta_k\}$ que describe las elecciones de los valores de la función de transición en cada paso hasta alcanzar el estado final (es decir, marca un camino en el sistema de transición). En ese caso, en alguna etapa esa cadena aparecerá en la primera cinta de trabajo de M_D y esta alcanzará el estado final, teniendo como contenido de la última cinta de trabajo el mismo que el de M, con lo que su resultado es el mismo (y no es necesario diferenciar

el caso en que M decide un lenguaje o computa una función, pues son esencialmente el mismo). □

Hemos querido hacer con todo detalle esta demostración para mostrar el estilo de las pruebas relacionadas con máquinas de Turing. Pese a su longitud, es trivial si la pensamos en términos del grafo del sistema de transición: lo único que estamos diciendo es que podemos extraer un camino que represente el sistema de transición de una máquina determinista del grafo en forma de árbol que representa el de la máquina indeterminista para una cierta configuración inicial. Nótese que nuestro método requiere probar cada una de las posibilidades de dicho árbol, es decir, estamos sustituyendo el indeterminismo por la fuerza bruta.

En la sección 3.1 veremos una noción de máquina de Turing indeterminista que resultará ser equivalente a la que acabamos de dar, y que puede ayudar a captar de forma más clara la esencia del indeterminismo.

1.4 LA MÁQUINA DE TURING UNIVERSAL Y EL PROBLEMA DE PARADA

Una de las primeras consideraciones sobre las máquinas de Turing que su creador hizo tras la publicación del artículo donde las definía fue la existencia de una máquina capaz de simular el funcionamiento de cualquier otra, a la que llamó *máquina universal*. Esta noción, que desdibuja las diferencias entre *software*, *hardware* y datos, motivó la aparición de los ordenadores personales que hoy en día manejamos, capaces de llevar a cabo una amplia gama de tareas.

1.4.1 Representación de máquinas de Turing como cadenas

Ya en el capítulo preliminar anticipamos que unos de los objetos matemáticos susceptibles de ser representados eran las máquinas de Turing. Por su sencillez, nos interesará representarlas como cadenas en el alfabeto binario $\{0,1\}$. El alfabeto que maneje la máquina a codificar no representa una dificultad, puesto que con un simple proceso de codificación podemos llevarlo al alfabeto binario, al igual que el espacio de estados (detallaremos estas cuestiones en la sección 2.2). Lo que verdaderamente determina la máquina de Turing es su función de transición, que podemos representar considerando la lista de todos sus posibles argumentos de entrada y sus correspondientes evaluaciones (que en el caso de máquinas indeterministas no tienen por qué ser únicas). Esta lista, expresada como una cadena, es fácilmente codificable en el alfabeto binario.

Por comodidad en el manejo de estas representaciones asumiremos que se cumplen las siguientes propiedades:

1. Toda cadena en $\{0,1\}^*$ representa alguna máquina de Turing. Podemos asegurarnos de que esto ocurre haciendo corresponder todas las cadenas que no son codificaciones válidas de ninguna máquina de Turing con alguna máquina prefijada, como aquella que en su primer paso se detiene y devuelve 0 para cualquier entrada.

2. Cada máquina de Turing puede ser representada por infinitas cadenas diferentes. Esto puede asumirse sin más que permitir que cada representación vaya seguida de un número arbitrario de 1's, que serán ignorados.

Denotaremos por $\llcorner M \lrcorner \in \{0,1\}^*$ a una representación de la máquina de Turing M como cadena binaria. Si $\alpha \in \{0,1\}^*$, denotamos por M_α a la máquina de Turing que α representa. En ocasiones abusaremos de la notación y denotaremos por M tanto a la máquina de Turing como a su representación como cadena binaria.

1.4.2 La máquina de Turing universal

Ahora estamos en condiciones de probar la existencia de la máquina capaz de simular cualquier otra máquina de Turing a partir de su representación como cadena.

Teorema 1.16 Existe una máquina de Turing $\mathscr{U}$ sobre el alfabeto $\{0,1\}$ tal que para todos $x, \alpha \in \{0,1\}^*$, $\mathrm{Res}_{\mathscr{U}}(\alpha, x) = \mathrm{Res}_{M_\alpha}(x)$. A esta máquina $\mathscr{U}$ la llamaremos *máquina de Turing universal*.

Demostración. Según veremos en el teorema 2.6 (de nuevo no incurrimos en fallos de lógica al utilizar este resultado posterior), podemos suponer que las máquinas de Turing que queremos simular tienen una cinta de entrada, de solo lectura, y dos cintas de trabajo; y que trabajan sobre el alfabeto binario ($\{0,1,\triangleright,\square\}$ en realidad), pues podemos transformar cualquier máquina de Turing en una equivalente con estas características, que será sobre la que trabajaremos.

Nuestra máquina $\mathscr{U}$ utilizará también el alfabeto $\{0,1,\triangleright,\square\}$, y tendrá como espacio de estados $Q = \{q_0, q_{prep}, q_{ej}, q_p\}$, donde q_0 es el estado inicial, q_{prep} marca los pasos destinados a preparar la simulación de la máquina M_α (copiar y adaptar el código, introducir la configuración inicial en x y otras tareas rutinarias menores), q_{ej} señala los pasos de simulación de la ejecución de M_α sobre la entrada x, y q_p es el estado final de la máquina $\mathscr{U}$, que se alcanza tras llegar a algún estado final de M_α en la simulación.

La máquina $\mathscr{U}$ tendrá una cinta de entrada (de solo lectura) y cuatro cintas de trabajo. La primera cinta de trabajo simulará la primera cinta de trabajo de M_α, la segunda cinta guardará la descripción de M_α, la tercera guardará el estado de la máquina simulada en cada momento, y la cuarta y última simulará la segunda cinta de trabajo de M_α, que es aquella en la que se escribe el resultado de la computación al finalizar. La figura 1.2 puede resultar útil para comprender la simulación.

De esta forma, en cada paso $\mathscr{U}$ busca en la descripción de M_α (segunda cinta de trabajo) lo que debe realizar (la evaluación de la función de transición), y una vez finalizada

la computación de M_α, $\mathscr{U}$ pasa al estado q_p, con lo que en la cinta de salida tenemos el mismo resultado que proporcionaría la máquina simulada. □

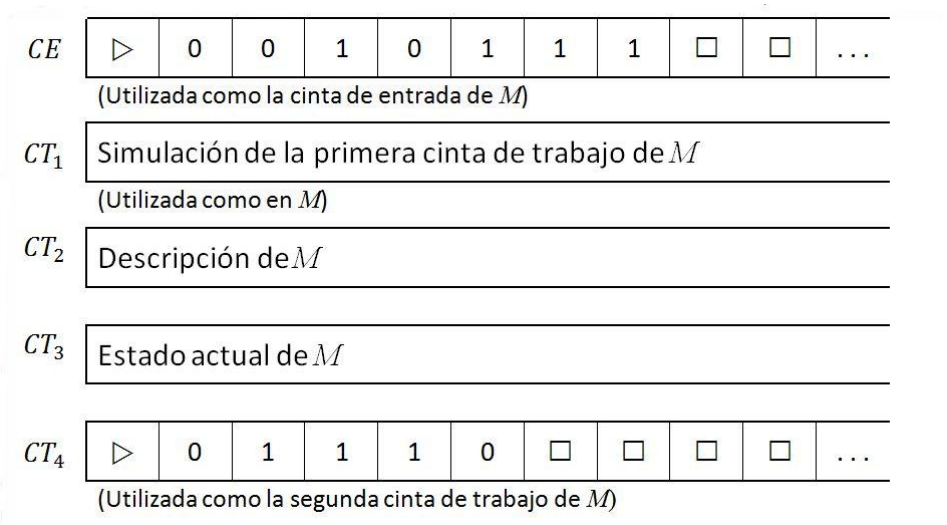

Figura 1.2: Esquema gráfico de cómo simula la máquina universal $\mathscr{U}$ el funcionamiento de otra máquina de Turing M (con dos cintas de trabajo).

A la vista de la demostración, queda claro que la máquina universal puede simular tanto máquinas deterministas como indeterministas, ya que se limita a utilizar sus funciones de transición.

1.4.3 El problema de parada

En la sección 1.1 introdujimos de forma sucinta la noción de incomputabilidad o indecidibilidad, llegando a demostrar, de forma no constructiva, la existencia de problemas indecidibles. Ahora ya nos encontramos en condiciones de dar ejemplos de funciones incomputables.

Teorema 1.17 Existe una función $\mathsf{UC} : \{0,1\}^* \longrightarrow \{0,1\}$ que no es computable por ninguna máquina de Turing.

Demostración. La función UC (del inglés *uncomputable*) se define de la forma siguiente: para cada $\alpha \in \{0,1\}^*$, si $\mathrm{Res}_{M_\alpha}(\alpha) = 1$ entonces $\mathsf{UC}(\alpha) = 0$; en otro caso ($\mathrm{Res}_{M_\alpha}(\alpha) \neq 1$ o no termina su computación) $\mathsf{UC}(\alpha) = 1$.

Razonemos por reducción al absurdo: supongamos que UC es computable, entonces existe una máquina de Turing M tal que $\mathrm{Res}_M(\alpha) = \mathsf{UC}(\alpha)$para todo $\alpha \in \{0,1\}^*$. En particular, $\mathrm{Res}_M(\llcorner M \lrcorner) = \mathsf{UC}(\llcorner M \lrcorner)$. Pero hemos llegado a una contradicción, por definición de UC, y por lo tanto no será una función computable. □

Esta función ha sido construida siguiendo un argumento de diagonalización, técnica muy utilizada en esta área de las matemáticas. Recibe este nombre porque si representáramos en la primera fila y columna de una tabla todas las posibles cadenas (binarias) ordenadas de igual forma e interpretáramos cada celda de la tabla como el resultado de ejecutar en la máquina cuya representación es la cadena de la fila correspondiente la entrada marcada por la columna, serían justamente las celdas de la diagonal principal las que nos interesarían. Se trata de un ejemplo que puede parecer artificial y carente de interés al lector (al fin y al cabo, ¿quién iba a querer computar una función así?). A continuación daremos otra función incomputable, mucho más interesante que la anterior.

La función HALT(parada, en inglés) toma como argumentos un par $(\alpha, x) \in \{0,1\}^* \times \{0,1\}^*$ y devuelve 1 si y solo si la máquina de Turing M_α termina su computación sobre la entrada x en un número finito de pasos. Computar esta función es lo que se conoce como *problema de parada*, y tiene un gran interés en la detección de fallos de programación, puesto que si los ordenadores pudieran computarla, dado un programa y una entrada para dicho programa podrían decidir si va a entrar en un bucle infinito. Desafortunadamente, es precisamente nuestro segundo ejemplo de función incomputable.

Teorema 1.18 La función HALT no es computable por ninguna máquina de Turing.

Demostración. De nuevo razonaremos por reducción al absurdo: supongamos que existe una máquina de Turing M_{HALT} que computa la función HALT (en un número finito de pasos). Consideremos la máquina de Turing M_{UC} que dada una entrada $\alpha \in \{0,1\}^*$ ejecuta M_{HALT} sobre la entrada (α, α). Si el resultado es 0 (lo cual significa que M_α no se detiene en la entrada α), entonces M_{UC} devuelve 1. En otro caso, M_{UC} utiliza la máquina de Turing universal $\mathscr{U}$ para calcular $\mathrm{Res}_{M_\alpha}(\alpha)$. Si $\mathrm{Res}_{M_\alpha}(\alpha) = 1$ entonces M_{UC} devuelve 0, en otro caso devuelve 1.

Entonces M_{UC} es una máquina de Turing que computa la función UC (resulta evidente sin más que revisar la definición de dicho lenguaje), pero esto contradice el teorema 1.17, por lo que HALT será también un lenguaje incomputable. □

La demostración que acabamos de dar utiliza una *reducción* para llevar un problema en otro, técnica muy utilizada en Complejidad. Estudiaremos más a fondo las reducciones en la sección 4.1.

Podemos identificar los problemas UC y HALT con los correspondientes lenguajes que definen, que denotaremos de la misma forma (el contexto nos permitirá diferenciarlos). Así $\mathsf{UC} := \{x \in \{0,1\}^* : \mathsf{UC}(x) = 1\}$ y $\mathsf{HALT} := \{x \in \{0,1\}^* : \mathsf{HALT}(x) = 1\}$ son dos lenguajes indecidibles, esto es, recursivamente enumerables pero no recursivos. Se pueden encontrar más detalles sobre el problema de parada en [34, sección 5.3].

Existen muchos otros ejemplos de problemas incomputables no relacionados directamente con las máquinas de Turing, como el problema de determinar si una ecuación diofántica tiene solución, el Décimo problema de Hilbert, cuya incomputabilidad probó Yuri Matiyasevich en 1970 (ver [38]).

Otro ejemplo famoso de lenguaje indecidible es el conjunto de los enunciados matemáticos ciertos, como demostró el propio Alan Turing en [45], al probar que el *Entscheidungsproblem* propuesto por Hilbert no tiene solución (consiste en encontrar un algoritmo para decidir la veracidad de un enunciado matemático).

1.5 MÁQUINAS DE TURING CON ORÁCULO

En su tesis doctoral de 1938, publicada luego como [46], Turing introduce un nuevo tipo de máquina modificada, la máquina con oráculo, que, dependiendo del oráculo elegido, puede tener una potencia computacional mayor que la de las máquinas consideradas hasta el momento. Burdamente, se trata de máquinas de Turing que, en cualquier paso de cálculo, pueden acudir a un oráculo para averiguar si una determinada palabra pertenece a un lenguaje prefijado, y a continuación proseguir con la computación.

Definición 1.19 Una *máquina de Turing con oráculo* $O \in \Sigma^*$ es una máquina de Turing (determinista o indeterminista) sobre el alfabeto Σ y con espacio de estados Q dotada de una cinta especial de lectura-escritura, llamada cinta del oráculo, y de tres estados especiales $q_{cons}, q_{sí}, q_{no} \in Q$, de forma que, en cualquier paso de cálculo, si la máquina accede al estado q_{cons} entonces lee (en un solo paso) el contenido de la cinta del oráculo $\omega \in \Sigma^*$ y cambia al estado $q_{sí}$ o q_{no} según $\omega \in O$ o no, continuando después la computación.

Las máquinas con oráculo suelen interpretarse como máquinas con una "caja negra", es decir, tales que no se conoce cómo se desarrolla una parte de su computación. Además de la potencia computacional añadida que pueden proporcionar dependiendo del lenguaje elegido, también son útiles como modelo en el que tenemos una máquina que precisa utilizar a su vez otras máquinas que no queremos detallar, y que englobamos en esa "caja negra".

1.6 OTRAS VARIANTES DE LA MÁQUINA DE TURING

Con el paso del tiempo la definición primitiva de máquina de Turing fue revelando algunas deficiencias, esencialmente en su modo de organizar la información (como ya comentamos, en un inicio solo se contemplaba una única cinta de trabajo infinita en ambas direcciones, por lo que la definición que hemos dado ya es una variación del modelo original, que simplifica su funcionamiento). Dichas deficiencias no afectaban al alcance del modelo computacional en el sentido de que toda función computable por cualquiera de las variaciones que veremos es también computable por la máquina primitiva; sino que se revelaban en el mayor número de pasos de cálculo necesarios para resolver algunos problemas, o bien desde el punto de vista teórico en la mayor complejidad de las demostraciones.

A continuación mostraremos algunas de las variantes de la máquina de Turing concebidas para enmendar dichas deficiencias, en la sección 2.2 demostraremos que son equivalentes a la defnición que hemos dado para la mayoría de los casos.

1.6.1 Máquinas ajenas a la entrada

Definición 1.20 Una *máquina de Turing ajena a la entrada* (u *oblivious*, en inglés) es una máquina de Turing cuyo movimiento de los cabezales en las cintas depende únicamente de la longitud de la entrada, y no de la propia entrada. Es decir, dada la entrada $x \in \Sigma^*$, la posición de un determinado cabezal en el paso i-ésimo depende solo de i y de $|x|$.

Estas máquinas son evidentemente más simples que las que hemos definido, y precisamente por ello nos resultará útil razonar con ellas en algunas demostraciones, como la del teorema de Cook-Levin (teorema 4.13). Suponiendo la entrada de la forma correcta (bit, símbolo de operación, bit, pues en otro caso se detiene), la máquina del ejemplo 1.7 que resolvía sumas y productos en $\mathbb{F}_2$ es de este tipo.

1.6.2 Máquinas con memoria de acceso aleatorio

Definición 1.21 Una *máquina de Turing con memoria de acceso aleatorio* (*random access memory, RAM* en inglés) es una máquina de Turing dotada de dos cintas especiales, una llamada cinta de memoria (inicialmente "vacía", es decir, rellena de $\square$) y la otra cinta de direcciones; además de poseer en su alfabeto dos símbolos especiales, L y E (leer y escribir, respectivamente) y un estado especial, $q_{acceder}$. Cuando la máquina alcanza el estado $q_{acceder}$, si la cinta de direcciones contiene $\llcorner i \lrcorner \mathsf{L}$ (donde $\llcorner i \lrcorner$ denota una representación de i en el alfabeto de la máquina), entonces el contenido de la i-ésima celda de la cinta de memoria es escrito en la celda de la cinta de direcciones contigua al símbolo L. Si en cambio la cinta de memoria contiene $\llcorner i \lrcorner \mathsf{E}\sigma$ (donde $\sigma \in \Sigma$), entoces σ es escrito en la i-ésima celda de la cinta de memoria.

Estas máquinas permiten realizar algunas tareas en menos pasos de cálculo que una máquina tradicional, sirviendo de inspiración para los ordenadores modernos. Para más detalles sobre estas máquinas ver [34, sección 4.4].

1.6.3 Máquinas con cintas multidimensionales

Uno de los mayores inconvenientes que plantean las máquinas de Turing tradicionales para resolver algunos problemas (pero que también simplifica mucho el modelo) es que los datos se almacenan de forma lineal en las cintas, con lo que la información relacionada con la que estamos utilizando puede estar almacenada en celdas lejanas, ya que cada celda solo tiene dos adyacentes. Una solución a esto es considerar cintas multidimensionales.

Definición 1.22 Una *máquina de Turing k-dimensional* es una máquina de Turing cuyas cintas son matrices k-dimensionales infinitas en un sentido por cada dimensión, y desde cada celda (llamamos celda a cada posición de la matriz) se permite al cabezal de la unidad de control acceder a las $2k$ celdas contiguas, excepto para aquellas que contengan el símbolo $\triangleright$, que marca los límites de la cinta (en cuyo caso no podrá desplazarse en ese sentido).

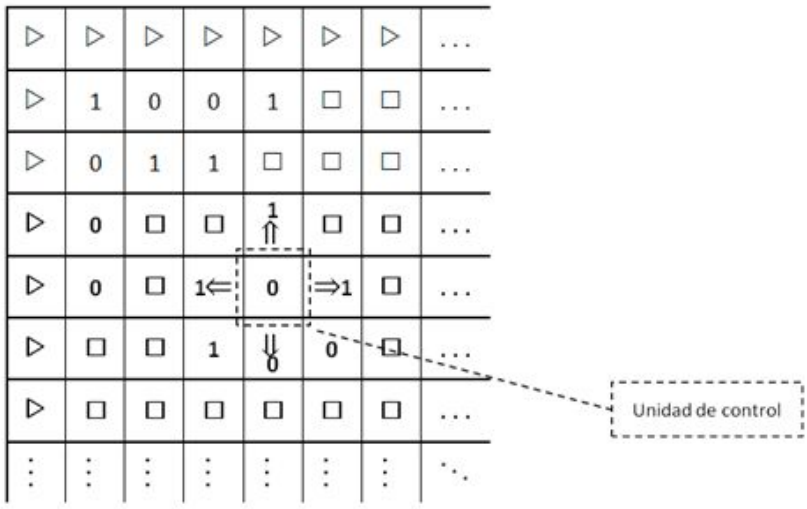

Figura 1.3: Cinta de una máquina de Turing 2-dimensional sobre el alfabeto binario, con los posibles movimientos del cabezal de la unidad de control.

CAPÍTULO 2

COMPLEJIDAD DE TIEMPO DETERMINISTA

En el capítulo anterior vimos con detalle la descripción de la máquina de Turing, modelo computacional sobre el que trabajaremos. Ahora ya podemos entrar plenamente en el ámbito que queremos estudiar: la Complejidad Computacional.

La Complejidad Computacional, como ya anunciamos en la sección 1.1, se dedica a la clasificación de los problemas según la eficiencia con la que pueden ser resueltos, o lo que es lo mismo, según la cantidad de recursos que necesitan. Los recursos fundamentales de una máquina de Turing son dos: el número de pasos de cálculo que debe efectuar antes de terminar la computación, que identificaremos con el tiempo (parece obvio que, suponiendo constante la velocidad con que realiza cada paso, el tiempo total de la computación será directamente proporcional al número de pasos); y por otra parte el espacio que ocupan los datos en las cintas. Precisamente esos son los recursos que intervienen en cualquier computación, ya sea con papel y lápiz o con un moderno ordenador.

La complejidad de una máquina de Turing se define como la medida de los recursos que precisa dicha máquina para llevar a cabo su computación, mientras que la complejidad de un problema es la medida de los mínimos recursos requeridos por cualquiera de las máquinas que lo resuelven.

Nos centraremos en el recurso del tiempo, que, a priori, resulta más valioso, ya que no es reutilizable como el espacio (las celdas de las cintas pueden sobreescribirse tantas veces como sea necesario) y por lo tanto limitará más los problemas que se pueden resolver, como ya avanzamos en la introducción.

2.1 PRIMEROS CONCEPTOS DE COMPLEJIDAD DE TIEMPO

Puesto que toda máquina no trivial necesitará al menos leer la entrada completa para llevar a cabo su computación, parece lógico expresar el número de pasos de cálculo como una función de la longitud de la entrada. Esta idea surgió a mediados de los años 60 de la mano de J. Hartmanis y R.E. Stearns (ver [25]), y también Blum trabajó sobre ella en [4] (aunque sin hacer referencia explícita al modelo computacional).

Definición 2.1 Sea M una máquina de Turing sobre el alfabeto Σ. Para cada palabra $x \in P(M) \subseteq \Sigma^*$, llamaremos *tiempo de cálculo de la máquina M sobre la entrada x* al número de pasos de cálculo que realiza sobre $I(x)$ hasta alcanzar alguna configuración final. Denotamos esta función por $t_M(x) : P(M) \longrightarrow \mathbb{N}$.

Definición 2.2 Sea M una máquina de Turing. Se define la *función de tiempo* $T_M(n) : \mathbb{N} \longrightarrow \mathbb{N}$ por $T_M(n) := \text{máx}\{t_M(x) : x \in P(M),\ |x| \leq n\}$. Diremos que la máquina M funciona en tiempo T_M (o en tiempo $T_M(n)$, para enfatizar la dependencia de la longitud de la entrada).

Esta es la función de complejidad de tiempo del caso peor, en el sentido de que su valor es el número de pasos de cálculo que tiene que realizar la máquina para terminar la computación dada la peor entrada posible. Aunque será la noción de tiempo con la que trabajaremos, en ocasiones puede distorsionar el estudio del tiempo de una máquina determinada, pues las entradas para las que el número de pasos de cálculo se disparan pueden ser muy escasas. Por ello para el análisis efectivo de algoritmos se suele considerar también la complejidad del caso medio, sobre la que se pueden encontrar más detalles en [3, capítulo 18].

EJEMPLO 2.3

Como ya comentamos en el ejemplo 1.8, la máquina allí descrita que decidía si una entrada era un palíndromo funciona en tiempo $3n$.

Funciones constructibles en tiempo

Para evitar anomalías, nos restringiremos a utilizar como cotas de tiempo para las máquinas de Turing funciones constructibles en tiempo.

