COMPUTABILIDAD Y

COMPLEJIDAD COMPUTACIONAL

VERSIÓN PRELIMINAR (A PUBLICARSE EN 2025)

Ricardo Rosenfeld

Facultad de Informática

UNLP

Índice

Prólogo

Decir que un libro de texto es fascinante suena un poco extraño. Sin embargo, es la sensación que sentí leyendo este trabajo, que tengo el gran gusto de presentar.

Hace poco menos de 40 años, un joven filósofo me señalaba que las ciencias de la computación no le interesaban porque finalmente todo su contenido puede reducirse al funcionamiento de una *máquina de Turing*, dispositivo trivial que solamente puede cambiar un símbolo por otro y moverse sobre una cinta…

Dos años después, E. Dijkstra, tal vez el más agudo y crítico pensador del mundo de las ciencias de la computación, escribía un artículo llamado *On the cruelty of really teaching computing science* (*Sobre la crueldad de enseñar realmente las ciencias de la computación*). Allí señalaba que las computadoras digitales modernas introducían *dos novedades radicales*:

Por un lado: “Desde un bit hasta cientos de megabytes, desde un microsegundo hasta media hora de computación, ¡nos enfrentamos a una proporción absolutamente desconcertante de 10^9! El programador se encuentra en la posición única de que su disciplina y su profesión son las únicas en las que una proporción tan gigantesca, que desconcierta totalmente nuestra imaginación, tiene que ser gestionada por una sola tecnología. Tiene que ser capaz de pensar en términos de jerarquías conceptuales mucho más profundas que las que una sola mente jamás tuvo que afrontar antes.”

Y por otro lado: “La segunda novedad radical es que el computador automático es nuestro primer aparato digital a gran escala. Es posible, e incluso tentador, considerar un programa como un mecanismo abstracto, como un dispositivo de algún tipo. Sin embargo, hacerlo es sumamente peligroso: la analogía es demasiado superficial porque un programa, como mecanismo, es totalmente diferente de todos los dispositivos analógicos familiares con los que crecimos. Como toda información codificada digitalmente, tiene inevitablemente la incómoda propiedad de que las perturbaciones más pequeñas posibles – es decir, los cambios de un solo bit – pueden tener las consecuencias más drásticas. (Para completar, agrego que el panorama no cambia esencialmente con la introducción de redundancia o corrección de errores.) En el mundo discreto de la informática no existe una métrica significativa en la que los cambios pequeños y los efectos pequeños vayan de la mano, y nunca la habrá.”

Tal vez, el joven filósofo no haya tenido en cuenta estas particularidades, tal vez no haya podido reflexionar sobre las novedades señaladas por Dijkstra (un universo abstracto de jerarquías de complejidad, abarcando muchos de los problemas y dilemas planteados por los matemáticos y filósofos durante siglos). Tal vez, percibir la máquina de Turing como un mecanismo abstracto, pero analógico en su esencia, le haya impedido ver la íntima relación entre la lógica y la computación, mucho más que instrumental por cierto.

Este libro transita exactamente por los senderos que acabo destacar, y en tal derrotero presenta la verdadera estructura, no solamente lógica sino también epistemológica, de las ciencias de la computación. Tal cual la planteó Dijkstra.

Es sumamente interesante, y muy poco habitual, encontrar un texto de ciencias de la computación que proponga como objeto de estudio los problemas y sus resoluciones (o la inexistencia de las mismas), y no los formalismos y los teoremas. Este trabajo lo hace, el énfasis está puesto en abordar los problemas y pensar en sus resoluciones, apostando a la intuición pero sin perder rigor matemático. Otro aspecto para subrayar son sus ejemplos. Habitualmente menospreciados, sin ellos es imposible aprender y menos aún aprehender los conceptos de ninguna materia. La cantidad, variedad y graduación de los ejemplos presentados sugieren muchos años de trabajo y reflexión del autor.

El libro propone un fascinante viaje a través de dos jerarquías de problemas, que plasman las ideas de Dijkstra acerca de las novedades radicales introducidas por la computadora digital moderna:

La primera jerarquía es la *jerarquía de dificultad computacional*, la graduación entre lo posible y lo imposible utilizando esta clase de dispositivos digitales, que involucra los límites de la deducción y la verdad del pensamiento lógico, corporizados en clases de problemas computacionales y sus relaciones, y que a la vez rescata la idea inherente de mecanismos digitales abstractos que constituye a la programación y la hermana con la tradición de la matemática constructiva y la física. Así es que dicha jerarquía se construye con reducciones realizables, las cuales conforman la estructura, el esqueleto del universo de lo computable hasta el límite de lo incomputable.

La segunda jerarquía corporiza, en términos de clases de problemas, aquella primera cuestión radical. Es decir, ¿cuáles son las consecuencias de poder, con una única técnica – la técnica de programar –, abarcar una impensable dimensión de problemas, tanto en tiempo como en espacio? La respuesta en el libro es la segunda etapa del viaje que propone, atravesando en verdad dos jerarquías densas de problemas que se interrelacionan, la *jerarquía de tiempo* y la *jerarquía de espacio*. La atracción principal de esta parte del trabajo se concentra en la gran pregunta, aún abierta, de las ciencias de la computación: *¿P = NP?* No adelanto nada al respecto, pero sí digo que vale la pena estudiar con muchísima atención los capítulos dedicados a este tema, porque tal vez en él se encuentre uno de los desafíos intelectuales más significativos de la ciencia contemporánea. Ciertamente, la respuesta más factible a esa gran pregunta estructura la jerarquía de clases de problemas computacionalmente resolubles que se muestra, ilustrada progresivamente en los capítulos mediante una sucesión de gráficos, teniendo en cuenta distintas capacidades computacionales de tiempo y espacio.

El libro termina con un *bonus track*, que contiene una serie de tópicos avanzados de gran valor conceptual y hasta práctico en algunos casos. Particularmente, muy interesantes son los abordajes que se hacen de la *computación probabilística* y, en especial, la *computación cuántica*. La computación tradicional, determinística, que se desarrolla en el cuerpo principal del texto, es heredera de la física clásica, en particular de la mecánica, y más allá de las observaciones de

Dijkstra, conserva lo esencial del paradigma *estado-estímulo-estado*. La física probabilística, y su culminación en la física cuántica, rompen tal paradigma, cambiando por completo la forma en que las ciencias ven el mundo material. Constituyen una verdadera revolución. Así es que cabe tener muy en cuenta una distinta forma de computar, basada en esta otra manera de percibir el mundo. ¿Acaso estará en ella el futuro de la computación? Es fascinante no saberlo todavía…

Gabriel Baum

La Plata, septiembre de 2024

Introducción

En este libro se describen, de manera introductoria y sistemática, elementos clásicos de la *teoría de la computabilidad* y la *teoría de la complejidad computacional.*

Ambas teorías forman parte de la *teoría de la computación*. La computabilidad estudia los límites de lo *computable*, es decir, qué *problemas* se pueden resolver en el marco de un *modelo computacional universal* (modelización de una computadora), entendiendo por resolver un problema obtener, dada cualquier *instancia* del problema (un número N, una fórmula booleana φ, un grafo G), una *solución* de la instancia (la factorización de N, una asignación de valores de verdad que satisface φ, un camino en G entre dos vértices determinados). Por su parte, la complejidad computacional se enfoca en el *costo computacional* de las resoluciones de los problemas que se pueden resolver, considerando los recursos del modelo computacional de referencia.

El modelo computacional utilizado en el texto para estudiar la computabilidad y la complejidad computacional es una máquina muy simple e intuitiva que ejecuta *algoritmos*, la *máquina de Turing*, y el costo computacional de la resolución de un problema que se puede resolver se define en términos del *tiempo* y el *espacio* requeridos por una máquina de Turing que la lleva a cabo (precisando, la cantidad de pasos ejecutados y la cantidad de celdas de memoria ocupadas por la máquina, respectivamente), escenario comprobadamente apropiado para analizar de manera robusta la dificultad *inherente* de los problemas.

Si bien los algoritmos existen desde hace más de 2000 años, y ya antes del siglo XX se diseñaron dispositivos computacionales, se considera que los trabajos fundacionales de la computabilidad (y de las ciencias de la computación en general) son los de A. Church y A. Turing de 1936. Turing, en particular, formalizó la noción de lo computable con su *máquina de Turing*, que enseguida tuvo aceptación general.

Un poco después, pasada la segunda guerra mundial, comenzaron a construirse las primeras computadoras, y como consecuencia casi inmediata, la eficiencia de los algoritmos cobró protagonismo. En 1964, A. Cobham definió la clase P de los problemas con resoluciones algorítmicas de tiempo polinomial, y sugirió que dicha clase era una buena formalización de las computaciones eficientes. Un año más tarde, algo similar planteó J. Edmonds, y simultáneamente, J. Hartmanis, R. Stearns y P. Lewis presentaron los primeros análisis sistemáticos de complejidad temporal y espacial. Estos hitos forman parte esencial de los orígenes de la complejidad computacional, que desde entonces se ha desarrollado de manera ininterrumpida a través de notables creaciones y descubrimientos, desde el concepto de NP-completitud a comienzos de los años 1970, pasando por los algoritmos probabilísticos unos pocos años más tarde, y llegando en décadas más recientes, sólo por nombrar algunos ejemplos,

a los algoritmos cuánticos, las pruebas chequeables probabilísticamente y los generadores pseudoaleatorios.

Transcurridos varios años desde la publicación de mis dos libros anteriores sobre computabilidad y complejidad computacional, que son (Rosenfeld e Irazábal, 2010) y (Rosenfeld e Irazábal, 2013), me pareció oportuno producir un nuevo texto, alineado con el formato de las dos materias semestrales que dicto actualmente en la Facultad de Informática de la Universidad Nacional de la Plata, Teoría de la Computación y Verificación de Programas, de la Licenciatura en Informática, y Fundamentos de Teoría de la Computación, de la Licenciatura en Sistemas. He agregado contenidos y quitado o modificado otros, y sobre todo, he puesto mucho más énfasis que antes en lo conceptual, teniendo en cuenta lecciones aprendidas de mi tarea docente (afortunadamente permanentes), además de cambios en la currícula, nuevo material bibliográfico, y avances tecnológicos que no hacen más que ratificar la necesidad de conocer cabalmente los temas tratados.

Más allá de los cambios mencionados, mi objetivo último es el de siempre: transmitir conocimientos de computabilidad y complejidad computacional de la mejor manera posible (los docentes sabemos muy bien que la enseñanza de los temas formales requiere un esfuerzo especial), y así lograr que el lector capte la importancia de los mismos en su real dimensión, ojalá *saboreando* como yo los fascinantes resultados que se derivan de ellos.

Contenidos generales y estructura del libro

Los objetos de estudio de este libro son los problemas. Por la computabilidad se sabe que existen problemas que se pueden resolver sobre todas sus instancias (problemas *computables decidibles*), mientras que otros se pueden resolver sólo sobre algunas instancias (problemas *computables semidecidibles*) o bien ni siquiera parcialmente (problemas *no computables*); en una clasificación más simplificada, el primer grupo corresponde a los problemas *decidibles*, y los otros dos a los problemas *indecidibles*. Adicionalmente, por la complejidad computacional, naturalmente restringida a los problemas decidibles, con el soporte de varias décadas de experiencia algorítmica, se ha acordado que algunos problemas decidibles son *tratables* y otros son *intratables*, en el sentido de que únicamente los primeros pueden resolverse con máquinas de Turing en un tiempo y un espacio *razonables*. Categorizaciones de este tipo, en *clases de problemas*, constituyen el eje temático de todos los capítulos.

El libro consta de tres partes estructuradas en nueve capítulos, una primera parte introductoria (capítulos 1 y 2), una segunda parte de computabilidad (capítulos 3 a 5), y una tercera parte dedicada a la complejidad computacional (capítulos 6 a 9). La siguiente es una descripción general de los contenidos de cada capítulo:

- *Capítulo 1*. Se incluyen definiciones y notaciones básicas utilizadas a lo largo del libro.

- *Capítulo 2*. Se introduce la máquina de Turing, el modelo computacional que se utiliza en todos los capítulos, salvo en algunas secciones del último. Se describen distintas variantes de máquinas de Turing: con una cinta, con varias cintas, y la máquina de Turing no determinística. Además, se definen los *problemas de decisión* o *de tipo sí/no*, cuyas resoluciones producen exclusivamente *aceptaciones* o *rechazos* de instancias (una instancia es aceptada sii tiene solución). Esto permite que los problemas puedan tratarse como *lenguajes*. En general, todo lo que se desarrolla en el libro se lleva a cabo con este tipo de problemas, facilitando las presentaciones pero sin perder generalidad con respecto a los conceptos que se quieren analizar. El capítulo incluye notas adicionales sobre máquinas de Turing particulares, los *autómatas finitos* y los *autómatas con pila*, y sobre un modelo computacional equivalente al de la máquina de Turing, la *máquina de acceso aleatorio* o *máquina RAM*.
- *Capítulo 3*. Se describe la *jerarquía de la computabilidad*, cuyas clases estructuran el universo completo de los problemas. Los problemas se clasifican de acuerdo a su grado de dificultad, en el sentido de la computabilidad. Así, quedan establecidos formalmente los límites de la computabilidad y la decibilidad. Se presentan ejemplos de aplicación del método de prueba de *construcción de una máquina de Turing*, el método natural para probar que un problema es computable (decidible o semidecidible). Una nota adicional describe las *gramáticas*, que en el caso más general constituyen un modelo computacional equivalente al de las máquinas de Turing.
- *Capítulo 4*. El tema central de este capítulo es la *indecidibilidad*. Se describe el método de prueba de *diagonalización*, que permite encontrar problemas indecidibles, incluyendo ejemplos de aplicación. En particular, en una nota adicional se incluye la prueba, por diagonalización, de la *indecibilidad del problema de la detención de las máquinas de Turing*, presentada por Turing en su artículo de 1936.
- *Capítulo 5*. Completando la parte de computabilidad, se describe un tercer método de prueba, la *reducción*, tan relevante para la computabilidad (y la complejidad computacional) como los dos métodos descriptos previamente. Se introducen dos tipos de reducciones, las *m-reducciones* y las *Turing-reducciones*, estas últimas basadas en máquinas de Turing con *oráculo*. Empleando reducciones se pueblan y relacionan de manera sistemática las distintas clases de problemas de la jerarquía de la computabilidad. En una nota adicional se prueba, por medio de una m-reducción, la *indecibilidad de la lógica de predicados* (se comenta la que presentó Turing también en su artículo de 1936, y se desarrolla una posterior de Church).
- *Capítulo 6*. Con este capítulo se inicia la parte de complejidad computacional del libro. Se introducen las dos métricas, ya mencionadas, que se utilizan para analizar el grado de dificultad de un problema en dicho ámbito, el tiempo de ejecución y el espacio ocupado por una máquina de Turing que lo resuelve. Además, se presentan las definiciones y las características más relevantes de la *complejidad temporal*, la primera métrica considerada, incluyendo el aspecto de la representación de los datos. Se introduce una primera versión de la *jerarquía temporal*, que establece cómo se distribuyen los problemas decidibles entre

distintas clases de acuerdo a su grado de dificultad, en términos de la métrica del tiempo. Se destacan las clases de problemas P y NP, cuyas soluciones se pueden obtener o verificar, respectivamente, en tiempo polinomial, que es el tiempo asociado a los problemas que se consideran tratables. En una nota adicional se demuestra, por diagonalización, que existen problemas con tiempo de resolución mínimamente no polinomial.

- *Capítulo 7*. Se describen los problemas *NP-completos*, los problemas más representativos de la clase NP, en el sentido de que identifican la dificultad de toda la clase. Para su definición se retoman las m-reducciones, que en este caso requieren ser ejecutadas por máquinas de Turing de tiempo polinomial. Entre otras pruebas clásicas con m-reducciones polinomiales se incluye la de la *NP-completitud del problema de la satisfactibilidad de las fórmulas booleanas* (el Teorema de Cook-Levin, uno sino el más importante de la complejidad computacional).
- *Capítulo 8*. Se profundiza en la jerarquía temporal, y se analizan particularmente dos problemas de interés computacional, los problemas del *isomorfismo de grafos* y la *factorización*. Luego se pasa a tratar la complejidad computacional con respecto a la métrica del espacio, la *complejidad espacial*. Al igual que en la parte de complejidad temporal, se plantean las definiciones y las características más relevantes de este segundo escenario, y se especifica la jerarquía de clases de problemas correspondiente, la *jerarquía espacial*. Al final se relacionan las dos jerarquías de clases de problemas descriptas anteriormente, la jerarquía temporal y la jerarquía espacial, para obtener una única *jerarquía espacio-temporal*. Una nota adicional destaca el distinto impacto del no determinismo sobre el espacio en comparación con el tiempo.
- *Capítulo 9*. Finalmente, en el último capítulo del libro se desarrollan breves secciones con tópicos avanzados de complejidad computacional. Se incluyen los problemas de *búsqueda* y de *conteo* de soluciones, los problemas de *optimización* y las *aproximaciones polinomiales* que se utilizan para resolverlos eficientemente (con error acotado), la *jerarquía polinomial*, los problemas con *resoluciones paralelas eficientes*, los *algoritmos probabilísticos* y la jerarquía de clases de problemas asociada, las *pruebas interactivas*, y los *algoritmos cuánticos*. Para el desarrollo de algunos temas se introducen modelos computacionales alternativos al de la máquina de Turing clásica, como el *circuito booleano*, la *máquina de Turing probabilística*, el *sistema de pruebas interactivas* y el *circuito cuántico*.

Todos los capítulos a partir del segundo tienen la misma estructura. Comienzan con un *cuerpo principal*. Continúan con *notas adicionales* dedicadas a aspectos específicos de los temas tratados en el cuerpo principal. Y después siguen una sección con *observaciones finales* que remarcan los conceptos más relevantes del capítulo, una sección de *referencias* bibliográficas e históricas, y por último una sección de *ejercicios*.

Los resultados considerados más relevantes se presentan en forma de teoremas o ejemplos. En general, un teorema formula un enunciado que se quiere destacar (la indecidibilidad del problema de la detención de la máquina de Turing, la NP-completitud del problema de la

satisfactibilidad de las fórmulas booleanas, etc.), y un ejemplo ilustra una definición o un método de prueba (la descripción de una máquina de Turing, una diagonalización para encontrar un problema indecidible, etc). Un ejemplo puede también complementar o sustituir la prueba de un teorema. Teoremas y ejemplos se rotulan para facilitar su acceso desde cualquier capítulo. En muchos casos comienzan con una breve explicación informal, para que el lector, si lo desea, saltee el detalle que sigue en una primera lectura. En (Rosenfeld, 2010) y (Rosenfeld, 2013) se puede encontrar mayor detalle sobre algunos de los teoremas y ejemplos presentados.

Algunas explicaciones se acompañan con figuras, sobre todo las que tienen que ver con jerarquías de clases de problemas.

Además de los ejercicios al final de los capítulos, se intercalan otros a lo largo de sus desarrollos. La idea es involucrar al lector en la formulación de las definiciones, las pruebas de los teoremas y la descripción de los ejemplos, con el objeto de reforzar su entendimiento acerca de lo que se está explicando. Todos los ejercicios son en general sencillos (se prioriza lo conceptual).

No se incluyen muchas referencias. Se consideran las de relación más estrecha con los temas presentados. El lector puede encontrar más información bibliográfica e histórica en las siguientes publicaciones, en las que se basan varios pasajes de este libro: sobre la máquina de Turing y la computabilidad, (Hopcroft y Ullman, 1979), (Sipser, 1997) y (Lewis y Papadimitriou, 1998), y sobre la complejidad computacional, (Bovet y Crescenzi, 1994), (Papadimitriou, 1994), (Goldreich, 2008), (Arora y Barak, 2009) y (Moore y Mertens, 2011).

Al final del libro se agrega un *apéndice*, que repasa los teoremas, ejemplos y figuras que presentamos (además de símbolos, abreviaturas, jerarquías, clases, problemas y lenguajes), y se muestra la *bibliografía* referida en los distintos capítulos.

Estilo de presentación

Se combina rigor matemático con informalidad, para introducir los conceptos de una manera precisa y al mismo tiempo didáctica. Hay una moderada utilización de símbolos.

Los temas se desarrollan con una mirada *estructural*. Es decir, el enfoque no es ingenieril (éste no es un libro de algorítmica), sino que la característica de los problemas más tenida en cuenta es su estructura, en definitiva la que determina su grado de dificultad inherente. De esta forma, se privilegia la caracterización de clases de problemas y las relaciones entre ellas, en lugar de hacer hincapié en problemas particulares, salvo cuando un problema es muy representativo de una clase. En realidad, esta mirada es insoslayable en la teoría de la complejidad computacional, dada la abundancia de conjeturas que sostiene, producto de una importante cantidad de preguntas abiertas en la disciplina (que la distinguen marcadamente de la teoría de la computabilidad).

Para exponer los temas de computabilidad y complejidad computacional de la manera más integrada posible, se recurre una y otra vez al mismo modelo computacional, la máquina de

Turing; a los mismos métodos de prueba, la construcción de una máquina de Turing, la diagonalización y la reducción; y a los mismos conceptos fundamentales, como la *completitud*, el *peor caso* y la *búsqueda exhaustiva de posibles soluciones*.

A los lectores

El libro está dirigido principalmente a los estudiantes de los últimos años de la carrera de ciencias de la computación, y a graduados que deseen profundizar en algunos aspectos de la computabilidad o la complejidad computacional.

Se asume un lector con cierta madurez matemática, básicamente con alguna experiencia en pruebas matemáticas y nociones de matemática discreta. También se asumen algunos conocimientos de algorítmica.

El capítulo 1, de definiciones y notaciones preliminares, puede omitirse y recurrirse a él como guía de consulta cuando fuese necesario. Las notas adicionales también se pueden omitir, todas o algunas, en una primera lectura. El capítulo 2 debe leerse antes de las partes de computabilidad y complejidad computacional, ya que introduce la máquina de Turing en la que se basan, y éstas a su vez deben leerse en dicho orden, porque muchas definiciones y resultados de la parte de computabilidad se reutilizan en la parte de complejidad computacional.

A los docentes

El libro puede ser utilizado como base para el dictado de una materia de introducción a la computabilidad y la complejidad computacional semestral o anual, considerando 15 clases teóricas y 15 clases prácticas por semestre.