Definición 2.4 Diremos que una función $f : \mathbb{N} \longrightarrow \mathbb{N}$ es *constructible en tiempo* si $f(n) \geq n$ y existe una máquina de Turing determinista M sobre el alfabeto unario $\Sigma = \{1\}$ tal que M termina su computación en todas las entradas, y para todo $n \in \{1\}^* = \mathbb{N}$, calcula $f(n) \in \{1\}^* = \mathbb{N}$ en tiempo $T_M(n) = O(f(n))$.

La condición $f(n) \geq n$ en la definición simplemente nos asegura que la máquina puede leer la entrada completa.

Para $k \in \mathbb{N}$ fijo son funciones constructibles en tiempo las polinómicas $f(n) = n^k$ y las expo-polinómicas $f(n) = 2^{n^k}$ o $f(n) = n\lfloor \log(n) \rfloor^k$. Para $c \in \mathbb{R}$ también lo son las exponenciales $f(n) = 2^{cn}$ o doblemente exponenciales $f(n) = 2^{2^{cn}}$. No pueden ser constructibles en tiempo las funciones $f(n) = o(n)$, pues no se cumpliría la primera condición de la definición (y por lo tanto la máquina no tendría suficiente tiempo para leer la entrada).

2.2 ROBUSTEZ DE LA DEFINICIÓN DE MÁQUINA DE TURING

Una vez hemos definido el tiempo de ejecución de una máquina de Turing, nos planteamos cómo le afectan las modificaciones sobre la máquina, algunas de las cuales ya vimos en el capítulo 1 y adelantamos que proporcionaban máquinas equivalentes en el sentido de que pueden ser simuladas por la máquina de nuestra definición. Se trata, pues, de comprobar que no afectan de forma desmedida al tiempo, hecho de gran relevancia al que daremos sentido en la sección 2.3.

Muchas de las demostraciones que daremos a continuación no están detalladas con total precisión (pues para ello tendríamos que dar la descripción formal de las correspondientes máquinas de Turing, que ya hemos visto que suele resultar tedioso y poco informativo), sino que se ofrece un esquema con la idea general.

2.2.1 Independencia de alfabetos

Teorema 2.5 (**Independencia de alfabetos**) Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo, sea una máquina de Turing M con alfabeto Γ (con al menos dos elementos) que calcula una función f definida sobre un subconjunto adecuado de Γ^* en tiempo $T(n)$, donde n es la longitud de la representación binaria de su entrada. Entonces f es computable (tras la correspondiente codificación) por una máquina $\tilde{M}$ sobre el alfabeto $\{0, 1, \triangleright, \square\}$ en tiempo (menor que) $4\log(\sharp\Gamma)T(n)$.

Esquema de la demostración: Sea M una máquina de Turing con alfabeto Γ, k cintas de trabajo y conjunto de estados Q que calcula la función f en tiempo $T(n)$. Describiremos una máquina equivalente $\tilde{M}$ con k cintas de trabajo sobre el alfabeto (binario, salvo los símbolos auxiliares) $\{0, 1, \triangleright, \square\}$ y espacio de estados $\tilde{Q}$ que calcule f (tras la correspondiente codificación de alfabetos).

La idea de la demostración es simple: podemos codificar cualquier elemento de Γ utilizando $\log(\sharp\Gamma)$ símbolos del alfabeto binario (recordemos que, por convenio, el logaritmo es en base 2, y que si no es un natural consideramos $\lceil \log(\sharp\Gamma) \rceil$). Entonces cada una de las

cintas de $\tilde{M}$ codificará una cinta de M, de forma que para cada celda de una cinta de M tendremos $\log(\sharp\Gamma)$ celdas en la correspondiente cinta de $\tilde{M}$, como se puede apreciar en la figura 2.1. Para simular un paso de cálculo M la máquina $\tilde{M}$ procederá de la siguiente forma:

1. Efectúa $\log(\sharp\Gamma)$ pasos de cálculo para leer de cada cinta los $\log(\sharp\Gamma)$ símbolos que codifican un símbolo de Γ.
2. Utiliza los estados para almacenar (en la unidad de control) los símbolos leídos.
3. Utiliza la función de transición de M para obtener los nuevos símbolos y el nuevo estado de M.
4. Almacena dicha información en la unidad de control.
5. Efectúa $\log(\sharp\Gamma)$ pasos de cálculo para escribir en cada cinta las codificaciones de dichos nuevos símbolos.
6. Efectúa a lo sumo $\log(\sharp\Gamma)$ pasos de cálculo para mover los cabezales a las celdas correspondientes (pues solo puede desplazarse una celda en cada paso).

La máquina $\tilde{M}$ necesitará entonces un espacio de estados que le permita guardar en la unidad de control en cada paso de cálculo el estado de M, k o $k+1$ (dependiendo de si está en la fase de lectura o de escritura) símbolos de Γ y un contador desde 1 hasta $\log(\sharp\Gamma)$. El espacio de estados $\tilde{Q}$ deberá tener entonces al menos $c(\sharp Q)(\sharp\Gamma)^{k+1}$ elementos, donde c es una constante conveniente que nos permitirá guardar otros datos más técnicos que podamos necesitar.

Entonces es claro que por cada paso de cálculo de M se realizarán a lo sumo $4\log(\sharp\Gamma)$ pasos de $\tilde{M}$ (esta cota no llega a alcanzarse, podría rebajarse a $3\log(\sharp\Gamma)$ aunque dejamos ese pequeño margen para asegurarnos de que otras operaciones de menor importancia no requieren más pasos de cálculo, y por lo tanto garantizar que la cota es válida). Entonces si para una entrada $x \in \Gamma^*$ el tiempo de M es $t_M(x)$, para $\tilde{M}$ será $t_M(\tilde{x}) < 4\log(\sharp\Gamma)\, t_M(x)$, y por lo tanto si el tiempo de M es $T(n)$, el de $\tilde{M}$ será $4\log(\sharp\Gamma)\, T(n)$. □

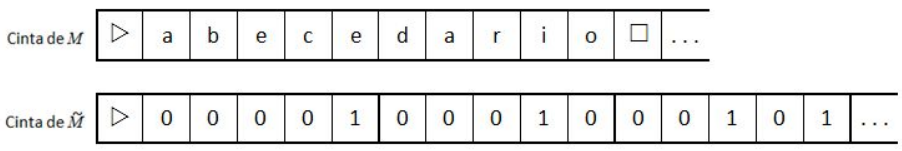

Figura 2.1: Ejemplo de codificación de una cinta de M, que utiliza el alfabeto español, de 27 símbolos, en el alfabeto binario según la codificación binaria obvia. Como $\lceil\log(27)\rceil = 5$ para codificar cada símbolo del alfabeto usual necesitamos 5 del binario

Además de la codificación entre alfabetos, la otra clave de la demostración es el aumento del nuevo espacio de estados, que nos permite suplir los símbolos que faltan en el alfabeto. En general se pueden simular varias unidades de control con sus respectivos espacios de estados por medio de una sola ampliando el espacio de estados de esta para que contenga todos los elementos del producto cartesiano de los espacios de estados originales. Recíprocamente, es inmediato reducir el espacio de estados sin más que añadir una nueva cinta de trabajo que simule la unidad de control, almacenando los estados convenientemente codificados al alfabeto.

Este teorema, al proporcionar una simulación con una cota de tiempo para el cambio entre cualquier alfabeto con más de dos elementos y el binario, nos la proporciona también entre dos alfabetos cualesquiera con más de dos elementos (sin más que utilizar solo dos símbolos del alfabeto de llegada).

2.2.2 Independencia del número de cintas y simulación por máquinas ajenas a la entrada y multidimensionales

Procedemos ahora a probar un resultado que ya hemos utilizado en otras demostraciones dado que nos permite asumir una simplificación importante:

Teorema 2.6 (**Reducción de cintas**) Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo, sea una máquina de Turing M con alfabeto Σ y k cintas ($k-1$ cintas de trabajo) que calcula una función f definida sobre un subconjunto adecuado de Σ^* en tiempo $T(n)$. Entonces f es computable por una máquina de Turing $\tilde{M}$ con una sola cinta de lectura-escritura (que actúa como cinta de entrada y de trabajo) en tiempo $5kT(n)^2$.

Esquema de la demostración: La idea fundamental es codificar las k cintas de M en una sola cinta de $\tilde{M}$ utlizando las posiciones $1, k+1, 2k+1, \ldots$ para los símbolos de la primera cinta, las celdas $2, k+2, 2k+2, \ldots$ para los de la segunda y así sucesivamente (ver figura 2.2). El alfabeto $\tilde{\Sigma}$ de $\tilde{M}$ contendrá, para cada símbolo $\sigma \in \Sigma$, los símbolos σ y $\hat{\sigma}$ (incluso para los símbolos auxiliares). En la codificación de cada cinta de M habrá exactamente un símbolo del tipo $\hat{}$, que marcará la posición de la cinta de M en la que se encuentra el cabezal de la unidad de control.

$\tilde{M}$ no modificará las $n+1$ primeras posiciones de la cinta, en las que se encuentra la entrada, sino que empleará $O(n^2)$ pasos de cálculo para copiar dicha entrada, símbolo a símbolo, en las posiciones de la cinta que le corresponden según lo explicado.

Para simular un paso de M, la máquina $\tilde{M}$ realizará las siguientes operaciones:

1. Recorre la cinta de izquierda a derecha guardando en la unidad de control (haciendo uso de sus estados) los k símbolos marcados con $\hat{}$.

2. Utiliza la función de transición de M para determinar el nuevo estado, los nuevos símbolos y los movimientos de los cabezales de la unidad de control en las cintas de M, que almacena por medio de los estados en la unidad de control.

3. Recorre la cinta de derecha a izquierda actualizando la información según corresponda.

Al finalizar la computación, se deberán efectuar de nuevo $O(n^2)$ pasos para copiar en las primeras posiciones de la cinta el contenido de las celdas correspondientes a la k-ésima cinta de M, borrando el resto de símbolos. El resultado de ambas máquinas es, claramente, el mismo.

Nótese además que, para entradas de longitud n, M nunca llega más allá de la posición $T(n)$ en ninguna de sus cintas, por lo que $\tilde{M}$ nunca irá más allá de la posición $2n + kT(n) \leq (k+2)T(n)$ en su cinta. Entonces para cada uno de los como máximo $T(n)$ pasos de cálculo de M, $\tilde{M}$ realizará como máximo $5kT(n)$ pasos (moverse adelante y atrás en la cinta requiere menos de $4kT(n)$ pasos, pues si $k \geq 2$ entonces $(k+2)T(n) \leq 2kT(n)$, y se recorre dos veces, una en cada sentido; y se necesitan algunos pasos más para actualizar la información). Por lo tanto el tiempo de la máquina $\tilde{M}$ será como máximo $5kT(n)^2$ (los pasos necesarios para situar la copia de la entrada y el resultado entran en el margen que hemos considerado). □

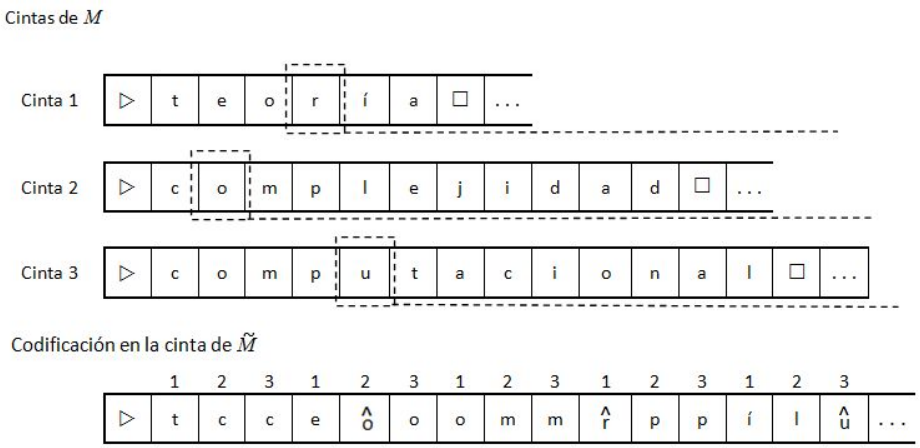

Figura 2.2: Codificación de tres cintas en una, marcando con ˆ las posiciones de los cabezales de la unidad de control de M.

En [25] podemos encontrar otra demostración basada en la misma idea, pero para máquinas bidireccionales (ver subsección 2.2.3). Obsérvese que la máquina de la demostración anterior es ajena a la entrada, esto es, el movimiento de su cabezal no depende de la entrada sino de su longitud. Esto demuestra, por lo tanto, el siguiente teorema:

Teorema 2.7 (**Simulación por máquinas ajenas a la entrada**) Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo, sea una máquina de Turing M con alfabeto Σ que calcula una función f definida sobre un subconjunto adecuado de Σ^* en tiempo $T(n)$. Entonces f es computable por una máquina de Turing $\tilde{M}$ ajena a la entrada en tiempo $O(T(n)^2)$.

Hay un resultado más fino sobre la simulación por máquinas ajenas a la entrada, que rebaja la cota de tiempo a $O(T(n)\log(T(n))$, demostrado en [42]. En este artículo también se puede encontrar la demostración del siguiente teorema.

Teorema 2.8 (**Simulación de máquinas multidimensionales**) Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo, sea una máquina de Turing k-dimensional M con alfabeto Σ que calcula una función f definida sobre un subconjunto adecuado de Σ^* en tiempo $T(n)$. Entonces f es computable por una máquina de Turing $\tilde{M}$ unidimensional en tiempo $O(T(n)^{2-\frac{1}{k}})$.

2.2.3 Máquinas con cintas bidireccionales

Definición 2.9 Se define una *máquina de Turing bidireccional* como una máquina cuyas cintas son infinitas en ambas direcciones.

Teorema 2.10 (**Simulación de máquinas bidireccionales**) Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo, sea una máquina de Turing bidireccional M con alfabeto Σ que calcula una función f definida sobre un subconjunto adecuado de Σ^* en tiempo $T(n)$. Entonces f es computable por una máquina de Turing $\tilde{M}$ unidireccional en tiempo $4T(n)$.

Esquema de la demostración: La idea de la demostración es la que se ilustra en la figura 2.3. El alfabeto de la máquina $\tilde{M}$ será Σ^2 (cada símbolo del alfabeto de $\tilde{M}$ se corresponde con un par de símbolos del alfabeto de M). Codificamos cada cinta de M utilizando una cinta infinita en un sentido "doblándola" por una posición cualquiera, con lo que cada celda de la cinta de $\tilde{M}$ codifica dos celdas de la cinta de M. $\tilde{M}$ ignorará uno de los símbolos de cada celda, siempre el mismo, aplicando la función de transición de M de forma normal hasta que esta requiera "pasar el pliegue" de alguna cinta, en cuyo caso se pasa a considerar en esa cinta el otro símbolo, ignorando aquel que hasta ese momento utilizaba. Será necesario guardar, por medio de los estados (consecuentemente será necesario ampliar el espacio de estados), si se considera el primer o el segundo símbolo de cada cinta, y en caso de que se considere el segundo, invertir el sentido de los movimientos (desplazar el cabezal hacia la izquierda en ese caso equivale a desplazarlo hacia la derecha en la cinta de $\tilde{M}$, y viceversa).

Entonces en cada paso de cálculo $\tilde{M}$ leerá los dos símbolos contenidos en la celda correspondiente, los guardará en un estado de la unidad de control, decidirá cuál de los dos es el que le interesa (hasta aquí un paso de cálculo), aplicará la función de transición de M y escribirá de nuevo en la celda el resultado junto con el símbolo que no utilizó (otro paso de cálculo), y finalmente moverá el cabezal en la dirección que corresponda, para lo cual necesitará comprobar, con el estado, en qué parte de la cinta se encuentra (lo que precisa un tercer paso de cálculo). Entonces el tiempo de la máquina $\tilde{M}$ será siempre menor que $4T(n)$ (consideramos un margen de $T(n)$ para hipotéticos pasos de menor importancia que no hayamos tenido en cuenta). □

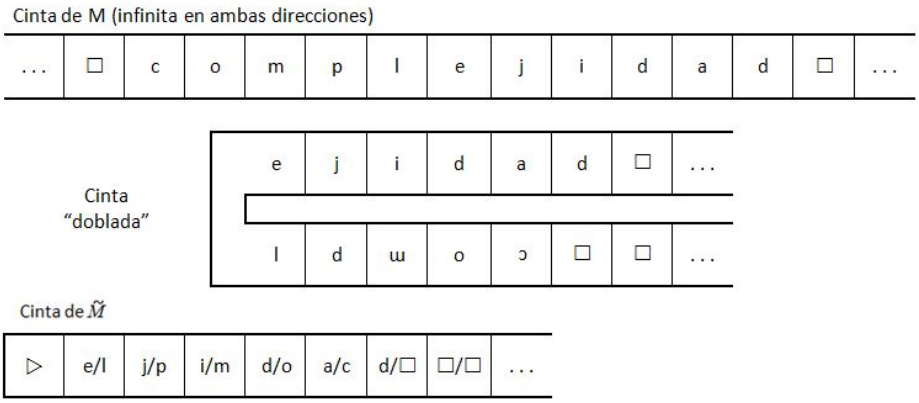

Figura 2.3: Representación gráfica del método de codificación de una cinta infinita en ambas direcciones por una infinita en una dirección, "doblándola".

2.2.4 Simulación por la máquina universal

En la sección 1.4 definimos la noción de máquina de Turing universal, demostrando (sin entrar en detalles excesivos) cómo podía simular cualquier otra máquina. Ahora nos interesa saber con qué sobrecoste de tiempo lo hace.

Teorema 2.11 (Simulación por máquina universal, versión relajada) Existe una máquina de Turing universal $\mathscr{U}$ sobre el alfabeto $\{0,1\}$ tal que para todos $x, \alpha \in \{0,1\}^*$, $\text{Res}_{\mathscr{U}}(\alpha, x) = \text{Res}_{M_\alpha}(x)$; y si M_α termina su ejecución sobre la entrada x en tiempo $t_{M_\alpha}(x)$ entoces $t_{\mathscr{U}}(\alpha, x) = C(t_{M_\alpha}(x))^2$, donde C es una constante que depende de las características de la máquina M_α, pero no de $|x|$.

Esquema de la demostración: La primera parte es el teorema 1.16, ya demostrado. Sobre aquella construcción de la máquina universal (reflejada en la figura 1.2) razonaremos ahora para calcular el tiempo de la simulación.

En primer lugar, como acabamos de ver en los teoremas 2.5 y 2.6, la suposición de que M_α tiene solo una cinta de trabajo (en este aspecto el teorema 2.6 es algo más fuerte) y trabaja sobre el alfabeto binario induce una ralentización cuadrática (si M_α funcionaba en tiempo $t_0 = t_{M_\alpha}(x)$ sobre x, ahora lo hará en tiempo $C' t_0^2$, donde C' es una constante que depende del tamaño del alfabeto de M_α y de su número de cintas antes de las reducciones). Además, cada paso de cálculo de M_α es simulado mediante C'' pasos de cálculo de $\mathscr{U}$, donde C'' es una constante que depende del tamaño de la representación de la función de transición de M_α que $\mathscr{U}$ guarda en su segunda cinta de trabajo.

Entonces $\mathscr{U}$ terminará su computación sobre (α, x) en tiempo menor que $C t_0^2$, donde C no depende de la longitud de la entrada de M_α sino de las características de dicha máquina. □

Aunque no lo demostraremos, el siguiente resultado de simulación por la máquina universal es más fino que el que acabamos de dar. La prueba, que puede encontrarse en [3, sección 1.7], se basa en la simulación de una máquina con k cintas por una con solo 2 cintas (incluyendo en ambos casos la de entrada) en tiempo $O(T\log(T))$, donde T es una función constructible en tiempo que denota el tiempo de la máquina simulada. Dicho resultado se debe a F.C. Hennie y R.E. Stearns y fue publicado en [26].

Teorema 2.12 **(Simulación por máquina universal, versión eficiente)** Existe una máquina de Turing universal $\mathscr{U}$ sobre el alfabeto $\{0,1\}$ tal que para todos $x, \alpha \in \{0,1\}^*$, $\mathrm{Res}_{\mathscr{U}}(\alpha, x) = \mathrm{Res}_{M_\alpha}(x)$; y si M_α termina su ejecución sobre la entrada x en tiempo $t_{M_\alpha}(x)$ entoces $t_{\mathscr{U}}(\alpha, x) = C\, t_{M_\alpha}(x)\log(t_{M_\alpha}(x))$, donde C es una constante que depende de las características de la máquina M_α, pero no de $|x|$.

2.2.5 Aceleración lineal

El siguiente resultado, aunque no compara variantes de la máquina de Turing como los anteriores, nos proporciona un método para "acelerar" de forma lineal el cómputo de un problema, demostrando que el papel que juegan las constantes en cuanto al tiempo de una máquina es irrelevante.

Teorema 2.13 (**de aceleración lineal**) Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo tal que $\liminf_{n\to\infty} \frac{T(n)}{n} = \infty$, sea una máquina de Turing con k cintas de trabajo $(k > 1)$ M con alfabeto Σ que calcula una función f definida sobre un subconjunto adecuado de Σ^* en tiempo $T(n)$. Entonces f es computable por una máquina de Turing $\tilde{M}$ con $k+1$ cintas de trabajo en tiempo $cT(n)$ para cualquier constante $c > 0$.

Demostración. La construcción de la máquina $\tilde{M}$ que simula a M es la siguiente:

- En primer lugar se divide el contenido de la cinta de entrada en grupos de m celdas, donde $m \in \mathbb{N}$ es un número grande que al final determinaremos (dependerá de c). $\tilde{M}$ copia la entrada en una cinta de trabajo, codificando m símbolos en uno (el alfabeto sobre el que $\tilde{M}$ trabaja contendrá a Σ^m). A partir de este punto, $\tilde{M}$ utilizará dicha cinta cada vez que quiera leer la entrada, ignorando la cinta de entrada (de ahí que necesitemos una cinta más, si la cinta de entrada fuera de lectura-escritura podríamos utilizarla como otra cinta de trabajo y no precisaríamos de esta cinta adicional). Los contenidos de las cintas de almacenamiento de M en cada paso también aparecerán codificados en "paquetes" de m símbolos cada uno, que serán un solo símbolo de $\tilde{M}$.
- Durante la computación, $\tilde{M}$ simulará un mayor número de pasos de cálculo de M en un *paso básico*, que constará de 8 pasos de cálculo de $\tilde{M}$. Cada uno de los cabezales de $\tilde{M}$ se sitúa en cada paso en una celda, que en realidad es un grupo, al que llamaremos *grupo hogar*, que codifica m *celdas hogar* de M. $\tilde{M}$ guarda en su unidad de control, mediante los estados, en cuál de las celdas hogar de M está situado el cabezal de cada cinta de dicha máquina. La figura 2.4 ilustra esta codificación de las cintas.

- En un paso básico se realizan las siguientes operaciones:

  1. $\tilde{M}$ mueve el cabezal una vez a la izquierda, dos veces a la derecha y de nuevo una vez a la izquierda, regresando al grupo hogar, y en cada uno de esos 4 pasos guarda (en la unidad de control) los contenidos de los dos grupos adyacentes, o *grupos vecinos*, junto con los del propio grupo hogar.
  2. $\tilde{M}$ acude a la función de transición de M (debidamente codificada en su propia función de transición) para simular todos los pasos de cálculo de M hasta que en alguna de las cintas el cabezal de M abandone la región formada por las celdas hogar y las celdas vecinas (que forman los grupos vecinos) a izquierda y derecha. Estos pasos de M no suponen ningún paso de cálculo para $\tilde{M}$, pues los simula mediante la función de transición. Si M se detuviera en alguno de estos pasos, $\tilde{M}$ también lo haría en este paso básico (después de realizar los pasos de cálculo que quedan).
  3. $\tilde{M}$ cambia en el grupo hogar y los grupos vecinos los símbolos que sea necesario de acuerdo con la simulación de la segunda fase, visitándolos en el orden correcto (ya sea el descrito antes o el simétrico) para acabar en el grupo correcto que alberga la celda en la que se encuentra ahora el cabezal de M en la cinta en cuestión (que será adyacente a uno de los grupos vecinos). Son necesarios otros 4 pasos de cálculo.

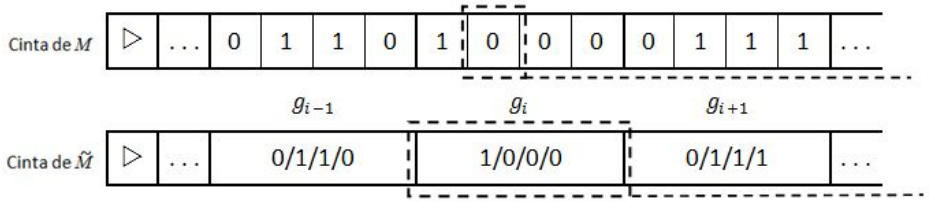

Figura 2.4: Representación de la correspondencia entre un grupo de celdas hogar y celdas vecinas con los grupos hogar (g_i) y vecinos (g_{i-1} y g_{i+1}).

Se necesitan al menos m pasos de cálculo de M para mover el cabezal de una cinta fuera de la región comprendida por las cintas hogar y las cintas vecinas de ambos lados. Así, en 8 pasos de cálculo $\tilde{M}$ ha simulado al menos m pasos de cálculo de M.

Si M realiza $T(n)$ pasos de cálculo, $\tilde{M}$ los simula en a lo sumo $8\left\lceil \frac{T(n)}{n} \right\rceil$ pasos de cálculo. Además $\tilde{M}$ debe copiar y codificar su entrada (m celdas en una) y devolver al extremo izquierdo el cabezal de la cinta de entrada simulada, lo cual precisa de $n + \lceil \frac{n}{m} \rceil$ pasos de cálculo.

Entonces en total $\tilde{M}$ realiza a lo sumo

$$n + \left\lceil \frac{n}{m} \right\rceil + 8 \left\lceil \frac{T(n)}{n} \right\rceil < n + \frac{n}{m} + 8\frac{T(n)}{n} + 2$$

pasos de cálculo.

Por hipótesis $\liminf_{n\to\infty} \frac{T(n)}{n} = \infty$, luego fijado $D > 0$ existe $n_D \in \mathbb{N}$ tal que para todo $n \geq n_D$, $n \in \mathbb{N}$, $\frac{T(n)}{n} \geq D$, es decir, $n \leq \frac{T(n)}{D}$.

Entonces si $n \geq 2$ (lo cual implica que $n+2 \leq 2n$) y $n \geq n_D$ se tiene que

$$n + \frac{n}{m} + 8\frac{T(n)}{n} + 2 \leq T(n)\left(\frac{8}{m} + \frac{2}{D} + \frac{1}{mD}\right).$$

Fijando $m = \frac{16}{c}$, $D = \frac{m}{4} + \frac{1}{8}$ en la expresión anterior se observa que, para todo $n \geq$ máx$(2, n_D)$,

$$T_{\tilde{M}}(n) < cT(n).$$

□

Nótese que es válido para cualquier valor de $c > 0$ porque este procede de m, que podemos variar libremente. De la demostración se deduce que, aunque consigamos "acelerar" la máquina, la función de transición se complica enormemente.

2.3 CLASES DE COMPLEJIDAD DETERMINISTAS

Una vez hemos introducido el concepto básico de la complejidad de tiempo, la función tiempo de una máquina, podemos abordar el tema central de esta teoría: la clasificación de problemas en función de los recursos (el tiempo, en este caso) que necesitan para ser resueltos. En estos términos se definen las clases de complejidad.

Definición 2.14 Una *clase de complejidad* **C** es un conjunto de problemas que pueden ser resueltos (por una máquina de Turing, sobre la que en ocasiones se imponen diversas condiciones) utilizando recursos limitados por una cota dada.

De la definición anterior se deduce que es la cota de recursos, junto con las condiciones impuestas a la máquina, las que determina la clase de complejidad. En este texto el recurso acotado será el tiempo, pero esta definición es igualmente válida si limitamos el espacio, dando lugar a la Teoría de la Complejidad de Espacio.

Aunque hemos dado la definición en el sentido más amplio posible, puesto que generalmente los problemas a los que nos enfrentamos son decisionales (es decir, determinar la pertenencia de una entrada a un lenguaje dado), o consisten en computar funciones, la mayoría de las veces consideraremos las clases de complejidad como conjuntos de lenguajes, o como conjuntos de funciones.

Definición 2.15 Una *clase de complejidad de lenguajes* es un conjunto de lenguajes cuyos correspondientes problemas decisionales están en una clase de complejidad **C**. Denotaremos a esta clase de complejidad de lenguajes también por **C**.

De forma análoga se definen las clases de complejidad de funciones (o aplicaciones, en general), que en ocasiones se denotan por **CF** (otras veces omitiremos la **F**, confundiendo así las notaciones para todas las clases, pues las de lenguajes y funciones son solo casos particulares del primero).

En lo sucesivo definiremos todas las clases de complejidad como clases de lenguajes, pero la generalización a clases de problemas es inmediata, sin más que fijarse en la cota del recurso correspondiente y en las características de la máquina. Por convenio consideraremos lenguajes sobre el alfabeto binario $\{0,1\}$ puesto que la independencia del alfabeto que nos otorga la codificación no es plena en este caso (ya que la longitud de las palabras no se conserva en general por codificaciones, y la utilizaremos constantemente). No obstante, podremos entender las definiciones en un sentido más amplio para cualquier alfabeto Σ, sin más que considerar las máquinas sobre ese mismo alfabeto.

2.3.1 Las clases DTIME

Dentro del estudio de la complejidad de tiempo determinista (es decir, problemas resolubles por máquinas de Turing deterministas) las clases básicas, y que servirán para definir el resto, son las clases **DTIME** (del inglés *deterministic time*, tiempo determinista).

Definición 2.16 Sea $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ una función monótona creciente. Se define la clase **DTIME**(f) como:

$$\mathbf{DTIME}(f) := \left\{ L \subseteq \{0,1\}^* : \exists M \text{ determinista sobre } \{0,1\} \text{ con } Res_M = \chi_L \text{ y } T_M \in O(f) \right\}.$$

Aunque, como ya hemos dicho, consideraremos solo funciones constructibles en tiempo para las cotas, la definición anterior se puede dar en términos de funciones monótonas crecientes cualesquiera con llegada en $\mathbb{R}^+$, sin más que tener en cuenta que $\mathbb{N} \subset \mathbb{R}^+$.

Obsérvese que la definición se da en términos de $O(f)$, con lo que los factores constantes son irrelevantes. Esto es una consecuencia del teorema 2.13 de aceleración lineal.

Resulta evidente que si $f(n) \leq g(n)\ \forall n \in \mathbb{N}$ entonces $\mathbf{DTIME}(f) \subseteq \mathbf{DTIME}(g)$. El siguiente resultado precisa más las contenciones entre estas clases.

Teorema 2.17 (de Jerarquía de Tiempo) Sean $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ dos funciones constructibles en tiempo tales que $g = \omega(f \log(f))$. Entonces

$$\mathbf{DTIME}(f) \subsetneq \mathbf{DTIME}(g).$$

Demostración. Consideremos la máquina determinista D sobre el alfabeto binario que funciona de la forma siguiente: Dada una entrada $x \in \{0,1\}^*$, ejecuta durante $g(|x|)$ pasos la máquina universal $\mathscr{U}$ del teorema 2.12 para simular la ejecución de M_x sobre x. Si $\mathscr{U}$ da como resultado un bit $b \in \{0,1\}$ (suponemos que el resultado, en caso de obtenerse, es siempre un bit) entonces $\mathrm{Res}_D(x) = 1 - b$. En otro caso $\mathrm{Res}_D(x) = 0$.

Por definición D termina su computación en $g(n)$ pasos de cálculo, y por lo tanto el lenguaje $L(D) \in \mathbf{DTIME}(g)$. Queremos ver que $L(D) \notin \mathbf{DTIME}(f)$. Razonaremos por reducción al absurdo, supongamos que existe una máquina de Turing determinista M y una constante c tal que, dada una entrada $x \in \{0,1\}^*$, termina su computación en $cf(|x|)$ pasos de cálculo y devuelve $\mathrm{Res}_D(x)$.

El número de pasos necesarios para simular la máquina M en la máquina universal $\mathscr{U}$ del teorema 2.12 es a lo sumo $c'cf(|x|)\log(f(|x|))$, donde c' es una constante independiente de $|x|$. Por hipótesis existe $n_0 \in \mathbb{N}$ tal que para todo $n \geq n_0$, $n_0 \in \mathbb{N}$, $g(n) > c'cf(|x|)\log(f(|x|))$. Sea $x \in \{0,1\}^*$ una cadena que represente a la máquina M de longitud mayor o igual que n_0 (dicha cadena existe, pues M está representada por una infinidad de cadenas, ver subsección 1.4.1). Entonces ejecutando D sobre x obtendrá $\mathrm{Res}_M(x) = b$ en menos de $g(|x|)$ pasos de cálculo, pero por definición de la máquina D tendremos $\mathrm{Res}_D(x) = 1 - b \neq \mathrm{Res}_M(x)$, absurdo, ya que la máquina M no devuelve $\mathrm{Res}_D(x)$ para todas las entradas. Entonces $L(D) \in \mathbf{DTIME}(g) \setminus \mathbf{DTIME}(f)$. □

Este teorema fue probado inicialmente por Hartmanis y Stearns en [25, teorema 9], aunque utilizaba la versión débil de la simulación por la máquina universal (ver teorema 2.11). Con la prueba de Hennie y Stearns del teorema 2.12 en [26], el teorema queda finalmente en la forma anterior (pues la clave de la demostración es justamente la simulación por la máquina universal).

El siguiente resultado, llamado teorema de la brecha, fue probado independientemente por B. Trakhtenbrot y A. Borodin (ver [5]), quien aprovechó la notación de M. Blum en su artículo [4], en que se aparta del modelo computacional, refiriéndose a los recursos como *medidas de complejidad*, con lo que buscaba la independencia del tipo de recurso (pues estos son inherentes a cada modelo computacional). La versión que daremos es una "traducción" de aquel resultado para el caso del tiempo.

Teorema 2.18 (de la brecha, para el tiempo) Dada una función computable $g : \mathbb{N} \longrightarrow \mathbb{N}$ tal que $g(n) \geq n$ para todo $n \in \mathbb{N}$, existe una cota de tiempo $T : \mathbb{N} \longrightarrow \mathbb{N}$ tal que $\mathbf{DTIME}(g(T(n))) = \mathbf{DTIME}(T(n))$.

Este teorema no contradice el teorema de Jerarquía de Tiempo, sino que pone de manifiesto la importancia de sus hipótesis, ya que la cota de tiempo T puede no ser una función constructible en tiempo. Lo que dice este teorema es que podemos encontrar "brechas" arbitrariamente grandes en la jerarquía temporal, es decir, que no hay una forma uniforme de aumentar la cota sobre un recurso que garantice un aumento de la potencia computacional.

2.3.2 La clase P y otras clases centrales

Una vez hemos visto las clases básicas de complejidad en tiempo determinista, podemos definir otras que nos permitan clasificar los problemas (o lenguajes) de acuerdo con los diversos grados de eficiencia que nos proporciona la intuición.

Definición 2.19 Se define la siguiente clase:

$$\mathbf{P} := \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(n^k) .$$

La importancia de esta clase en el ámbito de la Teoría de la Complejidad Computacional es mayúscula, pues, intuitivamente, es la clase de los problemas que se pueden resolver de manera eficiente. Precisaremos esto en la subsección siguiente.

Además de esta, otras clases centrales determinadas por el tiempo para máquinas deterministas son las siguientes:

Definición 2.20 Se definen las siguientes clases:

- $\mathbf{E} := \mathbf{DTIME}(2^{O(n)})$ (del inglés *exponential time*).
- $\mathbf{EXP} = \mathbf{EXPTIME} := \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(2^{n^k})$ (del inglés *expo-polynomial time*).
- $2\text{-}\mathbf{EXPTIME} := \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(2^{2^{n^k}})$ (y análogamente k-**EXPTIME**, $k \in \mathbb{N}$).

A la vista de las definiciones anteriores, y en virtud del teorema de Jerarquía de Tiempo, se tienen las siguientes contenciones entre clases:

$$\mathbf{P} \subsetneq \mathbf{E} \subsetneq \mathbf{EXP} \subsetneq 2\text{-}\mathbf{EXPTIME} .$$

EJEMPLO 2.21

Retomando el ejemplo 1.3 de la multiplicación, supongamos que cada operación básica necesita un número constante de pasos de cálculo (recordemos que los factores constantes no son relevantes) en una máquina de Turing con alfabeto $\{0,1,...,9\}$, suposición nada descabellada, por otra parte. Para dos números de n cifras (entrada de longitud $2n$), el número máximo de operaciones básicas necesarias para calcular su producto siguiendo el algoritmo de la suma repetida es (ligeramente inferior a) $(10^n-1)(2n)$. Puesto que $\lim_{n\to\infty} \frac{(10^n-1)2n}{2^{(2n)^2}} = 0$ se deduce que el problema de multiplicar dos enteros está en **EXP**.
Por otra parte, vimos que el algoritmo clásico resolvía el problema en a lo sumo $2n^2$ operaciones básicas, lo que implica que, de hecho, está en **P** (es más, está en $\mathbf{DTIME}(n^2)$, y si utilizamos la transformada rápida de Fourier está en $\mathbf{DTIME}(n\log(n)\log(\log(n)))$.

En el ejemplo anterior hemos sorteado una dificultad básica que surge al considerar problemas descritos en términos de un alfabeto cualquiera que luego querremos resolver en máquinas con un alfabeto fijado, usualmente el binario. Aquí, puesto que se trata de un ejemplo teórico, hemos utilizado dicho alfabeto como alfabeto de la máquina, pero en

otras condiciones tendremos que tener en cuenta el tamaño de la representación en el alfabeto correspondiente tanto de la entrada y el resultado como de los contenidos de todas las cintas en cada paso intermedio. Afortunadamente en este caso, y en muchos otros, esto no nos crea problemas.

EJEMPLO 2.22

El problema del viajante (ejemplo 1.4), en su versión más sencilla (suponiendo que puede desplazarse en línea recta entre dos ciudades cualesquiera), y requiriendo que la ciudad de partida sea la misma que la de destino, puede modelarse de forma que se reduzca a calcular un circuito que recorra todos los vértices de un grafo ponderado no orientado completo (las distancias entre ciudades serán los valores de las aristas). La entrada de una máquina de Turing que resuelva el problema deberá contener los k vértices junto con las $\binom{k}{2}$ distancias, y podemos considerar un alfabeto suficientemente grande de forma que la entrada tenga longitud $k+\binom{k}{2}$. También podemos suponer que, al aplicar el algoritmo de búsqueda exhaustiva, procesar cada ruta requiere un solo paso de cálculo (es decir, la elección de una ruta distinta de las ya procesadas y el cálculo de la distancia puede hacerse en un solo paso). Entonces necesitaremos $(k-1)!$ pasos de cálculo, en el peor de los casos. Por lo tanto este problema está en $\mathbf{DTIME}((n-1)!)$, pero no podemos asegurar que esté en ninguna de las clases que hemos visto en la definición 2.20 considerando este algoritmo.

El siguiente ejemplo es una muestra más de que los algoritmos más eficientes no son los más obvios, y pueden esconderse tras ideas simples pero brillantes.

EJEMPLO 2.23 *(Conectividad en grafos)*

Dado un grafo G y dos vértices s, t de G, el problema de la conectividad consiste en averiguar si s y t están conectados en G, es decir, si existe un camino en G que los une. Este problema está en $\mathbf{P}$, y el algoritmo que lo demuestra es el de *búsqueda en profundidad* (*depth-first search, DFS* en inglés).
Este algoritmo explora todas las aristas que parten de s, comprobando si llegan a t y marcándolas una vez visitadas. En caso afirmativo acepta. En otro caso continúa explorando cada una de las aristas adyacentes a las marcadas y que aún no hayan sido visitadas, y así sucesivamente. Tras a lo sumo $\binom{n}{2}$ pasos (donde n es el número de vértices) todas las posibles aristas conectadas con s han sido visitadas. Ahora bien, $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$, y por lo tanto el problema está en $\mathbf{P}$ (de hecho, en $\mathbf{DTIME}(n^2)$).

2.3.3 Comentarios sobre la clase P

Desde el comienzo del texto venimos subrayando la importancia de la eficiencia de los algoritmos. Resultaría ideal para que cada problema que nos planteamos existiera un

algoritmo eficiente que lo resolviera. Pero ¿qué consideramos que es eficiente? A pesar de la dificultad que entraña formalizar un concepto intuitivo como este, la propuesta de A. Cobham en [9] y J. Edmonds en [18, sección 2](en realidad Edmonds incluye esto en las digresiones previas a la presentación de un algoritmo eficiente para resolver el problema de los emparejamientos máximos en un grafo), más conocida como *tesis de Cobham* o de *Cobham-Edmonds*, es aceptada como patrón de tratabilidad (decimos que un problema es tratable si es resoluble por métodos algorítmicos de eficiencia razonable). Esta tesis puede resumirse en términos de las máquinas de Turing como sigue:

Tesis de Cobham-Edmonds: Un problema debe ser considerado tratable si existe una máquina de Turing determinista que lo resuelve en tiempo polinómico sobre la longitud de la entrada.

Esta tesis hace que la clase **P** tenga una especial relevancia, al convertirla en la clase de los problemas tratables.

No obstante, también es objeto de muchas críticas que señalan las deficiencias de una afirmación de tal magnitud, que pretende acercar la intuición a las matemáticas. Una de las más importantes estriba precisamente en la amplitud de la clase P, pues ¿hasta qué punto debe considerarse tratable un problema resoluble por una máquina con tiempo $O(n^{100})$? El tiempo de cálculo incluso para entradas de tamaño reducido se dispara enormemente, haciendo estos cálculos impracticables. Por otra parte, un problema que requiera $O(2^{0,00001})$ es considerado intratable, pero para entradas incluso de longitud 10^6 puede resolverse rápidamente.

Otro punto débil de esta tesis (y de toda la teoría que estamos desarrollando) es que no tiene en cuenta los factores constantes, que desde el punto de vista teórico no tienen relevancia (como consecuencia del teorema de Aceleración Lineal 2.13), pero sí desde el punto de vista práctico, pues normalmente el alfabeto de la máquina estará sujeto a limitaciones y querremos que esta sea lo más sencilla posible (coloquialmente, no se nos permite complicar la máquina para simplificar el algoritmo).

Afortunadamente, en la práctica la mayoría de los problemas en P admiten algoritmos (máquinas deterministas) que los resuelven en tiempo polinómico con un exponente y unas constantes razonables.

Otro problema que plantea la tesis de Cobham-Edmonds es su generalidad, pues dependiendo de la situación a la que nos enfrentemos (que puede abarcar desde las características del problema hasta las restricciones impuestas a la máquina) puede que incluso algoritmos de tiempo $O(n^3)$ sean considerados ineficientes (por ejemplo, porque las entradas tengan longitud muy grande, como en sistemas físicos en los que intervengan muchas variables).

También el hecho de considerar la complejidad del caso peor representa un problema en lo que se refiere a tratabilidad, pues un determinado algoritmo puede resultar eficiente para todas las entradas salvo algunas con escaso interés, y aún así sería considerado ineficiente. La complejidad del caso medio ofrece una clase alternativa a P como clase de los problemas tratables, la clase dist**P**.

Otra de las críticas a esta tesis se dirige contra la exigencia de determinismo, renunciando a las ventajas que pueden ofrecer a la eficiencia el aprovechamiento de la aleatoriedad o el modelo cuántico. En lo que respecta a la aleatoriedad, la consistencia de su uso en máquinas de Turing está comprobada y es aplicada en problemas reales (si no la aleatoriedad, al menos sí la pseudoaleatoriedad). Sin embargo, aunque está demostrado teóricamente que el modelo cuántico aventaja en potencia computacional a la máquina de Turing (al menos en ciertos problemas, como la factorización de enteros, que el *algoritmo de Shor* resuelve eficientemente en el caso cuántico y para el que aún no se conoce un algoritmo eficiente en ninguno de los modelos tradicionales), no está claro que se trate de un modelo físicamente realizable, y por lo tanto no se encuentra entre los contemplados por la tesis de Church.

A favor de la tesis de Cobham-Edmonds se cuenta, entre muchos otros argumentos, el hecho de que la mayoría de las modificaciones de las máquinas de Turing que respetan el determinismo (entre las que vimos en el capítulo 1 tendríamos que excluir las máquinas indeterministas y con oráculo) no cambian la condición de tratabilidad de un problema, es decir, dejan invariante la clase **P**, como nos aseguran los resultados de la sección 2.2. De ahí el título de dicha sección, que hace referencia a que la definición de máquina de Turing determinista es "robusta" no solo desde el punto de vista de la potencia computacional (es decir, todas las modificaciones proporcionan máquinas equivalentes, que computan las mismas funciones), sino desde el de la complejidad, pues las simulaciones producen un sobrecoste de tiempo a lo sumo polinómico, que no afecta a la pertenencia a la clase **P** de un determinado problema.

CAPÍTULO 3

COMPLEJIDAD DE TIEMPO INDETERMINISTA

En el capítulo anterior hemos hecho un estudio de la complejidad de tiempo para máquinas deterministas, que representan los algoritmos tradicionales. En la subsección 1.3.3 introdujimos la noción de máquina indeterminista, y adelantamos que supone un aumento de la potencia computacional, aunque no representan algoritmos en el sentido estricto (pues los pasos a realizar no están unívocamente determinados por la entrada).

Al enfrentarnos a un problema, suele ser mucho más difícil resolverlo que verificar que una solución dada es correcta, al igual que es más sencillo verificar una demostración de un teorema que obtenerla por nosotros mismos. La diferencia entre estas acciones es que la verificación de una solución o una demostración es un proceso mecánico, en el que debemos comprobar que cada paso del razonamiento es correcto de acuerdo con un sistema axiomático determinado (que puede ser más o menos complejo). En cambio, en la obtención de una demostración o en la resolución de un problema interviene la creatividad.

El indeterminismo puede entenderse como la formalización de la creatividad, haciendo extensiva a las máquinas de Turing esta habilidad de las personas al resolver problemas. En la sección 3.1 precisaremos este concepto.

Se puede hacer un estudio de la complejidad análogo al del caso determinista para obtener nuevas clases. De forma natural surge la cuestión de la relación entre las clases deterministas e indeterministas. Evidentemente las indeterministas contienen a las correspondientes deterministas, pero en la mayoría de los casos la contención opuesta continúa siendo un problema abierto, que equivale a preguntarnos si el aumento aparente de la potencia computacional es real, o en línea con la noción intuitiva planteada antes, si la creatividad puede sustituirse por métodos mecánicos "eficientes". Si nos planteamos esta cuestión restringiéndonos a las clases de los lenguajes tratables tanto en el caso determinista (la clase **P**) como en el indeterminista (la clase **NP**, que se definirá de forma análoga a la anterior, como veremos en la subsección 3.2.2) obtenemos uno de los problemas abiertos más importantes no sólo de la Teoría de la Complejidad, sino de todas las Matemáticas: la conjetura de Cook. Dedicaremos el capítulo 5 a comentarios sobre la misma.