En el caso de una materia semestral, los capítulos 1 a 5 pueden dictarse en 6 clases, los capítulos 6 a 8 en 5 clases, y el capítulo 9 en 4 clases. La idea es dedicar la mayor parte del tiempo a las reducciones, las clases P y NP, los lenguajes NP-completos y la complejidad espacial.

En el caso de una materia anual, el dictado se puede estructurar, agregando contenidos, en dos unidades temáticas, una básica para el primer semestre y una avanzada para el segundo semestre, con el siguiente esquema. La unidad temática básica puede incluir los capítulos 1 a 7, es decir hasta la NP-completitud inclusive, y contenidos adicionales de computabilidad, por ejemplo sobre autómatas, lenguajes formales, gramáticas y modelos computacionales. Por su parte, a la unidad temática avanzada, basada en los capítulos 8 y 9, se le pueden agregar contenidos de complejidad computacional sobre criptografía, cotas mínimas considerando circuitos booleanos, tiempo promedio, generadores pseudoaleatorios y desaleatorización.

Agradecimientos

Agradezco a las autoridades de la Facultad de Informática de la Universidad Nacional de La Plata, y a las autoridades de la misma Universidad, por permitirme publicar este nuevo libro.

A mi familia, por su apoyo constante.

Al equipo docente con el que comparto desde hace varios años el armado y el dictado de las materias referidas, ¡incluidas las polémicas del caso, siempre constructivas!

Al profesor Gabriel Baum, quien con su generosidad habitual aceptó nuevamente prologar un libro mío, lo que de por sí le agrega muchísimo valor a este trabajo.

A quienes leyeron borradores del libro en distintas versiones de avance y me sugirieron valiosas correcciones.

Otra mención especial es para los excelentes profesores que tuve la suerte de conocer durante mi formación, en la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata, en la Escuela Superior Latinoamericana de Informática y en el Instituto de Tecnología Technión de Israel. De sus enseñanzas surgió mi amor incondicional por las disciplinas que trato de enseñar hace ya bastante tiempo, cada vez más apasionadamente.

Y como siempre, dejo para el final mi enorme agradecimiento a mis alumnos, cuyo acompañamiento desde hace más de tres décadas enriquece día a día mi actividad docente.

Ricardo Rosenfeld

La Plata, julio de 2024

CAPÍTULO 1
Definiciones preliminares

En este primer capítulo incluimos algunas definiciones básicas que utilizamos en el libro. El lector puede optar por saltear el capítulo y recurrir a él como material de consulta cuando lo crea necesario. Para facilitar el acceso a las definiciones, las agrupamos por afinidad en distintas secciones. Algunas definiciones se repiten o reformulan en los siguientes capítulos.

También aprovechamos para incluir algunas notaciones básicas. A lo largo del libro tratamos de recurrir siempre a la notación más habitual, empleando indistintamente símbolos o palabras.

Conjuntos, relaciones y funciones

Conjuntos. Un *conjunto* es una colección de *elementos*. A = {a, b, c} denota un conjunto A con elementos *a*, b y c, los cuales le *pertenecen*. $a \in A$ expresa que *a* pertenece a A, y $d \notin A$ expresa que d no pertenece a A.

La *cardinalidad* de un conjunto A, denotada con |A|, es la cantidad de elementos de A. Un conjunto puede ser finito o infinito, según su cardinalidad sea finita o infinita, respectivamente. El conjunto con cardinalidad cero, es decir el conjunto sin elementos, se denomina *conjunto vacío*, y se denota con $\emptyset$.

Dos conjuntos A y B son iguales (A = B) si tienen los mismos elementos, y si no, son distintos ($A \neq B$). Un conjunto A es un *subconjunto* de un conjunto B, o está *incluido* en B ($A \subseteq B$), si todos los elementos de A pertenecen a B, y está incluido *estrictamente* en B ($A \subset B$) si $A \subseteq B$ y $A \neq B$.

Dados dos conjuntos A y B, su *intersección* ($A \cap B$) es el conjunto de los elementos que están en A y en B, su *unión* ($A \cup B$) es el conjunto de los elementos que están en A o en B, y su diferencia (A – B) es el conjunto de los elementos que están en A y no están en B. Dos conjuntos A y B son *disjuntos* si $A \cap B = \emptyset$.

El *conjunto de partes* de un conjunto A, denotado con 2^A (también se usa $\mathcal{P}(A)$), es el conjunto de todos sus subconjuntos. Si $A \neq \emptyset$, una *partición* de A es un subconjunto de 2^A, tal que sus elementos son dos a dos disjuntos, ninguno es el conjunto $\emptyset$ y su unión es A.

El *producto cartesiano* de k conjuntos $A_1, \ldots, A_k$ ($A_1 \times \ldots \times A_k$) es el conjunto formado por todas las tuplas $(a_1, \ldots, a_k)$ tales que $a_1 \in A_1, \ldots, a_k \in A_k$. Una tupla $(a_1, \ldots, a_k)$ es distinta de un conjunto $\{a_1, \ldots, a_k\}$, sólo en una tupla importa el orden de los componentes, los cuales además se pueden repetir. $(a_1, \ldots, a_k) = (b_1, \ldots, b_k)$ sii $a_i = b_i$ para todo i.

Si una operación, aplicada sobre los elementos de un conjunto A, da siempre como resultado un elemento de A, se dice que A es *cerrado*, o que se cumple la *clausura* de A, con respecto a dicha operación.

$A = \{a \mid a \in B \text{ y } P(a)\}$ denota un subconjunto A de un conjunto B, cuyos elementos satisfacen una propiedad P. Por ejemplo, $\mathcal{N}_i = \{n \mid n \in \mathcal{N} \text{ y } n \text{ es impar}\}$ denota el subconjunto de los números naturales impares del conjunto $\mathcal{N}$ de los números naturales.

Relaciones. Dado un conjunto $A = A_1 \times \ldots \times A_k$, cualquier subconjunto R de A es una *relación* k-aria entre los conjuntos $A_1, \ldots, A_k$. $R(a_1, \ldots, a_k)$ expresa que la tupla $(a_1, \ldots, a_k)$ pertenece a la relación R.

Una relación binaria R incluida en A x A es *reflexiva* si para todo primer componente *a* cumple R(a, a), e *irreflexiva* si en cambio cumple $\neg R(a, a)$; es *simétrica* si para todo par (a, b) cumple $R(a, b) \rightarrow R(b, a)$, y *antisimétrica* si en cambio cumple $(R(a, b) \wedge R(b, a)) \rightarrow a = b$; y es *transitiva* si para todo par (a, b) y todo par (b, c) cumple $(R(a, b) \wedge R(b, c)) \rightarrow R(a, c)$.

Funciones. Una relación binaria f incluida en A x B, con a lo sumo un par (a, b) para todo primer componente *a*, es una *función* de A a B, y se denota con $f : A \rightarrow B$. Los conjuntos A y B son el *dominio* y el *codominio* de f, respectivamente. f(a) = b expresa que el par (a, b) pertenece a f, donde *a* es el *argumento* y b es el *valor*.

Una función $f : A \rightarrow B$ es *total* si está definida para todos los elementos de A, y *parcial* en caso contrario; es *inyectiva* si para todo par de argumentos a_1 y a_2 cumple $f(a_1) \neq f(a_2)$, *suryectiva* si está definida para todos los elementos de B, y *biyectiva* si es inyectiva y suryectiva. En este último caso también se dice que existe una *biyección* entre A y B.

La función *identidad* asigna a todo argumento *a* el valor *a*.

Dadas dos funciones $f : A \rightarrow B$ y $g : B \rightarrow C$, la *composición* de la función f con la función g es la función, denotada con $g \circ f$, que se calcula aplicando sucesivamente f y g. Es decir que dado *a*, cuando los valores f(a) y g(f(a)) están definidos, se cumple $g \circ f(a) = g(f(a))$.

Toda función inyectiva $f : A \rightarrow B$ tiene función *inversa* $f^{-1} : B \rightarrow A$, definida de la siguiente manera: si f(a) = b, entonces $f^{-1}(b) = a$.

Una función natural f es del *orden* de una función natural g, lo que se expresa con la notación f = O(g), conocida como *notación asintótica* u O Grande (*Big O*), si existe una constante c > 0 tal que $f(n) \leq c.g(n)$ para todo n mayor o igual que un determinado n_0. Una definición equivalente contempla la relación $f(n) \leq c.g(n)$ para todo n mayor o igual que cero. La notación asintótica la utilizamos en la parte de complejidad computacional del libro.

Algunos resultados sobre la cardinalidad de los conjuntos finitos. Dado un conjunto finito C de n elementos:

La cantidad de subconjuntos del *conjunto de partes* de C es 2^n.

La cantidad de tuplas del *producto cartesiano* C x C es n^2. Generalizando a k conjuntos no necesariamente iguales C_1, …, C_k, la cantidad de tuplas de C_1 x … x C_k es $|C_1|\ldots|C_k|$.

La cantidad de subconjuntos de k elementos de C (*combinaciones sin repetición* de n elementos tomados de a k) es $n!/((n - k)!.k!)$. La notación h! denota el factorial del número h, es decir: $h! = h.(h - 1)\ldots 2.1$.

La cantidad de tuplas de k elementos distintos de C (*variaciones sin repetición* de n elementos tomados de a k) es $n!/(n - k)!$

La cantidad de tuplas de k elementos no necesariamente distintos de C (*variaciones con repetición* de n elementos tomados de a k) es n^k.

La cantidad de tuplas de n elementos distintos de C (*permutaciones sin repetición* de n elementos) es n!

Conjuntos contables e incontables. Dos conjuntos tienen igual cardinalidad cuando se puede definir entre ellos una biyección.

Dados dos conjuntos A y B tales que $A \subset B$, si los conjuntos son finitos naturalmente se cumple $|A| < |B|$, pero si son infinitos no necesariamente sucede lo mismo.

Por ejemplo, dados el conjunto $\mathcal{N}_p$ de los números naturales pares y el conjunto $\mathcal{N}$ de los números naturales (que incluye estrictamente a $\mathcal{N}_p$), se cumple que sus cardinalidades coinciden, porque la función que asigna, para todo $k \geq 0$, a todo número natural par 2k el número natural k, es una biyección entre $\mathcal{N}_p$ y $\mathcal{N}$. Un ejemplo de conjuntos infinitos con cardinalidades diferentes lo constituyen $\mathcal{N}$ y el conjunto $\mathcal{R}$ de los números reales: se prueba que $|\mathcal{N}| < |\mathcal{R}|$ (intuitivamente, existen muchos más números reales que números naturales).

Los conjuntos finitos, y los conjuntos infinitos con la cardinalidad de $\mathcal{N}$, se conocen como *contables* (o *enumerables*), y los restantes como *incontables* (o *no enumerables*). Por lo tanto, $\mathcal{R}$ es incontable, y también lo es $2^{\mathcal{N}}$, porque se prueba que $|\mathcal{R}| = |2^{\mathcal{N}}|$, es decir que se puede definir una biyección entre el conjunto de los números reales y el conjunto de todos los subconjuntos de los números naturales. La Hipótesis del Continuo formula que no existe una cardinalidad intermedia entre $|\mathcal{N}|$ y $|\mathcal{R}|$, y se prueba que agregando la hipótesis o su negación a la teoría de conjuntos estándar no se producen contradicciones en la teoría (se dice que ambos enunciados son *independientes*).

Alfabetos y lenguajes

Alfabetos. Un *alfabeto* es un conjunto finito de símbolos. A partir de un alfabeto se pueden generar *cadenas de símbolos* (o directamente *cadenas*), tuplas de símbolos que se representan yuxtaponiéndolos. Por ejemplo, dado el alfabeto $\Sigma = \{a, b, c\}$, a, ab y bbac son cadenas que se pueden formar a partir de Σ.

El *tamaño* de una cadena w, denotado con |w|, es la cantidad de símbolos de w. La cadena *vacía*, denotada con λ, es la cadena de tamaño cero, es decir la cadena sin símbolos.

Dada una cadena $w = s_1 \ldots s_k$, la cadena $w^R = s_k \ldots s_1$ es la cadena *inversa* de w. Una cadena w es un *palíndromo* si $w = w^R$.

Lenguajes. Un *lenguaje formal*, o directamente un *lenguaje*, es un conjunto de cadenas con símbolos de un alfabeto determinado. El lenguaje *vacío*, denotado con $\emptyset$, es el lenguaje que no tiene cadenas. Y el lenguaje Σ^* es el lenguaje de todas las cadenas con símbolos del alfabeto Σ. Por ejemplo, a partir del alfabeto $\Sigma = \{a, b, c\}$ se obtiene el lenguaje $\Sigma^* = \{\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, \ldots\}$. De esta manera, Σ^* es un lenguaje infinito y todas sus cadenas son finitas. Además, Σ^* es contable, y por lo tanto 2^{Σ^*}, que es el conjunto de todos los lenguajes que se pueden obtener a partir del alfabeto Σ, es incontable.

El orden de las cadenas de Σ^* mostrado en el ejemplo anterior es el que utilizamos en el libro. Lo denominamos *canónico*. Formalmente, establece que una cadena precede a otra de longitud mayor, y que en caso de longitudes iguales desempata el orden lexicográfico.

La *función censo* $c_L(n)$ de un lenguaje L asigna al argumento n la cantidad de cadenas de L de tamaño a lo sumo n.

Como los lenguajes son conjuntos, sobre ellos se pueden aplicar las operaciones que definimos anteriormente (intersección, unión, diferencia, etc). Otras operaciones sobre lenguajes son el *complemento* y la *concatenación*:

El *complemento* de un lenguaje L_1 con respecto a un lenguaje L_2 es el lenguaje $L_2 - L_1$, y se lo denota con L_1^C (precisando el lenguaje L_2 que se toma como referencia). Por ejemplo, dado el alfabeto $\Sigma = \{a, b\}$ y el lenguaje $L_1 = \{\lambda, a, b\}$, el complemento del lenguaje L_1 con respecto al lenguaje Σ^* es $L_1^C = \{aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots\}$.

Por su parte, la *concatenación* de dos lenguajes L_1 y L_2, denotada con $L_1.L_2$, es el lenguaje que se obtiene yuxtaponiendo las cadenas de L_1 con las cadenas de L_2. Por ejemplo, dados los lenguajes $L_1 = \{00, 11, 101\}$ y $L_2 = \{0, 1\}$, se cumple $L_1.L_2 = \{000, 001, 110, 111, 1010, 1011\}$.

Cadenas que codifican números. En el libro consideramos los siguientes dos sistemas de codificación de números:

El *sistema de codificación unaria*, en el que los números se representan con cadenas de unos. Por ejemplo, el número 5 se representa con la cadena 11111. De esta manera, el tamaño de una cadena que codifica un número n es n.

El *sistema de codificación binaria*, en el que los números se representan con cadenas de unos y ceros. Por ejemplo, el número 5 se representa con la cadena 101. En este caso, el tamaño de una cadena que codifica un número n es $O(\log_2 n)$.

Pruebas matemáticas y métodos de prueba

Pruebas matemáticas. Una *prueba matemática* es una secuencia finita de pasos, cuyo propósito es determinar la verdad de un enunciado matemático.

Métodos de prueba generales. Entre los métodos de prueba generales a los que recurrimos en el libro se encuentran la *prueba por contradicción* (o *método de la reducción al absurdo*), el *método del contrarrecíproco* y la *inducción matemática*.

La prueba por contradicción se basa en el *principio del tercero excluido*, según el cual si un enunciado afirma algo y otro lo contradice, necesariamente uno debe ser verdadero y el otro falso. En base a esto, la prueba consiste en suponer la falsedad del enunciado que se quiere demostrar, llegar a una contradicción, y entonces concluir que el enunciado es verdadero. Formalmente, si p es un enunciado verdadero, la prueba de un enunciado q consiste en asumir $\neg(p \longrightarrow q)$, o lo que es lo mismo $p \wedge \neg q$, y llegar a una contradicción.

El método del contrarrecíproco consiste en probar, en lugar de $p \longrightarrow q$, la implicación equivalente $\neg q \longrightarrow \neg p$. De esta forma es similar a la prueba por contradicción, salvo que en vez de partir de $p \wedge \neg q$ parte de $\neg q$.

Con respecto a la inducción matemática, establece que para probar que una propiedad se cumple para todos los números naturales, hay que probar que se cumple para el 0, y que si se cumple para el número natural k entonces también se cumple para el número natural k + 1. Por ejemplo, la siguiente es una prueba por inducción matemática de que la suma de los primeros n números naturales es $n.(n + 1)/2$: $\Sigma_{i=0,0}\ i = 0.(0 + 1)/2$ (se cumple para el 0), $\Sigma_{i=0,k}\ i = k.(k + 1)/2$ (hipótesis para k), $\Sigma_{i=0,k+1}\ i = \Sigma_{i=0,k}\ i + (k + 1) = k.(k + 1)/2 + (k + 1) = (k + 1).(k + 2)/2$ (se cumple para k + 1).

Otros métodos de prueba. Los métodos de prueba específicos de las áreas de computabilidad y complejidad computacional que utilizamos son la *construcción de algoritmos* (implementados por *máquinas de Turing*), la *diagonalización* y la *reducción*.

En algunos pasajes del libro nos referimos también a los *métodos de prueba axiomáticos* (o *sistemas axiomáticos*), que permiten mecanizar las pruebas produciendo, a partir de fórmulas llamadas *axiomas* y *reglas de inferencia*, otras fórmulas identificadas como *teoremas* (las pruebas son sintácticas en lugar de semánticas). Por ejemplo, la *lógica de predicados* (o *lógica de primer orden*, o *cálculo de predicados*) es un sistema axiomático con fórmulas con constantes, variables, funciones, relaciones, conectivos lógicos y cuantificadores sobre las variables. Y agregándole axiomas *extralógicos* se obtienen distintas *teorías axiomáticas*, como la teoría de conjuntos, la teoría de los números naturales (aritmética) y la teoría de los números reales. Un sistema axiomático es *consistente* si no produce teoremas contradictorios (es decir teoremas y sus negaciones), y es *decidible* si cuenta con un algoritmo para decidir si una fórmula es o no uno de sus teoremas. Por ejemplo, la lógica de predicados es consistente e indecidible.

La relación entre la sintaxis y la semántica en un sistema axiomático se establece con una (función de) *interpretación*, determinando la semántica adoptada para las fórmulas. Por ejemplo, la interpretación (estándar) en la aritmética asigna al símbolo 0 el número 0, al símbolo + la operación de suma, etc. Una interpretación en la que todos los axiomas son verdaderos es un *modelo*. En el marco de una interpretación, un sistema axiomático es *sensato* si todos sus teoremas son verdaderos (naturalmente, un sistema axiomático insensato no tiene razón de ser),

y es *completo* si permite probar todas las fórmulas verdaderas. Por ejemplo, la aritmética, considerando la interpretación estándar, es sensata e incompleta, y la lógica de predicados es sensata y completa para toda interpretación (sus teoremas coinciden con las fórmulas *válidas*).

Fórmulas booleanas

Un lenguaje lógico más simple que el de la lógica de predicados es el de las *fórmulas booleanas*, que se obtienen combinando *variables booleanas* de valor de verdad *verdadero* (abreviado con V) o *falso* (abreviado con F), o alternativamente 1 o 0, respectivamente, con los operadores lógicos *and* ($\wedge$), *or* ($\vee$) y *not* ($\neg$). Por ejemplo, $\varphi = (x_1 \vee x_2) \wedge x_3 \wedge (\neg x_1 \vee \neg x_2)$ es una fórmula booleana.

La semántica de los operadores lógicos es la siguiente: $\varphi = x_1 \wedge x_2$ es verdadera sii x_1 y x_2 son verdaderas, $\varphi = x_1 \vee x_2$ es verdadera sii x_1 o x_2 son verdaderas, y $\varphi = \neg x$ es verdadera sii x es falsa. De estas definiciones se obtiene, razonando inductivamente, la semántica de cualquier fórmula booleana.

Una fórmula booleana φ es *satisfactible* si evaluada con al menos una asignación de valores de verdad a sus variables resulta verdadera, es *insatisfactible* en caso contrario, y es *válida* o es una *tautología* si con toda asignación su evaluación resulta verdadera.

Las fórmulas booleanas también pueden incluir *cuantificadores*, de tipo *existencial* ($\exists$) o *universal* ($\forall$). Por ejemplo, $\theta = (\exists x_1 \forall x_2: x_1 \vee x_2) \vee x_3$ es una fórmula booleana cuantificada. En una fórmula cuantificada, una variable está *libre* si no está alcanzada por ningún cuantificador, y está *ligada* en caso contrario. La fórmula $\theta = \exists x: \varphi$ es verdadera sii φ resulta verdadera con al menos alguno de los dos valores de verdad de x, y la fórmula $\theta = \forall x: \varphi$ es verdadera sii φ resulta verdadera con los dos valores de verdad de x.

Grafos, árboles y circuitos booleanos

Grafos. Un *grafo no dirigido* G es un par (V, E), tal que V es un conjunto finito de *vértices* y E es un conjunto finito de *arcos*, cada uno determinado por un par de vértices, sus *extremos*, que se dicen *adyacentes*. E también se define como una relación simétrica de V x V. Dos arcos son *adyacentes* si comparten un vértice.

Por ejemplo, G = ({1, 2, 3, 4}, {(1, 2), (1,3), (2, 4)}) es un grafo con cuatro vértices y tres arcos, y los vértices 1 y 2 son adyacentes, al igual que los arcos (1, 2) y (1, 3).

A veces los arcos de un grafo pueden tener asociado un número, en cuyo caso el grafo es *ponderado*. La cantidad de vértices adyacentes a un vértice i determina el *grado* de i. Un grafo $G_1 = (V_1, E_1)$ es un *subgrafo* de un grafo $G_2 = (V_2, E_2)$ cuando $V_1 \subseteq V_2$ y $E_1 \subseteq E_2$.

Un *camino* del vértice i al vértice j en un grafo G es una secuencia de vértices i, k_1, …, k_m, j tal que (i, k_1), …, (k_m, j) son arcos de G. Un camino es *simple* si no repite vértices. Si empieza y termina en el mismo vértice, el camino es un *ciclo* o *circuito*, el cual es *simple* si no repite ningún otro vértice. Un grafo G es *conexo* si entre todo par de vértices de G existe un camino, y es *completo*, o es un *clique*, si entre todo par de vértices de G existe un arco.