3.1 NOCIÓN EQUIVALENTE DE MÁQUINA DE TURING INDETERMINISTA

A simple vista, la noción intuitiva de indeterminismo que acabamos de presentar, relacionada con la creatividad, poco parece tener que ver con la definición 1.14 de una máquina de Turing indeterminista. A continuación precisaremos esa idea intuitiva y probaremos la equivalencia de ambas nociones.

Definición 3.1 Una *máquina de Turing adivinatoria* es una máquina de Turing determinista M que, inmediatamente después de recibir la entrada $x \in \Sigma^*$ (donde Σ es el alfabeto de la máquina) "adivina" de forma indeterminista una palabra $y \in \Sigma^*$ que escribe a continuación de una copia de x en la primera cinta de trabajo, y después inicia la computación normalmente (de forma determinista) utilizando dicha primera cinta de trabajo como cinta de entrada, con lo que equivale a ejecutar la misma máquina determinista sin el proceso de adivinación sobre la entrada $xy \in \Sigma^*$.

Como ya adelantamos en el encabezado de la definición anterior, las máquinas adivinatorias no son más que una versión de máquinas indeterministas, por lo que el siguiente teorema garantiza la equivalencia de ambas definiciones.

Teorema 3.2 Las definiciones de máquina indeterminista tradicional (definición 1.14) y adivinatoria (definición 3.1) son equivalentes (en el sentido de que una puede simular a la otra de forma razonablemente eficiente).

Demostración. Supongamos que la máquina determinista tradicional M_T recibe una entrada $x \in \Sigma^*$ y produce una salida $r \in \text{Res}_{M_T}(x)$ en tiempo $O(f(|x|))$. Queremos simular su funcionamiento por una máquina adivinatoria M_A. La máquina tradicional realiza una secuencia de a lo sumo $O(f(|x|))$ pasos de cálculo elegidos de forma indeterminista. Recuperando la idea de la demostración del teorema 1.15, numeramos todos los posibles valores de la función de transición de M_T, y los incluimos en la función de transición de la parte determinista de M_A, con un argumento adicional para la numeración (por lo tanto para cada configuración y cada número o símbolo la función de transición toma un único valor, y es una aplicación). La máquina M_A adivinará una palabra y de longitud a lo sumo $O(f(|x|))$ que represente la secuencia de los pasos de cálculo de M_T y después la parte determinista reproduce los pasos de la máquina M_T en a lo sumo $O(f(|x|))$, con lo que el resultado es r, y la máquina M_A funciona en tiempo $O(f(|x|))$.

Supongamos ahora que es la máquina M_A la que recibe la entrada $x \in \Sigma^*$, "adivina" $y \in \Sigma^*$ de longitud a lo sumo $O(g(|x|))$ y obtiene como resultado $r \in \text{Res}_{M_A}(x)$ en tiempo $O(f(|x|))$. Entonces mediante su función de transición, M_T obtiene en una de sus cintas de trabajo la palabra y (por ejemplo dando en cada paso de cálculo la posibilidad de escribir un símbolo de Σ, y escribiéndola en los sucesivos pasos de cálculo, lo cual requiere $O(g(|x|))$ pasos.). Una vez obtenida y, basta con escribir en la primera cinta de trabajo xy

y aplicar la función de transición de la parte determinista de M_A, que se puede integrar en la función de transición indeterminista de M_T, obteniendo el resultado r en a lo sumo otros $O(f(|x|))$ pasos, con lo que el tiempo de cálculo de la máquina M_T en la entrada x es $t_{M_T}(x) = O(g(|x|)) + O(f(|x|))$. □

En la demostración hemos utilizado el concepto de tiempo de una máquina indeterminista. En el capítulo 2 definimos el tiempo de cálculo de una máquina sobre una entrada y la función de tiempo de una máquina (definiciones 2.1 y 2.2), sin especificar si era determinista o no, a pesar de que en dicho capítulo considerábamos máquinas deterministas.

Puede entenderse que la "palabra adivinada" marca, para una entrada dada, la rama del sistema de transición de la máquina por la que debe discurrir la computación; la demostración anterior justifica esta intuición.

Es el paso de "adivinar" cierta palabra que luego permita realizar la computación de manera determinista el que se asocia con la idea de creatividad, y con esta definición de indeterminismo queda justificada la analogía que hacíamos en la introducción del capítulo.

En lo que sigue cuando nos refiramos a máquinas indeterministas estaremos considerando las máquinas tradicionales de la definición 1.14, cuando queramos tratar las de la definición 3.1 nos referiremos a ellas como máquinas adivinatorias.

3.2 CLASES DE COMPLEJIDAD INDETERMINISTAS

De forma análoga a como lo hicimos en la sección 2.3 podemos definir una serie de clases de complejidad para máquinas indeterministas con tiempo acotado (generalmente por funciones constructibles en tiempo). Por supuesto, la definición de clase de complejidad es la misma (ver definiciones 2.14 y 2.15).

Aunque de nuevo definiremos las clases de complejidad como clases de lenguajes sobre el alfabeto binario $\{0,1\}$, la generalización a clases de problemas es la obvia en cada caso.

Como en el caso determinista, las clases de funciones (cuando nos interese diferenciarlas) las representaremos como las de lenguajes seguidas de **F** (por ejemplo **NTIMEF**(f)).

3.2.1 Las clases NTIME

Análogamente a las clases **DTIME** (definición 2.16) se pueden definir para el caso de máquinas indeterministas unas clases básicas que servirán para construir el resto de clases.

Definición 3.3 Sea $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ una función monótona creciente. Se define la clase **NTIME**(f) como:

$$\textbf{NTIME}(f) := \left\{ L \subseteq \{0,1\}^* : \exists M \text{ indeterminista sobre } \{0,1\} \text{ con } Res_M = \chi_L \text{ y } T_M \in O(f) \right\}.$$

Ahora bien, las máquinas adivinatorias nos proporcionaban otra noción de indeterminismo, con lo que cabe esperar que podamos definir las clases en esos términos.

Definición 3.4 Sea $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ una función monótona creciente. Se define la clase **NTIME**(f) como el conjunto de los lenguajes $L \subseteq \{0,1\}^*$ tales que existe una máquina determinista M con $T_M = O(f)$ de forma que para cada $x \in \{0,1\}$ se tiene que

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{O(f(|x|))} \text{ tal que } M(x,u) = M(xu) = 1$$

Llamaremos *verificador* para L a la máquina M y si $x \in L$ y $u \in \{0,1\}^{O(f(|x|))}$ cumple que $M(x,u) = 1$ entonces llamaremos a *u certificado* para x (respecto del lenguaje L y la máquina M).

El certificado para una palabra $\in \{0,1\}$ se puede entender como la parte que "adivina" una máquina adivinatoria, y por lo tanto la definición anterior puede enunciarse en los términos siguientes:

Definición 3.5 Sea $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ una función monótona creciente. Se define la clase **NTIME**(f) por

$$\textbf{NTIME}(f) := \left\{ L \subseteq \{0,1\}^* : \exists M \text{ máquina adivinatoria sobre } \{0,1\} \text{ con } Res_M = \chi_L \text{ y } T_M \in O(f) \right\},$$

cumpliendo además que la palabra adivinada para cada $x \in L$ tiene longitud $O(f(|x|))$.

La equivalencia entre estas dos últimas definiciones es evidente, y ya fue explicada en el párrafo anterior. Es más común utilizar la definición 3.4 certificado-verificador, que hace que, si entendemos las clases como clases de problemas, **NTIME**(f) se corresponda de manera intuitiva con los problemas que tienen una solución (certificado) "suficientemente corta" (de longitud "del orden de f") que es verificable en un tiempo "suficientemente reducido" (también "del orden de f").

Falta comprobar que estas dos últimas definiciones son a su vez equivalentes a la primera dada, pero esto es una mera consecuencia del teorema 3.2 y de su demostración (téngase en cuenta que $O(f)+O(f) = O(f)$, propiedad que se deduce fácilmente de la definición 0.7).

Si $f(n) \leq g(n)\ \forall n \in \mathbb{N}$ evidentemente **NTIME**$(f) \subseteq$ **NTIME**(g), y puesto que toda máquina determinista puede considerarse como indeterminista, **DTIME**$(f) \subseteq$ **NTIME**(f).

Como en el caso determinista, también existe un teorema de Jerarquía de Tiempo para el caso indeterminista, publicado por Stephen Cook en [11], y que toma la siguiente forma (ligeramente distinta a la planteada por Cook):

Teorema 3.6 (de Jerarquía de Tiempo Indeterminista) Sean $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ dos funciones constructibles en tiempo tales que $g = \omega(f(n+1))$. Entonces

$$\textbf{NTIME}(f) \subsetneq \textbf{NTIME}(g)\ .$$

Demostración. Sea $\{M_i\}_{i\in\mathbb{N}}$ una numeración de las máquinas de Turing indeterministas (recordemos que toda máquina de Turing puede representarse por infinitos elementos de $\{0,1\}^*$, que es un conjunto numerable; el hecho de que infinitos naturales denoten a una misma máquina no supone un problema).

Definimos una máquina de Turing indeterminista D que procede de la forma siguiente sobre una entrada $w = 1^i01^m0y$ (y es la "palabra adivinada" por la máquina para la entrada 1^i01^m0, al tratarse de una máquina indeterminista tenemos que explicar su funcionamiento para cada una de las ramas del sistema de transición, representadas por dichas palabras):

1. Si $|y| < f(i+m+2)$ acepta si y solo si M_i acepta las dos entradas 1^i01^m0y0 y 1^i01^m0y1 (las palabras adivinadas son respectivamente $y0$ y $y1$) en $g(|w|)$ pasos de cálculo.
2. Si $|y| = f(i+m+2)$ acepta si y solo si M_i rechaza la entrada 1^i01^m0y o no termina su computación en $g(|w|)$ pasos de cálculo.

Evidentemente esta máquina funciona en tiempo $T_D(n) = O(g(n))$.

Razonaremos por reducción al absurdo: supongamos que $\mathbf{NTIME}(f) \nsubseteq \mathbf{NTIME}(g)$, entonces existe una máquina indeterminista, digamos M_i, que decide el lenguaje $L(D)$ en tiempo $O(f(n))$. Como $f(n+1) = o(g(n))$, para m suficientemente grande se tiene la siguiente cadena de equivalencias:

$$1^i01^m0 \in L(D) \overset{(a)}{\Leftrightarrow} 1^i01^m0y \in L(D)\ \forall y \text{ tal que } |y| = 1 \overset{(a)}{\Leftrightarrow} 1^i01^m0y \in L(D)\ \forall y \text{ tal que } |y| = 2 \overset{(a)}{\Leftrightarrow}$$
$$\overset{(a)}{\Leftrightarrow} \ldots \overset{(a)}{\Leftrightarrow} 1^i01^m0y \in L(D)\ \forall y \text{ tal que } |y| = f(i+m+2)-1 \overset{(a)}{\Leftrightarrow}$$
$$\overset{(a)}{\Leftrightarrow} M_i \text{ acepta las dos palabras } 1^i01^m0y0 \text{ y } 1^i01^m0y1\ \forall y \text{ tal que } |y| = f(i+m+2)-1 \overset{(b)}{\Leftrightarrow}$$
$$\overset{(b)}{\Leftrightarrow} M_i \text{ acepta } 1^i01^m0y\ \forall y \text{ tal que } |y| = f(i+m+2) \overset{(c)}{\Leftrightarrow}$$
$$\overset{(c)}{\Leftrightarrow} 1^i01^m0y \in L(D)\ \forall y \text{ tal que } |y| = f(i+m+2)\,.$$

Pero también se tiene que:

$$M_i \text{ acepta } 1^i01^m0y\ \forall y \text{ tal que } |y| = f(i+m+2) \overset{(d)}{\Leftrightarrow}$$
$$\overset{(d)}{\Leftrightarrow} D \text{ rechaza} 1^i01^m0y\ \forall y \text{ tal que } |y| = f(i+m+2) \overset{(e)}{\Leftrightarrow}$$
$$\overset{(e)}{\Leftrightarrow} 1^i01^m0y \notin L(D)\ \forall y \text{ tal que } |y| = f(i+m+2)\,,$$

y llegamos a una contradicción.

Las equivalencias marcadas con (a) se deducen de aplicar el caso 1 de la descripción del funcionamiento de la máquina D, pues D acepta cada palabra si M_i acepta esa misma

palabra ampliando la parte adivinada con cualquiera de los dos símbolos del alfabeto, y como partimos de una palabra para la que no hay parte adivinada, se deduce que M_i debe aceptar dicha palabra para todas las partes adivinadas posibles de longitud hasta $f(i+m+2)$.

La equivalencia marcada con (b) es una trivialidad. La (c) se deduce del hecho de que M_i decide el lenguaje $L(D)$, y por lo tanto $L(D) = L(M_i)$.

La equivalencia (d) se obtiene aplicando el caso 2 de la descripción de D, y (e) es obvia. □

El razonamiento de esta demostración es diferente del empleado en el teorema análogo para el caso determinista, ya que, aunque existe una máquina universal indeterminista con unas propiedades de simulación similares a las de la determinista, el "cambio de respuesta" dada la descripción de una máquina indeterminista no es trivial.

3.2.2 La clase NP (y otras clases centrales, pero menos relevantes)

De forma totalmente paralela a como lo hicimos en la subsección 2.3.2 definiremos algunas clases de complejidad de tiempo indeterminista a partir de las clases **NTIME**.

Definición 3.7 Se define la siguiente clase:

$$\mathbf{NP} := \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k) .$$

Si consideramos la noción de indeterminismo basada en las máquinas adivinatorias, la clase **NP** agrupa los problemas cuya solución es suficientemente corta y verificable de forma eficiente, frente a la clase **P**, formada por los problemas que pueden ser resueltos eficientemente. Explícitamente, la definición en términos de certificados y verificadores que se obtiene inmediatamente de la definición anterior y la 3.4, y cuya consisitencia está garantizada por el teorema 3.2 es la siguiente:

Definición 3.8 Se define la clase **NP** como el conjunto de los lenguajes $L \subseteq \{0,1\}^*$ tales que existe un polinomio $p : \mathbb{N} \longrightarrow \mathbb{N}$ y una máquina determinista M que termina su computación en tiempo polinómico en la longitud de la entrada de forma que para cada $x \in \{0,1\}$ se tiene que

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ tal que } M(x,u) = M(xu) = 1 .$$

Los términos *verificador* y *certificado* se utilizarán como en la definición 3.4.

Otras clases menos relevantes que la **NP** de complejidad de tiempo indeterminista son las recogidas en la siguiente definición:

Definición 3.9 Se definen las siguientes clases:

- $\mathbf{NE} := \mathbf{NTIME}(2^{O(n)})$.
- $\mathbf{NEXP} = \mathbf{NEXPTIME} := \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(2^{n^k})$.
- 2-$\mathbf{NEXPTIME} := \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(2^{2^{n^k}})$ (y análogamente k-**NEXPTIME**, $k \in \mathbb{N}$).

El teorema de Jerarquía de Tiempo para el caso indeterminista nos permite establecer la siguiente cadena de contenciones:

$$\mathbf{NP} \subsetneq \mathbf{NE} \subsetneq \mathbf{NEXP} \subsetneq 2\text{-}\mathbf{NEXPTIME}\,.$$

EJEMPLO 3.10

De las definiciones se deduce inmediatamente que $\mathbf{P} \subseteq \mathbf{NP}$. Entonces sabemos que el problema de la multiplicación de enteros (ejemplo 2.21) y el de conectividad en grafos (ejemplo 2.23) están en **NP**.

El siguiente ejemplo es más interesante:

EJEMPLO 3.11 *(Conjunto independiente)*

Supongamos que queremos organizar una fiesta con el máximo número posible de amigos de tal forma que todos los invitados se lleven bien entre sí. Podemos modelizar esta situación a través de un grafo cuyos vértices identificamos con nuestro conjunto de amigos, de forma que dos de ellos están unidos por una arista si las dos personas que representan se llevan mal. Entonces, fijado un número mínimo de invitados k, la elaboración de la lista de invitados se reduce a encontrar en el grafo un conjunto independiente de al menos k vértices, es decir, tales que ningún par de entre ellos está unido por una arista. Si denotamos por $G = (V_G, A_G)$ a un grafo genérico (donde V es el conjunto de vértices y A el de aristas), nos interesa decidir el siguiente lenguaje:

$$\mathsf{INDSET} = \left\{(G,k) : \exists S \subseteq V_G \text{ tal que } |S| \geq k \text{ y } \forall\, u,v \in S, \overline{uv} \notin A_G\right\}.$$

Un algoritmo que decida este lenguaje nos permitirá saber si podemos organizar una fiesta con el número de invitados deseado, con lo que nos da una forma de encontrar el máximo número posible de invitados para nuestra fiesta sin más que ir aumentando el valor de k para nuestro grafo de amigos hasta que el par (G,k) no esté en INDSET.

La siguiente máquina de Turing M actúa como un verificador de dicho lenguaje: dado un par (G,k), donde G es un grafo y $k \in \mathbb{N}$, y una palabra $u \in \{(0,1)^*\}$, acepta si u es la codificación de una lista de al menos k vértices de G tales que no hay parejas unidas por una arista. Entonces $(G,k) \in \mathsf{INDSET}$ si y solo si existe una palabra u tal que $M((G,k),u) = 1$, que por lo tanto actuará como certificado. Nótese que si $n = V_G$ entonces una lista de k vértices puede codificarse utilizando $O(k\log(n))$ bits (símbolos del alfabeto binario), y la máquina M necesita sólo un número de pasos polinómico en el tamaño de la entrada para hacer las correspondientes comprobaciones (contar y buscar en el grafo proporcionado como entrada cada uno de los vértices listados en u y las aristas entre ellos), por lo que concluimos que $\mathsf{INDSET} \in$ **NP**, y también el problema de organizar la fiesta estará en **NP**.

Otros ejemplos de problemas en **NP** son los siguientes (no entraremos en detalles sobre el verificador y el tamaño del certificado, asumiremos que se ajustan al tiempo y tamaño polinómicos en la longitud de la entrada):

- *Circuito del viajante*: recuperando el problema del viajante (ejemplos 1.4 y 2.22), y considerando un modelo similar al del ejemplo de la fiesta que acabamos de ver, tendremos un grafo ponderado completo con n vértices, que se corresponderán con las n ciudades que debe visitar el viajante, y $\binom{n}{2}$ valores que se corresponderán con las distancias entre las ciudades. Dado $k \in \mathbb{R}$, nos interesará saber si existe un circuito cerrado que pase una vez por cada vértice y tal que la suma de las distancias a recorrer sea menor que k. Un certificado será una lista válida de vértices del grafo.

- *Suma de subconjunto*: dada una lista de números $n_1,...,n_k \in \mathbb{R}$ y $s \in \mathbb{R}$, decidir si existe un subconjunto de índices $\{i_1,...,i_m\} \subseteq \{1,...,k\}$ tal que $\Sigma_{j=1}^{m} n_{i_j} = s$. El certificado es la lista de índices de dicho subconjunto.

- *Programación lineal*: Dada una lista de m desigualdades lineales con coeficientes racionales sobre n variables, decidir si existe alguna asignación racional de las variables que satisfaga todas las desigualdades. El certificado es dicha asignación. Trabajamos en $\mathbb{Q}$ para evitar problemas de representación (que debe ser finita).

- *Números compuestos*: Dado $n \in \mathbb{N}$, decidir si n es compuesto. El certificado es la factorización de n.

- *Factorización*: dados $n,a,b \in \mathbb{Z}$, decidir si n tiene algún factor primo p en el intervalo $[a,b]$. El certificado es p (y el verificador utilizará la división euclídea).

- *Isomorfismo de grafos*: dados dos grafos con n vértices (por ejemplo, a través de sus matrices de adyacencia), decidir si son isomorfos (es decir, si son el mismo grafo salvo reordenación de los vértices). El certificado es la permutación de S_n que proporciona la reordenación correcta de los vértices del primer grafo, de forma que coincida con el segundo (que coincidan sus matrices de adyacencia).

Se ha demostrado que la programación lineal y el problema de los números compuestos están en **P**, dando algoritmos apropiados.

3.2.3 La clase de los complementarios (co-C). La clase co-NP

Dado un subconjunto de otro conjunto, en muy diversas áreas de las matemáticas resulta interesante estudiar su complementario, ya sea poque aporta información adicional sobre el conjunto original, porque facilite el tratamiento de aquel o por el interés que él mismo suscita. Puesto que hemos definido los lenguajes como subconjuntos del conjunto de palabras sobre un alfabeto, podemos considerar sus complementarios (es decir, el conjunto de las palabras sobre el mismo alfabeto que no están en el lenguaje original). Resulta evidente que estos nuevos lenguajes pueden ser clasificados en clases de complejidad, según hemos visto hasta ahora, pero nos interesará también clasificarlos según la complejidad de sus complementarios.

Definición 3.12 Sea **C** una clase de complejidad. Se define la *clase de los complementarios* **co-C** como sigue:

$$\textbf{co-C} := \left\{ L \subseteq \Sigma^* : \overline{L} \in \textbf{C} \right\}.$$

Resulta de gran interés comparar una clase **C** con la correspondiente clase **co-C**.

Lema 3.13 **P** = **co-P**.

Demostración. La idea básica de la demostración consiste en "dar la vuelta" a la respuesta de la máquina determinista, como ya hiciéramos en la prueba del teorema 2.17.

co-P ⊆ P

Sea $L \in$ **co-P**, por definición $\overline{L} \in$ **P** y por lo tanto existe una máquina determinista M que funciona en tiempo polinómico en la longitud de la entrada tal que dado $x \in \Sigma^*$, $x \in \overline{L} \Leftrightarrow Res_M(x) = 1$, con lo que $x \in L \Leftrightarrow Res_M(x) = 0$.

Consideremos la máquina $\tilde{M}$ que, sobre una entrada $y \in \Sigma^*$ devuelve $Res_{\tilde{M}}(y) = 1 - Res_M(x)$. Evidentemente se trata de una máquina determinista que funciona en tiempo polinómico sobre la longitud de la entrada, y cumple que $y \in L \Leftrightarrow Res_{\tilde{M}}(y) = 1$, de donde se deduce que $L \in$ **P**.

P ⊆ co-P

Sea $L \in$ **P**, entonces $\overline{L} \in$ **co-P** $\subseteq$ **P** y por lo tanto $L = \overline{\overline{L}} \in$ **co-P**. □

Mucho más interesante que la clase **co-P** es la clase **co-NP**, de complementarios de lenguajes en **NP**. El argumento de la demostración precedente, dar la vuelta a la respuesta, no es válido en el caso indeterminista, puesto que una de estas máquinas puede dar diferentes resultados sobre la misma entrada. De hecho la relación entre **NP** y **co-NP** es un problema abierto. Es sencillo ver que se tiene la siguiente relación:

Lema 3.14 **P** $\subseteq$ **NP** $\cap$ **co-NP**.

Demostración. Ya sabemos que **P** $\subseteq$ **NP**, solo nos falta comprobar que **P** $\subseteq$ **co-NP**. Sea $L \in$ **P**, como **P** = **co-P** se tiene que $L \in$ **co-P** (lema 3.13), con lo que $\overline{L} \in$ **P** $\subseteq$ **NP**, y por lo tanto $L \in$ **co-NP**. □

Dada la relevancia que tiene la clase **co-NP**, daremos una definición alternativa, más intuitiva y que simplificará el razonamiento para ver cuándo un lenguaje está en esta clase.