Además de los grafos no dirigidos también se definen los *grafos dirigidos*, en los que existe un sentido de recorrido en sus arcos (para simplificar la escritura en el libro, en ambos casos denotamos los arcos con pares de vértices). En cada arco se identifica un *vértice inicial* o *cola* y un *vértice final* o *cabeza*. El vértice final es *adyacente* al vértice inicial y el vértice inicial es *incidente* al vértice final. Todo vértice i tiene un *grado de entrada* y un *grado de salida*, que corresponden a la cantidad de arcos que llegan a i (*arcos de entrada*) y salen de i (*arcos de salida*), respectivamente. Dos arcos son *adyacentes* cuando el vértice final de uno coincide con el vértice inicial del otro.

Existen distintas representaciones de grafos. Dos representaciones habituales son la *matriz de adyacencia*, con celdas [i, j] que tienen el valor 1 o 0 según el vértice j sea o no adyacente al vértice i, respectivamente, y la *lista de adyacencia*, que es un vector de listas, una por vértice, conteniendo sus vértices adyacentes. La representación que utilizamos en el libro es más simple, consiste directamente en la enumeración de los vértices (números naturales consecutivos desde el 1) seguida de la enumeración de los arcos (pares de números naturales).

Arboles. Un *árbol* es un grafo conexo sin ciclos. Sus vértices se denominan *nodos*. En los árboles dirigidos se distinguen distintos tipos de nodos. Dado un árbol dirigido T, su nodo *raíz* r es su único nodo con grado de entrada cero. Un *ancestro* de un nodo i ($i \neq r$), es un nodo j intermedio entre r e i; en este caso, i es un *descendiente* de j. Los ancestros inmediatos se denominan *padres*, y los descendientes inmediatos, *hijos*. Todo nodo i ($i \neq r$), tiene un único padre. Los nodos sin hijos son las *hojas* de T (tienen grado de salida cero). La longitud entre r y un nodo i es la *profundidad* de i, y la profundidad de T es la máxima profundidad considerando todos sus nodos.

Circuitos booleanos. Un *circuito booleano* es otro grafo particular, dirigido, conexo, sin ciclos, y con vértices diferenciados. Sus *vértices de entrada* (con grado de entrada cero) representan variables booleanas, que pueden adoptar los valores 1 o 0; sus *vértices internos* (con grado de entrada uno o dos) representan operaciones lógicas (*and* ($\wedge$), *or* ($\vee$) o *not* ($\neg$)); y su *vértice de salida* (con grado de salida cero) también representa una operación lógica, de la que resulta un valor 1 o 0, producto de la evaluación de la fórmula booleana representada por el circuito a partir de los valores de sus variables de entrada.

Teoría de números

El conjunto de los *números enteros* {…, –2, –1, 0, 1, 2, …}, identificado con $\mathcal{Z}$, incluye estrictamente al conjunto $\mathcal{N}$ de los números naturales {0, 1, 2, …}.

La división de un número entero m por un número entero n ≠ 0 se puede expresar con la igualdad m = n.c + r, siendo c y r números enteros, con r entre 0 y n – 1. Si r = 0, se dice que n *divide* a m, o que n es un *divisor* de m, lo que se denota con n|m. Los divisores mayores que cero se denominan *factores*.

Un número entero p > 1 es un número *primo* si sus únicos factores son 1 y p (en caso contrario, se dice que el número es *compuesto*). El Teorema Fundamental de la Aritmética establece que todo número entero m > 1 se puede expresar de una única manera como un producto de números primos, identificado como la *factorización* de m. Se prueba además que para todo número primo p y todo número entero m entre 1 y p – 1 se cumple $m^{p-1} = 1 \pmod p$, y que la cantidad Π(m) de números primos entre 1 y m satisface las igualdades $\Pi(m) = O(m/\log_2 m)$ y $m/\log_2 m = O(\Pi(m))$.

Dados dos números enteros m y n tales que al menos uno no es el cero, su *máximo común divisor* es el mayor número entero d que cumple d|m y d|n, y se denota con mcd(m, n). Cuando se cumple mcd(m ,n) = 1, se dice que m y n son *primos relativos* o *coprimos*.

Teoría de probabilidades

Un *espacio de probabilidad* es un conjunto finito $\Omega = \{\omega_1, \ldots, \omega_n\}$ que modeliza un *experimento aleatorio*. Los elementos ω_i de Ω representan los resultados del experimento. A cada ω_i se le asigna un número real p_i entre 0 y 1, que representa su *probabilidad de ocurrencia*. La suma de los p_i es 1. Si no se especifica ninguna distribución sobre los elementos de Ω, se asume la distribución *uniforme* (es decir, $p_i = 1/n$ para todo i).

Una *variable aleatoria* X es una función definida de un espacio de probabilidad Ω al conjunto $\mathcal{R}$ de los números reales. Un *evento* A, definido sobre un espacio de probabilidad Ω, es un subconjunto de Ω. La probabilidad de ocurrencia de un evento A es la suma de las probabilidades de ocurrencia de sus elementos, y se denota con Pr(A).

Por ejemplo, si se define Ω = {1, 2, 3, 4, 5, 6} para modelizar el lanzamiento de un dado, entonces la probabilidad de ocurrencia de cada resultado de Ω es 1/6 (asumiendo la distribución uniforme), y la del evento de que se obtenga un número menor que 6 es 5/6.

Dos eventos A y B son *independientes* si Pr(A ∩ B) = Pr(A).Pr(B). La definición se puede generalizar a cualquier cantidad de eventos.

En los ejemplos de algoritmos probabilísticos que mostramos en el libro hacemos referencia a la *cota de Chernoff*, cuya utilidad es la siguiente. Dado un conjunto U de objetos, se quiere estimar cuál es la proporción *u* de los que tienen una determinada propiedad (por ejemplo, cuántos profesionales en informática de la ciudad de La Plata estudiaron lógica). El procedimiento natural es muestrear aleatoriamente n objetos de U, calcular la cantidad k de los que tienen la propiedad especificada, y establecer u = k/n. Pero como la estimación depende del tamaño de la muestra (no es igual encuestar a 100 que a 1000 personas), se admite un error Ɛ, con Ɛ > 0. Adicionalmente, como también existen aspectos cualitativos que pueden influir en la estimación (siguiendo con el ejemplo, en qué institución estudió el profesional, durante qué años

hizo la carrera, etc.), se admite un segundo parámetro, correspondiente a la probabilidad δ de que la estimación quede afuera del intervalo [u – Ɛ, u + Ɛ]. Con estas consideraciones, la cota de Chernoff establece que el orden de magnitud de las muestras necesarias para estimar *u* hasta un error Ɛ con probabilidad al menos 1 – δ es $O(\log_2(1/\delta)/\varepsilon^2)$.

Números complejos y álgebra lineal

El conjunto de los *números complejos*, identificado con C, incluye estrictamente al conjunto $\mathcal{R}$ de los números reales.

Un número complejo z es un par (a, b) de números reales, siendo *a* su *parte real* y b su *parte imaginaria*. Otra representación de z es a + i.b, con $i = \sqrt{-1}$.

Dado un número complejo z = a + i.b, la *norma* de z, denotada con |z|, es la raíz cuadrada de $a^2 + b^2$, y el *conjugado* de z, denotado con z*, es a – i.b. Se cumple $z.z^* = |z|^2$.

El producto de una matriz M_1 de m x n por una matriz M_2 de n x k es una matriz M_3 de m x k, tal que $M_3[i, j] = \Sigma_{h=1,n}\ M_1[i, h].M_2[h, j]$. La expresión M^i, con $i \geq 2$, denota el producto de la matriz M por sí misma i – 1 veces.

Dada una matriz M de m x n de números complejos, la matriz *transpuesta conjugada* de M, denotada con M*, es la matriz de n x m tal que M*[i, j] es el conjugado de M[j, i].

Una matriz M de números complejos es *unitaria* si M.M* = I, donde I es la matriz *identidad*, matriz cuadrada de números complejos con todos (1, 0) en su diagonal y todos (0, 0) en sus celdas restantes.

Referencias

Para profundizar en los temas mencionados en este capítulo se puede consultar:

El libro (Mendelson, 2010): cubre la lógica booleana, la lógica de predicados, la aritmética y la teoría de conjuntos. Incluye además un capítulo de computabilidad.

Los capítulos 4 a 6 de (Papadimitriou, 1994): sobre la lógica booleana, la lógica de predicados y la aritmética, con foco en los problemas computacionales que se plantean sobre ellas.

El capítulo 3 de (Martínez y Piñeiro, 2009): trata los sistemas axiomáticos en general y la aritmética en particular.

El libro (Bondy y Murty, 2008): libro introductorio sobre teoría de grafos.

El apéndice de (Arora y Barak, 2009): abarca casi todos los temas que consideramos en este capítulo, incluyendo bastante detalle sobre teoría de números, teoría de probabilidades, números complejos y álgebra lineal.

CAPÍTULO 2
La máquina de Turing

El modelo computacional con el que vamos a estudiar la computabilidad y la complejidad computacional es la *máquina de Turing*. La máquina de Turing (utilizaremos la abreviatura MT) es la modelización más sencilla e intuitiva que se conoce de las computadoras. La definió A. Turing, uno sino el más destacado pionero de las ciencias de la computación, en su famoso artículo de 1936 *Sobre los números computables, con una aplicación al Entscheidungsproblem* (*On computable numbers, with an application to the Entscheidungsproblem*).

A pesar de su simplicidad, a casi un siglo de su creación se mantiene firme la tesis, conocida como Tesis de Church-Turing, de que una máquina de Turing puede simular cualquier dispositivo computacional físicamente realizable. En lo que hace a su eficiencia, en cambio, la situación es menos clara: según la versión *fuerte* de la Tesis de Church-Turing (Tesis Fuerte de Church-Turing), los retardos en las simulaciones de las máquinas de Turing en relación a los tiempos de ejecución de los dispositivos originales no son significativos, pero esto podría no ser así incluyendo entre los dispositivos a las computadoras cuánticas (como comentaremos en el último capítulo).

Vista como una caja negra, en el caso más general una máquina de Turing *resuelve un problema* devolviendo, a partir de toda *instancia* del problema, su *solución*. Si la instancia tiene más de una solución, la máquina devuelve una cualquiera, a menos que se especifique una condición especial (por ejemplo que sea la óptima de acuerdo a la definición del problema), y si no tiene ninguna solución, en el mejor de los casos la máquina responde *no* (en el peor de los casos *no responde nada*, como aclararemos enseguida). Esta es la visión de la máquina de Turing *calculadora*, que resuelve los problemas más generales, conocidos como *problemas de búsqueda (search problems)*. Las instancias y las soluciones se representan por medio de cadenas de símbolos, que identificaremos de manera genérica, respectivamente, como *cadenas de entrada* w y *cadenas de salida* M(w) (o directamente *entradas* w y *salidas* M(w)), siendo M la MT considerada.

Por ejemplo, una máquina de Turing que resuelve el *problema de la accesibilidad en los grafos no dirigidos*, o directamente el *problema de accesibilidad* (recién en el capítulo 8 vamos a considerar grafos dirigidos), a partir de un grafo G y dos vértices i y j de G devuelve, si existe, un camino en G del vértice i al vértice j, y si no existe ningún camino responde *no* (figura 2.1).

Otro ejemplo es el de una máquina de Turing que resuelve el *problema de la satisfactibilidad de las fórmulas booleanas sin cuantificadores,* o directamente el *problema de satisfactibilidad* (recién en el capítulo 8 vamos a utilizar fórmulas booleanas con cuantificadores):

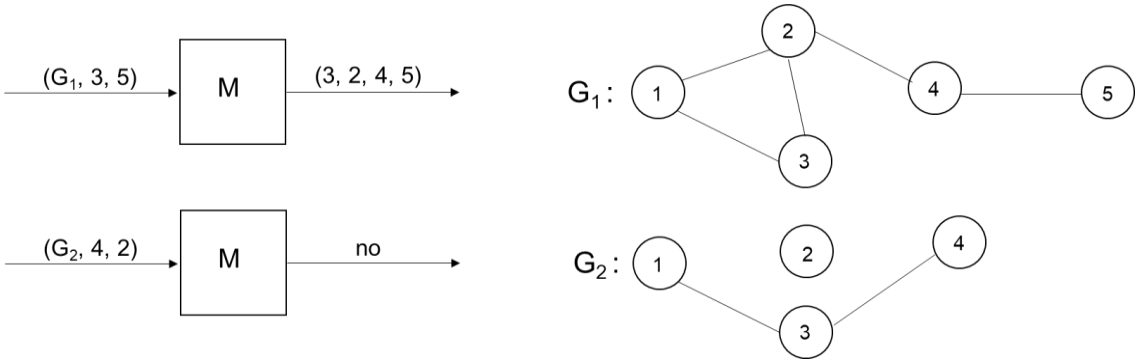

Figura 2.1. Dos resultados de una MT M que resuelve el problema de accesibilidad.

Dada una fórmula booleana φ la máquina devuelve, si existe, una asignación de valores de verdad $\mathcal{A}$ con valores *verdadero* (o V) y *falso* (o F) que la satisface (al aplicarse sobre φ se obtiene el valor *verdadero*), y si no existe, responde *no* (figura 2.2).

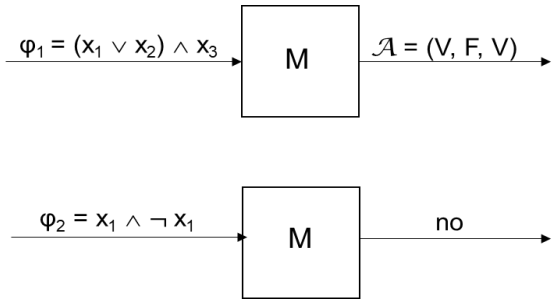

Figura 2.2. Dos resultados de una MT M que resuelve el problema de satisfactibilidad.

Una visión de caja negra más restringida de una máquina de Turing es la de una máquina sólo capaz de *aceptar* o *rechazar* las instancias del problema que resuelve, respondiendo *sí* o *no* (o *nada* en el peor de los casos) según tengan o no solución, respectivamente. En este caso no resuelve un problema de búsqueda sino un *problema de decisión (decision problem)*. Notar que de esta manera la máquina *reconoce* un *lenguaje*, el lenguaje de las cadenas que representan las instancias *positivas* del problema, es decir las instancias que tienen solución. Por eso esta visión de máquina de Turing se conoce como *reconocedora* (reconoce lenguajes que representan problemas).

Por ejemplo, volviendo al problema de accesibilidad pero en su forma de problema de decisión, una máquina de Turing que lo resuelve, a partir de toda terna (G, i, j) responde *sí* o *no* según el grafo G tenga o no un camino del vértice i al vértice j, respectivamente, y el lenguaje que reconoce es el de las cadenas (G, i, j) tales que G tiene un camino del vértice i al vértice j (figura 2.3).

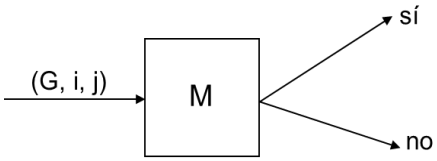

Figura 2.3. Esquema de una MT M que reconoce el lenguaje del problema de accesibilidad.

De la misma manera se puede definir una MT M que reconozca el lenguaje que representa el problema de satisfactibilidad.

Ejercicio. Definir la MT M.

Para estudiar tanto la computabilidad como la complejidad computacional consideraremos los problemas de decisión, y en consecuencia las máquinas de Turing reconocedoras, porque facilitan la presentación de los temas sin que se pierda generalidad con respecto a los aspectos que pretendemos analizar (en el último capítulo trataremos los problemas de búsqueda, y también una tercera clase de problemas, los *problemas de conteo (counting problems)*, que consisten en determinar la cantidad de soluciones de las instancias). De este modo, vamos a emplear indistintamente los términos *problema* y *lenguaje*. Con la expresión L(M) denotaremos el lenguaje que reconoce una MT M.

Para completar esta sección introductoria nos faltan describir dos situaciones que pueden ocurrir en el marco de los problemas y las máquinas de Turing (una ya anticipamos). En efecto, para progresar gradualmente en las definiciones hemos comenzado considerando sólo los *problemas computables decidibles*, que como su nombre lo indica cuentan con máquinas de Turing que los *deciden*, es decir que responden apropiadamente a partir de todas sus instancias, pero desafortunadamente, en lo que hace a la computabilidad, el universo completo de los problemas es más amplio:

Un segundo grupo de problemas es el de los *problemas computables semidecidibles*, que cuentan con máquinas de Turing que responden apropiadamente a partir de todas sus instancias positivas, pero que a partir de algunas de sus instancias negativas, que son las instancias que no tienen solución, *no se detienen*, *entran en un bucle infinito* (*infinite loop*).

Y peor aún, existe un tercer grupo de *problemas no computables*. En este caso, los problemas directamente no cuentan con máquinas de Turing que los resuelvan, es decir que respondan apropiadamente a partir de todas sus instancias positivas.

Desde el capítulo siguiente profundizaremos en los distintos grados de dificultad de los problemas, en el sentido de la computabilidad. En lo que sigue de éste consideramos solamente problemas computables decidibles, y así máquinas de Turing que siempre se detienen, para familiarizarnos con su definición, características y uso.

Estructura y funcionamiento de una máquina de Turing

Una MT M tiene cuatro componentes:

- Una *función de transición*.
- Una *unidad de control*.
- Una *cinta* infinita en ambos extremos dividida en *celdas*.
- Un *cabezal*.

La función de transición, identificada con δ, es el componente de M que especifica el *algoritmo* que ejecuta. En todo momento de la ejecución de M: la unidad de control tiene un *estado*, perteneciente a un conjunto finito de estados Q, entre los que se encuentran el *estado de aceptación* q_A y el *estado de rechazo* q_R; cada celda de la cinta tiene un *símbolo*, perteneciente a un alfabeto finito Γ, que siempre incluye al símbolo blanco B; y el cabezal apunta a alguna celda. Estado, contenido de la cinta y posición del cabezal, conforman en conjunto al comienzo y después de cada paso, una *configuración* de M (figura 2.4).

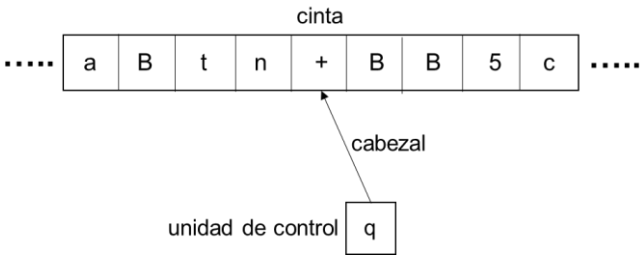

Figura 2.4. Ejemplo de una configuración de una máquina de Turing.

En particular, la *configuración inicial* de M: está formada por el *estado inicial*, comúnmente identificado con q_0; la cinta tiene la entrada w, delimitada por infinitos blancos (w no incluye blancos y puede ser la cadena vacía λ); y el cabezal apunta al primer símbolo de w.

A partir de la configuración inicial, M avanza paso a paso de acuerdo a lo especificado por su función de transición δ. Todo paso consiste en leer un estado, el *estado corriente*, y un símbolo, el *símbolo corriente*, eventualmente modificar uno o ámbos, y moverse un lugar a la derecha o a la izquierda o no moverse. Sólo éstas son las capacidades de M. Las transiciones tienen la forma:

$$\delta(q_i, s) = (q_j, t, d)$$

siendo q_i el estado corriente y s el símbolo corriente, q_j el estado y t el símbolo que reemplazan eventualmente a los anteriores, respectivamente, y d el movimiento del cabezal, que puede ser

R (por *right*) si es a la derecha, L (por *left*) si es a la izquierda, o S (por *stay*) si el cabezal permanece en el mismo lugar.

Por ejemplo, si M tiene la transición $\delta(q_0, a) = (q_1, a, R)$, cuando su estado es q_0 y su cabezal apunta a un símbolo *a*, en el paso siguiente pasa al estado q_1, el símbolo *a* apuntado no se modifica, y su cabezal se mueve un lugar a la derecha (figura 2.5).

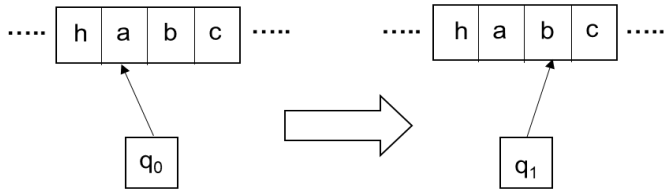

Figura 2.5. Ejemplo de un paso de una máquina de Turing.

La función δ se puede describir también con 5-tuplas y de manera matricial. Cuando y si alcanza el estado q_A o el estado q_R, M se detiene. Se define que si M se detiene en q_A, *acepta* su entrada w, y si se detiene en q_R o no se detiene, *rechaza* w.

Ejemplo 2.1. La siguiente MT M resuelve un *problema de reconocimiento sintáctico de cadenas*. M acepta toda cadena de la forma $a^n b^n$, con $n \geq 1$, es decir que L(M) = {ab, aabb, aaabbb, …}.

La idea general del algoritmo es la siguiente. M lee la primera *a* de la entrada w, la reemplaza por una α y avanza para buscar la primera b. Cuando la encuentra, la reemplaza por una β y retrocede para repetir el ciclo a partir de la nueva primera *a*. Sigue así ciclo tras ciclo hasta que no quedan *a* por procesar. Si nunca encuentra un símbolo distinto de *a* o de b, y logra reemplazar todas las *a* y las b por α y β, respectivamente, acepta w; en caso contrario rechaza w. La figura 2.6 ilustra la idea general del algoritmo, considerando la cadena de entrada w = aaabbb.

aaabbb
αaabbb
αaaβbb
ααaβbb
ααaββb
αααββb
αααβββ

Figura 2.6. Idea general del algoritmo definido para reconocer las cadenas $a^n b^n$, con $n \geq 1$.