Definición 3.15 Alternativamente, se define la clase **co-NP** como el conjunto de los lenguajes $L \subseteq \{0,1\}^*$ tales que existe un polinomio $p : \mathbb{N} \longrightarrow \mathbb{N}$ y una máquina determinista M que termina su computación en tiempo polinómico en la longitud de la entrada de forma que para cada $x \in \{0,1\}$ se tiene que

$$x \in L \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)} \text{ se tiene que } M(x,u) = M(xu) = 1\,.$$

Se trata de una reescritura de la definición de **NP** en términos de certificados (definición 3.8), cambiando el cuantificador existencial $\exists$ por el universal $\forall$. La dualidad entre cuantificadores existenciales y universales permite generalizar estas clases definidas mediante cuantificadores, que se estructuran en la jerarquía polinómica **PH**, que no trataremos en este texto (para más detalles ver [3, capítulo 5]).

Proposición 3.16 Las dos definiciones de **co-NP**(3.12 y 3.15) son equivalentes.

Demostración. $L \in$ **co-NP** según la definición 3.12 si y solo si $\overline{L} \in$ **NP**, que es equivalente, de acuerdo con la definición 3.8, a que existan un polinomio $p : \mathbb{N} \longrightarrow \mathbb{N}$ y una máquina determinista M que termina su computación en tiempo polinómico en la longitud de la entrada de forma que $x \notin L \Leftrightarrow x \in \overline{L} \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)}$ tal que $M(xu) = 1$, y aplicando el contrarrecíproco, $x \in L \Leftrightarrow x \notin \overline{L} \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)}$ se tiene que $M(xu) \neq 1$. Puesto que M es determinista y termina su ejecución en todas las entradas, podremos asumir que $x \in L \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)}$ se tiene que $M(xu) = 0$, y consideramos la máquina $\tilde{M}$ que "da la vuelta" a la respuesta de M, es decir, tal que $Res_{\tilde{M}}(x) = 1 - Res_M(x)$; esta máquina obviamente termina su computación en tiempo polinómico en la longitud de la entrada, y se cumple que $x \in L \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)}$ se tiene que $\tilde{M}(xu) = 1$, es decir, si y solo si $L \in$ **co-NP** según la definición 3.15.

Puesto que el paso de modificar la máquina para "dar la vuelta" a la respuesta se puede efectuar en cualquiera de los dos sentidos manteniendo el tiempo de funcionamiento, se obtiene la equivalencia entre ambas definiciones. □

Al igual que la clase **NP** podía interpretarse como la clase de los lenguajes con un certificado corto de pertenencia, **co-NP** puede verse como aquella clase de lenguajes con un certificado corto de no pertenencia, es decir, las palabras que no están en el lenguaje poseen un certificado corto permite comprobarlo eficientemente.

EJEMPLO 3.17 *(TAUT)*

Se define el lenguaje TAUT como el conjunto de (codificaciones en un alfabeto fijado, por ejemplo el binario, de) las fórmulas booleanas $\Phi(X_1, ..., X_n)$ que son tautologías:

$$\text{TAUT} := \{\Phi : \forall x \in \{0,1\}^n, \Phi(x) = 1\}$$

Puesto que una fórmula booleana se expresa en términos de las funciones booleanas conjunción, disyunción y negación, se puede definir (no lo haremos explícitamente, aunque desde el punto de vista intuitivo es claro) una máquina de Turing determinista que, dada una fórmula booleana $\Phi(X_1, ..., X_n)$ y una asignación de variables booleanas $x \in \{0,1\}^n$, tenga como resultado $\Phi(x)$, que será 0 o 1. Entonces se cumple la definición 3.15 identificando la fórmula con la palabra y la asignación con la "parte adivinada" u, y por lo tanto TAUT $\in$ **co-NP**.

EJEMPLO 3.18 *(PRIMOS)*

Se define PRIMOS como el conjunto de (palabras sobre un alfabeto, por ejemplo el binario, que representan) números naturales primos. Aunque no lo detallamos, en la subsección 3.2.2 ya pusimos como ejemplo de lenguaje en **NP** el de los números compuestos, y como su complementario (dentro del conjunto de los naturales) es precisamente el conjunto de los primos, PRIMOS, se deduce que PRIMOS $\in$ **co-NP**. Un certificado corto de no pertenencia sería un divisor propio del número que consideremos, el verificador sería una máquina que ejecute el algortimo de la división euclídea.

CAPÍTULO 4

NP-COMPLETITUD

En el capítulo 3 introdujimos la noción de indeterminismo, definimos la clase **NP**, y anunciamos que se trataba de uno de los tópicos más relevantes de la Teoría de la Complejidad Computacional; es por ello que dedicaremos este capítulo a estudiarla con detalle.

En la subsección 3.2.2 vimos varios ejemplos de lenguajes en **NP**, en particular todos aquellos que están en **P**, además de otros aparentemente no tan eficientes de resolver, como el del conjunto independiente en un grafo (ejemplo 3.11) o el del circuito del viajante. Aunque no podemos asegurar categóricamente que esta apariencia sea real (precisamente sobre esto trata la conjetura de Cook, a la que dedicaremos el capítulo 5), la intuición nos incita a diferenciar dentro de la propia clase **NP** los problemas que, por el momento, parecen ser "verdaderamente difíciles" del resto. Esto nos conduce a definir la **NP**-dureza, cualidad de los problemas que son "al menos tan difíciles" como cualquier otro de la clase, y con ella la **NP**-completitud. Para ello necesitaremos una forma de relacionar problemas distintos, las reducciones. A estas definiciones dedicaremos las dos primeras secciones. Tal y como hemos hecho en los capítulos precedentes, aunque intuitivamente podemos hablar de problemas, las definiciones precisas las daremos en términos de lenguajes.

La siguiente sección consistirá en un estudio del lenguaje **NP**-completo por excelencia, SAT, y una variante suya, 3SAT, que serán de utilidad para probar que muchos otros lenguajes en **NP** son, de hecho, **NP**-completos, cosa que haremos en la sección 4.4, donde daremos algunos de los ejemplos más relevantes.

Por último, la sección 4.5 estará dedicada a presentar algunas técnicas para enfrentarnos a problemas **NP**-duros de forma razonable, aunque esto no implique obtener la solución a dichos problemas (la que nos proporcionaría, por ejemplo, el algoritmo de búsqueda exhaustiva en el problema del viajante, ver ejemplos 1.4 y 2.22).

4.1 REDUCCIONES

A continuación pretendemos formalizar en el concepto de reducción la noción de que un problema (un lenguaje) es "al menos tan difícil" o "tan duro" (en el sentido de la complejidad) como otro, de la que ya hablamos en la introducción del capítulo.

Definición 4.1 Dados dos lenguajes A y B, decimos que (el problema decisional asociado a) A *se reduce* o *es reducible* (al problema decisional asociado) a B si existe una transformación R, llamada reducción, tal que, para toda palabra $x \in \Sigma^*$, $x \in A \Leftrightarrow R(x) \in B$.

En otras palabras, para decidir si $x \in A$ solo tenemos que calcular $R(x)$ y comprobar si está en B (ver la figura 4.1).

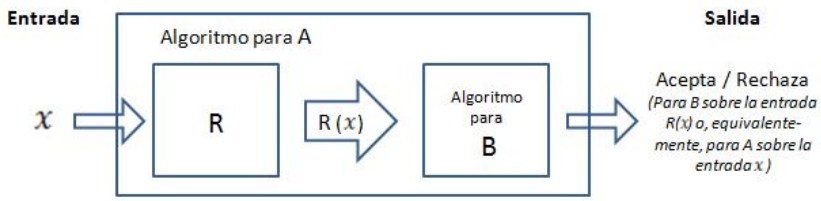

Figura 4.1: Reducción de A a B.

Sin embargo, para que esta definición se corresponda con la idea intuitiva que queremos precisar, debemos imponer sobre la reducción R la condición de que no sea "demasiado dura" de calcular, esto es, que no sea más dura que los lenguajes que relaciona.

EJEMPLO 4.2

Si consideramos el problema del viajante (ejemplo 1.4), podemos definir la siguiente reducción: se calcula la distancia de cada uno de los posibles itinerarios y se ordenan dichas distancias. Entonces el problema del viajante se reduciría al trivial de, dada una lista, devolver el primer elemento. Obviamente, lo que falla es que la reducción considerada requiere tiempo exponencial.

Entonces dada una reducción apropiada, decimos que un problema B es "al menos tan duro" como A si A se reduce a B.

Podemos definir varios tipos de reducciones, que inducen relaciones entre los lenguajes, a las que pediremos que cumplan las propiedades reflexiva (que es trivial, sin más que considerar la transformación que no modifica la entrada) y transitiva (aquellas reducciones que no cumplan esta propiedad no resultan tan interesantes).

Definición 4.3 Llamamos *reducción de Karp* o *en tiempo polinómico* a toda reducción f que sea una aplicación computable en tiempo polinómico en la longitud de la entrada (por una máquina de Turing determinista), $f \in \mathbf{PF}$. En este caso, dados dos lenguajes A y B decimos que A es Karp-reducible o reducible en tiempo polinómico a B, y lo denotamos por $\mathsf{A} \leq_p \mathsf{B}$.

Proposición 4.4 Sean A, B y C tres lenguajes. Se cumplen las siguientes propiedades:

1. (Transitividad) Si $\mathsf{A} \leq_p \mathsf{B}$ y $\mathsf{B} \leq_p \mathsf{C}$ entonces $\mathsf{A} \leq_p \mathsf{C}$.
2. Si $\mathsf{A} \leq_p \mathsf{B}$ y $\mathsf{B} \in \mathbf{P}$ entonces $\mathsf{A} \in \mathbf{P}$.
3. Si $\mathsf{A} \leq_p \mathsf{B}$ y $\mathsf{B} \in \mathbf{NP}$ entonces $\mathsf{A} \in \mathbf{NP}$.

Demostración.

1. Téngase en cuenta que si tenemos dos funciones $f(n) \leq n^c$ y $g(n) \leq n^d$ entonces su composición $g \circ f(n) \leq (n^c)^d = n^{cd}$, luego la composición de dos funciones (acotadas por) polinómicas es (acotada por una) polinómica. Si f_1 es una reducción de Karp de A a B y f_2 es una reducción de Karp de B a C, la aplicación $f_2 \circ f_1$ es computable en tiempo polinómico, pues las funciones de tiempo de las máquinas que las implementan son (acotadas por) polinómicas, y lo será su composición, puesto que en el peor de los casos la longitud de la salida de f_1 es polinómica en la longitud de la entrada, y esta coincide con la entrada de f_2. Además, dada una palabra $x \in \Sigma^*$, $f_2 \circ f_1(x) = f_2(f_1(x)) \in \mathsf{C} \Leftrightarrow f_1(x) \in \mathsf{B} \Leftrightarrow x \in \mathsf{A}$, luego $f_2 \circ f_1$ es una reducción de Karp de A a C, y por lo tanto $\mathsf{A} \leq_p \mathsf{C}$.

2. Sea $x \in \Sigma^*$, $x \in \mathsf{A} \Leftrightarrow f(x) \in \mathsf{B}$, donde f es una reducción de Karp, y por lo tanto computable en tiempo polinómico. Además existe una máquina de Turing determinista M que decide B en tiempo polinómico, con lo que la máquina $\tilde{M}$ que, dada la entrada x, calcula $f(x)$ y ejecuta M sobre $f(x)$ es determinista (pues lo era M y $f \in \mathbf{PF}$), y lo que acabamos de ver en la demostración del apartado anterior sobre la composición de funciones computables en tiempo polinómico se generaliza (utilizando el mismo argumento) a la composición (aplicación sucesiva) de máquinas de Turing.

3. Se demuestra análogamente al caso anterior, sustituyendo las máquinas deterministas por indeterministas.

□

Los apartados segundo y tercero de la proposición anterior motivan la importancia de estas reducciones, ya que permiten relacionar lenguajes en las clases **P** y **NP** sin salir de ellas. Puesto que estas son precisamente las clases que nos interesa estudiar, las reducciones con las que trabajaremos serán de este tipo.

Otra familia importante de reducciones es el siguiente:

Definición 4.5 Dados dos lenguajes A y B decimos que A es *Cook-reducible* a B si existe una máquina de Turing M con oráculo B que finaliza sus computaciones en tiempo polinómico en la longitud de la entrada y tal que $L(M) = A$.

Es posible expresar estas reducciones en los términos de la definición 4.1, aunque esta es más clara.

Resulta evidente que toda reducción de Karp es también una reducción de Cook, sin más que tomar como máquina M la que calcula la reducción f, y que tiene como cinta del oráculo la cinta de salida, con lo que una vez que acaba su ejecución sobre una entrada x, contiene en esa cinta $f(x)$ y accede al estado q_{cons}, devolviendo 1 o 0 según $f(x) \in B$ o no. Sin embargo no se sabe si las reducciones de Cook son más fuertes que las de Karp, de hecho se desconoce si la clase **NP** es cerrada bajo reducciones de Cook.

Definición 4.6 (*Reducción de Levin*) Una *reducción de Levin* de un lenguaje A en otro B es una reducción de Karp f que aplica los certificados de pertenencia de una palabra x a A en certificados de pertenencia de $f(x)$ a B, y viceversa.

Generalmente las reducciones de Karp que daremos serán, de hecho, reducciones de Levin. Las reducciones de Levin permiten transformar problemas de búsqueda en problemas decisionales.

Aunque imponen una limitación sobre el espacio necesario para computar la reducción, también en el estudio de clases de complejidad definidas por el tiempo (en especial en **P** y sus subclases) se utilizan en ocasiones las reducciones log-space.

A continuación introduciremos la noción de reducibilidad entre clases:

Definición 4.7 Decimos que una *clase de complejidad* **C** *es reducible* (Karp, Cook, Levin, log-space...) a otra clase **C'** si todos los problemas de la primera son reducibles (Karp, Cook, Levin, log-space...) a problemas en la segunda.

4.2 NP-COMPLETITUD

Una vez definidas las reducciones y formalizada la noción de un problema "al menos tan duro" como otro, nos interesa encontrar problemas que sean "al menos tan duros" como todos los de una clase, en lo que supone un primer paso hacia nuestro objetivo de encontrar problemas que "capten la esencia" de una cierta clase de complejidad.

Definición 4.8 (***C**-dureza*) Sea **C** una clase de complejidad, decimos que un lenguaje L es ***C**-duro* para un cierto tipo de reducciones si todos los lenguajes de la clase **C** son reducibles (para ese tipo de reducciones) a L.

Definición 4.9 (***C**-completitud*) Sea **C** una clase de complejidad, decimos que un lenguaje L es ***C**-completo* para un cierto tipo de reducciones si $L \in$ **C** y L es **C**-duro.

Las dos definiciones anteriores precisan, respectivamente, las nociones de lenguajes "al menos tan duros" como todos los de una clase y lenguajes que "captan la esencia de una clase" de las que hablábamos antes. Los lenguajes **C**-completos son "los más duros" de la clase **C**, con lo que podremos razonar sobre las clases por medio de estos lenguajes "representativos", como se aprecia en la siguiente proposición.

Proposición 4.10 Sean **C** y **C'** dos clases de complejidad con **C'** $\subseteq$ **C** y consideremos un cierto tipo de reducciones que dejan estable la clase **C'**.

1. Si un lenguaje L es **C**-duro y $L \in$ **C'** entonces **C** = **C'**.
2. Si un lenguaje L es **C**-completo entonces $L \in$ **C'** si y solo si **C** = **C'**.

Demostración.

1. Sea $A \in$ **C** un lenguaje, puesto que L es **C**-duro A es reducible a L, y como $L \in$ **C'** y la reducción f deja estable la clase **C'** (es decir, el cómputo de la reducción es un problema en **C'**), entonces la máquina M que, sobre una entrada x, calcula $f(x)$ y simula la ejecución de la máquina que decide L, decide A con los recursos acotados correspondientes a la clase **C'**, luego $A \in$ **C'** y por lo tanto **C** = **C'**.
2. Por ser L un lenguaje **C**-completo en particular es **C**-duro, y del apartado anterior se deduce una implicación. La otra implicación es inmediata, ya que si L es **C**-completo entonces $L \in$ **C**, y como por hipótesis **C**=**C'** se deduce que $L \in$ **C'**.

□

Esta proposición destaca la relevancia de los problemas **C**-completos para una cierta clase **C**: concentran en sí mismos todo el potencial de la clase y si uno cualquiera de ellos "cae" en una clase inferior (en el sentido de las contenciones) pero estable por las reducciones consideradas entonces toda la clase "cae" en esa subclase, en cuyo caso diremos que ambas clases *colapsan*.

En particular nos interesarán los problemas **NP**-completos bajo reducciones de Karp o en tiempo polinómico, puesto que precisamente pretendemos estudiar la tratabilidad de los problemas, y la clase **NP** representa precisamente el umbral de la intratabilidad. Resulta llamativa la presencia de dichos problemas en prácticamente todas las áreas de las matemáticas, lo que no hace sino aumentar su interés, ya que esto supone, tal y como acabamos de ver, que problemas de muy diversa índole concentran en sí mismos todo el potencial y la esencia de la clase **NP**. A continuación presentamos el primer ejemplo de lenguaje **NP**-completo.

Teorema 4.11 El siguiente lenguaje es **NP**-completo:

$$\mathsf{TMSAT} := \left\{(\alpha, x, 1^n, 1^t) : \exists u \in \{0,1\}^n \text{ tal que } \mathrm{Res}_{M_\alpha}(x,u) = 1 \text{ y } t_{M_\alpha}(x,u) \leq t\right\}$$

donde 1^n y 1^t son, respectivamente, las representaciones unarias (en el alfabeto $\{1\}$) de n y t.

Demostración. El hecho de que TMSAT $\in$ **NP** se deduce directamente de la definición 3.8, pues $(\alpha, x, 1^n, 1^t) \in$ TMSAT si y solo si existe $u \in \{0,1\}^n$ tal que $\text{Res}_{M_\alpha}(x,u) = 1$ y $t_{M_\alpha}(x,u) \leq t$ y bastará con modificar la máquina para que acepte como argumentos no sólo x y u, sino también $\alpha, 1^n$ y 1^t sin cambiar su funcionamiento en la práctica. Evidentemente dicha máquina termina en tiempo polinómico en la longitud de la entrada, pues uno de sus argumentos es justamente 1^t (que tiene longitud t), y también u tiene longitud polinómica en el tamaño de la entrada, ya que tiene longitud n y 1^n es parte de la entrada.

Veamos ahora que TMSAT es **NP**-duro. Sea $\mathsf{L} \in$ **NP**, según la definición 3.8 existe un polinomio $p : \mathbb{N} \longrightarrow \mathbb{N}$ y una máquina de Turing M que actúa como verificador tal que $x \in \mathsf{L}$ si y solo si existe un certificado $u \in \{0,1\}^{p(|x|)}$ tal que $\text{Res}_M(x,u) = 1$ y M termina su computación en tiempo polinómico en la longitud de la entrada, es decir, $t_M(x,u) \leq q(|x| + p(|x|))$ para algún polinomio $q : \mathbb{N} \longrightarrow \mathbb{N}$.

La reducción consistirá entonces en enviar cada palabra $x \in \{0,1\}^*$ a la correspondiente cuaterna $(\llcorner M \lrcorner, x, 1^{p(|x|)}, 1^{q(|x|+p(|x|))})$, esta asignación puede realizarse en tiempo polinómico (pues cada uno de los elementos se pueden obtener en tiempo polinómico a partir de lo anterior) y de acuerdo con la definición de TMSAT se tiene que

$$\begin{aligned} x \in \mathsf{L} \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ tal que } \text{Res}_M(x,u) = 1 \text{ y } t_M(x,u) \leq q(|x| + p(|x|)) \Leftrightarrow \\ \Leftrightarrow (\llcorner M \lrcorner, x, 1^{p(|x|)}, 1^{q(|x|+p(|x|))}) \in \text{TMSAT}\,. \end{aligned}$$

□

Este primer ejemplo de lenguaje **NP**-completo puede resultar un tanto artificial, construido especialmente para que cumpla la definición, pero al menos nos asegura la existencia de lenguajes **NP**-completos.

4.3 EL TEOREMA DE COOK-LEVIN

En esta sección veremos el primero de tales lenguajes **NP**-completos naturales que anunciamos en la sección anterior, en el que viene llamándose teorema de Cook o de Cook-Levin, puesto que fue probado de forma casi simultánea por estos dos matemáticos en sus artículos [10] (donde Cook introduce también la noción de **NP**-completitud) y [33] (en este caso Levin había dado una definición alternativa de la **NP**-completitud, basada en los problemas de búsqueda, y el lenguaje que prueba que es **NP**-completo es una variante del que veremos aquí).

Definición 4.12 *(SAT, SAT-CNF y 3SAT)*

- Se define el lenguaje SAT como el conjunto de las fórmulas booleanas Φ que son satisfactibles:

$$\text{SAT} := \left\{\Phi \text{ fórmula booleana} : \exists z \in \{0,1\}^* \text{ tal que } \Phi(z) = 1\right\}.$$

- Se define el lenguaje SAT-CNF como el conjunto de las fórmulas de SAT que son CNF.
- Se define el lenguaje 3SAT como el conjunto de las fórmulas de SAT que son 3CNF.

Teorema 4.13 (de Cook-Levin) SAT es **NP**-completo.

Demostración. Es evidente que SAT $\in$ **NP**, puesto que una asignación de variables que haga verdadera la fórmula sirve como certificado (que será suficientemente corto, de igual longitud que el número de variables de la fórmula), y el hecho de que esta asignación hace cierta la fórmula puede comprobarse en tiempo polinómico por una máquina de Turing que tenga implementadas las funciones $\wedge$, $\vee$ y $\neg$.

Entonces nos falta ver que SAT es **NP**-duro.

Sea $L \in$ **NP**, de acuerdo con la definición 3.8 existe una máquina de Turing determinista M tal que para todo $x \in \{0,1\}^*$, $x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)}$ tal que $\text{Res}_M(x,u) = 1$, donde $p : \mathbb{N} \longrightarrow \mathbb{N}$ es un polinomio. Probaremos que L es reducible Karp a SAT describiendo una transformación en tiempo polinómico (que será la reducción) $x \longmapsto \varphi_x$ de cadenas binarias a fórmulas booleanas tal que $x \in L$ si y solo si φ_x es satisfactible. Es decir, $\varphi_x \in \text{SAT} \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)}$ tal que $\text{Res}_M(x,u) = 1$.

La función booleana que aplica $u \in \{0,1\}^{p(|x|)}$ en $\text{Res}_M(x,u)$ no es válida en este caso, pues puede resultar demasiado larga, con lo que tendremos que buscar un método que nos proporcione fórmulas más cortas.

En primer lugar, de los teoremas 2.6 y 2.7 nos permiten suponer que la máquina M tiene solo dos cintas, una de entrada de solo lectura, y que es ajena a la entrada, manteniendo el tiempo polinómico en la longitud de la entrada (la simulación por una máquina ajena a la entrada en ese teorema no es compatible con la existencia de una única cinta, debe haber al menos dos, una de las cuales puede ser de solo lectura). Por ser M ajena a la entrada, podemos ejecutarla en la entrada trivial $(x, 0^{p(|x|)})$ para determinar la posición de los cabezales de ambas cintas en cualquier entrada de la misma longitud.

Sea Q el conjunto de estados de M y Σ su alfabeto. Dada una entrada y, en un determinado paso de cálculo i nos interesará la terna $(a,b,q) \in \Sigma \times \Sigma \times Q$, donde a y b son los símbolos contenidos en las celdas en las que se encuentra el cabezal en la primera y segunda cintas, respectivamente, y q es el estado almacenado en la unidad de control en ese paso de cálculo. Es evidente que esa terna puede codificarse como una cadena binaria, de longitud c, que es una constante dependiente de $|Q|$ y $|\Sigma|$. Para cada entrada y, esta terna en un paso de cálculo i depende del estado en el paso previo y del contenido de las celdas en cada una de las cintas.