Formalmente, M utiliza los siguientes estados (además de q_A y q_R): q_0, el estado inicial, en el que M espera leer una *a*; q_a, en el que M también espera leer una *a* (la necesidad de usar dos

estados para este caso se aclara enseguida); q_b, en el que M avanza para buscar una b; q_L, en el que M retrocede para buscar la siguiente *a*; y q_H, en el que M avanza, habiendo detectado que no quedan *a* por procesar, para chequear que sólo hay β hasta el final. Las transiciones de la función *δ* son (leerlas por filas):

$\delta(q_0, a) = (q_b, \alpha, R)$	$\delta(q_b, a) = (q_b, a, R)$	$\delta(q_b, b) = (q_L, \beta, L)$	$\delta(q_L, a) = (q_L, a, L)$
$\delta(q_L, \alpha) = (q_a, \alpha, R)$	$\delta(q_a, a) = (q_b, \alpha, R)$	$\delta(q_b, \beta) = (q_b, \beta, R)$	$\delta(q_L, \beta) = (q_L, \beta, L)$
$\delta(q_a, \beta) = (q_H, \beta, R)$	$\delta(q_H, \beta) = (q_H, \beta, R)$	$\delta(q_H, B) = (q_A, B, S)$	

Para facilitar la interpretación de δ ordenamos sus transiciones tal como describimos la idea general del algoritmo, considerando primero su primer ciclo y luego los ciclos restantes. Además, para simplificar la escritura no incluimos las transiciones con el estado q_R.

En el estado inicial q_0 sólo se admite leer una *a*, mientras que en el estado q_a también se admite leer una β; por eso utilizamos dos estados cuando se espera una *a*, diferenciando el primer ciclo del resto. La diferencia entre el primer ciclo y los otros también se observa cuando se espera una b o se vuelve para iniciar un nuevo ciclo (estados q_b y q_L, respectivamente): en el primer ciclo sólo se saltean *a*, mientras que en los siguientes también se saltean β.

La prueba de correctitud del algoritmo no reviste mayor dificultad. Se puede utilizar *inducción matemática*, probando que M acepta ab, y que si acepta $a^k b^k$ entonces también acepta $a^{k+1}b^{k+1}$.
Fin del ejemplo

Ejercicio. Modificar la MT M para que también acepte la cadena vacía, y probar por inducción matemática la correctitud del algoritmo modificado.

La máquina de Turing con varias cintas

Existen varios modelos con poder computacional equivalente a la máquina de Turing, en el sentido de que una máquina (o expresión, o función, o programa, etc.) de cada uno de dichos modelos reconoce un lenguaje sii existe una máquina de Turing que reconoce el mismo lenguaje (ver las referencias). A su vez, la propia máquina de Turing contiene distintas variantes computacionalmente equivalentes, como por ejemplo la variante de máquina de Turing con *varias cintas*, que permite especificar algoritmos más fácilmente. La describimos a continuación.

Como antes, la unidad de control es única. La diferencia es que existen varias cintas, cada una apuntada por un cabezal distinto, que se procesan independientemente. Más en detalle: en un solo paso se pueden modificar eventualmente el estado y los símbolos apuntados en cada cinta, y cada cabezal puede moverse de una manera distinta (uno puede moverse a la derecha, otro a la izquierda, etc). Asumiendo K cintas, las transiciones tienen la forma:

$$\delta(q_i, (s_1, \ldots, s_K)) = (q_j, (t_1, d_1), \ldots (t_K, d_K))$$

siendo q_i el estado corriente; s_1, …, s_K los símbolos corrientes; q_j el estado y t_1, …, t_K los símbolos que reemplazan eventualmente a los anteriores, respectivamente; y d_1, …, d_K los movimientos de los cabezales en cada cinta.

De esta manera, ahora una configuración está formada por un estado, los contenidos de las distintas cintas y las posiciones de cada uno de los cabezales (figura 2.7).

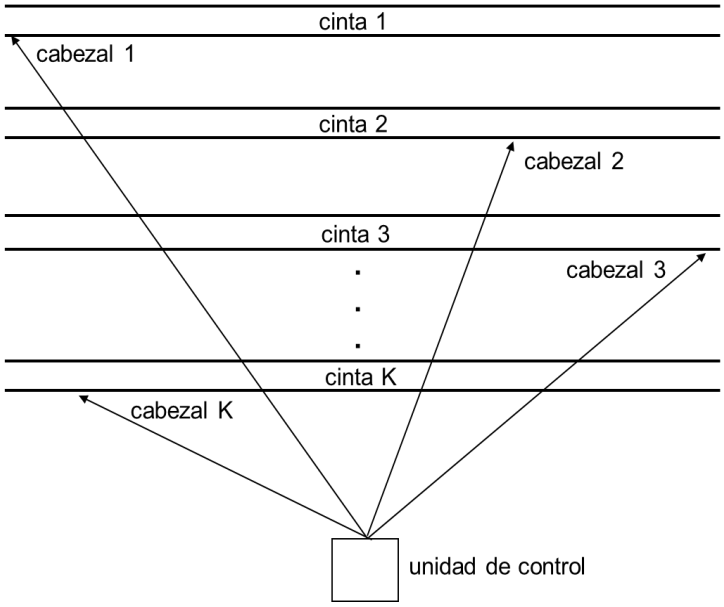

Figura 2.7. Esquema de una configuración de una máquina de Turing con varias cintas.

La entrada se ubica en la primera cinta. Al comienzo, el cabezal de la primera cinta apunta al primer símbolo de la entrada, y las otras cintas tienen blancos. En el caso de que la máquina genere una salida, se designa una cinta para tal fin (*cinta de salida*).

Ejemplo 2.2. Mostramos a continuación una MT M con varias cintas que reconoce el mismo lenguaje del ejemplo anterior: $L(M) = \{a^n b^n \mid n \geq 1\}$.

La idea general del algoritmo es la siguiente. M tiene dos cintas. En la cinta 1 tiene la entrada w. En una primera etapa, M recorre en la cinta 1, de izquierda a derecha, todas las *a* (se detiene cuando encuentra una b u otro símbolo; en este último caso rechaza), al tiempo que en la cinta 2, por cada *a* leída escribe de izquierda a derecha un símbolo X. En una segunda etapa, M recorre en la cinta 1 desde donde quedó antes el cabezal, de izquierda a derecha, todas las b (se detiene cuando encuentra un blanco u otro símbolo; en este último caso rechaza), al tiempo que en la cinta 2, desde donde quedó la última X escrita, por cada b leída retrocede una celda. Si al final de la etapa los dos cabezales apuntan a un blanco (en la cinta 1 al blanco de la derecha de la última b y en la cinta 2 al blanco de la izquierda de la primera X), significa que hay tantas b como X, y así tantas b como *a*, por lo que M acepta w; en caso contrario, M rechaza w.

Formalmente, M utiliza los siguientes estados (además de q_A y q_R): q_a, el estado inicial, en el que M recorre las *a* en la cinta 1 y escribe las X en la cinta 2; y q_b, en el que M recorre las b en la cinta 1 y las X en la cinta 2. Las transiciones de la función δ, descriptas matricialmente, son:

	(a, B)	(b, B)	(b, X)	(B, B)
q_a	(q_a, (a, R), (X, R))	(q_b, (b, S), (B, L))		
q_b			(q_b, (b, R), (X, L))	(q_A, (B, S), (B, S))

Las celdas de la matriz contienen los valores de la función de transición correspondientes a las distintas combinaciones de estados y pares de símbolos. Las celdas vacías representan las transiciones con el estado q_R.

Cuando la MT M pasa al estado q_b, el cabezal de la cinta 1 no se mueve, para quedar apuntando a la primera b encontrada, mientras que el cabezal de la cinta 2 retrocede una celda para posicionarse en la última X escrita.

Tampoco en este caso la prueba de correctitud del algoritmo reviste mayor dificultad.

Fin del ejemplo

Ejercicio. Comparar la cantidad de pasos que hacen las dos máquinas de Turing que describimos para reconocer el lenguaje de las cadenas a^nb^n, con $n \geq 1$, una con una cinta y la otra con dos cintas, tomando como referencia una cadena determinada (más adelante generalizamos y formalizamos esta relación).

Ejemplo 2.3. En este otro ejemplo se aprecia mucho mejor la ventaja de utilizar una máquina de Turing con varias cintas. Consideramos nuevamente el problema de accesibilidad. Vamos a construir una MT M con varias cintas que dada una entrada (G, i, j), compuesta por un grafo G y dos vértices i y j de G, la acepta sii G tiene un camino de i a j.

G es un par (V, E), donde V es el conjunto de sus vértices (por convención de tamaño m), y E es el conjunto de sus arcos, identificados por medio de los dos vértices que los delimitan. La representación que elegimos para G, que utilizaremos de acá en adelante, es la representación de V seguida de la representación de E. Por ejemplo, considerando el primero de los dos grafos mostrados en la figura 2.1, su representación sería:

({1,2,3,4,5},{(1,2),(1,3),(2,3),(2,4),(4,5)})

siendo V = {1, 2, 3, 4, 5} (los vértices se representan con una secuencia de números del 1 al m), y E = {(1, 2), (1,3), (2, 3), (2, 4), (2, 5)} (el orden de los vértices de los arcos es irrelevante porque el grafo es no dirigido). La representación de la terna (G, i, j) se completa con los números que representan los vértices i y j. Los números se codifican en notación binaria, lo que no tiene importancia en esta parte pero sí en la parte de complejidad computacional, así que aprovechamos este ejemplo para dejarlo establecido.

El algoritmo propuesto se basa en el método de *búsqueda en profundidad* o *DFS* (*depth-first search*), que plantea avanzar desde el vértice inicial i en la búsqueda del vértice final j mientras se pueda, y retroceder para buscar otra alternativa cuando sea imposible continuar. M tiene tres cintas. En la cinta 1 tiene la terna de entrada. En la cinta 2 guarda los vértices que van determinando un camino de i a j (la cinta 2 se procesa como una pila). Y en la cinta 3 marca los arcos procesados, para no pasar por ellos más de dos veces, por la ida y la vuelta (cada celda de la cinta 3 se asocia a un arco del grafo de la cinta 1). Dada una entrada w, M hace lo siguiente (por la complejidad de los algoritmos que vamos a presentar en lo sucesivo, recurriremos de ahora en más a una notación algorítmica de alto nivel, con bloques de pasos que agrupan pasos elementales de una máquina de Turing):

1. Valida que la entrada w sea sintácticamente correcta, es decir que tenga la forma (G, i, j). Si no, rechaza.
2. Hace v_1 := i y apila el vértice v_1 en la cinta 2.
3. Busca en la cinta 1 un arco (v_1, v_2) o (v_2, v_1) que no esté marcado en la cinta 3.
   Si encuentra algún arco:
      Lo marca en la cinta 3 y apila el vértice v_2 en la cinta 2.
      Si v_2 = j, acepta.
      Si no, hace v_1 := v_2 y vuelve al bloque 3.
   Si no encuentra ningún arco:
      Desapila el vértice del tope de la cinta 2.
      Si la cinta 2 queda vacía, rechaza.
      Si no, si v_3 es el nuevo vértice del tope de la cinta 2, hace v_1 := v_3 y vuelve al bloque 3.

La implementación de las acciones del algoritmo en términos de transiciones de una máquina de Turing se puede hacer sin mayores inconvenientes. Notar que M no sólo decide si el grafo de entrada tiene un camino del vértice i al vértice j, sino que también, si acepta, devuelve el camino en la cinta 2.

Fin del ejemplo

Probamos a continuación que las máquinas de Turing con varias cintas no tienen más poder computacional que las máquinas de Turing con una cinta:

Teorema 2.1. Una MT M_1 con K > 1 cintas se puede simular con una MT M_2 con una cinta.

Prueba. Sólo describimos la idea general. La simulación se basa en el hecho de que la MT M_2 puede representar las K cintas de la MT M_1 con una cinta de 2K pistas (asumimos, sin perder generalidad, que una cinta de una máquina de Turing tiene varias *pistas*, o lo que es lo mismo, que una celda tiene varias *subceldas*). La idea es que las pistas 1 y 2 de M_2 representen la cinta 1 de M_1, las pistas 3 y 4 de M_2 representen la cinta 2 de M_1, y así sucesivamente, de modo tal

que las pistas impares guarden los contenidos de las cintas, y las pistas pares, en las subceldas correspondientes, las posiciones de los cabezales (con marcas X).

Por ejemplo, si M_1 tiene dos cintas y en algún momento de su ejecución la celda i de su cinta 1 guarda una *a* y está apuntada por el cabezal, y la celda i de su cinta 2 guarda una b y no está apuntada por el cabezal, entonces la celda i de M_2 contendrá la 4-tupla (a, X, b, B). En la figura 2.8 mostramos otro ejemplo, con más detalle.

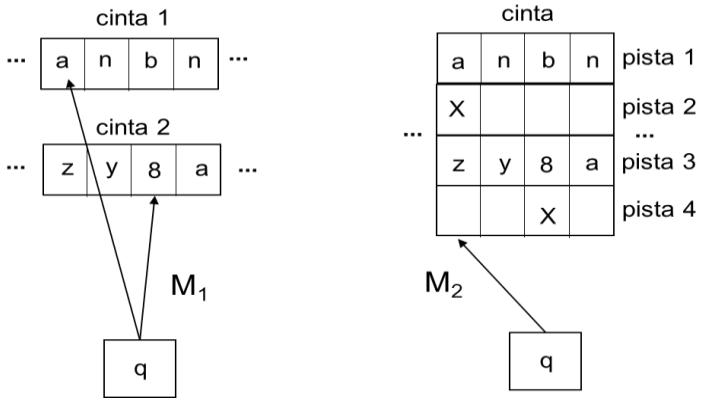

Figura 2.8. Ejemplo de una representación de dos cintas mediante una cinta con cuatro pistas.

La simulación de M_1 por parte de M_2 empieza con la adaptación de la entrada al formato de 2K-tuplas descripto previamente. Luego, M_2 continúa con la simulación de cada paso de M_1 posicionándose en la 2K-tupla con la X más a la izquierda y ejecutando un ciclo de varios pasos. Primero va hacia la derecha, hasta la 2K-tupla con la X más a la derecha, memorizando por medio de sus estados los símbolos con una X y a qué cinta pertenecen. Por ejemplo, si encuentra en el camino un símbolo s con una X correspondiente al cabezal de la cinta j, tendrá un estado que incluya el índice (j, s). Completado el camino de ida hacia la derecha, emprende el camino de vuelta hacia la izquierda, hasta la 2K-tupla con la X más a la izquierda, y se comporta de acuerdo a la función de transición de M_1. Es decir, cuando encuentra una X, considera la información obtenida a la ida (cinta y símbolo), y modifica eventualmente el símbolo y la posición de la X de acuerdo al comportamiento especificado para M_1. Finalmente, cierra el ciclo de pasos modificando su estado según cómo lo modifica M_1, y si en particular pasa a un estado de aceptación o rechazo, acepta o rechaza, respectivamente.

Fin del teorema

De acuerdo a la simulación anterior, como los cabezales de la MT M_1 después de h pasos pueden distanciarse hasta 2h celdas, la MT M_2 puede llegar a ejecutar unos 2**.**2 pasos para simular el primer paso de M_1 (2 de ida y 2 de vuelta), 4**.**2 pasos para simular el segundo paso (4 de ida y 4 de vuelta), 6**.**2 pasos para simular el tercero (6 de ida y 6 de vuelta), y así siguiendo hasta llegar a 2h**.**2 pasos para simular el paso h (2h de ida y 2h de vuelta), lo que da un total de

4.(1 + 2 + … + h) = O(h^2). Es decir que la cantidad de pasos de M_2 puede superar a lo sumo en un orden *cuadrático* a la cantidad de pasos de M_1. Se prueba además que si la simulación se hace con una máquina de Turing con dos cintas en lugar de una, la cota O(h^2) se puede reducir a O($\log_2 h.h$). Estos resultados se tendrán en cuenta en la parte de complejidad computacional.

La máquina de Turing no determinística

Otra variante de máquina de Turing equivalente computacionalmente es la máquina de Turing *no determinística* (o MTN), que también facilita la especificación de algoritmos. Se caracteriza por permitir que haya más de una continuación a partir de un estado q y un símbolo s. Su comportamiento se define por una *relación de transición* Δ, con transiciones de la forma:

$$\Delta(q, s) = \{(q_1, t_1, d_1), \ldots, (q_K, t_K, d_K)\}$$

que expresan que desde el par (q, s), la máquina puede continuar, no determinísticamente, con alguna alternativa (q_1, t_1, d_1), …, (q_K, t_K, d_K). Así, queda establecido un *árbol de computaciones*, cuyo nodo raíz contiene el estado inicial y el primer símbolo de la entrada (figura 2.9).

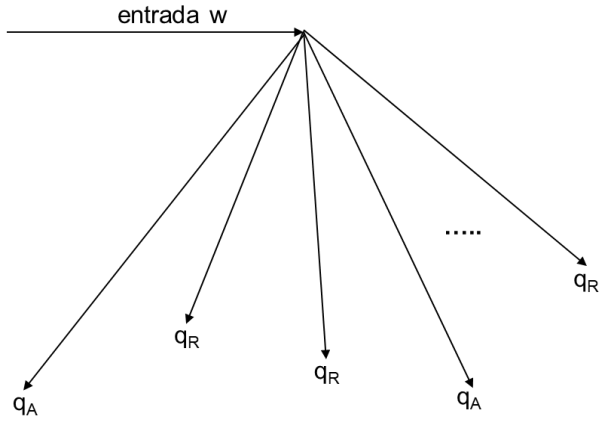

Figura 2.9. Árbol de computaciones de una máquina de Turing no determinística.

Se define que una máquina de Turing no determinística acepta una entrada w sii a partir de w al menos una de sus computaciones se detiene en q_A.

La interpretación natural del comportamiento de esta variante es que ejecuta en paralelo todas sus computaciones, aceptando en caso de que alguna acepte. Por lo tanto, no puede tomarse como un modelo real de una computadora, dada la gran cantidad de procesadores que requeriría en general, imposible de concebir en la práctica. En realidad, una máquina de Turing no determinística debe entenderse como una *abstracción* de una máquina de Turing *determinística*

(o MTD), como una simplificación de la descripción de la ejecución secuencial de las ramas del árbol de computaciones asociado.

Ejemplo 2.4. Volvemos una vez más al problema de accesibilidad. La siguiente MT M, a diferencia de la que construimos en el ejemplo 2.3, reconoce no determinísticamente el lenguaje que representa el problema (la presentamos con menor nivel de detalle). Dada una entrada w, M hace:

1. Valida que la entrada w sea sintácticamente correcta, es decir que tenga la forma (G, i, j). Si no, rechaza.
2. Genera no determinísticamente una secuencia de a lo sumo m vértices entre 1 y m (recordar que definimos la convención de que los grafos tienen m vértices).
3. Acepta (rechaza) si la secuencia generada en el bloque 2 es (no es) un camino en el grafo G del vértice i al vértice j.

La implementación de las acciones definidas, en términos de una relación de transición, se puede efectuar sin mayor dificultad. En el bloque 2 se deben contemplar todas las secuencias posibles de a lo sumo m vértices del grafo G. El bloque 3 consiste en validar si la secuencia de vértices generada en el bloque anterior empieza con el vértice i y termina con el vértice j, y si todo par de vértices consecutivos de la secuencia corresponde a un arco de G.
Fin del ejemplo

Las máquinas de Turing no determinísticas no tienen más poder computacional que las que utilizamos hasta ahora:

Teorema 2.2. Una MTN M_1 se puede simular con una MTD M_2.

Prueba. Si el grado de no determinismo de la MTN M_1 es K, es decir si M_1 tiene a lo sumo K continuaciones a partir de cualquier estado q y cualquier símbolo s, la simulación por parte de la MTD M_2 se basa en representar las computaciones de M_1 con secuencias de números entre 1 y K, denominadas *discriminantes*. Por ejemplo, el discriminante (3,4,1,3,2) representa una computación de M_1 que en el primer paso avanza por su tercera alternativa no determinística, en el segundo paso por su cuarta alternativa, en el tercer paso por la primera, etc. Puede ser que un discriminante no represente una computación (puede contener alternativas que no estén definidas en M_1).

M_2 tiene tres cintas. En la cinta 1 está la entrada. En la cinta 2 genera los discriminantes, ordenados de menor a mayor longitud, y en orden lexicográfico cuando tienen igual longitud. Por ejemplo, los discriminantes correspondientes a las computaciones de hasta dos pasos se ordenan así:

(1), (2), …, (K), (1,1), (1,2), …, (1,K), …, (K,1), (K,2), …, (K,K)

A este orden lo utilizaremos en varios pasajes del libro, y lo identificaremos como *orden canónico*. M_2 lleva a cabo la simulación en la cinta 3, por ciclos. En cada ciclo, copia la entrada de la cinta 1 a la cinta 3 (previo borrado de su contenido, producto de la ejecución anterior), genera en la cinta 2 el siguiente discriminante de acuerdo al orden canónico, y simula M_1 sobre la cinta 3 de acuerdo a las alternativas especificadas por el discriminante (sí éste es válido). Acepta la entrada sii en algún ciclo alcanza el estado de aceptación de M_1.

Como se observa, M_2 recorre el árbol de computaciones de M_1 con el método de *búsqueda a lo ancho* o *BFS* (*breadth-first search*) (ejecuta un paso de cada rama, después dos, después tres, etc., hasta llegar eventualmente a la aceptación), y no con el método de búsqueda en profundidad o DFS, porque M_1 puede tener computaciones que no terminan.

Fin del teorema

A diferencia de la simulación de la sección anterior, la que mostramos recién puede alcanzar una cantidad de pasos exponencialmente mayor que la cantidad de pasos de la máquina simulada: simular h pasos de la MTN M_1, con grado de no determinismo K, puede implicar considerar K^h computaciones de h pasos. Este es otro resultado que utilizaremos en la parte de complejidad computacional.

Ejercicio. La MTD M_2 anterior termina en el estado q_A o no termina. Modificarla de modo tal que termine en q_R cuando pueda hacerlo.

En los ejercicios finales mencionamos otras variantes de máquinas de Turing con poder computacional equivalente. La variante que utilizaremos en los próximos capítulos es la *máquina de Turing determinística con varias cintas* (así, salvo mención explícita, una MT M deberá entenderse como una *MTD M con K cintas*). Físicamente realizable, esta variante constituye un escenario robusto para el estudio de la computabilidad y la complejidad computacional. Cuando recurramos a las máquinas de Turing no determinísticas lo haremos para facilitar la formulación de definiciones o la descripción de algoritmos.

Notas adicionales

Complementamos la descripción de las máquinas de Turing con la introducción de dos variantes *restringidas* de su visión reconocedora, es decir variantes con poder computacional limitado en comparación con el modelo general. Se trata de los *autómatas finitos* y los *autómatas con pila*. Estas variantes permiten reconocer determinados lenguajes de una manera más simple que utilizando máquinas de Turing sin restricciones. También presentamos un modelo

computacional equivalente a la máquina de Turing, la *máquina de acceso aleatorio* o *máquina RAM*, mucho más cercano a una computadora.