La idea de la demostración consiste en que, como certificado de la existencia de la palabra u tal que $\text{Res}_M(x,u) = 1$, podemos dar la sucesión de las ternas que vamos obteniendo en la ejecución de M sobre la entrada (x,u), ya que en cada paso de cálculo modificamos a lo sumo una celda, es decir, es un proceso local en el que no se ve afectada toda la configuración, sino solo una pequeña parte. Para ver que dicha sucesión de ternas representa una

computación válida basta con comprobar que para cada paso de cálculo hasta que finaliza la ejecución, la terna correspondiente es válida dadas las ternas de cada uno de los pasos anteriores, pero en realidad es suficiente con fijarse en dos de dichas ternas anteriores (ya que a lo sumo se modifica el símbolo de la cinta de trabajo en cada paso). Por comodidad designaremos a la terna en el i-ésimo paso por z_i, entonces para comprobar que z_i es válida solo necesitamos z_{i-1}, $y_{posent(i)}$ y $z_{prev(i)}$, donde $y = xu$ (la concatenación de palabras). $posent(i)$ contiene la posición del cabezal de la cinta de entrada en el i-ésimo paso (recordemos que por ser la cinta de lectura no se modifica su contenido, que será siempre xu), y $prev(i)$ es el último paso de cálculo anterior al i-ésimo en el que el cabezal de la cinta de trabajo/salida estaba en la misma posición que en este último paso (si el i-ésimo es el primer paso en que el cabezal visita una cierta celda, fijamos $prev(i) = 1$). Esta información es suficiente, ya que el contenido de la celda actual no ha sido modificado desde el paso $prev(i)$ hasta el i (véase la figura 4.2).

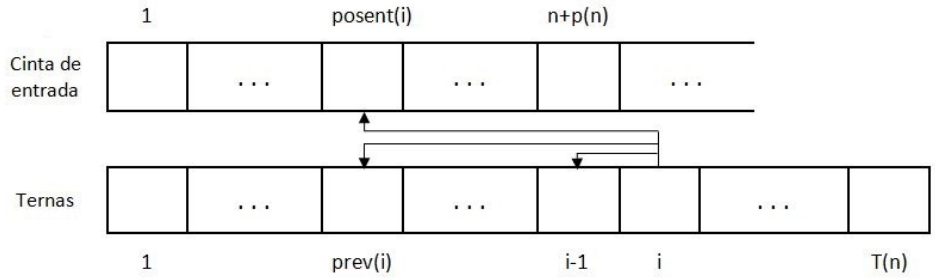

Figura 4.2: La terna correspondiente al i-ésimo paso depende del estado del paso anterior (contenido en la terna del $(i-1)$-ésimo paso), y de los símbolos que los cabezales leen de la cinta de entrada (es decir, el símbolo de la posición $posent(i)$) y de la cinta de trabajo/salida, que fue modificado por última vez en el paso $prev(i)$(y por lo tanto podemos encontrarlo en la terna correspondiente a dicho paso).

Puesto que M es una máquina determinista, una vez tenemos z_{i-1}, $y_{posent(i)}$ y $z_{prev(i)}$ a lo sumo una terna z_i es válida. Entonces existe una función $F : \{0,1\}^{2c+d}$ (obtenida a partir de la función de transición de M) donde d es la longitud de la escritura binaria de $y_{posent(i)}$ (la máxima para cualquier índice i, para así poder utilizarla como cota de dicha longitud), tal que una terna correcta z_i satisface $z_i = F(z_{i-1}, z_{prev(i)}, y_{posent(i)})$.

Por ser M ajena a la entrada, los valores $y_{posent(i)}$ y $prev(i)$ no dependen de la entrada $y = xu$ concreta, y pueden calcularse en tiempo polinómico simulando M sobre una entrada trivial, como indicamos previamente.

Recordemos que una entrada $x \in \{0,1\}^n$ está en L si y solo si $\text{Res}_M(xu) = 1$ para algún $u \in \{0,1\}^{p(n)}$, lo cual sucede si y solo si existen una cadena $y \in \{0,1\}^{n+p(n)}$ y una sucesión de cadenas (ternas codificadas) $z_1, ..., z_{T(n)} \in \{0,1\}^c$ (donde $T(n)$ es el número de pasos de cálculo que M realiza sobre una entrada de longitud $n + p(n)$) que satisfacen las siguientes condiciones:

1. Los primeros n símbolos (bits) de y son iguales a los de x.

2. La cadena z_1 codifica la terna inicial de M, esto es, la terna $(\triangleright, \triangleright, q_0)$.

3. Para cada $i \in \{2, ..., T(n)\}$, $z_i = F(z_{i-1}, z_{prev(i)}, y_{posent(i)})$.

4. $z_{T(n)}$ es la codificación de una terna en la cual M termina su ejecución y devuelve 1.

La fórmula φ_x tomará variables $y \in \{0,1\}^{n+p(n)}$ y $z \in \{0,1\}^{cT(n)}$ y deberá verificar que y, z cumplen la conjunción de las 4 condiciones anteriores, con lo que $x \in \mathsf{L} \Leftrightarrow \varphi_x \in \mathsf{SAT}$.

La condición 1 puede expresarse como una fórmula CNF de tamaño $4n$ (ver ejemplo 0.6). Las condiciones 2 y 4 dependen de c variables y por lo tanto pueden expresarse en términos de fórmulas CNF de tamaño $c2^c$, según el lema 0.5. La condición 3 es una conjunción de $T(n)$ condiciones, cada una de las cuales depende a lo sumo de $3c+d$ variables, con lo que puede expresarse como una fórmula de tamaño a lo sumo $T(n)(3c+d)2^{3c+d}$. Entonces la conjunción de todas estas condiciones puede expresarse como una fórmula CNF de tamaño $k(n+T(n))$, donde k es una constante que solo depende de la máquina M. Además, esta fórmula CNF puede computarse en tiempo polinómico en el tiempo de ejecución de M. □

De hecho, puesto que las fórmulas booleanas obtenidas son CNF, la demostración anterior es válida para el siguiente teorema:

Teorema 4.14 SAT-CNF es **NP**-completo.

Y aún podemos refinar más el resultado, viendo que 3SAT es **NP**-completo.

Teorema 4.15 3SAT es **NP**-completo.

Demostración. Siguiendo el mismo razonamiento que para SAT, se deduce que 3SAT $\in$ **NP**. Para ver que 3SAT es **NP**-duro, según la proposición 4.4 bastará con ver que SAT-CNF $\leq_p$ 3SAT. Queremos dar una transformación (en tiempo polinómico) que aplique cada fórmula CNF φ en una fórmula 3CNF ψ tal que ψ sea satisfactible si y solo si lo es φ.

Sea C una cláusula de φ con $k \geq 3$ literales, $C = u_1 \vee u_2 \vee \cdots \vee u_k$. Añadimos una nueva variable z y reemplazamos C por el par de cláusulas $C_1 = u_1 \vee u_2 \vee z$ y $C_2 = u_3 \vee \cdots \vee u_k \vee \overline{z}$, con 3 y $k-1$ literales respectivamente. Evidentemente si para una cierta asignación de las variables u_i, $i \in \{1, \ldots k\}$, C es cierta, existe una asignación de z que hace ciertas las dos cláusulas C_1 y C_2. Repitiendo el proceso tantas veces como sea necesario (rebajamos el número de literales en uno cada vez, aunque duplicamos el número de cláusulas) obtenemos una fórmula ψ 3CNF equivalente a φ, y esta fórmula puede obtenerse en tiempo polinómico en la longitud de φ, con lo que es una reducción de Karp válida. □

Comentarios sobre el teorema de Cook-Levin

Como ya hemos visto, la prueba del teorema de Cook-Levin es algo más fuerte de lo necesario, puesto que prueba de hecho que SAT-CNF es **NP**-completo. Esta prueba tiene dos puntos a los que se puede sacar más partido. Utilizando el resultado más fino de simulación por máquinas ajenas a la entrada demostrado en [42] podemos reducir el tamaño de la fórmula φ_x de $O(T(|x|)^2)$ a $O(T(|x|)\log(T(|x|))$.

Además, la reducción dada en la demostración no solo satisface que $x \in \mathsf{L} \Leftrightarrow \varphi_x \in \mathsf{SAT}$, sino que proporciona un procedimiento eficiente para transformar un certificado para x en una asignación que satisfaga φ_x, es decir, es una reducción de Levin (ver definición 4.6).

Nótese también que, pese a que probar que SAT es **NP**-completo podía haber resultado más fácil, hemos querido demostrar que también SAT-CNF y 3SAT lo son. Esto se debe a que estos lenguajes, en especial el último, resultarán mucho más útiles a la hora de probar la **NP**-completitud de otros lenguajes debido a la mayor simplicidad de la estructura combinatoria subyacente, lo que facilitará su uso en reducciones (recordemos que para probar que otro lenguaje es **NP**-duro bastará con reducir en tiempo polinómico cualquiera de estos a él, como consecuencia de la transitividad establecida por la proposición 4.4). El motivo de que tanto Cook como Levin prestaran atención a este tipo de problemas es la enorme importancia que la lógica proposicional juega en el ámbito de la lógica matemática.

4.4 LA RED DE REDUCCIONES: EJEMPLOS DE PROBLEMAS NP-COMPLETOS

Como acabamos de ver, para probar la **NP**-completitud de otros lenguajes solo tendremos que probar que están en **NP** y que existe una reducción de alguno de los lenguajes que ya hemos probado que son **NP**-completos en estos. Es por esto que podemos establecer una "red de reducciones" (técnicamente es suficiente con un "árbol de reducciones") para probar la **NP**-dureza de unos problemas a partir de otros.

Esta red empezó a construirse muy poco después de que Cook definiera la noción de **NP**-completitud y probara que SAT es **NP**-completo en [10]. La primera referencia en este sentido es [27], donde Karp demuestra la **NP**-completitud de 21 problemas de naturaleza combinatoria a través de reducciones, que se pueden ver en la figura 4.3.

En [3, figura 2.4] se puede ver un árbol más completo. En [22] Garey y Johnson presentan muchos más ejemplos de problemas **NP**-completos dando las correspondientes reducciones entre ellos. En [12] encontramos una compilación de problemas relacionados con **NP**, muchos de los cuales son **NP**-completos, y también existe una lista de problemas **NP**-completos en Wikipedia, con enlaces a los artículos de cada problema, [48].

A continuación daremos algunos ejemplos de lenguajes **NP**-completos detallando la reducción.

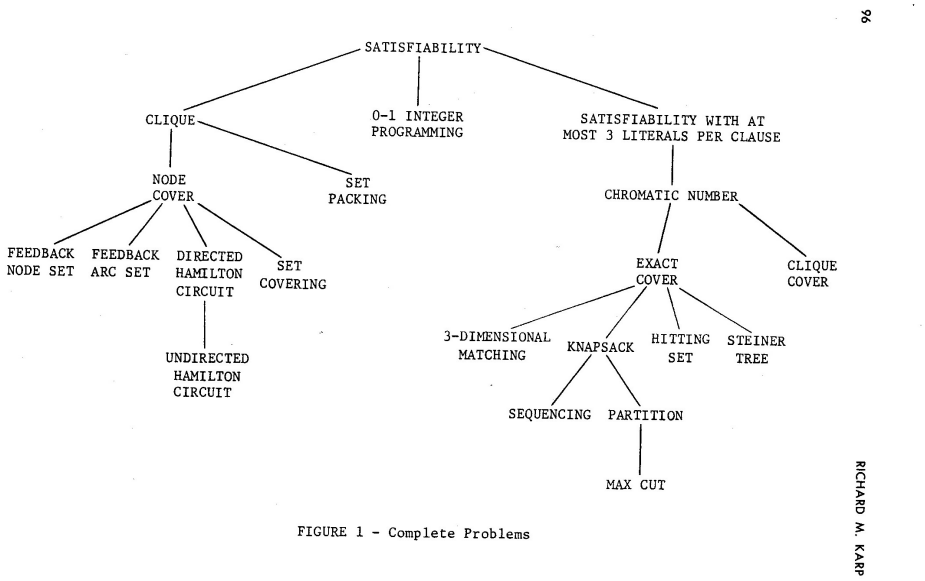

Figura 4.3: Árbol con las reducciones que Karp probó en [27], extraído del mismo artículo.

Teorema 4.16 $\mathsf{INDSET} = \{(G,k) : \exists S \subseteq V_G \text{ tal que } |S| \geq k \text{ y } \forall u,v \in S, \overline{uv} \notin A_G\}$ es **NP**-completo (V_G denota el conjunto de vértices del grafo G y A_G su conjunto de aristas).

Demostración. Ya tratamos este lenguaje en el ejemplo 3.11, y probamos que estaba en **NP**, con lo que solo nos falta ver que es **NP**-duro, para lo cual daremos una reducción de Karp de 3SAT en él. Es decir, presentaremos una transformación que se puede llevar a cabo en tiempo polinómico que transforma cada fórmula 3CNF φ con m cláusulas en un grafo G con $7m$ vértices tal que φ es satisfactible si y solo si G tiene un conjunto independiente de tamaño al menos m.

Para construir el grafo G, asociaremos un conjunto de 7 vértices a cada cláusula de φ, de forma que cada uno corresponda con una de las $2^3 - 1 = 7$ posibles asignaciones booleanas de las variables que hacen cierta la cláusula (puesto que es una disyunción no exclusiva de tres literales, la única de las 2^3 posibilidades de que no sea cierta es que los tres literales tomen el valor 0, y de ellos se recuperan los valores de las variables según fueran estas o su negación). Si alguna cláusula tuviera menos de 3 variables, se repite alguna de las asignaciones parciales y se hacen corresponder vértices distintos a asignaciones parciales iguales. Dos vértices de G estarán unidos por una arista si corresponden a asignaciones parciales inconsistentes, es decir, si dan valores distintos a alguna de las variables que comparten. También estarán unidos si corresponden al mismo grupo, es decir, si son posibles asignaciones parciales para la misma cláusula.

Evidentemente esta construcción puede hacerse en tiempo polinómico en el tamaño de la fórmula φ, luego solo nos falta probar que φ es satisfactible si y solo si G tiene un conjunto independiente de tamaño m:

- Supongamos que φ es satisfactible, es decir, existe una asignación u tal que $\varphi(u) = 1$. Definimos un subconjunto S de m vértices de G de la forma siguiente: para cada cláusula C de φ incluimos en S el vértice del grupo asociado a C que corresponde con la restricción de u a las variables de las que depende C. Puesto que todos los vértices de S se corresponden a restricciones de la asignación u estas no pueden ser inconsistentes y como elegimos uno de cada grupo se deduce que ninguno de dichos vértices estará conectado a otro de S, y formarán un conjunto independiente de tamaño m dentro de G.

- Supongamos ahora que G tiene un conjunto independiente S de tamaño m, querremos construir a partir de él una asignación $u \in \{0,1\}^n$ que haga cierta la fórmula φ, donde n denota el número de variables que intervienen en φ. La definimos de la forma siguiente: para cada $i \in \{1,\dots,n\}$ si existe algún vértice en S cuya asignación asociada dé a u_i el valor 1 fijamos $u_i = 1$, en otro caso $u_i = 0$. Puesto que S es un conjunto independiente las asignaciones parciales asociadas son todas consistentes, y por lo tanto cada u_i toma a lo sumo un valor en todas estas asignaciones, por lo que u está bien definido. Además, como hay aristas que unen los vértices del mismo grupo, correspondientes a asignaciones parciales de verdad de distintas cláusulas, no habrá dos vértices en S asociados a asignaciones de una misma cláusula, y como S tiene m elementos y hay m cláusulas, habrá un vértice por cada cláusula, que representa una asignación parcial de verdad para ella, y entonces la asignación u hará cierta la conjunción de todas las cláusulas, es decir, la fórmula φ.

□

Aunque en la subsección 3.2.2 afirmamos que el problema de la programación lineal está en **P** (como consecuencia del algoritmo del elipsoide de Khachiyan, ver [28]), podemos considerar el problema de la programación entera 0/1, es decir, donde las variables solo pueden tomar esos dos valores. Se define el lenguaje 0/1 IPROG como el conjunto de las desigualdades lineales con coeficientes racionales en n variables tales que existe una asignación de las variables en el conjunto $\{0,1\}$ que satisface todas las desigualdades. Entonces tenemos el siguiente teorema:

Teorema 4.17 0/1 IPROG es **NP**-completo.

Demostración. 0/1 IPROG $\in$ **NP**, pues la asignación de variables sirve como certificado (y evidentemente es suficientemente corto y verificable en tiempo polinómico).

Ahora es fácil reducir 3SAT a 0/1 IPROG en tiempo polinómico, pues cada fórmula 3CNF φ puede expresarse como un conjunto de desigualdades, una por cada cláusula, que será de la forma $\sum_{i=1}^{3} \tilde{u}_i \geq 1$, donde $\tilde{u}_i = u_i$ si en el literal correspondiente aparece u_i y

$\tilde{u_i} = 1 - u_i$ si en el literal aparece $\overline{u_i}$. Es evidente que cada una de las cláusulas admite una asignación parcial que la hace cierta si dicha asignación verifica la desigualdad correspondiente, y por lo tanto su conjunción (la fórmula φ) admite una asignación que la hace cierta si y solo si dicha asignación cumple todas las desigualdades del conjunto.

Entonces 0/1 IPROG es **NP**-duro y, por lo tanto, **NP**-completo. □

En cuanto al resto de ejemplos expuestos en la subsección 3.2.2, ya adelantamos que la programación lineal y el problema de los números compuestos están en **P**. El problema de la suma de subconjunto es **NP**-completo, así como el del circuito del viajante tal y como lo hemos definido en esa sección, que es la versión decisional del problema del viajante. Por contra, el problema de búsqueda, es decir, dado un grafo encontrar el circuito de longitud mínima, es **NP**-duro pero no está en **NP**, por lo que no es **NP**-completo. En cuanto al problema de la factorización y el del isomorfismo de grafos, aunque están en **NP** no se ha demostrado que sean **NP**-duros, y por lo tanto **NP**-completos. Hay algunos problemas interesantes más en esta situación, lo que sugiere la existencia de problemas **NP**-intermedios, a los que dedicaremos la sección 5.2.

Un ejemplo de gran interés está relacionado con el Nullstellensatz de Hilbert, cuya versión decisional puede interpretarse como la pertenencia de una lista de polinomios al siguiente lenguaje:

$$\mathsf{HN} := \left\{(f_1, \ldots, f_s) \in (\mathbb{K}[X_1, \ldots, X_n]_d)^s : \exists x \in \mathbb{K}^n \text{ tal que } f_1(x) = 0, \ldots f_n(x) = 0\right\},$$

donde por $\mathbb{K}[X_1, \ldots, X_n]_d$ denotamos a los polinomios en n variables de grado menor o igual que d sobre el cuerpo $\mathbb{K}$. Tenemos el siguiente resultado:

Teorema 4.18 Si $\mathbb{K}$ es un cuerpo finito entonces HN es **NP**-completo.

Demostración. Es evidente que $\mathsf{HN} \in$ **NP**, pues un punto de anulación de todos los polinomios es un certificado (corto y verificable en tiempo polinómico). Para ver que es **NP**-duro veremos que 3SAT es reducible en tiempo polinómico a él. Tendremos que dar un método para asociar a cada fórmula 3CNF φ un conjunto de polinomios en $\mathbb{K}[X_1, \ldots, X_n]_d$ tales que φ sea satisfactible si y solo si el sistema de ecuaciones polinómicas asociado a los polinomios tiene solución. Consideremos para cada variable u_i que aparece en φ el polinomio $X_i(1 - X_i)$, y para cada cláusula C de φ el polinomio $\prod_{u_i \in C} (1 - X_i) \prod_{\overline{u_j} \in C} X_j$.

Queremos ver que el sistema de ecuaciones polinómicas obtenido al igualar a 0 todos los polinomios anteriores tiene solución si y solo si φ es satisfactible. Si φ es satisfactible debe existir una asignación booleana que hace ciertas todas las cláusulas. Fijemos $x_i = u_i$ para todo i, evidentemente esta asignación de x satisface el primer tipo de ecuaciones polinómicas. Consideremos ahora una cláusula C de φ, será cierta si contiene al menos un literal cierto, es decir, $u_i = 1$ si aparece u_i o bien $u_i = 0$ si aparece $\overline{u_i}$ en C. Entonces si aparece u_i se tiene que $x_i = 1$ y por lo tanto $1 - x_i = 0$, y si aparece $\overline{u_i}$ se tiene que $x_i = 0$, y en ambos casos será cierta la ecuación polinómica correspondiente a dicha cláusula C, y esto debe ocurrir para todas las cláusulas, por lo que x es solución del sistema de ecuaciones polinómicas.

Supongamos ahora que tenemos una solución x del sistema de ecuaciones polinómicas. Por las ecuaciones del primer tipo, cada componente x_i tomará los valores 0 o 1, con lo que es correcto definir la asignación booleana u tal que $u_i = x_i$ para cada i. Las ecuaciones correspondientes a las cláusulas fuerzan a que cada cláusula tenga al menos un literal igual a 1 (ya sea una variable o su negación), con lo que todas las cláusulas serán ciertas con esta asignación, y será cierta la fórmula φ.

Esta asignación de un conjunto de polinomios a cada fórmula es evidentemente polinómica en el tamaño de la fórmula, luego la reducción es válida. □

El motivo de trabajar en cuerpos finitos es que necesitamos representar los coeficientes (y todo lo demás) en un alfabeto finito, el binario en este caso. El sentido de fijar el grado máximo de los polinomios es solo para hacer hincapié en el hecho de que, tal y como se deduce de la demostración, basta con considerar $d \geq 3$, puesto que estamos reduciendo 3SAT, que consiste en fórmulas CNF con tan solo 3 literales por cláusula.

El correspondiente problema de búsqueda (es decir, encontrar el conjunto de puntos de anulación de una colección finita de polinomios) es **NP**-duro, pero no **NP**-completo, pues no está en **NP**.

En [36] podemos encontrar un estudio más profundo de la relación del Nullstellensatz con la **NP**-completitud, incluyendo más reducciones de otros problemas **NP**-completos a HN (entre ellos INDSET).

Hemos visto ejemplos de problemas **NP**-completos de índole lógica, combinatoria e incluso algebraica. Esta es solo una pequeña muestra de la ubicuidad de estos problemas. Como curiosidad, en [1] se demuestra la **NP**-dureza de la generalización de algunos de los juegos clásicos de Nintendo, como los de la saga Super Mario Bros, Donkey Kong, Legend of Zelda, Metroid y Pokémon. Para Super Mario Bros y Donkey Kong se prueba, de hecho, que son **NP**-completos. En todos los casos las demostraciones de la **NP**-dureza se basan en la construcción de reducciones de 3SAT en estos problemas, es decir, lo mismo que nosotros hemos hecho en nuestros ejemplos.

4.5 LIDIANDO CON PROBLEMAS NP-DUROS

Después de lo que hemos visto en la sección anterior entendemos que no es extraño, sino más bien habitual, que tengamos que enfrentarnos a algún tipo de problema **NP**-duro. En este caso no conocemos ningún algoritmo que lo resuelva en tiempo polinómico, y cabe esperar que la solución sea difícil de encontrar. ¿Qué podemos hacer ante esta situación? Rendirnos es una opción, pero hay muchas otras antes. Después de todo, miles de niños hemos jugado en los últimos 20 años a los juegos de Nintendo a los que hacíamos referencia en la sección anterior sin saber que eran computacionalmente tan difíciles de resolver (aunque en momentos de frustración casi lo intuyéramos), y no parece probable que después de saberlo vayan a dejar de hacerlo.

Resolver un problema **NP**-duro es una tarea ardua, en general. Pero puede que para nuestros intereses sea suficiente resolverlo en la mayoría de las situaciones, o tan solo en una determinada, o que baste con obtener una "solución" suficientemente parecida a la que buscábamos, o incluso que no sea necesario que el problema que resolvamos sea exactamente el propuesto. Presentaremos entonces una pequeña colección de las técnicas más utilizadas para resolver estos problemas **NP**-duros.