El autómata finito

Un *autómata finito* (abreviado con AF) es una máquina de Turing que tiene las siguientes características:

- Tiene una cinta, que es de *sólo lectura.*
- Su cabezal se mueve únicamente a la derecha desde el primer símbolo de la entrada.
- En lugar de los estados finales de aceptación y rechazo q_A y q_R, cuenta con un *conjunto de estados finales* F.
- Cuando su cabezal llega al blanco de la derecha de la entrada se detiene, y la acepta sii el estado alcanzado pertenece al conjunto F.

Una aplicación típica de los autómatas finitos es el análisis lexicográfico que realizan los compiladores. También se emplean para el diseño de procesos de control, circuitos, algoritmos de verificación de software, redes neuronales, procesadores de textos, etc.

Ejemplo 2.5. El siguiente AF M, con estado inicial q_0 y conjunto de estados finales $\{q_0\}$, reconoce el lenguaje de las cadenas de *a* y b con una cantidad par de b, incluyendo la cadena vacía:

$$\delta(q_0, a) = q_0 \qquad \delta(q_0, b) = q_1 \qquad \delta(q_1, a) = q_1 \qquad \delta(q_1, b) = q_0$$

Los estados q_0 y q_1 indican que el AF M lleva reconocidos una cantidad par o impar de b, respectivamente. Por eso, M sólo acepta en el estado q_0.
Fin del ejemplo

Como se refleja en el ejemplo anterior, el único recurso que tiene esta variante de máquina de Turing para memorizar información lo constituyen sus estados. Así por ejemplo, un autómata finito no puede reconocer las cadenas de la forma a^nb^n que tratamos previamente, ya que necesariamente tiene que memorizar la cantidad de *a* que lee para contrastarla luego con la cantidad de b, lo que es imposible utilizando únicamente estados.

Los lenguajes reconocidos por los autómatas finitos se conocen como *regulares*. Existe un lema, el Lema de Bombeo (*Pumping Lemma*), que caracteriza la forma de las cadenas de un lenguaje regular.

El autómata con pila

Un *autómata con pila* (abreviado con AP) es otra variante restringida de máquina de Turing, con más poder computacional que un autómata finito. Sus características son:

- Tiene dos cintas y un cabezal a la cinta 1 en la que guarda la entrada.
- También en este caso la cinta de la entrada es de sólo lectura y su cabezal se mueve únicamente a la derecha.
- La cinta 2 es de lectura/escritura, pero se comporta exclusivamente como una *pila*. Se inicializa con el símbolo especial Z.
- En un paso puede: leer el estado corriente, el símbolo corriente de la cinta 1 y el símbolo del tope de la cinta 2; modificar eventualmente el estado corriente y el símbolo del tope de la cinta 2 (reemplazándolo, desapilándolo o apilando un nuevo símbolo); y moverse un lugar a la derecha en la cinta 1. Pero también puede, alternativamente, omitir la lectura de alguna de las dos cintas y correspondientemente no desplazarse en la cinta 1. Por lo tanto es no determinístico, y se demuestra que lo es intrínsecamente: algunos lenguajes sólo pueden ser reconocidos de este modo.
- Acepta la entrada sii luego de recorrerla completamente, la cinta 2 está vacía (otra alternativa de aceptación es por estado final, como en los autómatas finitos).

Entre las aplicaciones de los autómatas con pila se destaca el análisis sintáctico que llevan a cabo los compiladores.

Ejemplo 2.6. Mostramos que alcanza con un autómata con pila para reconocer el lenguaje de las cadenas $a^n b^n$, con $n \geq 1$, antes reconocido con una máquina de Turing general. El siguiente AP M reconoce el lenguaje. Es determinístico, su estado inicial es q_a y acepta por pila vacía (leer por filas):

$$\delta(q_a, a, Z) = (q_a, \alpha Z) \qquad \delta(q_a, a, \alpha) = (q_a, \alpha\alpha) \qquad \delta(q_a, b, \alpha) = (q_b, \lambda)$$
$$\delta(q_b, b, \alpha) = (q_b, \lambda) \qquad \delta(q_b, B, Z) = (q_b, \lambda)$$

El estado q_a indica que se están leyendo *a*, y el estado q_b, que se están leyendo b. Por cada *a* leída, M apila una α para actualizar la cantidad de *a* recorridas, y así poder validar al final si coincide con la cantidad de b. Cuando encuentra la primera b, M pasa al estado q_b y comienza a desapilar las α, lo que se denota con la cadena vacía. En todos los pasos avanza una celda en la cinta 1. M acepta su entrada sii llega al blanco que le sigue y el tope de la pila es Z.
Fin del ejemplo

La limitación de un autómata con pila de tener que utilizar su cinta de lectura/escritura sólo como una pila se nota, por ejemplo, en el problema de reconocimiento de las cadenas de la forma

$a^n b^n c^n$, Una máquina de Turing general las puede reconocer (ejercicio 2.1.e), pero un autómata con pila no. Intuitivamente, luego de comparar las *a* con las b usando la pila, como mostramos en el ejemplo anterior, se queda sin información para validar si la cantidad de c es la correcta.

Los lenguajes que son reconocidos por los autómatas con pila se denominan *libres de contexto*. Todos los lenguajes de programación habituales en la industria pertenecen a este tipo, para los cuales también existe un Lema de Bombeo que caracteriza la forma de sus cadenas.

Ejercicio. ¿Puede un autómata con pila determinístico reconocer el lenguaje de las cadenas de *a* y b de la forma $w = vv^R$, siendo v^R la cadena *inversa* de v?

La máquina RAM

En esta última nota adicional describimos un modelo computacional equivalente a la máquina de Turing mucho más parecido a una computadora, la *máquina de acceso aleatorio o* máquina *RAM* (*random access machine*).

A diferencia de una máquina de Turing, una máquina RAM permite acceder directamente a su memoria, compuesta por infinitas *palabras*. Las palabras almacenan números enteros, codificados en binario, que representan instrucciones de programa (carga o LOAD, almacenamiento o STORE, suma o ADD, terminación o HALT, condicional, salto incondicional, etc.) o datos, en este orden, de izquierda a derecha. Los primeros dígitos de una instrucción codifican su tipo, y los dígitos restantes constituyen la dirección del operando de la instrucción. La máquina RAM tiene además una cantidad finita de registros, que también almacenan números enteros, siempre codificados en binario. Los registros son R_1, R_2, …, más dos registros especiales, el registro C, que guarda la dirección a la próxima instrucción a ejecutar, y el registro A, que guarda la dirección al próximo dato a procesar.

Mostramos en lo que sigue la equivalencia de los dos modelos computacionales:

Teorema 2.3. Las máquinas de Turing y las máquinas RAM tienen poder computacional equivalente.

Prueba. Desarrollamos solamente la idea general. Utilizamos, sin perder generalidad, máquinas de Turing con una cinta finita a izquierda e infinita a derecha (en el ejercicio 2.4.d pedimos probar la equivalencia computacional entre esta variante y la variante con una cinta infinita en ambos extremos).

Primero mostramos cómo una máquina RAM D puede simular una MT M:

- La simulación consiste en iteraciones. Al comienzo de cada iteración, el registro C apunta al inicio de la secuencia de las instrucciones, las cuales codifican las 5-tuplas de la función de transición de M; el registro A apunta a la palabra que contiene (el código de) el símbolo corriente de M; y el registro R_1 contiene (el código de) el estado corriente de M.

- En cada iteración, D recorre las instrucciones, variando el registro C, hasta encontrar eventualmente la que codifica la 5-tupla de M que empieza con el estado corriente y el símbolo corriente. Si no la encuentra, termina rechazando. Y si la encuentra, por ejemplo la que corresponde a una 5-tupla (q_i, s, q_j, t, R), la ejecuta, quedando el registro R_1 con el estado q_j, la palabra apuntada por el registro A con el símbolo t, y el registro A con su valor anterior incrementado en uno (en el caso de los movimientos L y S, el registro A queda con su valor anterior decrementado en uno o inalterado, respectivamente).
- Si completada una iteración alcanza el estado q_A (q_R) de M, D acepta (rechaza).

Ahora mostramos cómo una MT M puede simular una máquina RAM D:

- M tiene varias cintas. En la cinta 1 guarda las instrucciones y los datos de D. El comienzo de dicha cinta tiene la forma: #0*v_0#1*v_1#2*v_2#3*v_3#…#i*v_i#..., tal que v_i es el contenido de la palabra i de D (i y v_i se codifican en binario). Las otras cintas de M tienen los contenidos de los registros R_1, R_2, …, C y A de D (C está inicializado en cero).
- Considerando el valor i de la cinta correspondiente al registro C, M hace:

1. Recorre la cinta 1 desde el comienzo hasta encontrar la cadena #i*v_i.
2. Procesa v_i. Por ejemplo, si v_i codifica la instrucción ADD TO R_1 j (cuya semántica es sumar al valor de R_1 el valor de la palabra j), entonces:
   2.1. Suma 1 al valor de la cinta correspondiente al registro C.
   2.2. Copia el número j a la cinta correspondiente al registro A.
   2.3. Recorre la cinta 1 desde el comienzo hasta encontrar la cadena #j*v_j. Luego suma el número v_j al valor de la cinta correspondiente al registro R_1.

- M simula así todas las instrucciones de D, considerando la semántica de cada una y siguiendo el orden determinado por el contenido de la cinta correspondiente al registro C, hasta que eventualmente termina.

Fin del teorema

Observaciones finales

- El modelo computacional que utilizaremos para estudiar la computabilidad y la complejidad computacional es la máquina de Turing.
- Según la Tesis de Church-Turing, una máquina de Turing puede simular cualquier modelo computacional físicamente realizable.
- Emplearemos la variante de máquina de Turing determinística con cualquier cantidad de cintas.

- Consideraremos los problemas de decisión, salvo en algunas secciones del último capítulo. En consecuencia, utilizaremos máquinas de Turing que sólo aceptan o rechazan instancias de problemas, y por lo tanto que reconocen lenguajes, los lenguajes de las cadenas que representan las instancias positivas de los problemas correspondientes. De este modo, usaremos de manera indistinta los términos *problema* y *lenguaje*.

Referencias

La máquina de Turing fue introducida por A. Turing en (Turing, 1936), para definir un modelo general de computación. En 1936, la noción de computadora era la de una persona que hacía cálculos con lápiz y papel. La idea de Turing era mostrar que la tarea del calculista podía ser simulada por una máquina, desarrollando los cálculos y escribiendo los resultados en algún lugar. Como punto de partida consideró una cuadrícula donde guardar un símbolo en cada cuadrado, y sin pérdida de generalidad la adaptó a una cinta, tan larga como fuera necesario. El artefacto apenas podía leer y modificar símbolos de la cinta, uno por vez, moverse de a un cuadrado, decidir el paso siguiente a partir del símbolo leído y la configuración interna o estado resultante del cálculo hasta ese momento, y detenerse al finalizar el cálculo. Los conjuntos de símbolos y estados eran finitos, de modo tal que la máquina fuera físicamente realizable.

A partir de la misma época se plantearon otras formalizaciones con poder computacional equivalente, como el *λ-cálculo* (Church, 1936a), las *funciones recursivas parciales* (Kleene, 1936), la *máquina de Post* (Post, 1936), los *sistemas de Post* (Post, 1946) y los *algoritmos de Markov* (Markov, 1954). Pero la máquina de Turing es considerada la modelización más sencilla de todo lo que es computable, pese a lo cual sería capaz de simular todos los modelos computacionales físicamente realizables con poca pérdida de eficiencia (tesis que podrían refutar las computadoras cuánticas).

Las máquinas de Turing no determinísticas surgieron más como una necesidad para analizar la complejidad computacional de los problemas que su computabilidad. Las primeras referencias a las mismas son de finales de la década de 1950 (Rabin y Scott, 1959).

En los capítulos 2 y 5 de (Hopcroft y Ullman, 1979), 1 y 2 de (Sipser, 1997), y 2 y 3 de (Lewis y Papadimitriou, 1998), se describen con mucho detalle los autómatas finitos y los autómatas con pila.

Las máquinas RAM se definieron en (Shepherdson y Sturgis, 1963).

Las máquinas de Turing ejecutan computaciones con números enteros, pero existen algoritmos que se especifican más naturalmente operando con números reales o complejos. Para leer sobre los mismos se puede consultar el capítulo 16 de (Arora y Barak, 2009).

Otras lecturas recomendadas:

(Hodges, 1992), (Herken, 1994) y (Cooper y van Leeuwen, 2013): sobre la vida y obra de Turing.

El capítulo 7 de (Moore y Mertens, 2011): sobre la historia y los pioneros de la teoría de la computación.

Ejercicios

2.1 Construir máquinas de Turing que reconozcan los siguientes lenguajes:

a. El lenguaje de todas las cadenas con los símbolos de un alfabeto Γ.
b. El lenguaje *vacío* ∅. *Comentario: tomar como referencia algún alfabeto Γ*.
c. Un lenguaje *finito.*
d. El lenguaje de las cadenas w, con los símbolos *a* y b, que son *palíndromos*, es decir que cumplen la condición $w = w^R$ (w es igual a su cadena inversa).
e. El lenguaje de las cadenas $a^n b^n c^n$, con $n \geq 0$.
f. El lenguaje de las cadenas x#y#z, tales que x, y, z son números codificados en unario y cumplen z = x + y.

2.2 Construir una máquina de Turing determinística, en notación de alto nivel, que reconozca el lenguaje que representa el problema de satisfactibilidad. *Comentario: asumir que la representación de una fórmula booleana incluye variables x_i, operadores lógicos $\wedge$, $\vee$, ¬, y paréntesis, como por ejemplo $(x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$*.

2.3 Lo mismo que el ejercicio anterior, pero ahora utilizando una máquina de Turing no determinística.

2.4 Probar la equivalencia en poder computacional entre las máquinas de Turing con una cinta infinita en ambos extremos y estados finales q_A y q_R, y las máquinas de Turing con una cinta tales que:

a. Aceptan si se detienen en un estado perteneciente a un conjunto de estados finales.
b. El cabezal se desplaza en todos los pasos, es decir que sólo tienen los movimientos R y L.
c. Si la entrada w no es la cadena vacía, al comienzo no saben a qué símbolo de w apunta el cabezal.
d. La cinta es finita a izquierda e infinita a derecha. *Ayuda: utilizar una cinta con pistas*.

2.5 Probar que toda MTN M_1 tiene una MTN M_2 equivalente de grado de no determinismo 2.

2.6 Construir autómatas finitos (descriptos en una nota adicional) que reconozcan los siguientes lenguajes:

a. El lenguaje de las palabras *if, then, else, fi*.

b. El lenguaje de las cadenas de *a* y b tales que a toda *a* le sigue una b, incluyendo la cadena vacía.
c. El lenguaje de las cadenas de *a* y b con tres *a* consecutivas.

2.7 Construir autómatas con pila (descriptos en una nota adicional) que reconozcan los siguientes lenguajes:

a. El lenguaje de las cadenas $a^n c b^n$, con $n \geq 1$.
b. El lenguaje de las cadenas de *a* y b con igual cantidad de los dos símbolos.
c. El lenguaje de las cadenas $a^n b^m$, con $1 \leq n \leq m$.

CAPÍTULO 3
La jerarquía de la computabilidad

Según su grado de dificultad, en el sentido de la computabilidad, los problemas pueden ser *computables decidibles*, *computables semidecidibles* o *no computables*. También es usual denominar *decidibles* a los primeros e *indecidibles* a los otros.

En este capítulo, con el que iniciamos la parte del libro dedicada a la computabilidad, formalizamos y caracterizamos dicha clasificación, que identificamos como la *jerarquía de la computabilidad*. Precisamos las definiciones de sus clases de problemas, probamos propiedades de cada una y mostramos ejemplos.

Como establecimos en el capítulo anterior, consideramos sólo problemas de decisión, por lo que describimos la jerarquía de la computabilidad en términos de lenguajes (los que representan dichos problemas) y de máquinas de Turing reconocedoras.

Los lenguajes recursivamente enumerables y los lenguajes recursivos

Sea $\mathcal{L}$ el conjunto de todos los lenguajes, representante del conjunto de todos los problemas de decisión. Y sea Σ el alfabeto de los símbolos que integran las cadenas de los lenguajes de $\mathcal{L}$ (toda cadena de un lenguaje de $\mathcal{L}$ pertenece al conjunto Σ*). Para definir la jerarquía de la computabilidad, comenzamos distinguiendo en $\mathcal{L}$ los siguientes dos conjuntos de lenguajes:

- Los lenguajes *recursivamente enumerables*. Un lenguaje L es recursivamente enumerable (L pertenece al conjunto RE) si existe una MT M que lo reconoce (L(M) = L), es decir, si a partir de toda cadena $w \in \Sigma^*$, se cumple que si $w \in L$ entonces M se detiene en q_A, y si $w \notin L$ entonces M se detiene en q_R o no se detiene.
- Los lenguajes *recursivos*. Un lenguaje L es recursivo (L pertenece al conjunto R) si existe una MT M que lo reconoce y se detiene siempre, es decir, si a partir de toda cadena $w \in \Sigma^*$, se cumple que si $w \in L$ entonces M se detiene en q_A, y si $w \notin L$ entonces M se detiene en q_R.

Los lenguajes recursivamente enumerables se denominan así porque tienen la propiedad de que sus cadenas pueden *enumerarse* (profundizamos sobre esto en una nota adicional). Diremos también que una máquina de Turing *decide* un lenguaje recursivo.

En términos de la clasificación mencionada previamente, los lenguajes recursivamente enumerables representan los problemas computables, tanto decidibles como semidecidibles, ya que las máquinas de Turing que los reconocen pueden no detenerse. En cambio, los lenguajes recursivos representan sólo los problemas computables decidibles, porque las máquinas de Turing que los reconocen se detienen siempre. Teniendo en cuenta estas definiciones, en la figura 3.1 mostramos una primera versión de la jerarquía de la computabilidad.

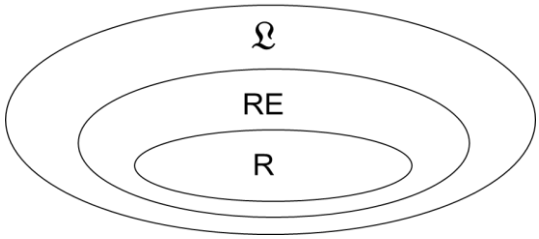

Figura 3.1. Primera versión de la jerarquía de la computabilidad.

Por definición, $R \subseteq RE \subseteq \mathcal{L}$. En el próximo capítulo probaremos que las dos inclusiones son estrictas. En lo que sigue de éste demostramos algunas propiedades básicas de los conjuntos RE y R, y además una relación entre ellos que es central en la jerarquía de la computabilidad.

Propiedades de los lenguajes recursivamente enumerables y los lenguajes recursivos

Nos enfocaremos en propiedades de *clausura*, muy útiles a la hora de caracterizar conjuntos. Empezamos por los lenguajes recursivos:

Teorema 3.1. Si $L \in R$, entonces $L^C \in R$.

Prueba. El teorema establece que R es *cerrado con respecto al complemento* (en todos los casos consideraremos el complemento L^C de un lenguaje L con respecto a Σ^*). La demostración es muy sencilla. Si M es una MT que decide el lenguaje L, entonces existe una MT M^C que decide el lenguaje L^C. Dada una entrada w, M^C hace:

1. Ejecuta M.
2. Si M acepta, entonces rechaza, y si M rechaza, entonces acepta.

La figura 3.2 muestra la máquina construida.

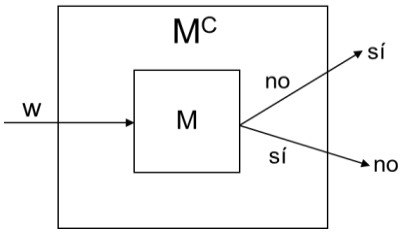

Figura 3.2. La MT M^C decide el complemento del lenguaje que decide la MT M.

M^C decide el lenguaje L^C: se detiene siempre porque M se detiene siempre, y reconoce L^C porque responde al revés que M, que reconoce L.

Fin del teorema

La prueba anterior, por *construcción*, es típica para demostrar la pertenencia de un lenguaje al conjunto R (y también al conjunto RE). Se construye una máquina de Turing. A veces, la construcción de la máquina se hace a partir de otras máquinas que se tienen como hipótesis, en cuyo caso sus funciones de transición se incluyen, adaptadas o no, en la función de transición de la máquina nueva. Por ejemplo, en la demostración que desarrollamos recién, la MT M^C se construye a partir de la MT M permutando sus estados q_A y q_R.

Teorema 3.2. Si $L_1 \in R$ y $L_2 \in R$, entonces $L_1 \cap L_2 \in R$ y $L_1 \cup L_2 \in R$.

Prueba. El teorema establece que R es *cerrado con respecto a la intersección y a la unión*. Las pruebas de estas propiedades también son muy sencillas. Desarrollamos la correspondiente a la intersección. Si M_1 es una MT que decide el lenguaje L_1 y M_2 es una MT que decide el lenguaje L_2, entonces existe una MT M que decide el lenguaje $L_1 \cap L_2$. Dada una entrada w, M hace:

1. Ejecuta M_1.
2. Si M_1 rechaza, entonces rechaza.
3. Ejecuta M_2.
4. Si M_2 acepta, entonces acepta, y si M_2 rechaza, entonces rechaza.

La figura 3.3 muestra la máquina construida.

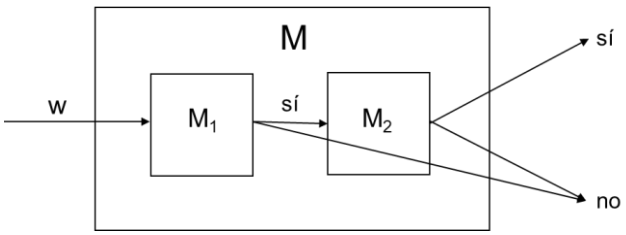

Figura 3.3. La MT M decide la intersección de los lenguajes que deciden las MT M_1 y M_2.

M decide el lenguaje $L_1 \cap L_2$: se detiene siempre porque M_1 y M_2 se detienen siempre, y reconoce $L_1 \cap L_2$ porque acepta una entrada sii M_1 y M_2 la aceptan.

M puede implementarse básicamente con dos cintas. En la cinta 1 tiene la entrada w y en la cinta 2 ejecuta secuencialmente las MT M_1 y M_2 (podemos asumir, sin perder generalidad, que ambas máquinas tienen una cinta). Antes de cada ejecución, copia w de la cinta 1 a la cinta 2, en particular borrando antes de la segunda copia el contenido de la cinta 2.

Ejercicio. Probar que R es cerrado con respecto a la unión. *Ayuda: adaptar la prueba anterior.*
Fin del teorema

Reflexión. Antes de pasar a caracterizar al conjunto RE, cabe recordar que además de reconocer lenguajes, estamos *resolviendo problemas*. En efecto, hay una doble interpretación, a menos que el problema correspondiente sea directamente de reconocimiento sintáctico de cadenas (por ejemplo, el de las cadenas de la forma a^nb^n, con $n \geq 1$, que vimos en el capítulo anterior). Por lo tanto, en este marco las operaciones entre lenguajes deben entenderse también como operaciones entre problemas. En este sentido, acabamos de probar que si un problema es decidible también lo es el problema contrario, y que si dos problemas son decidibles también es decidible el problema común a ambos.

Por ejemplo, como se puede decidir si un grafo G tiene un camino de su vértice i a su vértice j, entonces también se puede decidir si G no lo tiene. Y como se puede decidir, dada una fórmula booleana φ, tanto si tiene una asignación de valores de verdad que la satisface como si tiene una asignación de valores de verdad que no la satisface, entonces también se puede decidir si φ es una fórmula booleana satisfactible no tautológica.
Fin de la reflexión

En las demostraciones con lenguajes recursivamente enumerables hay que tomar mayores recaudos, dado que las máquinas de Turing que intervienen pueden no detenerse.

Lo primero para destacar, porque marca una diferencia fundamental entre los conjuntos RE y R, es que RE *no es cerrado con respecto al complemento*, es decir, existen lenguajes recursivamente enumerables cuyos complementos no lo son. Esta diferencia prueba que RE incluye estrictamente a R, que formalizamos en el capítulo siguiente. Intuitivamente, la construcción que hicimos en la prueba del teorema 3.1 para R no sirve para RE, porque una máquina de Turing que reconoce un lenguaje recursivamente enumerable puede rechazar una cadena no deteniéndose, acción que entonces no puede sustituirse por una aceptación.

En cambio, RE se comporta como R en lo que hace a la intersección y a la unión:

Teorema 3.3. Si $L_1 \in RE$ y $L_2 \in RE$, entonces $L_1 \cap L_2 \in RE$ y $L_1 \cup L_2 \in RE$.

Prueba. El teorema establece que RE es *cerrado con respecto a la intersección y a la unión*. Primero desarrollamos la prueba relacionada con la intersección, muy similar a la que hicimos para R.

Si M_1 es una MT que reconoce el lenguaje L_1 y M_2 es una MT que reconoce el lenguaje L_2, entonces existe una MT M que reconoce el lenguaje $L_1 \cap L_2$. Dada una entrada w, M hace:

1. Ejecuta M_1.
2. Si M_1 rechaza, entonces rechaza (no se detiene si M_1 no se detiene).
3. Ejecuta M_2.
4. Si M_2 acepta, entonces acepta, y si M_2 rechaza entonces rechaza (no se detiene si M_2 no se detiene).

La figura 3.4 muestra la máquina construida.

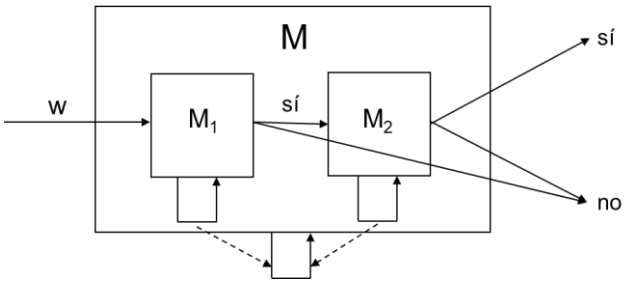

Figura 3.4. La MT M reconoce la intersección de los lenguajes que reconocen las MT M_1 y M_2.

M reconoce el lenguaje $L_1 \cap L_2$: aun existiendo la posibilidad de que M_1 o M_2 entren en un bucle infinito, en cuyo caso M también entra en un bucle infinito, la construcción de M se puede efectuar de la misma manera que en la prueba del teorema anterior, haciendo que acepte una entrada sii M_1 y M_2 la aceptan.

En lo que hace a la prueba correspondiente a la unión, la situación cambia. La MT M que se construya a partir de las MT M_1 y M_2 debe aceptar una cadena si M_1 o M_2 la aceptan, y si éstas se ejecutan secuencialmente puede suceder que una cadena w, aceptada por M_2, produzca un bucle infinito cuando la procesa M_1, por lo que M_2 nunca llegará a procesarla. Así que en este caso se debe recurrir a otra técnica, típica de cuando se utilizan máquinas de Turing que pueden no detenerse. La idea es que M ejecute alternadamente pasos de M_1 y pasos de M_2, o en otras palabras, que M ejecute M_1 y M_2 *en paralelo*. Describimos a continuación dicho esquema. Dada una entrada w, M hace:

1. Hace i := 1.
2. Ejecuta los primeros i pasos de M_1 a partir de w, y luego los primeros i pasos de M_2 a partir de w. Si M_1 o M_2 aceptan, entonces acepta.
3. Hace i := i + 1 y vuelve al bloque 2.

La figura 3.5 muestra la máquina construida.

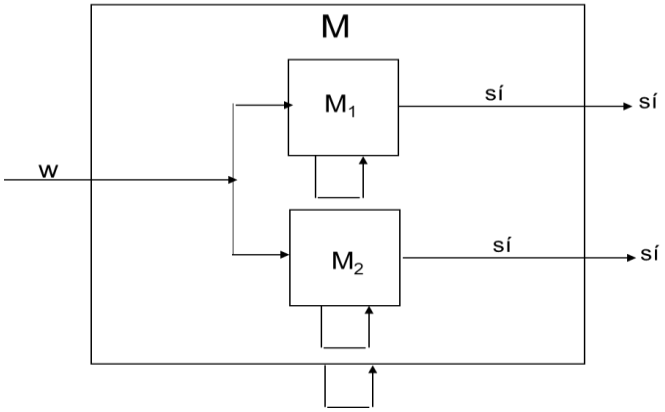

Figura 3.5. La MT M reconoce la unión de los lenguajes que reconocen las MT M_1 y M_2.

M reconoce el lenguaje $L_1 \cup L_2$: M acepta la cadena w sii M_1 o M_2 la aceptan; si la primera (o única) MT que acepta w lo hace al cabo de k pasos, entonces M acepta w en su iteración k-ésima (i = k); y si ninguna de las dos máquinas acepta w, entonces M tampoco la acepta.

M puede implementarse básicamente con cuatro cintas. En la cinta 1 tiene la entrada w, en las cintas 2 y 3 ejecuta M_1 y M_2, respectivamente (podemos asumir sin perder generalidad que las dos máquinas tienen una cinta), y en la cinta 4 mantiene un contador i de pasos. En la primera iteración, antes de ejecutar las máquinas, copia w de la cinta 1 a las cintas 2 y 3 e inicializa i en 1 en la cinta 4. Y en las siguientes iteraciones, antes de las ejecuciones, borra las cintas 2 y 3, copia w en ellas desde la cinta 1 e incrementa i en una unidad en la cinta 4.

Fin del teorema

Ejercicio. La MT M construida en la prueba anterior acepta o no se detiene. El algoritmo se puede optimizar haciendo que M se detenga rechazando si en alguna iteración M_1 y M_2 se detienen rechazando. Otra optimización posible de M, para evitar que ejecute en cada iteración M_1 y M_2 desde el comienzo, es que toda vez ejecute sólo el paso siguiente de M_1 y el paso siguiente de M_2, para lo cual, después de cada ejecución de M_i, debe memorizar cierta información de su configuración. Modificar la MT M aplicando ambas optimizaciones.

Completamos la caracterización de los conjuntos R y RE probando una relación entre ellos que, como anticipamos, resulta central en la jerarquía de la computabilidad. Si CO-RE es el conjunto de los complementos de los lenguajes de RE, es decir, si $L \in RE$ sii $L^C \in$ CO-RE, entonces se demuestra:

$$R = RE \cap \text{CO-RE}$$

Esta relación permite definir el conjunto $\mathcal{L}$ como una partición de cuatro conjuntos de lenguajes con distintos grados de dificultad. Profundizamos sobre esto después de demostrar la relación:

Teorema 3.4. R = RE ∩ CO-RE.

Prueba. Expresado con máquinas de Turing, el teorema establece que existe una MT M que decide un lenguaje L (lo reconoce y se detiene siempre) sii existen dos MT M_1 y M_2 que reconocen L y L^C, respectivamente (que no necesariamente se detienen).

La prueba de que R ⊆ RE ∩ CO-RE se obtiene directamente a partir de las definiciones de R, RE y CO-RE, y de que el conjunto R es cerrado con respecto al complemento.

Ejercicio. Probar que R ⊆ RE ∩ CO-RE.

La prueba de la otra inclusión, RE ∩ CO-RE ⊆ R, es un poco más compleja. Su significado, que ya comentamos, es muy interesante: contando con una MT M_1 que reconoce un lenguaje L y puede no detenerse, y una MT M_2 que reconoce L^C y puede no detenerse, se puede construir una MT M que reconoce L y se detiene siempre. En otras palabras, si existen algoritmos para reconocer un lenguaje L y su complemento, entonces también existe un algoritmo para decidir L.

La prueba consiste en construir M a partir de M_1 y M_2 recurriendo a la técnica de ejecución de máquinas de Turing en paralelo que utilizamos previamente. Dada una cadena w, M hace:

1. Ejecuta en paralelo M_1 y M_2.
2. Si M_1 acepta, entonces acepta, y si M_2 acepta, entonces rechaza.

La figura 3.6 muestra la máquina construida.

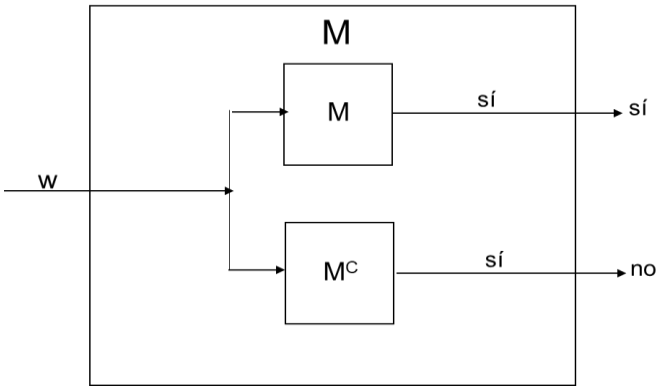

Figura 3.6. La MT M decide el lenguaje L utilizando las MT M_1 y M_2 que reconocen L y L^C.

M reconoce L porque acepta w cuando M_1 acepta w, la cual reconoce L, y rechaza w cuando M_2 acepta w, la cual reconoce L^C. Además, M se detiene siempre, porque toda cadena w pertenece a L o a L^C, y por lo tanto, cualquiera sea w, M_1 o M_2 la aceptan (a diferencia de la prueba de la clausura de RE con respecto a la unión, en esta prueba M_1 y M_2 no son MT

cualesquiera, sino que reconocen un lenguaje y su complemento, lo que asegura que M se detenga siempre).

Fin del teorema

Del teorema 3.4 se obtiene una versión de la jerarquía de la computabilidad más detallada que la anterior, que consideraremos definitiva. Consiste en definir $\mathcal{L}$ como una partición de cuatro conjuntos, que ordenados por el grado de dificultad de sus lenguajes, de menor a mayor, son: R, RE – R, CO-RE – R y $\mathcal{L}$ – (RE ∪ CO-RE). Los mostramos en la figura 3.7.

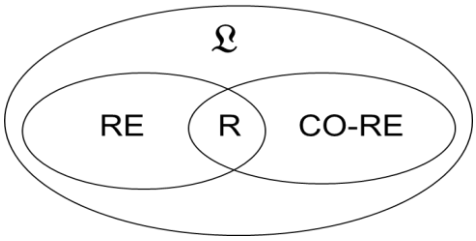

Figura 3.7. Versión definitiva de la jerarquía de la computabilidad.

Las características principales de los cuatro conjuntos se resumen de la siguiente manera:

- *Conjunto R*. Lo constituyen los lenguajes recursivos, los más fácilmente reconocibles de la jerarquía porque cuentan con máquinas de Turing que siempre se detienen. Si un lenguaje L está en R entonces también lo está su complemento L^C.
- *Conjunto RE – R*. Lo constituyen los lenguajes recursivamente enumerables no recursivos, reconocidos por máquinas de Turing que en al menos un caso (negativo) no se detienen. Si un lenguaje L está en RE – R entonces su complemento L^C está en CO-RE – R.
- *Conjunto CO-RE – R*. Lo constituyen los lenguajes no recursivamente enumerables tales que sus complementos son lenguajes recursivamente enumerables no recursivos. No cuentan con máquinas de Turing que los reconozcan. Si un lenguaje L está en CO-RE – R entonces su complemento L^C está en RE – R.
- *Conjunto $\mathcal{L}$ – (RE ∪ CO-RE)*. Lo constituyen los lenguajes de mayor grado de dificultad, porque ni ellos ni sus complementos son recursivamente enumerables. No cuentan con máquinas de Turing que los reconozcan. Si un lenguaje L está en $\mathcal{L}$ – (RE ∪ CO-RE) entonces también lo está su complemento L^C.

De las relaciones R ⊂ RE ⊂ $\mathcal{L}$ y R = RE ∩ CO-RE no se infiere que $\mathcal{L}$ – (RE ∪ CO-RE) tenga lenguajes, porque podría cumplirse $\mathcal{L}$ = RE ∪ CO-RE. De todas maneras esto no sucede. En la siguiente sección mostramos un lenguaje de $\mathcal{L}$ – (RE ∪ CO-RE), y en otro capítulo encontramos formalmente algunos de sus miembros. Si bien tanto sus lenguajes como los de CO-RE – R no

cuentan con máquinas de Turing que los reconozcan, la idea de asignarles a estos últimos un grado de dificultad menor se justifica porque sus complementos sí cuentan con máquinas de Turing, lo cual constituye información algorítmica útil (existe otra justificación, relacionada con las máquinas de Turing restringidas descriptas en las notas adicionales del capítulo anterior, que comentaremos en la sección del capítulo 5 dedicada a las Turing-reducciones).

Ejemplos de lenguajes no recursivos

Completando este capítulo introductorio de la jerarquía de la computabilidad, presentamos a continuación algunos ejemplos de lenguajes no recursivos. Antes de embarcarnos, en los capítulos siguientes, en pruebas y técnicas relacionadas con dichos lenguajes, preferimos empezar con esta primera aproximación, para familiarizarnos con el concepto de no recursividad. Formulamos los lenguajes directamente en términos de los problemas que representan. Primero los definimos y clasificamos, y luego comentamos algunas de sus características:

1. *El problema de la detención.* El problema se refiere a las propias máquinas de Turing. Consiste en determinar, dadas una MT M (codificada de alguna manera) y una entrada w, si M se detiene a partir de w. El problema es *computable semidecidible*, lo que extrapolado al ámbito de la programación significa que en general no se puede decidir si un programa termina.

2. *El problema de la resolución de las ecuaciones diofánticas.* Una ecuación diofántica es una ecuación algebraica con constantes, variables y coeficientes del dominio de los números enteros. El problema consiste en determinar, dada una ecuación diofántica, si tiene solución. Por ejemplo, $3xy = 27$ tiene varias soluciones (como $x = 3$, $y = 3$), y $x^2 = 5$ no tiene ninguna. El problema es *computable semidecidible*.

3. *El problema de decisión en la lógica de predicados.* El problema también se conoce como el *Entscheidungsproblem*. Consiste en determinar, dada una fórmula ψ, si ψ es un teorema de la lógica de predicados, es decir si a partir de los axiomas y las reglas de inferencia de la lógica de predicados se puede derivar ψ. El problema es *computable semidecidible*.

4. *El problema de las palabras para semigrupos.* El problema se define de la siguiente manera. Se parte de un conjunto finito C de palabras (cadenas de letras de un alfabeto específico) y un conjunto finito de reglas de igualdad entre cadenas de letras, y se tiene que determinar, dadas una palabra p_1 de C y una palabra nueva p_2, si según las reglas de igualdad se puede agregar a partir de la palabra p_1 la palabra p_2 a C. Por ejemplo, dadas p_1 = boa y p_2 = bola, y las reglas las = as, aso = a, naso = ron, san = liron y gas = del, se cumple que a partir de p_1 puede agregarse p_2 a C. El problema es *computable semidecidible*. Se lo puede interpretar

como una abstracción del problema de decisión en la lógica de predicados, equiparando las palabras iniciales con los axiomas, las relaciones de igualdad con las reglas de inferencia y las palabras nuevas con los teoremas.

5. *El problema de correspondencia de Post*. El problema consiste en determinar, dado un conjunto finito de pares de cadenas $\{(u_1, v_1), \ldots, (u_k, v_k)\}$, si existe una forma de ordenar los pares, permitiendo repeticiones, de modo tal que la secuencia resultante de concatenar las cadenas u_i sea igual a la secuencia resultante de concatenar las cadenas v_i. Por ejemplo, dado el conjunto {(a, aaa), (abaaa, ab), (ab, b)}, ordenando los pares de la forma (abaaa, ab), (a, aaa), (a, aaa), (ab, b), se obtienen secuencias iguales abaaaaaab. El problema es *computable semidecidible*. Se lo puede interpretar como una abstracción del problema de traducción, equiparando el conjunto de pares de cadenas con un diccionario bilingüe y las secuencias obtenidas con las oraciones con igual significado en los dos idiomas.

6. *El problema de pertenencia al Conjunto de Mandelbrot.* El *Conjunto de Mandelbrot* es un *fractal* (figura geométrica cuya estructura se repite en diferentes escalas). Se define en el dominio de los números complejos con la sucesión $z_0 = 0$ y $z_{n+1} = z_n^2 + c$, tal que c pertenece al conjunto sii la sucesión está *acotada*, es decir si todos sus términos se encuentran entre dos números específicos. El problema consiste en determinar, dado un número c, si c pertenece al Conjunto de Mandelbrot. Por ejemplo, si c = 1, la sucesión obtenida es 0, 1, 2, 5, 26, …, que no está acotada, por lo que c no pertenece al conjunto. En cambio, c = –1 pertenece al conjunto porque se obtiene la sucesión 0, –1, 0, –1, …, que está acotada entre –1 y 0. El problema no es *computable*, y el problema contrario es *computable semidecidible*. Dibujado en el plano, el Conjunto de Mandelbrot revela un contorno muy complejo, que ilustra la dificultad para determinar la ubicación de los puntos del plano cercanos a él (al graficar un problema en el plano, el contorno de la figura correspondiente es muy ilustrativo del grado de dificultad del problema).

7. *El problema de decisión en la aritmética.* El problema consiste en determinar, dada una fórmula ω, si ω es un teorema de la aritmética (la teoría de los números naturales). Por lo tanto, se define como el problema de decisión en la lógica de predicados, salvo que el sistema axiomático no sólo incluye los axiomas lógicos sino también los axiomas de la aritmética. El problema no es *computable*, y el problema contrario tampoco lo es.

Como se puede apreciar de los ejemplos anteriores, los problemas representados por los lenguajes no recursivos no son necesariamente artificiales. Muchos problemas naturales de interés computacional son indecidibles.

Los lenguajes asociados a los primeros cinco problemas pertenecen al conjunto RE – R. Ninguno de ellos cuenta con un algoritmo que lo decida (a menos que se restrinja el dominio de las cadenas, alternativa que comentamos enseguida). No existe un método *analítico general* para

reconocerlos, el único método es la *fuerza bruta*, que consiste en *barrer* el espacio completo de soluciones posibles, el cual al ser infinito puede provocar computaciones infinitas.

Por ejemplo, para determinar si una MT M se detiene a partir de una entrada w, el único método general es ejecutar M a partir de w. Así, si M se detiene a partir de w se puede probar su detención, pero si no, se entra en un bucle infinito. En otras palabras, dada una MT M arbitraria y una entrada w arbitraria, hay que ejecutar un paso de M a partir de w, dos pasos, tres, etc., hasta alcanzar eventualmente su estado de aceptación o rechazo.

En el caso del problema de la resolución de las ecuaciones diofánticas, para determinar si una ecuación diofántica tiene solución hay que probar con todas las combinaciones posibles de números enteros. Así, si existe alguna combinación se la puede encontrar, pero si no, se entra en un bucle infinito. Y lo mismo sucede con el resto de los ejemplos de este tipo.

El lenguaje correspondiente al sexto problema, el problema de pertenencia al Conjunto de Mandelbrot, pertenece al conjunto CO-RE – R. No existe un algoritmo para reconocerlo, pero sí para reconocer su complemento, necesariamente por medio de la fuerza bruta: dado un número complejo c, si c no pertenece al Conjunto de Mandelbrot se puede detectar que la sucesión obtenida con c no está acotada, y así se puede probar que c no pertenece al conjunto, pero si c pertenece al conjunto se entra en un bucle infinito.

Finalmente, el lenguaje que representa el problema de decisión en la aritmética pertenece al conjunto $\mathcal{L}$ – (RE ∪ CO-RE). De este modo, el problema está entre los más difíciles dentro del universo de los problemas, no existen métodos para reconocer todas sus instancias positivas (los enunciados aritméticos verdaderos) ni todas sus instancias negativas (los enunciados aritméticos falsos).

En los siguientes capítulos probaremos la no recursividad de algunos de estos lenguajes, con mayor o menor nivel de detalle. Las demostraciones van a contemplar el *peor caso*, en el sentido de que asumiremos que una máquina de Turing puede recibir cualquier cadena de Σ*. Naturalmente, restringiendo el dominio de las cadenas de entrada, que tiene sentido cuando el interés se centra en un determinado tipo de instancias, la no recursividad puede transformarse en recursividad, al evitarse las cadenas que provocan las computaciones infinitas. Sólo por dar algunos ejemplos, son recursivos los lenguajes representantes de los siguientes problemas: el problema de la detención cuando las máquinas de Turing se mueven en un espacio acotado de celdas, el problema de la resolución de las ecuaciones diofánticas cuando las ecuaciones son de la forma ax + by = c, y el problema de decisión en la lógica de predicados cuando los predicados de las fórmulas son de un solo argumento. Seguiremos considerando los conceptos de peor caso y fuerza bruta en la parte de complejidad computacional.