1. **Fuerza bruta.** La forma más sencilla posible (y también la más ineficiente) de abordar un problema cualquiera (y en particular uno **NP**-duro) es la fuerza bruta, es decir, probar todas las posibles soluciones hasta dar con la correcta (búsqueda exhaustiva), que analizamos en el caso del problema del viajante en el ejemplo 1.4, o bien cualquier otro método basado en la repetición de procedimientos sencillos que nos den finalmente la solución correcta (como el algoritmo de suma repetida para la multiplicación de enteros, que analizamos en el ejemplo 1.3).

   Con el avance de los microprocesadores, capaces de realizar cientos de miles de millones de operaciones básicas por segundo, las técnicas de fuerza bruta han adquirido mayor relevancia, ya que pueden resolver en un tiempo razonable algunos problemas que, aunque no tienen un tamaño excesivo, se encontraban fuera de la capacidad computacional de una persona (por ejemplo el problema del viajante para 10 ciudades, la multiplicación de dos enteros de 4 cifras mediante la suma repetida o el problema de satisfabilidad de una fórmula booleana con 25 variables). Como curiosidad, el primer microprocesador de Intel, el Intel 4004, fue presentado en 1971 (el mismo año en que Cook definió la **NP**-completitud y probó que SAT es **NP**-completo). Con sus 92000 operaciones básicas por segundo, suponiendo que cada asignación de variables para una fórmula booleana con 40 variables es comprobada en 100 operaciones básicas, habría tardado hasta 2009 en comprobar cada una de las posibles asignaciones, mientras que un microprocesador de la misma compañía de 2009, por ejemplo el Intel i7-870, habría tardado aproximadamente 10 horas en realizar la misma tarea.

   Sin embargo no conviene fiarlo todo al avance de la capacidad computacional de los ordenadores, pues siempre seguirá habiendo problemas aún más grandes que nos interese resolver, y que continúen fuera de la capacidad de cálculo de cualquier ordenador.

2. **Búsqueda del algoritmo eficiente.** Tras descartar la fuerza bruta, una de las primeras opciones que un matemático barajaría al enfrentarse a un problema (quizás antes de saber que es **NP**-duro) es, en nuestro afán por abarcar siempre el caso más general posible, el desarrollo de un algoritmo que lo resuelva para cualquier conjunto de datos, y que sea eficiente, es decir, polinómico. Esto equivaldría a probar que el problema está en **P**. Esa sería la solución ideal, pero desgraciadamente no suele ser sencilla, especialmente para problemas **NP**-duros, en cuyo caso, como razonaremos en el capítulo 5, será mejor buscar otras opciones.

3. **Métodos heurísticos.** Una vez hemos descartado las dos primeras opciones, puede resultar interesante buscar métodos *ad hoc* para resolver nuestro problema concreto, analizando sus peculiaridades y buscando patrones que nos permitan acotar los casos en que tiene solución, o los tipos de solución que puede tener. Sin embargo, estos métodos, aunque suelen ser eficientes y proporcionar soluciones en muchos casos, de hecho en la mayoría de las situaciones prácticas, no funcionan en todas las circunstancias, por lo que no pueden ser considerados algoritmos que resuelven el problema. Entraríamos en la disputa entre la complejidad del caso peor y la del caso medio.

   Por ejemplo, en el problema de la coloración de mapas (que es **NP**-completo) podemos considerar la versión decisional del problema de los tres colores, esto es, dado un mapa, decidir si puede colorearse con 3 colores (además del azul para masas de agua) de forma que países vecinos estén coloreados de tonos distintos. Un simple análisis nos permite deducir la siguiente regla heurística: un mapa puede ser coloreado con 3 colores salvo que haya algún país completamente rodeado (es decir, sin salida al mar) por un número impar de países (técnicamente la condición es que haya un número impar de fronteras, un país puede tener con otro más de una frontera, o componente conexa de la frontera, como ocurre con España y Francia, que tienen en medio a Andorra). Esta regla heurística falla, por ejemplo, cuando hay cuatro países que comparten un punto de su frontera (que recibe el nombre de cuatrifinio), como ocurre con los estados americanos de Utah, Colorado, Nuevo México y Arizona (no está claro si Namibia, Botsuana, Zimbabue y Zambia forman otro cuatrifinio, o dos trifinios separados por apenas unos metros).

4. **Aproximación.** En ocasiones no podremos resolver un problema de forma exacta, pero sí podremos obtener una solución aproximada suficientemente buena para lo que necesitamos. Por ejemplo, en el problema del viajante tal vez no podamos encontrar la solución óptima, pero una solución un 1 % más larga será admisible para la mayoría de casos prácticos. De hecho en [2] se demuestra que para cada ε existe un algoritmo polinómico en $n(log(n))^{poly(1/\varepsilon)}$ para n ciudades que proporciona un circuito a lo sumo $1+\varepsilon$ veces peor que el óptimo ($poly(x)$ significa que es una expresión polinómica en x). Otra posibilidad es superponer sobre el mapa con las ciudades a visitar una malla tan fina como queramos, y resolver el problema del viajante para un único nodo por cuadro que determina la malla, con lo que disminuimos el número de puntos a visitar y se puede resolver por otros métodos (incluso el de fuerza bruta). Después se resuelve en cada cuadro el problema del camino hamiltoniano (que también es **NP**-completo), fijando los puntos inicial y final que nos interesen según la solución sobre el conjunto de los cuadros, y combinamos la solución sobre la malla total con las soluciones sobre cada cuadro, obteniendo una aproximación a la solución óptima más o menos buena en función de la malla. Una de las principales referencias sobre el problema del viajante, que incluye descripciones de métodos heurísticos y de aproximación, es [32].

5. **Modificar el problema.** Otras veces el problema que tenemos es muy difícil de resolver, y o bien no se admiten aproximaciones o bien no son suficientemente buenas. En algunos de esos casos existe la posibilidad de modificar el problema para convertirlo en otro que se pueda resolver fácilmente, o al menos al que se pueda aplicar alguna de las técnicas anteriores. Nótese que esto no implica que la solución a este problema sea una aproximación de la solución del problema original, simplemente estamos considerando otro problema distinto que puede servir igualmente para nuestro propósito original. Por ejemplo, los algoritmos criptográficos de clave privada están diseñados para que la decodificación sin la clave sea un problema difícil, que podríamos entender como **NP**-duro. Pero un delincuente podría tratar de modificar este problema (mediante procedimientos informáticos que afecten al algoritmo o físicos que afecten al dispositivo en el que está implementado) de forma que sea más fácil de resolver, obteniendo el mismo resultado que si hubiera roto el criptosistema encontrando la clave.

Hay muchas otras técnicas que pueden resultar útiles en la resolución, aunque sea parcial, de problemas **NP**-duros, por lo que en la mayoría de los casos prácticos (que son los que interesan a los no matemáticos) podremos encontrar una solución satisfactoria, aunque no sea óptima en términos de eficiencia y de exactitud. Rendirse siempre es una posibilidad, pero debería ser la última opción.

CAPÍTULO 5

LA CONJETURA DE COOK

El padre de Veruca, el señor Salt, había explicado a los periodistas con todo detalle cómo se había encontrado el billete. —Veréis, muchachos —había dicho—, en cuanto mi pequeña me dijo que tenía que obtener uno de esos Billetes Dorados, me fui al centro de la ciudad y empecé a comprar todas las chocolatinas de Wonka que pude encontrar. Debo haber comprado miles de chocolatinas. ¡Cientos de miles! Luego hice que las cargaran en camiones y las transportaran a mi propia fábrica. Yo tengo un negocio de cacahuetes, ¿comprendéis?, y tengo unas cien mujeres que trabajan para mí allí en mi local, pelando cacahuetes para tostarlos y salarlos. Eso es lo que hacen todo el día esas mujeres, se sientan allí a pelar cacahuetes. De modo que les digo: «Está bien, chicas, de ahora en adelante podéis dejar de pelar cacahuetes y empezar a pelar estas ridículas chocolatinas.» Y eso es lo que hicieron. Puse a todos los obreros de la fábrica a arrancar los envoltorios de esas chocolatinas a toda velocidad de la mañana a la noche. Pero pasaron tres días y no tuvimos suerte. ¡Oh, fue terrible! Mi pequeña Veruca se ponía cada vez más nerviosa, y cuando volvía a casa me gritaba: «¿Dónde está mi Billete Dorado? ¡Quiero mi Billete Dorado!» Y se tendía en el suelo durante horas enteras, chillando y dando patadas del modo más inquietante. Y bien, señores, a mí me desagradaba tanto ver que mi niña se sentía tan desgraciada, que me juré proseguir con la búsqueda hasta conseguir lo que ella quería. Y de pronto..., en la tarde del cuarto día, una de mis obreras gritó: «¡Aquí está! ¡Un Billete Dorado!» Y yo dije: «¡Dámelo, de prisa!», y ella me lo dio, y yo lo llevé a casa corriendo y se lo di a mi adorada Veruca, y ahora la niña es toda sonrisas y una vez más tenemos un hogar feliz.

Roald Dahl, *Charlie y la fábrica de chocolate*, 1964.

¿P=NP?

La respuesta a esta pregunta, en la que intervienen tan solo 4 símbolos si obviamos los de interrogación, está valorada en un millón de dólares, aunque podrían ser muchos más si fuera positiva. Esto se debe a que, como muchos de los lectores ya sabrán, se trata de uno de los siete problemas del milenio por los que el Clay Mathematics Institute ofrece dicha recompensa desde el año 2000, y de los cuales solo la hipótesis de Poincaré ha sido resuelta. Sin saber nada de Complejidad Computacional, esto ya nos da una idea de la importancia que tiene no solo dentro de esta área, sino en el conjunto de las Matemáticas. Pero después de haber leído los capítulos precedentes sabemos que hay más.

Supongamos que demostramos que $\mathbf{P} \neq \mathbf{NP}$. Adiós a nuestra esperanza de un mundo utópico (fantasías catastrofistas sobre robots inteligentes ansiosos por dominar el mundo aparte) donde las máquinas pueden resolver gran cantidad de problemas difíciles de muy

diversos ámbitos, aplicables a las comunicaciones, los transportes, la medicina, la economía... Al menos habremos ganado un millón de dólares, y durante algún tiempo seremos famosos.

Ahora supongamos que demostramos que **P** = **NP** (de hecho, puesto que evidentemente **P** ⊆ **NP** ⊆ **EXP** será suficiente con pobar la contención **NP** ⊆ **P**). A la vista de la proposición 4.10, bastará con probar que uno de los numerosos problemas **NP**-duros está en **P**, es decir, bastará con encontrar un algoritmo (una máquina de Turing) que lo resuelva en tiempo polinómico. Ahora bien, esto significará que todos los problemas en **NP** admiten un algoritmo que los resuelve en tiempo polinómico. Pero como el problema para el que hemos obtenido el algoritmo era **NP**-duro, todos los problemas en **NP** son reducibles en tiempo polinómico a él, y componiendo la reducción (o las sucesivas reducciones si hemos probado su **NP**-dureza mediante una sucesión de reducciones a través de diversos problemas **NP**-duros) con el algoritmo ¡obtenemos un algoritmo en tiempo polinómico para cada problema en **NP**! Y eso, como a estas alturas todos los lectores intuirán, vale mucho más de un millón de dólares.

La mala noticia es que en el resto del capítulo daremos gran cantidad de argumentos contra la igualdad, aunque nunca se debe perder la esperanza, especialmente teniendo en cuenta que la mayoría de los expertos en la Teoría de la Complejidad coinciden en que no se resolverá en este siglo, haciendo honor a su condición de problema del milenio.

La pregunta, conocida también como conjetura de Cook, o simplemente la Conjetura si estamos en el ámbito de la Complejidad Computacional, surgió de forma natural tras las definiciones modernas que Stephen Cook dio de **NP** y los problemas **NP**-completos, junto con la de **P**. A un nivel abstracto, la Conjetura puede verse como una pregunta sobre el potencial del indeterminismo en el modelo computacional de la máquina de Turing, ya resuelta para modelos más simples como los autómatas finitos.

No obstante, la definición de **NP** en términos de certificados y verificadores 3.8 sugiere que la Conjetura captura un fenómeno con una cierta importancia filosófica, el hecho de que verificar la validez de una respuesta es más simple que obtener dicha respuesta, al igual que demostrar un teorema es más difícil que comprobar que una demostración es correcta. En muchas ocasiones, la solución de problemas **NP**-completos parece pasar forzosamente por la búsqueda exhaustiva en un conjunto de tamaño exponencial en la talla del problema o por algoritmos no mucho mejores en términos de eficiencia. Mucha gente intuye que la fuerza bruta (equivalente al concepto *perebor* utilizado por los rusos) no puede evitarse, aunque formalizar esta intuición en una demostración de la Conjetura ha resultado ser una ardua tarea. En resumen, creen que el señor Salt no adoptó un método tan malo para conseguir aplacar la rabia de su caprichosa hija Veruca, aunque quizás debería haber leído antes la sección 4.5 y pensado una forma más barata de conseguir el preciado Billete Dorado.

5.1 OTROS PROBLEMAS ABIERTOS RELACIONADOS CON LA CONJETURA

En esta sección veremos algunos problemas abiertos cuya solución implicaría una respuesta a la conjetura de Cook, pero que, evidentemente, son al menos igual de difíciles de resolver que esta.

Teorema 5.1 Si **NP** $\neq$ **co-NP** entonces **P** $\neq$ **NP**.

Demostración. Veamos el contrarrecíproco: si **P** = **NP** entonces **NP** = **co-NP**. Pero esto es trivial, puesto que si **P** = **NP** entonces **co-P** = **co-NP** y tenemos la cadena de igualdades **co-NP** = **co-P** = **P** = **NP**. □

Evidentemente la mayoría de los investigadores piensan que **NP** $\neq$ **co-NP**. La intuición es casi tan clara como en el caso de **P** y **NP**: parece difícil creer que hay un certificado corto y verificable en tiempo polinómico que acredite que determinada fórmula booleana es una tautología, es decir, que certifique que todas las posibles asignaciones de variables la satisfacen. En la sección 5.3.3 ahondaremos más en las implicaciones que esta igualdad tendría de ser cierta.

Sabemos que **P** $\subseteq$ **NP** $\subseteq$ **EXPNEXP**, y a partir de esto surge la duda natural de si el estudio de la relación entre las clases exponenciales determinista e indeterminista, pese a comprender problemas completamente intratables, puede resultar de interés. El siguiente teorema está dirigido en ese sentido:

Teorema 5.2 Si **EXP** $\neq$ **NEXP** entonces **P** $\neq$ **NP**.

Demostración. De nuevo probaremos el contrarrecíproco: suponiendo que **P** = **NP** querremos ver que **EXP** = **NP**. Sea $L \in$ **NTIME**(2^{n^k}) y sea M una máquina de Turing indeterminista que lo decide. Entonces el lenguaje $L_{rell} := \left\{\left(x, 1^{2^{|x|^k}}\right) : x \in L\right\}$ está en **NP**, daremos una máquina de Turing indeterminista que lo decide: dada una entrada y, en primer lugar se comprueba si existe una cadena z tal que $y = \left(z, 1^{2^{|z|^k}}\right)$; si la respuesta es negativa devuelve 0. Si y sí es de esa forma entonces nuestra máquina simula la ejecución de M sobre la entrada z durante $2^{|z|^k}$ pasos y devuelve $\text{Res}_M(z)$. Evidentemente el tiempo de ejecución de esta máquina es polinómico en $|y|$, y por lo tanto $L_{rell} \in$ **NP**. Entonces como **P** = **NP** se tiene que $L_{rell} \in$ **NP**, y por lo tanto $L \in$ **EXP**, pues para determinar si una entrada $x \in L$ simplemente lo "rellenamos" y comprobamos con nuestra máquina si está en L_{rell}. □

5.2 ¿PROBLEMAS NP-INTERMEDIOS?

Como ya avanzamos en la sección 4.4, hay ciertos problemas interesantes para los que no se ha podido demostrar por el momento que estén en **P** ni que sean **NP**-completos, y aunque querríamos poder decir rigurosamente que son difíciles hay muy pocas formas de cuantificar esto, de ahí que en algunas ocasiones ciertos investigadores les otorguen sus propias clases de complejidad entre **P** y **NP** (por ejemplo **PPAD**, introducida por Christos Papadimitriou, que capta la esencia del problema de encontrar equilibrios de Nash en juegos de dos personas, pues es **PPAD**-completo).

En ocasiones se prueba que alguno de dichos problemas "no parece que" vaya a ser **NP**-completo, esto se razona demostrando que si lo fuera entonces contradiría alguna conjetura (de las que tenemos buenas razones para creer que son ciertas, como $\mathbf{P} \neq \mathbf{NP}$).

En base a todo lo anterior, tenemos razones para creer en la existencia de problemas **NP**-intermedios, es decir, que no son **NP**-completos pero tampoco están en **P**. Obviamente, esto solo tiene sentido si $\mathbf{P} \neq \mathbf{NP}$. El siguiente teorema, publicado por Richard Ladner en [31], asegura la existencia de dichos problemas en este caso.

Teorema 5.3 (**de Ladner**) Supongamos que $\mathbf{P} \neq \mathbf{NP}$, entonces existe un lenguaje $L \in \mathbf{NP} \setminus \mathbf{P}$ que no es **NP**-completo.

Demostración. Para cada función $H : \mathbb{N} \to \mathbb{N}$ definimos el lenguaje SAT_H como aquel que contiene todas las fórmulas booleanas satisfactibles de longitud n que hemos "rellenado" con $n^{H(n)}$ 1's:

$$\mathsf{SAT}_H := \left\{ \varphi 01^{n^{H(n)}} : \varphi \in \mathsf{SAT} \text{ y } n = |\varphi| \right\},$$

donde $|\varphi|$ denota la longitud de la fórmula.

Definimos la función $H : \mathbb{N} \longrightarrow \mathbb{N}$ de forma que $H(n)$ sea el menor entero $i < \log(\log(n))$ tal que para todo $x \in \{0,1\}^*$ con $|x| \leq \log(n)$, M_i (considerando ahora la representación binaria de i) devuelve $\mathsf{SAT}_H(x)$ en a lo sumo $i|x|^i$ pasos. En caso de que no exista tal entero i definimos $H(n) = \log(\log(n))$. H está bien definida puesto que $H(n)$ (es la representación de una máquina de Turing determinista que) determina la pertenencia a SAT_H de cadenas de longitud mayor que n, mientras que para definirla solo hay que comprobar la pertenencia de cadenas de longitud no superior a $\log(n)$ (en realidad tenemos un algoritmo "recursivo", es decir, que utiliza sus propios resultados sobre otras entradas en su proceso de computación sobre una entrada dada; que calcula $H(n)$ a partir de n y funciona en tiempo $O(n^3)$, aunque no lo necesitamos, pues basta con que el método de cálculo de H sea consistente). La función H así definida cumple el siguiente lema:

Lema 5.4 $\mathsf{SAT}_H \in \mathbf{P}$ si y solo si $H(n) = O(1)$ (es decir, H está acotada por una constante). Además si $\mathsf{SAT}_H \notin \mathbf{P}$ entonces $\lim_{n\to\infty} H(n) = \infty$.

Demostración del lema: Veamos que si $\mathsf{SAT}_H \in \mathbf{P}$ entonces $H(n) = O(1)$. Supongamos que existe una máquina de Turing determinista M que decide SAT_H en a lo sumo cn^c pasos de cálculo. Como M admite una infinidad de cadenas binarias que la representan, sea $i > c$ tal que $M = M_i$. Por definición de $H(n)$, para $n > 2^{2^i}$, $H(n) \leq i$, y por lo tanto $H(n) = O(1)$.

Recíprocamente, supongamos ahora que $H(n) = O(1)$, en ese caso la imagen de H está en un conjunto finito y debe existir i tal que $H(n) = i$ para infinitos valores de $n \in \mathbb{N}$. Pero esto implica que la máquina de Turing determinista M_i decide SAT_H en tiempo in^i, pues en otro caso existiría una entrada x sobre la cual M_i no devuelve la respuesta correcta en ese tiempo, y para todo $n > 2^{|x|}$ tendríamos que $H(n) \neq i$, con lo que llegamos a una contradicción. Esto es válido aún rebajando la hipótesis: basta suponer que $H(n)$ está acotado por una constante para infinitos valores de $n \in \mathbb{N}$, con lo que se demuestra la segunda parte del lema (que si $\mathsf{SAT}_H \notin \mathbf{P}$ entonces $\lim_{n\to\infty} H(n) = \infty$). □

Ahora utilizaremos este lema para probar que si $\mathbf{P} \neq \mathbf{NP}$ entonces SAT_H no está en $\mathbf{P}$ y tampoco es **NP**-completo:

- Supongamos que $\mathsf{SAT}_H \in \mathbf{P}$, entonces por el lema $H(n) \leq C$ para alguna constante C, y por lo tanto SAT_H es simplemente SAT "relleno" con a lo sumo una cantidad polinómica de 1's (n^C). Entonces una máquina de Turing determinista que resuelva SAT_H en tiempo polinómico puede utilizarse para resolver SAT (también en tiempo polinómico), y por lo tanto tendríamos $\mathbf{P} = \mathbf{NP}$, absurdo.

- Supongamos que SAT_H es **NP**-completo, entonces existe una reducción de SAT en SAT_H en tiempo $O(n^i)$ para $i \in \mathbb{N}$. Dado que ya hemos visto que $\mathsf{SAT}_H \notin \mathbf{P}$, el lema nos dice que $\lim_{n\to\infty} H(n) = \infty$. Puesto que la reducción funciona solo en tiempo $O(n^i)$, para n suficientemente grande aplicará fórmulas de SAT de longitud n en elementos de SAT_H de longitud menor que $n^{H(n)}$. Entonces aplicará una fórmula suficientemente larga φ en una cadena de la forma $\Psi 01^{|\Psi|^{H(|\Psi|)}}$, con $\Psi \in \mathsf{SAT}$ de longitud menor que n según cierto factor polinómico. Pero entonces esta reducción nos proporciona un algoritmo recursivo en tiempo polinómico para SAT, que implica que $\mathbf{P}$=$\mathbf{NP}$, con lo que tenemos una contradicción.

□

La idea de esta demostración se basa en que el lenguaje SAT_H para la función H que hemos definido "codifica" en sí mismo la dificultad de decidirlo. Aunque la demostración es perfectamente válida no ha podido ser extendida a ningún otro lenguaje más natural. Hay realmente muy pocos lenguajes candidatos a ser **NP**-intermedios, aunque entre estos destacan los dos que ya anunciamos previamente: la factorización de enteros y el isomorfismo de grafos (siendo rigurosos, los lenguajes asociados a estos problemas). En ambos casos no se ha logrado encontrar un algoritmo que los resuelva de forma eficiente, y hay una fuerte evidencia contra el hecho de que sean **NP**-completos.

5.3 LA UTOPÍA DE P=NP

Ya sabemos que basta con encontrar un algoritmo eficiente que resuelva cualquier problema **NP**-duro para que se tenga la igualdad **P** = **NP**, con lo que todos los problemas en **NP**, que son muchos y muy diversos, tendrían un algoritmo eficiente que los resuelve, lo que cambiaría radicalmente el mundo en que vivimos.

En general, si se da la igualdad en la conjetura de Cook, todo problema cuyas soluciones son verificables de forma eficiente (por medio de certificados cortos) puede ser resuelto también en tiempo polinómico. En particular esto será cierto para los problemas de búsqueda, como veremos en la subsección 5.3.1.