Notas adicionales

Describimos una tercera visión de máquina de Turing, la máquina de Turing *generadora*, equivalente a la visión reconocedora en el sentido de que existe una máquina de Turing que

reconoce un lenguaje sii existe una máquina de Turing que lo genera. También presentamos sucintamente las *gramáticas*, las cuales constituyen el mecanismo habitual para especificar la sintaxis de los lenguajes y conforman en el caso más general otro modelo computacional equivalente a la máquina de Turing. A partir de ambos modelos establecemos nuevas caracterizaciones de los lenguajes recursivamente enumerables y los lenguajes recursivos.

La máquina de Turing generadora

Otra visión de máquina de Turing, además de las visiones calculadora y reconocedora, es la visión *generadora*. En este caso, la máquina escribe todas las cadenas correspondientes a las instancias positivas del problema que resuelve (todas las ternas formadas por un grafo G y dos vértices i y j tales que G tiene un camino del vértice i al vértice j, todas las fórmulas booleanas satisfactibles, todos los teoremas de la lógica de predicados, etc).

Una MT M generadora escribe las cadenas del lenguaje que genera, denotado con G(M), en una *cinta de salida de sólo escritura* (después de escribir una cadena y un símbolo separador, se mueve un lugar a la derecha). Las cadenas se escriben sin ningún orden específico, y se pueden repetir. La entrada de M es irrelevante porque no interviene en la ejecución.

Probamos a continuación que las visiones generadora y reconocedora son equivalentes:

Teorema 3.5. Un lenguaje es recursivamente enumerable sii existe una máquina de Turing que lo genera.

Prueba. Primero probamos que a partir de una MT M_1 que reconoce un lenguaje, se puede construir una MT M_2 que lo genera:

Dada M_1, M_2 genera una a una todas las cadenas w de Σ^* de menor a mayor longitud y en orden lexicográfico cuando tienen igual longitud (es decir, en el *orden canónico* que definimos en el capítulo anterior), ejecutando M_1 a partir de w luego de cada generación (apropiadamente, porque M_1 puede no detenerse), y escribiendo w en su cinta de salida si M_1 la acepta. Formalmente, M_2 hace:

1. Hace i := 1.
2. Genera todas las cadenas w de Σ^* de longitud a lo sumo i en el orden canónico.
3. Ejecuta a lo sumo i pasos de M_1 a partir de cada w generada en el bloque 2.
4. Escribe en su cinta de salida las w aceptadas en el bloque 3.
5. Hace i := i + 1 y vuelve al bloque 2.

La técnica empleada para evitar los bucles infinitos ya la vimos previamente. En el bloque 3, M_2 ejecuta a lo sumo i pasos de M_1 porque M_1 puede terminar antes. M_2 genera $L(M_1)$ porque todas y sólo las cadenas que M_1 acepta, tarde o temprano M_2 las encuentra y las escribe en su

cinta de salida. Las cadenas escritas no siguen ningún orden, el cual depende de las características de M_1, y se pueden repetir (pero se puede evitar memorizando en una cinta auxiliar lo que se va escribiendo).

Ejercicio. Modificar la MT M_2 para que no escriba cadenas repetidas.

Probamos ahora que a partir de una MT M_1 que genera un lenguaje, se puede construir una MT M_2 que lo reconoce:

M_2 se comporta como M_1, salvo que cuando M_1 escribe una cadena en su cinta de salida, M_2 la compara con su entrada, y si son iguales la acepta. Y si en el transcurso de la ejecución M_1 termina, entonces M_2 rechaza su entrada.

Ejercicio. Detallar la MT M_2.

M_2 reconoce $G(M_1)$ porque si su entrada pertenece a $G(M_1)$, tarde o temprano M_1 la escribe en su cinta de salida y M_2 la acepta, y si su entrada no pertenece a $G(M_1)$, M_1 no la escribe y por lo tanto M_2 no la acepta (terminando sii M_1 termina).

Fin del teorema

La prueba del teorema anterior, adaptándola un poco, sirve para establecer una característica distintiva de los lenguajes recursivos:

Teorema 3.6. Un lenguaje es recursivo sii existe una máquina de Turing que lo genera en el orden canónico.

Prueba. Si M_1 es una MT que decide un lenguaje recursivo, claramente la siguiente MT M_2 genera $L(M_1)$ en el orden canónico: genera en el orden canónico una a una las cadenas w de Σ^*, y luego de cada generación ejecuta M_1 a partir de w y escribe w en su cinta de salida si M_1 la acepta.

Recíprocamente, si M_1 es una MT que genera un lenguaje en el orden canónico, y el lenguaje es infinito, claramente la siguiente MT M_2 decide $G(M_1)$: dada una entrada w, ejecuta M_1; cuando M_1 escribe una cadena v, si v = w, acepta, si v > w en el orden canónico, rechaza, y si v < w en el orden canónico, espera la próxima escritura de M_1.

Ejercicio. Falta probar la implicación recíproca cuando $G(M_1)$ es finito. La construcción anterior no sirve porque M_2 puede no detenerse. Explicar cuándo puede suceder esto, y por qué de todos el teorema sigue valiendo ($G(M_1)$ es recursivo). *Ayuda: ver el ejercicio 3.1.*

Fin del teorema

Las gramáticas

Las gramáticas constituyen el mecanismo habitual para especificar la sintaxis de los lenguajes. Se definen en términos de tres conjuntos finitos, un conjunto V_N de *símbolos no*

terminales, un conjunto V_T de *símbolos terminales* y un conjunto P de reglas, denominadas *producciones*. En el conjunto V_N se distingue un símbolo especial conocido como *axioma*, denotado con S. Las producciones tienen la forma $\alpha \rightarrow \beta$, siendo α y β cadenas de símbolos de V_N y V_T, con $|\alpha| \geq 1$.

Una gramática también puede verse como un generador de un lenguaje, en este caso aplicando producciones desde el axioma S hasta llegar cada vez a una cadena de símbolos terminales, lo que se define como una *derivacion*. La expresión $S \rightarrow^*_G w$ denota una derivación de una gramática G que produce la cadena w.

Ejemplo 3.1. La siguiente gramática $G = (V_N, V_T, P)$ genera el lenguaje constituido por las cadenas de la forma $a^n b^n$, con $n \geq 1$:

$$V_N = \{S\}$$
$$V_T = \{a, b\}$$
$$P = \{1.\ S \rightarrow aSb,\ 2.\ S \rightarrow ab\}$$

S es el axioma de G (en este caso, su único símbolo no terminal). Las cadenas del lenguaje se obtienen aplicando cero o más veces la producción 1 y luego una vez la producción 2. En la figura 3.8 mostramos la derivación $S \rightarrow^*_G$ aaabbb, utilizando un *árbol de derivación*.

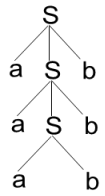

Figura 3.8. Árbol de derivación correspondiente a la derivación $S \rightarrow^*_G$ aaabbb de la gramática G.

Fin del ejemplo

En su forma más general, las gramáticas constituyen otro modelo computacional equivalente a la máquina de Turing. Por lo tanto, otra caracterización de los lenguajes recursivamente enumerables es que cuentan con gramáticas que los generan.

Dos tipos particulares de gramáticas, con restricciones en la forma de sus producciones, son las *gramáticas regulares* y las *gramáticas libres de contexto*, las cuales se relacionan con los autómatas y los lenguajes que describimos en notas adicionales del capítulo anterior de la siguiente manera: las gramáticas regulares tienen poder computacional equivalente a los *autómatas finitos* y generan los *lenguajes regulares*, y las gramáticas libres de contexto tienen poder computacional equivalente a los *autómatas con pila* y generan los *lenguajes libres de contexto*. Las gramáticas regulares son un caso particular de las gramáticas libres de contexto,

y éstas a su vez son un caso particular de otro tipo de gramáticas, las *gramáticas sensibles al contexto*, con producciones de la forma $\alpha \rightarrow \beta$, con $|\alpha| \leq |\beta|$ (como excepción pueden tener también la producción especial $S \rightarrow \lambda$ para contemplar la cadena vacía λ, siempre que S no figure en la parte derecha de ninguna producción).

Debido a la forma de las producciones de las gramáticas sensibles al contexto, se cumple que los lenguajes que generan, los *lenguajes sensibles al contexto*, y por lo tanto también los lenguajes libres de contexto y los lenguajes regulares, son recursivos:

Teorema 3.7. Los lenguajes sensibles al contexto son recursivos.

Prueba. La recursividad de los lenguajes sensibles al contexto proviene del hecho de que la parte derecha de toda producción de una gramática sensible al contexto G tiene tantos o más símbolos que su parte izquierda (salvo la eventual producción especial $S \rightarrow \lambda$), lo que permite decidir, para toda cadena w, si existe una derivación de G que la genera.

Formalmente, dada una cadena w de longitud n, se puede determinar si w pertenece al lenguaje L generado por G del siguiente modo. Se construye un grafo (en este caso particular dirigido), con vértices α_1, α_2, etc., que corresponden a todas las cadenas de símbolos de G, no terminales y terminales, de longitud a lo sumo n, y con arcos (α_i, α_j) si se puede pasar de la cadena α_i a la cadena α_j por la aplicación de alguna producción de G. De esta manera, w estará en L sii existe un camino en el grafo construido desde el vértice asociado al axioma S de G hasta el vértice asociado a la cadena w, lo cual se puede determinar por medio de algún algoritmo que resuelva el problema de accesibilidad (en este caso sobre grafos dirigidos).
Fin del teorema

Observaciones finales

- Los lenguajes recursivamente enumerables (o lenguajes del conjunto RE) son los lenguajes que cuentan con máquinas de Turing que los reconocen. En particular, los lenguajes recursivos (o lenguajes del conjunto R) son los lenguajes que cuentan con máquinas de Turing que los reconocen y se detienen siempre (diremos indistintamente que los deciden).
- La jerarquía de la computabilidad clasifica a los lenguajes (conjunto $\mathcal{L}$), representantes de los problemas de decisión, según su grado de dificultad, en el sentido de la computabilidad. Se define como una partición de cuatro conjuntos: R, RE – R, CO-RE – R y $\mathcal{L}$ – (RE ∪ CO-RE). CO-RE incluye a los complementos de los lenguajes de RE, y los lenguajes de CO-RE – R y de $\mathcal{L}$ – (RE ∪ CO-RE) no cuentan con máquinas de Turing que los reconozcan.
- El camino más natural para probar la pertenencia de un lenguaje al conjunto RE o al conjunto R es construyendo una máquina de Turing que lo reconozca, en el segundo caso deteniéndose siempre.

- Los lenguajes recursivamente enumerables también se caracterizan por contar con máquinas de Turing y gramáticas que los generan.

Referencias

Los contenidos de este capítulo sobre las propiedades de los lenguajes recursivamente enumerables y los lenguajes recursivos se basan fundamentalmente en el capítulo 8 de (Hopcroft y Ullman, 1979).

Varios de los problemas indecidibles que mencionamos están relacionados con el origen de la computación. En el Congreso Internacional de Matemáticos de París de 1900, D. Hilbert, uno de los matemáticos más importantes de fines del siglo XIX y comienzos del siglo XX, planteó 23 problemas que influirían notoriamente en la evolución de las matemáticas. Sólo por nombrar algunos: la Hipótesis del Continuo, la consistencia de la aritmética, la Hipótesis de Riemann, la Conjetura de Goldbach, y la resolución de las ecuaciones diofánticas. Casi tres décadas más tarde, el mismo Hilbert volvió a desafiar a la comunidad matemática con un problema mucho más abarcador, el problema de decisión en la lógica de predicados (el *Entscheidungsproblem*), en el marco de un ambicioso proyecto que lideraba para mecanizar las pruebas matemáticas, con el objeto de asegurar la consistencia de los sistemas axiomáticos en uso sin depender de la intuición. Los *Principia Mathematica* (Whitehead y Russell, 1913), y la prueba de K. Gödel, uno de los lógicos más importantes de todos los tiempos, de la completitud de la lógica de predicados (Gödel, 1929), eran indicios alentadores con respecto a una respuesta positiva al Entscheidungsproblem. Pero la prueba en 1931 del propio Gödel de la incompletitud de cualquier sistema axiomático consistente de los números naturales con suficiente expresividad, enunciado que se conoce como Primer Teorema de Incompletitud (Gödel, 1931), acabó abruptamente con el proyecto: una de las teorías más sencillas de las matemáticas resultaba incompleta. Peor aún, cinco años más tarde, A. Church en (Church, 1936a) y (Church, 1936b), y de manera independiente A. Turing en su artículo de 1936 al que ya nos referimos, probaron la indecibilidad del Entscheidungsproblem. En particular, la prueba de Turing se basaba en la indecibilidad del problema de la detención que él mismo definió. Estos resultados de Church y Turing, además de responder al desafío de Hilbert, clarificaron el concepto de lo computable y se consideran fundacionales en las ciencias de la computación.

Para leer en detalle acerca de las gramáticas, se pueden consultar los capítulos 4 y 9 de (Hopcroft y Ullman, 1979). Otra caracterización de los lenguajes regulares son las *expresiones regulares* (ver por ejemplo el capítulo 1 de (Sipser, 1997) y el capítulo 2 de (Lewis y Papadimitriou, 1998)).

Los conjuntos de lenguajes generados por gramáticas regulares, gramáticas libres de contexto, gramáticas sensibles al contexto y gramáticas sin restricciones, cada uno incluido en el siguiente, conforman una jerarquía de lenguajes alternativa a la que presentamos, conocida como *jerarquía de Chomsky*. Se describe en (Chomsky, 1956) y (Chomsky, 1959). Otra jerarquía

de lenguajes alternativa es la *jerarquía aritmética*, definida por S. Kleene. Dicha jerarquía es una sucesión infinita de conjuntos de lenguajes Σ_0, Σ_1, Σ_2, …, también cada uno incluido en el siguiente, con $\Sigma_0 = R$ y $\Sigma_1 = RE$. Su especificación y propiedades pueden encontrarse en las notas finales del capítulo 3 de (Papadimitriou, 1994).

Otras lecturas recomendadas:

El capítulo 4 de (Penrose, 1989): incluye una sección dedicada a las matemáticas no recursivas, con explicaciones sobre la indecibilidad de los problemas de la resolución de las ecuaciones diofánticas, las palabras para semigrupos y la pertenencia al Conjunto de Mandelbrot, entre otros.

(Chaitín, 2015): analiza la incompletitud, la indecibilidad, y en general los límites del razonamiento matemático, desde el punto de vista de la aleatoriedad y la teoría de la información.

(Martínez y Piñeiro, 2009): plantea la incompletitud de la aritmética en términos de la capacidad de un lenguaje aritmético para definir una operación de concatenación.

Ejercicios

3.1 Probar que todo lenguaje finito es recursivo.

3.2 Probar, utilizando máquinas de Turing no determinísticas, que el conjunto RE es cerrado con respecto a la unión.

3.3 Probar que el conjunto R es cerrado con respecto a la concatenación.

3.4 Probar que el conjunto RE también es cerrado con respecto a la concatenación.

3.5 Probar que si L_1, L_2, …, L_k, forman una partición de Σ^* de lenguajes recursivamente enumerables, entonces cada uno de ellos es recursivo. *Ayuda: considerar la ejecución en paralelo de las máquinas de Turing que reconocen L_1, L_2, …, L_k.*

3.6 Dados dos lenguajes L_1 y L_2:

a. Si $L_1 \subseteq L_2$ y $L_2 \in R$, ¿$L_1 \in R$?
b. Si $L_1 \subseteq L_2$ y $L_2 \in RE$, ¿$L_1 \in RE$?
c. Si $L_1 \in R$ y se le agrega una cantidad finita de cadenas, ¿el lenguaje resultante sigue siendo recursivo?
d. Si $L_1 \in$ CO-RE y $L_2 \in$ CO-RE, ¿$L_1 \cap L_2 \in$ CO-RE?
e. Si $L_1 \cap L_2 \in RE$, ¿$L_1 \in RE$ o $L_2 \in RE$?
f. Si $L_1 \cup L_2 \in RE$, ¿$L_1 \in RE$ o $L_2 \in RE$?

3.7 Explicar cómo una MT M_1 puede determinar si otra MT M_2 acepta al menos una cadena. *Ayuda: M_1 tiene que determinar si M_2 acepta alguna cadena de tamaño i en j pasos.*

3.8 Explicar cómo una MT M_1 puede determinar si otra MT M_2 se detiene, asumiendo que M_2 se mueve en un espacio acotado de celdas. *Ayuda: la cantidad de configuraciones distintas de una máquina de Turing que se mueve en un espacio acotado de celdas es finita.*

3.9 Justificar por qué si existiera una máquina de Turing capaz de reconocer el lenguaje de los enunciados aritméticos verdaderos, dicho lenguaje sería recursivo. *Ayuda: un enunciado aritmético es verdadero o falso*.

3.10 Utilizando las igualdades del ejemplo del capítulo sobre el problema de las palabras para semigrupos, es decir las = as, aso = a, naso = ron, san = liron y gas = del, explicar por qué no se puede obtener la palabra *delatar* a partir de la palabra *gastar*. *Ayuda: en toda igualdad, la suma de las a, r, d, de un miembro coincide con la del otro miembro.*

3.11 Construir una máquina de Turing que genere todas las fórmulas booleanas satisfactibles. *Comentario: se pide construir una máquina de Turing generadora (descripta en una nota adicional). Asumir que ya se cuenta con una máquina de Turing que decide si una cadena es una fórmula booleana y si la misma es satisfactible (los símbolos que integran una fórmula booleana los mostramos en el ejercicio 2.2)*.

3.12 Definir gramáticas (descriptas en una nota adicional) con producciones que tengan la forma $A \rightarrow wB$ y $A \rightarrow w$, siendo A y B símbolos no terminales y w una cadena de símbolos terminales (puede ser la cadena vacía), para generar los siguientes lenguajes regulares (en el ejercicio 2.6 se pide reconocerlos con autómatas finitos):

a. El lenguaje de las palabras *if, then, else, fi*.
b. El lenguaje de las cadenas de *a* y b tales que a toda *a* le sigue una b, incluyendo la cadena vacía.
c. El lenguaje de las cadenas de *a* y b con tres *a* consecutivas.

3.13 Definir gramáticas con producciones que tengan la forma $A \rightarrow \alpha$, siendo A un símbolo no terminal y α una cadena de símbolos no terminales y terminales, para generar los siguientes lenguajes libres de contexto (en el ejercicio 2.7 se pide reconocerlos con autómatas con pila):

a. El lenguaje de las cadenas $a^n c b^n$, con $n \geq 1$.
b. El lenguaje de las cadenas de *a* y b con igual cantidad de ambos símbolos.
c. El lenguaje de las cadenas $a^n b^m$, con $1 \leq n \leq m$.

CAPÍTULO 4
Indecibilidad

Planteado el escenario para estudiar la computabilidad, ya podemos poblar los conjuntos de lenguajes que lo integran. Esta tarea constituye la columna vertebral de nuestro análisis en este capítulo y el siguiente. La tarea no sólo sirve para establecer en qué parte de la jerarquía de la computabilidad se ubican distintos lenguajes de interés computacional y cómo éstos se relacionan entre sí, sino que también ayuda a entender por qué algunos lenguajes son más difíciles que otros. Además, es un buen camino para introducir dos métodos de prueba fundamentales que se utilizan en este marco, la *diagonalización* y la *reducción* (que también utilizaremos en la parte de complejidad computacional).

En realidad, ya poblamos la jerarquía de la computabilidad con varios lenguajes recursivos, construyendo máquinas de Turing, el método de prueba más natural en este caso. Así que vamos a enfocarnos en los lenguajes no recursivos. Si bien mencionamos algunos ejemplos, todavía no comprobamos formalmente la no recursividad de ningún lenguaje, es decir que no demostramos, para ningún lenguaje concreto, que efectivamente no existe una máquina de Turing que lo decida (porque no se detiene siempre, o directamente porque la máquina no existe), lo que naturalmente no puede lograrse construyendo máquinas.

Empezamos encontrando lenguajes no recursivos en los conjuntos RE – R y CO-RE – R, dejando para el capítulo siguiente el conjunto $\mathcal{L}$ – (RE U CO-RE). Utilizamos uno de los métodos de prueba referidos, la diagonalización (el próximo capítulo lo dedicaremos a las reducciones). Como los lenguajes representan problemas sobre máquinas de Turing, primero necesitamos introducir un tipo de máquina de Turing especial y un estándar de codificación de máquinas.

La máquina de Turing universal

Una *máquina de Turing universal* es una máquina de Turing como cualquier otra, pero con la particularidad de que sus entradas representan máquinas de Turing, y su función de transición especifica cómo ejecutarlas (lo cual se ajusta más al funcionamiento de una computadora).

En el caso más general, una máquina de Turing universal tiene como entradas pares de la forma (<M>, w), siendo <M> la codificación de una MT M y w una entrada de M. Dada una MT universal U y una entrada (<M>, w), U ejecuta M a partir de w y responde como M (deteniéndose o no deteniéndose). Como toda MT, U tiene un conjunto de estados, un alfabeto y una función de transición. Esta última especifica cómo ejecutar M a partir de w, lo que básicamente consiste

en: recorrer las 5-tuplas de la función de transición de M hasta encontrar eventualmente una que empiece con el par formado por su estado corriente y su símbolo o símbolos corrientes (M puede tener cualquier cantidad de cintas, por lo que U debe ejecutarla de la manera que vimos en el teorema 2.1); si se encuentra una 5-tupla de estas características, proceder de acuerdo a su descripción; y si no se la encuentra, rechazar. La entrada de U también puede ser sólo un código <M>, sin ninguna cadena w que lo acompañe.

Existen varias maneras de codificar una máquina de Turing. En todos los casos, dado que una MT universal U puede recibir una MT M con un conjunto de estados y un alfabeto de cualquier tamaño, los cuales entonces pueden ser más grandes que los conjuntos correspondientes de la propia MT U, lo que se hace es codificar los estados y símbolos con números enteros. Las características más relevantes de la codificación que elegimos son las siguientes (casi ni vamos a recurrir a ella, lo que nos interesa es que el lector se convenza de que las máquinas de Turing se pueden codificar, y como veremos enseguida, que se pueden identificar unívocamente a través de sus códigos):

- La codificación de los números es la binaria.
- El código del estado inicial es el número 1, el del estado q_A el 2 y el del estado q_R el 3.
- El código del símbolo blanco es el 1.
- Los movimientos R, L y S también se codifican con números: 1, 2 y 3, respectivamente.
- Con las consideraciones anteriores, el código <M> de una MT M consiste en la secuencia de las 5-tuplas que conforman su función de transición, precedida por cualquier cantidad de ceros, y con paréntesis y comas que se disponen de la manera habitual.