El software de inteligencia artificial sería prácticamente perfecto, ya que podría efectuar constantemente búsquedas exhaustivas en un árbol de posibilidades de tamaño exponencial de forma eficiente, tomando siempre la mejor decisión posible. Todos los problemas de optimización podrían resolverse siempre, con lo que las empresas tomarían siempre las decisiones que maximizaran lo beneficios, los ingenieros podrían obtener el diseño óptimo de piezas para cualquier cometido, los médicos prescribirían siempre las dosis óptimas de cualquier tratamiento, lo que supondría un aumento de nuestra calidad y esperanza de vida; la conducción autónoma sería posible, disminuyendo e incluso erradicando las muertes en la carretera, y un sinfín más de aplicaciones útiles.

También se podrían hacer predicciones mucho más exactas en diversos ámbitos, desde la meteorología (imagínense las repercusiones económicas que tendría conocer el tiempo con un año de antelación para el sector agrario o el turístico), a la economía (con predicciones en los mercados que permitirían detectar y corregir a tiempo las inestabilidades, en caso de que hubiera voluntad). Asimismo no habría necesidad de algoritmos aleatorios, puesto que no aportarían ninguna ventaja en términos de eficiencia sobre los deterministas.

Cada vez que un científico obtuviera datos experimentales, podría obtener eficientemente la teoría más simple que los explica que, según el principio de la navaja de Occam, pensador de origen inglés que vivió en los siglos XIII y XIV, será usualmente la correcta; este principio fue utilizado por ejemplo por Isaac Newton para describir sus tres famosas leyes de la dinámica, que sin embargo se ven invalidadas por la Teoría de la Relatividad, pues, como dijo el propio Einstein: “Hazlo todo tan simple como sea posible, pero no más simple”. Aún así, es innegable que las leyes de Newton han tenido una gran utilidad durante siglos, por lo que la posibilidad de obtener estas teorías simples de forma eficiente a partir de la experimentación servirá para ampliar considerablemente las fronteras del conocimiento.

En el ámbito matemático, no nos resultará difícil asumir que la resolución de ecuaciones diferenciales es un problema en **NP**, donde la solución es el certificado (técnicamente necesitaríamos que fuera suficientemente corta, si es necesario nos restringiremos solo a aquellas ecuaciones diferenciales con soluciones de longitud polinómica en la talla de la ecuación, las derivadas a realizar para la verificación se puede suponer en ese caso que son

calculables en tiempo polinómico). Entonces existiría un algoritmo eficiente que resolvería de forma exacta estas ecuaciones. ¡Adiós a los métodos numéricos!

Este mundo hasta ahora utópico también presenta algunas desventajas, como por ejemplo el fin de la criptografía de clave pública, basada en que para obtener la clave privada es necesario resolver problemas que, por el momento, están en **NP**, aunque no se ha probado que sean **NP**-completos (como la factorización de enteros en RSA o el logaritmo discreto en un grupo apropiado, generalmente el asociado a una curva elíptica, para el criptosistema de ElGamal). Tendremos que ingeniárnoslas para mantener la privacidad en la red de otra forma.

Seguramente algunas de las posibilidades que acabamos de ver en caso de que se dé la igualdad en la conjetura de Cook le parecerán demasiado buenas para ser ciertas. Eso significa que, aunque sea inconscientemente, usted piensa que $\mathbf{P} \neq \mathbf{NP}$.

A continuación indagaremos un poco más en algunas consecuencias de la igualdad en la Conjetura que tienen gran interés matemático.

5.3.1 Problemas decisionales y de búsqueda

Hemos definido de forma precisa las clases de complejidad como conjuntos de lenguajes que pueden ser decididos por máquinas de Turing con recursos acotados (ver definición 2.15), y toda la construcción y clasificación posterior se refiere a dichos problemas decisionales.

Por otro lado tenemos problemas de búsqueda, en los que nos planteamos encontrar la solución a un determinado problema, ya no nos limitamos a problemas cuya respuesta es solo verdadero o falso. Claramente el problema de búsqueda es más difícil que el correspondiente problema decisional (es decir, en el caso de SAT, decidir si una fórmula es satisfactible es más sencillo que encontrar una asignación que la haga cierta). Entonces si $\mathbf{P} \neq \mathbf{NP}$ ni el problema decisional ni el de búsqueda pueden ser resueltos en tiempo polinómico para un cierto problema **NP**-completo. No obstante, resulta que para problemas **NP**-completos los problemas decisionales y de búsqueda de certificados son equivalentes, es decir, si el problema decisional puede resolverse en tiempo polinómico (y por lo tanto **P**=**NP**), entonces la versión de búsqueda de certificados de cualquier problema **NP**-completo puede ser resuelto en tiempo polinómico, como nos asegura el siguiente teorema.

Teorema 5.5 Supongamos que $\mathbf{P} = \mathbf{NP}$. Entonces para cada lenguaje $L \in \mathbf{NP}$ y cada verificador M para L (que será una máquina de Turing determinista, según la definición 3.8) existe una máquina de Turing determinista N que funciona en tiempo polinómico tal que para cada entrada $x \in L$ devuelve un certificado para x respecto del lenguaje L y el verificador M.

Demostración. Tenemos que probar que si **P**=**NP**, entonces para cada máquina de Turing determinista M que funciona en tiempo polinómico y cada polinomio $p : \mathbb{N} \longrightarrow \mathbb{N}$ existe una máquina de Turing determinista D tal que para cada $y \in \{0,1\}^n$, si existe $u \in \{0,1\}^{p(n)}$ que cumple que $\mathrm{Res}_M(y,u) = 1$ (es decir, si existe un certificado de pertenencia de y a L verificable por M), se tiene que $|\mathrm{Res}_D(y)| = p(n)$ y $\mathrm{Res}_M(y, \mathrm{Res}_D(y)) = 1$.

En primer lugar probaremos el teorema para el lenguaje SAT: dada una máquina M que decida SAT en tiempo polinómico, construiremos una máquina D que al recibir una fórmula satisfactible φ en n variables $X_1, \ldots, X_n$ devuelva una asignación $x \in \{0,1\}^n$ que la haga cierta llamando $n+1$ veces a M y realizando algunos cálculos más en tiempo polinómico en el tamaño de φ.

D funciona de la forma siguiente: en primer lugar ejecuta M para comprobar que φ es satisfactible, en caso negativo se detiene, pues no existe el certificado que buscábamos. En otro caso sustituimos $X_1 = 0$ en φ y ejecutamos M sobre dicha fórmula modificada (en particular acortada, depende solo de $n-1$ variables), si está en SAT fijamos $x_1 = 0$, y si no está fijamos $x_1 = 1$. Repetimos el proceso ahora para la fórmula $\varphi(x_1, X_2, \ldots, X_n)$ en $n-1$ variables (pues una está fijada), obteniendo x_2 tal que $\varphi(x_1, x_2, X_3 \ldots, X_n) \in \mathsf{SAT}$. Tras n iteraciones tendremos la asignación $x = (x_1, x_2, \ldots, x_n)$ que hace cierta φ. El proceso de fijar una variable es polinómico en el tamaño de φ, por lo que D funciona como habíamos previsto.

Para extender esta solución del problema de búsqueda de certificados basta con recordar que la reducción del teorema de Cook-Levin de un lenguaje $L \in$ **NP** cualquiera en SAT es, de hecho, una reducción de Levin, y por lo tanto transforma en tiempo polinómico certificados de pertenencia a L en certificados de pertenencia a SAT, luego para obtener un certificado de pertenencia de y a L bastará con aplicarle la reducción correspondiente f, ejecutar D sobre $f(y)$ y recuperar (haciendo la contraimagen por f) el certificado para y en L, todo ello en tiempo polinómico en el tamaño de la entrada. □

Nótese que esta demostración implica que SAT es *autorreducible* (no utilizamos la palabra "reducción" en el sentido en que venimos haciéndolo usualmente), es decir, dado un algoritmo que lo resuelve para fórmulas con un múmero de variables menor que n, podemos resolverlo para fórmulas con n variables. Utilizando que todos los lenguajes **NP**-completos se reducen a este por reducciones de Levin, se puede probar que todos ellos son autorreducibles.

5.3.2 NP y las demostraciones matemáticas

Hemos estado manejando durante los últimos capítulos la noción de certificados, que a los matemáticos nos resultará vagamente familiar al tratarse de una reminiscencia de la de demostración matemática. En principio las matemáticas pueden ser axiomatizadas (véanse los axiomas de Peano o de Zermelo-Fraenkel), por lo que las demostraciones son simples manipulaciones formales de los axiomas, y la verificación de una demostración es relativamente sencilla de llevar a cabo, basta con comprobar que cada línea es consecuencia de

las anteriores aplicando los axiomas. Resultan interesantes los avances en el desarrollo y aplicaciones de demostradores automáticos y asistentes de demostración, como la familia HOL (del inglés *High Order Logic*, desarrollado por John Harrison) o Isabelle (desarrollado originalmente por la Universidad de Cambridge y el Technische Universität München).

Para los sistemas axiomáticos más conocidos esta verificación se puede realizar en tiempo polinómico en la longitud de la demostración, y por lo tanto el siguiente problema está en **NP** para los principales sistemas axiomáticos $\mathscr{A}$:

$$\mathsf{TEO} := \left\{(\Phi, 1^n) : \Phi \text{ admite una demostración formal de longitud } \leq n \text{ en el sistema } \mathscr{A}\right\},$$

considerando alguna medida apropiada de la longitud de una demostración.

Es bien conocido que el *Entscheidungsproblem* o problema de decisión, que plantea encontrar un algoritmo general que decida si una fórmula de cálculo de primer orden es un teorema (la formulación moderna en estos términos se debe a David Hilbert, en 1928), es incomputable (Alan Turing lo demostró en [45]). Atendamos a la siguiente cita de Kurt Gödel, fragmento de una carta enviada a John von Neumann en 1956 pero hecha pública dos décadas después:

> One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\Psi(F,n)$ be the number of steps the machine requires for this and let $\varphi = \max_F \Psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq kn$. If there really were a machine with $\varphi(n) \approx kn$ (or even $\varphi(n) \approx kn^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.
>
> Kurt Gödel en una carta a John von Neumann, 1956.

Gödel observa que el lenguaje que nosotros hemos definido como TEO es una versión finita del Entscheidungsproblem, y concluye que si pudiera ser resuelto en tiempo lineal o cuadrático (o polinómico en n en general, es decir, si $\mathsf{TEO} \in \mathbf{P}$) entonces la incomputabilidad del Entscheidungsproblem no sería tan desesperanzadora, pues en general estamos interesados en teoremas cuya demostración no es excesivamente larga.

Ciertamente usted ha experimentado en sus propias carnes que demostrar un teorema es mucho más difícil que comprobar que una demostración es correcta, lo que de nuevo significa que, intuitivamente, cree que $\mathbf{P} \neq \mathbf{NP}$.

5.3.3 ¿Qué ocurriría si NP=co-NP?

El teorema 5.1 nos dice que la igualdad **NP** = **co-NP** es una consecuencia de la igualdad en la conjetura de Cook, luego en particular en nuestro mundo utópico ocurriría todo

aquello que se pudiera deducir de dicha igualdad. Se desconoce también el recíproco, es decir, si **NP** = **co-NP** implica **P** = **NP**, por lo que merece la pena estudiar aparte algunas de las implicaciones que tendría **NP** = **co-NP**.

Un primer análisis de la igualdad en términos de certificados nos permite deducir que esta igualdad implicaría la existencia de certificados cortos y verificables en tiempo polinómico para problemas que intuitivamente no parece que los vayan a tener (véase el ejemplo ya indicado antes de las tautologías). Insistamos en el ejemplo de índole algebraica relacionado con el Nullstellensatz que vimos en la sección 4.4:

$$\mathsf{HN} := \{(f_1,\dots,f_s) \in (\mathbb{K}[X_1,\dots,X_n]_d)^s : \exists x \in \mathbb{K}^n \text{ tal que } f_1(x) = 0,\dots f_n(x) = 0\}.$$

Ya probamos que HN es **NP**-completo, si existe una solución del sistema de ecuaciones polinómicas hará las veces de certificado (siempre que pueda describirse en longitud polinómica). Evidentemente el problema de decidir si el sistema no tiene solución está en **co-NP**. Del teorema de los Ceros de Hilbert se deduce que el sistema no tiene solución si y solo si $1 \in \langle f_1,\dots,f_s\rangle$ (1 el polinomio constantemente igual a 1 y $\langle f_1,\dots,f_s\rangle$ el ideal generado por dichos polinomios), y esto ocurre si y solo si existen polinomios $g_1,\dots,g_s$ tales que $\sum_i f_i g_i = 1$. Entonces los polinomios $g_1,\dots,g_s$ son un cerificado de que el sistema no tiene solución. ¿Significa esto que el teorema de los Ceros de Hilbert demuestra que **NP** = **co-NP**? Lamentablemente la respuesta es negativa (como era de prever), pues el grado de los polinomios g_i puede ser exponencial en s y el número de variables, y por lo tanto no cumplen la condición de ser "cortos".

Si **NP** = **co-NP** existiría alguna otra noción de certificado corto que para los casos en que el sistema no tiene solución, resultado que podría superar en relevancia al teorema de los Ceros de Hilbert.

5.4 INDAGACIONES MÁS PROFUNDAS EN LA COMPLEJIDAD DE TIEMPO

Para concluir, resulta conveniente detenernos a analizar hasta dónde llega la teoría que hemos expuesto y esbozar qué hay más adelante. Desde que en el capítulo 2 definiéramos las primeras clases de complejidad, entre ellas la clase **P**, y enunciáramos la tesis de Cobham-Edmonds hemos estado estudiando la diferencia entre problemas resolubles (de forma determinista) en tiempo polinómico y no polinómico; la propia conjetura de Cook está orientada en ese sentido, pues se pregunta si todos los problemas verificables en tiempo polinómico son resolubles dentro de esa cota de tiempo.

Como ya hemos mostrado en el resto del capítulo, tenemos fuertes argumentos en contra de que **P**=**NP**, y por lo tanto resultará interesante poder decir algo más sobre la compleji-

dad de ciertos problemas en **NP**, más allá de la simple distinción entre resolubles en tiempo polinómico o no. La hipotética existencia de problemas **NP**-intermedios (ver sección 5.2), que como hemos visto es razonable, avala el interés de esta cuestión.

Volvamos al ejemplo del lenguaje INDSET, hemos probado que es **NP**-completo y por lo tanto creemos que no puede resolverse en tiempo polinómico. Pero ¿cuál es exactamente su complejidad? ¿Es $n^{O(\log(n))}$, o $2^{n^{0,2}}$ o $2^{n/10}$? Se cree que es $2^{\Omega(n)}$, es decir, el algoritmo de búsqueda exhaustiva está próximo a ser óptimo.

Ahora bien, podemos fijar el tamaño máximo del conjunto independiente, digamos k; entonces enumerar todos los conjuntos requeriría $\binom{n}{k}$ pasos, y $\binom{n}{k}$ puede aproximarse por n^k para $k \ll n$. La pregunta es entonces si este problema, en el que hemos fijado el parámeto k, puede resolverse de forma más eficiente, por ejemplo en tiempo $f(k)poly(n)$ ($poly(n)$ denota alguna expresión polinómica en n), con $f(k)$ una función dependiente de k.

Estas y otras cuestiones se abordan en la *teoría de la (in)tratabilidad con parámetro fijo*, definiendo la clase **FPT** de los lenguajes tratables al fijar un parámetro y su propio tipo de reducciones. De hecho, INDSET es uno de los múltiples problemas **FPT**-completos con respecto a dichas reducciones. Una buena referencia sobre esta teoría es [19], también lo son [16] y su expansión [15].

Con esta última sección se pretende poner de relieve la profundidad, que no completitud (los problemas abiertos son innumerables, y cada vez surgen más preguntas a las que no se encuentra respuesta) de la Teoría de la Complejidad Computacional, de la cual este texto no pretende ser sino una pequeña introducción a su parte más sencilla y conocida, (aunque no suficientemente): la Complejidad de Tiempo.

BIBLIOGRAFÍA

[1] ALOUPIS, G, DEMAINE, E.D.,GUO, A. y VIGLIETTA, G. Classic Nintendo Games are (Computationally) Hard. *Fun with Algorithms* en la serie *Lecture Notes in Computer Science*. 2014; 8496: 40-51.

[2] ARORA, S. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the Association for Computing Machinery*. 1998; 45:753-782.

[3] ARORA, S. y BARAK, B. *Computational complexity: a modern approach*. Nueva York, EEUU: Cambridge University Press; 2009.

[4] BLUM, M. A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery*. 1967; 14(2):322-336.

[5] BORODIN, A. Computational complexity and the existence of complexity gaps. *Journal of the Association for Computing Machinery*. 1972; 19(1):158-174.

[6] BUCHMANN, J.A. *Introduction to Cryptography*. 2ª ed. Nueva York, EEUU: Springer-Verlag; 2004.

[7] BROOKSHEAR, J.G. *Teoría de la Computación: Lenguajes formales, autómatas y complejidad*. Wilmington, Delaware, EEUU: Addison-Wesley Iberoamericana; 1993. Traducción del inglés por Ernesto Morales Peake, título original *Theory of Computation: Formal languages, automata and complexity*. Redwood City, California, EEUU: The Benjamin/Cummings Publishing Company; 1989.

[8] CHURCH, A. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*. 1936; 58(2):345-363.

[9] COBHAM, A. The intrinsic computational difficulty of functions. *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*. North-Holland, 1964; 24-30.

[10] COOK, S.A. The complexity of theorem proving procedures. *Proceedings Third Annual ACM Symposium Theory of Computing*. 1971; 151-158.

[11] COOK, S.A. A hierarchy for nondeterministic time complexity. *Journal of computer and system sciences*. 1973; 7(4):343-353.

[12] CRESCENZI, P. y KANN, V. *A compendium of NP optimization problems*. Recuperado el 28 de junio de 2016, desde: http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html

[13] DAVIS, M.D. y WEYUKER, E.J. *Computability, Complexity and Languages*. San Diego, California, EEUU: Academic Press, Inc.; 1983.

[14] DENNING, P.J., DENNIS, J.B. y QUALITZ, J.E. *Machines, Languages and Computation*. Englewood Cliffs, New Jersey, EEUU: Prentice-Hall; 1978.

[15] DOWNEY, R.G. y FELLOWS, M.R. *Fundamentals of Parameterized Complexity*. Londres, Reino Unido: Springer; 2013.

[16] DOWNEY, R.G. y FELLOWS, M.R. *Parameterized Complexity*. Berlín, Alemania: Springer; 1999.

[17] DU, D.Z. y KO, K.I. *Theory of computational complexity*. Nueva York, EEUU: John Wiley and Sons, Inc.; 2000.

[18] EDMONDS, J. Paths, trees and flowers. *Canadian Journal of Mathematics*. 1965; 17(1):449-467.

[19] FLUM, J. y GROHE, M. *Parameterized Complexity theory*. Berlín, Alemania: Springer; 2006.

[20] FORTNOW, L. *The golden ticket: P, NP, and the search for the impossible*. Princeton, New Jersey, EEUU: Princeton University Press; 2013.

[21] FORTNOW, L. y HOMER, S. A Short History of Computational Complexity. *Bulletin of the European Association for Theoretical Computer Science*. 2003; 80:95-133.

[22] GAREY, M.R. y JOHNSON, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, California, EEUU: W.H.Freeman and Company; 1979.

[23] GÖDEL, K. *On formally undecidable Propositions of Principia Mathematica and related Systems*. Mineola, Nueva York, EEUU: Dover Publications, Inc.; 1992. Traducción del alemán por B. Meltzer, título original Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*. 1931; 38:173-198.

[24] GÓMEZ, D. y PARDO, L.M. *Teoría de Autómatas y Lenguajes Formales (para Ingenieros Informáticos)*. Universidad de Cantabria; 2015. Recuperado el 12 de febrero de 2016, desde: http://personales.unican.es/pardol/Docencia/TALF2012.pdf

[25] HARTMANIS, J. y STEARNS, R.E. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*. 1965; 117:285-306.

[26] HENNIE, F.C. y STEARNS, R.E. Two-tape simulation of multitape Turing machines. *Journal of the Association for Computing Machinery*. 1966; 13(4):533-546.

[27] KARP, R.M. Reducibility among combinatorial problems. *Complexity of Computer Computations*. Editado por R.E. Miller y J.W. Thatcher. Nueva York, EEUU: Plenum Press; 1972.85-103

[28] KHACHIYAN, L.G. A polinomial algorithm in linear programming. *Soviet Math. Doklady*. 1979; 20:191-194.

[29] KLEENE, S.C. λ-definability and recursiveness. *Duke Mathematical Journal*. 1936; 2:340-353.

[30] KNUTH, D.E. Big omicron and big omega and big theta. *ACM SIGACT News*. 1976; 8(2):18-23.

[31] LADNER, R.E. On the structure of polynomial time reducibility. *Journal of the ACM*. 1975; 22(1):155-171.

[32] LAWLER, E.L., LENSTRA, J.K., RINNOOY KAN, A.H.G. y SHMOYS, D.B. (editores). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Nueva York, EEUU: John Wiley and Sons, Inc.; 1985.

[33] LEVIN, L.A. Universal sequential search problems. *PINFTRANS: Problems of Information Transmission* (traducción del ruso desde *Problemy Peredachi Informatsii*). 1973; 9(3):265-266.

[34] LEWIS, H.R. y PAPADIMITRIOU, C.H. *Elements of the theory of computation*. 2ª ed. Upper Saddle River, New Jersey, EEUU: Prentice-Hall; 1998.

[35] MANNA, Z. *Mathematical theory of computation*. Mineola, Nueva York, EEUU: Dover Publications, Inc.; 2003.

[36] MARGULIES, S. *Computer Algebra, Combinatorics, and Complexity: Hilbert's Nullstellensatz and **NP**-complete Problems*. (Tesis doctoral) University of California in Davis; 2008.

[37] MARTIN, J.C. *Lenguajes formales y teoría de la computación*. 3ª ed. México D.F., México: McGraw-Hill Interamericana; 2004. Traducción del inglés por Jorge Luis Blanco y Correa Magallanes, título original *Introduction to languages and the theory of computation*. 3ª ed. Nueva York, EEUU: McGraw-Hill; 2003.

[38] MATIYASEVICH, Y.V. *Hilbert's Tenth Problem*. Cambridge, Massachusetts, EEUU: MIT Press; 1993.

[39] PAPADIMITRIOU, C.H. *Computational Complexity*. Reading, Massachusetts, EEUU: Addison-Wesley Publishing Company; 1994.

[40] PARDO, L.M. La Conjetura de Cook (¿P=NP?). Parte I: Lo Básico. *La Gaceta de la RSME*. 2012; 15(1):117-147.

[41] PARDO, L.M. La Conjetura de Cook (¿P=NP?). Parte II: Probabilidad, Interactividad y Comprobación Probabilística de Demostraciones. *La Gaceta de la RSME*. 2012; 15(2):303-333.

[42] PIPPENGER, N. y FISCHER, M. J. Relations among complexity measures. *Journal of the Association for Computing Machinery*. 1979; 26(2):361-381.

[43] RAMOS, J. Sobre la naturaleza de la Tesis de Church. *Ideas y Valores*. 1993; 42(92-93):157-167.

[44] SALOMAA, A. *Formal Languages*. Nueva York, EEUU: Academic Press; 1973.

[45] TURING, A.M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings, London Mathematical Society*. 1936; 2(42):230-265.

[46] TURING, A.M. Systems of logic based on ordinals. *Proceedings, London Mathematical Society*. 1939; 2(45):161-228.

[47] VON ZUR GATHEN, J. y GERHARD, J. *Modern Computer Algebra*. 3ª ed. Nueva York, EEUU: Cambridge University Press; 2013.

[48] *List of NP-complete problems*. Wikipedia. Recuperado el 28 de junio de 2016, desde: https://en.wikipedia.org/wiki/List_of_NP-complete_problems

ÍNDICE ALFABÉTICO