Por ejemplo, la cadena:

(1,2,1,3,2),(1,3,1,2,2),(1,1,2,1,3)

con los números en binario, codifica una MT M con una cinta y 5-tuplas con el estado inicial y el estado q_A, el blanco y otros dos símbolos del alfabeto de M, y los movimientos L y S.

Por su parte, consistentemente con lo anterior, los códigos de las entradas w contienen los códigos numéricos de cada uno de sus símbolos, separados por comas (para simplificar la escritura no utilizamos la notación <w> para identificarlos).

Y para separar <M> de w se usa el numeral. Siguiendo con el ejemplo anterior, la cadena:

(1,2,1,3,2),(1,3,1,2,2),(1,1,2,1,3)#2,2,3,3

con los números en binario, incluye <M> y una entrada w con los dos símbolos del alfabeto de M distintos del blanco.

Codificar las máquinas de Turing permite identificarlas unívocamente:

Ejemplo 4.1. Mostramos una MT M que genera en su cinta de salida el i-ésimo código de máquina de Turing, $\langle M_i \rangle$, correspondiente a la MT M_i, según el orden canónico. Dada una entrada i, M hace:

1. Hace n := 0.
2. Crea la siguiente cadena v según el orden canónico.
3. Verifica si v es el código de una máquina de Turing. Si no, vuelve al bloque 2.
4. Si n = i, escribe v en su cinta de salida y termina.
5. Hace n := n + 1.
6. Vuelve al bloque 2.

En el bloque 3, M verifica si la cadena generada en el bloque 2 es un código de máquina de Turing sintácticamente correcto. Cuando en el bloque 4 se cumple n = i, significa que M obtuvo el código $\langle M_i \rangle$.

Fin del ejemplo

Ejercicio. Adaptar la MT M del ejemplo anterior para que genere todos los códigos de máquinas de Turing (en el orden canónico). *Comentario: hay que construir una máquina de Turing con la visión generadora que describimos en una nota adicional del capítulo anterior*.

Primeros lenguajes no recursivos

Ahora sí estamos en condiciones de empezar a poblar la jerarquía de la computabilidad con lenguajes no recursivos. Con el siguiente teorema formalizamos la relación entre los conjuntos de lenguajes R, RE y $\mathcal{L}$, lo cual había quedado pendiente del capítulo anterior. Lo hacemos encontrando un primer lenguaje de RE – R y un primer lenguaje de CO-RE – R:

Teorema 4.1. $R \subset RE \subset \mathcal{L}$.

Prueba. Para probar las inclusiones formuladas definimos la siguiente tabla T de unos y ceros:

T	w_0	w_1	w_2	w_3	…
M_0	1	0	0	1	…
M_1	1	1	0	1	…
M_2	0	1	0	0	…
M_3	0	1	1	0	…
…	…	…	…	…	…

La tabla T representa el comportamiento de todas las máquinas de Turing (identificadas con M_i) a partir de todas las cadenas de Σ^* (identificadas con w_j), enumeradas unas y otras en el orden canónico. $T[M_i, w_j] = 1$ significa que M_i acepta w_j, y $T[M_i, w_j] = 0$, que M_i rechaza w_j (los valores incluidos en T son arbitrarios, no tienen por qué coincidir con la realidad).

De este modo, la fila i de la tabla T representa el lenguaje reconocido por la MT M_i, que podemos especificar así: $L_i = \{w_j \mid M_i \text{ acepta } w_j\}$. Por ejemplo, utilizando los valores que hemos incluido en T, sería $L_0 = \{w_0, w_3, \ldots\}$, $L_1 = \{w_0, w_1, w_3, \ldots\}$, etc. Y como T incluye a todas las máquinas de Turing, entonces todas sus filas en conjunto reúnen a todos los lenguajes reconocidos por máquinas de Turing, es decir a todos los lenguajes recursivamente enumerables, el conjunto RE.

Veamos cómo a partir de esta observación podemos probar que los siguientes dos lenguajes no son recursivos:

1. El lenguaje representado por la diagonal de T. Dicho lenguaje incluye toda cadena w_i aceptada por M_i, que podemos especificar así: $D = \{w_i \mid M_i \text{ acepta } w_i\}$ (D es por *lenguaje diagonal*). Usando los valores incluidos en T, sería $D = \{w_0, w_1, \ldots\}$.
2. El lenguaje representado por la diagonal de T pero cambiando unos por ceros y ceros por unos. Por lo tanto, el lenguaje incluye toda cadena w_i rechazada por M_i, y así no es otro que D^C, es decir el complemento del lenguaje diagonal D: $D^C = \{w_i \mid M_i \text{ rechaza } w_i\}$. Según los valores de T sería $D^C = \{w_2, w_3, \ldots\}$.

Primero vamos a probar que D^C no es recursivamente enumerable, y por lo tanto que se cumple $RE \subset \mathcal{L}$:

Intuitivamente, no parece posible que se pueda construir una máquina de Turing que acepte una cadena w_i sii M_i la rechaza, porque M_i puede no detenerse. Formalmente, se puede comprobar que D^C no es recursivamente enumerable directamente a partir de su definición: D^C difiere de todos los lenguajes L_i representados en T, que son justamente todos los lenguajes recursivamente enumerables:

- D^C difiere de L_0 en la cadena w_0: si $w_0 \in L_0$ entonces $w_0 \notin D^C$ y si $w_0 \notin L_0$ entonces $w_0 \in D^C$.
- D^C difiere de L_1 en la cadena w_1: si $w_1 \in L_1$ entonces $w_1 \notin D^C$ y si $w_1 \notin L_1$ entonces $w_1 \in D^C$.
- D^C difiere de L_2 en la cadena w_2: si $w_2 \in L_2$ entonces $w_2 \notin D^C$ y si $w_2 \notin L_2$ entonces $w_2 \in D^C$.
- Y así siguiendo con todos los lenguajes L_i.

Ahora probaremos que D es recursivamente enumerable pero no recursivo, y por lo tanto que se cumple $R \subset RE$:

También este resultado es inmediato. D no es recursivo porque si lo fuera también lo sería D^C, por la propiedad de clausura del conjunto R con respecto al complemento (teorema 3.1). En este caso, a diferencia de D^C, D es recursivamente enumerable. La siguiente MT U reconoce D. Dada una entrada w, U hace:

1. Calcula el índice i tal que w = w_i, según el orden canónico.
2. Genera el código <M_i> correspondiente a la MT M_i, según el orden canónico.
3. Ejecuta M_i a partir de w_i y responde como M_i.

La MT U reconoce D. En el bloque 1 genera en el orden canónico todas las cadenas hasta encontrar la entrada w, y así obtiene su índice i. El bloque 2 ya lo implementamos en el ejemplo 4.1. Finalmente, en el bloque 3 la MT U procesa el par de entrada (<M_i>, w_i), y al responder como M_i efectivamente reconoce D.

Fin del teorema

Así hemos encontrado formalmente nuestros primeros dos lenguajes no recursivos, el lenguaje D = {w_i | M_i acepta w_i} del conjunto RE – R y el lenguaje D^C = {w_i | M_i rechaza w_i} del conjunto CO-RE – R, probando en consecuencia que R $\subset$ RE $\subset$ $\mathcal{L}$. La figura 4.1 muestra las ubicaciones de D y D^C en la jerarquía de la computabilidad.

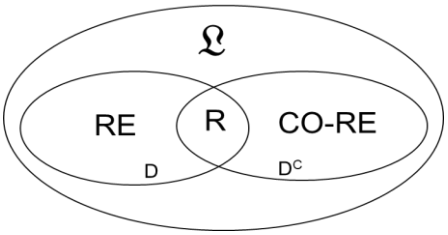

Figura 4.1. El lenguaje diagonal D y su complemento D^C en la jerarquía de la computabilidad.

La diagonalización

La prueba de que el lenguaje D^C = {w_i | M_i rechaza w_i} no es recursivamente enumerable la hicimos por *diagonalización*, método muy efectivo para diferenciar conjuntos, permitiendo encontrar un elemento *separador* (mostramos recién que D^C *separa* el conjunto RE de $\mathcal{L}$ – RE, perteneciendo sólo a este último conjunto).

La diagonalización se basa en un principio muy simple. Expresado en términos de la prueba anterior, establece que en una tabla cuadrada de unos y ceros, si cambiamos los unos por ceros y los ceros por unos de su diagonal, la diagonal resultante difiere de todas sus filas: difiere de la primera fila en su primer elemento, de la segunda fila en su segundo elemento, de la tercera fila en su tercer elemento, y así sucesivamente.

También se pueden usar otros tipos de tablas y se pueden realizar otras modificaciones sobre las diagonales, como mostramos en el siguiente ejemplo, históricamente el primero, presentado por el creador del método, G. Cantor:

Ejemplo 4.2. Cantor creó la diagonalización para demostrar que $|\mathcal{R}|$, la cardinalidad del conjunto de los números reales, es mayor que $|\mathcal{N}|$, la cardinalidad del conjunto de los números naturales Concretamente, demostró que no se pueden enumerar los números reales, o dicho de otra forma, que no se puede definir una biyección entre $\mathcal{N}$ y $\mathcal{R}$, que existe al menos un número real al que no se le puede asociar ningún número natural. Mostramos a continuación una adaptación de la prueba:

Supongamos que la siguiente es una enumeración de todos los números reales entre 0 y 1 (alcanza con considerar solamente este intervalo):

0,1035762718…
0,1432980611…
0,0216609521…
0,4300535777…
0,9255048910…
…….

Sea T la tabla determinada por los decimales de los números reales enumerados, es decir la tabla con las filas 1035762718…, 1432980611…, 0216609521…, etc. Sea d_1 la diagonal de T, es decir la secuencia 14100… Y sea d_2 la secuencia que se obtiene de d_1 reemplazando en ella los dígitos 1 por 2 y los dígitos distintos de 1 por 1, es decir la secuencia 21211…

De esta manera, d_2 difiere de todas las filas de T. Difiere de la primera fila en el primer dígito, de la segunda fila en el segundo dígito, de la tercera fila en el tercer dígito, y así sucesivamente. Pero entonces, el número real con parte entera 0 y parte decimal d_2 no está en la enumeración, lo que es una contradicción, porque supuestamente partimos de una enumeración de todos los números reales entre 0 y 1. Esto significa que tal enumeración (o cualquier otra) es imposible, y en consecuencia, que hay más números reales entre 0 y 1 que números naturales. Generalizando: $|\mathcal{R}| > |\mathcal{N}|$.

Fin del ejemplo

Ejercicio. A diferencia de los números reales, los números enteros y los números fraccionarios se pueden enumerar. Mostrar cómo hacerlo.

La similitud de la prueba de Cantor con la que hicimos antes para probar que el lenguaje D^C no es recursivamente enumerable (teorema 4.1) es evidente. En el caso de la prueba de Cantor, el elemento separador es un número real. Por otra parte, comparar cardinalidades infinitas como las que se consideran en dicha prueba permite obtener relaciones entre los conjuntos de la jerarquía de la computabilidad, como mostramos en una nota adicional.

El problema de la detención

En un capítulo sobre la indecibilidad no podemos dejar de incluir una sección dedicada al *problema de la detención*, que ya mencionamos en el capítulo anterior.

El problema de la detención consiste en determinar, dadas una MT M y una entrada w, si M se detiene a partir de w. El lenguaje que representa el problema se conoce como HP (*halting problem*), y se define de la siguiente manera:

$$HP = \{(<M>, w) \mid M \text{ se detiene a partir de } w\}$$

A. Turing lo definió, y probó su no recursividad por diagonalización (en otra nota adicional mostramos una adaptación de la prueba).

El lenguaje HP es recursivamente enumerable, se puede reconocer mediante una MT M_{HP} que, dada una entrada sintácticamente correcta (<M>, w), ejecuta M a partir de w y acepta sii M se detiene (a las cadenas que no representan pares (<M>, w) las rechaza). En particular, HP está entre los lenguajes recursivamente enumerables más difíciles. En efecto, si HP fuera recursivo, también lo serían todos los lenguajes recursivamente enumerables, es decir que se cumpliría RE = R.

Ejercicio. Probar la última observación. *Ayuda: asumir que existe una MT M_{HP} que decide HP, considerar una MT M_1 que reconoce un lenguaje L cualquiera del conjunto RE, y construir a partir de M_{HP} y M_1 una MT M_2 que decida L*.

Teniendo en cuenta dicha característica de HP, se define que es un lenguaje *RE-completo*, lo que significa que su grado de dificultad, en el sentido de la computabilidad, identifica el grado de dificultad del conjunto RE (en el capítulo anterior mostramos ejemplos que ilustran tal condición de HP, con lenguajes reconocibles únicamente recorriendo todo el espacio de soluciones posibles asociado). Gráficamente, una ubicación apropiada para HP en la jerarquía de la computabilidad sería, entonces, dentro de RE, al lado de su frontera y lo más lejos posible de R, señalando el límite de lo computable (figura 4.2).

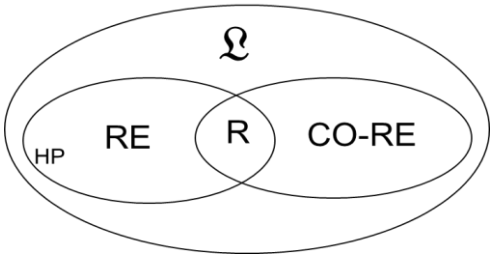

Figura 4.2. El lenguaje HP en la jerarquía de la computabilidad.

En el capítulo siguiente retomaremos el concepto de RE-completitud de HP, en el marco de las reducciones.

Reflexión. A esta altura resulta conveniente repasar qué significa y qué no significa que HP no sea recursivo (que en realidad se puede generalizar a cualquier lenguaje no recursivo).

Que HP no sea recursivo significa que no existe una máquina de Turing que *siempre* pueda decidir, a partir de *cualquier* máquina de Turing M y *cualquier* entrada w, si M se detiene a partir de w. *No* significa que no se pueda decidir la detención de *ciertas* máquinas de Turing a partir de *ciertas* entradas. En otras palabras, la no recursividad de HP determina que no existe un *algoritmo general* para decidir el problema de la detención, pero *nada establece* con respecto a la existencia de *algoritmos particulares* para decidir lenguajes de pares de máquinas de Turing y entradas *particulares*, que de hecho existen.

Llevado al ámbito de la programación, la no recursividad de HP implica que la verificación de la terminación de programas no es en general automatizable, pero no implica que no pueda serlo para cierta clase de programas.

Naturalmente, siempre se puede decidir si *una MT M particular* se detiene a partir de *una entrada w particular*. Este problema tiene *una sola instancia*, o es positiva o es negativa. Resolverlo puede requerir más o menos ingenio, según como sean M y w, pero el algoritmo de decisión que lo resuelve siempre existe.

Fin de la reflexión

A propósito de la última reflexión, en el siguiente ejemplo mostramos un tipo de máquina de Turing, a la que ya nos referimos previamente, cuya detención siempre se puede decidir:

Ejemplo 4.3. En el ejercicio 3.8 pedimos explicar cómo una MT M_1 puede decidir si otra MT M_2 se detiene, asumiendo que M_2 se mueve en un espacio acotado de celdas. En este ejemplo resolvemos un ejercicio similar. Probamos la recursividad del lenguaje L_{20} = {<M> | M es una MT con una cinta, y ejecutada a partir de la entrada vacía su cabezal nunca sale del fragmento de la cinta delimitado por las primeras 20 celdas en el sentido de izquierda a derecha} (figura 4.3):

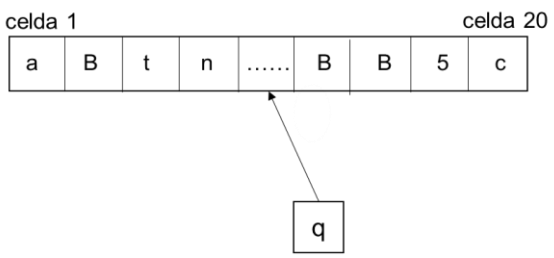

Figura 4.3. El cabezal de una MT que decida el lenguaje L_{20} no debe sobrepasar las celdas 1 a 20.

La idea es que una MT M de las características definidas, con |Q| estados y |Γ| símbolos, en una ejecución como máximo puede atravesar h = 20.|Q|.|Γ|^{20} configuraciones distintas (si atraviesa más quiere decir que entró en un bucle infinito). Por lo tanto, una MT M_{L20} puede decidir el lenguaje L_{20} de la siguiente forma: si la entrada no es un código <M>, rechaza, y si lo es, ejecuta M a partir de la cadena vacía, chequeando con la ayuda de un contador de pasos y un contador de número de celda, luego de cada paso, primero si el cabezal de M salió del fragmento de 20 celdas, en cuyo caso rechaza, y después si M terminó o alcanzó los h pasos, en cuyo caso acepta.

Fin del ejemplo

La máquina de Turing que sólo se mueve a la derecha es otro ejemplo de máquina cuya detención siempre se puede decidir (ejercicio 4.11.b). En los ejercicios finales mencionamos más ejemplos.

La importancia del problema de la detención se aprecia especialmente en las matemáticas. Si fuera decidible, muchos problemas matemáticos se hubieran resuelto o se resolverían con facilidad. Dos ejemplos clásicos en este sentido son el Ultimo Teorema de Fermat, relacionado con el problema de la resolución de las ecuaciones diofánticas, y la Conjetura de Goldbach (mencionado en las referencias del capítulo anterior).

El Ultimo Teorema de Fermat establece que las ecuaciones diofánticas que tienen la forma $x^n + y^n = z^n$, con n > 2, no tienen solución, salvo las soluciones triviales con los valores 0 y 1. El teorema fue formulado por P. de Fermat en 1637, y se demostró recién en 1995.

Con respecto a la Conjetura de Goldbach, fue enunciada por C. Goldbach en 1742, y aún no se comprobó. Establece que todo número natural par mayor que 2 es la suma de dos números primos (números naturales mayores que 1 que tienen como divisores naturales sólo al 1 y a ellos mismos). Por ejemplo: 4 = 2 + 2, 6 = 3 + 3, 8 = 3 + 5, etc.

Ejercicio. Explicar cómo se podrían resolver los dos problemas matemáticos mencionados si el problema de la detención fuera decidible.

Notas adicionales

Presentamos otros tres ejemplos de diagonalización. De la primera diagonalización se deriva la inclusión RE $\subset \mathcal{L}$ de una manera alternativa a la que vimos antes. El segundo ejemplo permite establecer una característica del conjunto R en el marco de la Jerarquía de Chomsky (ver las referencias del capítulo anterior). Y en el tercer ejemplo mostramos, adaptada, la prueba de Turing por diagonalización de que el lenguaje HP no es recursivo.

Diagonalización que prueba que $|2^{\mathbb{N}}| > |\mathbb{N}|$

La diagonalización del ejemplo siguiente permite probar que $|2^{\mathcal{N}}| > |\mathcal{N}|$, es decir que la cardinalidad del conjunto de partes del conjunto de los números naturales $\mathcal{N}$ es mayor que la cardinalidad del conjunto $\mathcal{N}$ (existen más subconjuntos de números naturales que números naturales):

Ejemplo 4.4. Mostramos a continuación que $|2^{\mathcal{N}}| > |\mathcal{N}|$.

Definimos una tabla T, tal que sus filas representan todos los subconjuntos C_i de números naturales ($i \geq 0$), y sus columnas, todos los números naturales n_j ($j \geq 0$), con $T[C_i, n_j] = 1$ si $n_j \in C_i$ y $T[C_i, n_j] = 0$ si $n_j \notin C_i$.

De esta forma, la diagonal de T representa el conjunto $X = \{n_i \mid n_i \in C_i\}$, y si a la misma le cambiamos los unos por ceros y los ceros por unos, obtenemos la representación del conjunto $X^C = \{n_i \mid n_i \notin C_i\}$, que no puede ser ningún subconjunto C_i, porque difiere de C_0 en n_0, de C_1 en n_1, de C_2 en n_2, etc.

Pero entonces llegamos a una contradicción, porque partimos del supuesto de que las filas de T representan el conjunto de todos los subconjuntos de números naturales. Esto quiere decir que el conjunto de partes de $\mathcal{N}$ no puede enumerarse, es decir que $|2^{\mathcal{N}}| > |\mathcal{N}|$.

Fin del ejemplo

A partir de $|2^{\mathcal{N}}| > |\mathcal{N}|$ podemos probar de una manera alternativa a la que mostramos antes que $RE \subset \mathcal{L}$. Por un lado, $|\Sigma^*| = |RE| = |\mathcal{N}|$, porque las cadenas de Σ^* y las máquinas de Turing se pueden enumerar, y así, $|2^{\Sigma^*}| = |2^{\mathcal{N}}|$. Por otro lado, $|2^{\Sigma^*}| = |\mathcal{L}|$, porque 2^{Σ^*} es el conjunto de todos los subconjuntos de cadenas de Σ^*, que es el conjunto $\mathcal{L}$ de todos los lenguajes. De esta manera, $|\mathcal{L}| = |2^{\mathcal{N}}| > |\mathcal{N}| = |RE|$, es decir que hay más lenguajes que lenguajes recursivamente enumerables, o lo que es lo mismo, $RE \subset \mathcal{L}$.

Como se prueba que $|\mathcal{R}| = |2^{\mathcal{N}}|$, de lo anterior también se deriva que la relación que existe entre los tamaños de $\mathcal{L}$ y RE es la misma que existe entre los tamaños de $\mathcal{R}$ y $\mathcal{N}$.

Diagonalización que prueba que hay lenguajes recursivos no sensibles al contexto

En una nota adicional del capítulo anterior definimos las gramáticas sensibles al contexto y probamos que los lenguajes que generan, los lenguajes sensibles al contexto, son recursivos

(teorema 3.7). Con la diagonalización del siguiente ejemplo vamos a probar que existen lenguajes recursivos que no son sensibles al contexto:

Ejemplo 4.5. Probaremos que el lenguaje $L = \{w_i \mid w_i \notin L(G_i)$ y G_i es una gramática sensible al contexto$\}$ es recursivo pero no sensible al contexto.

Consideramos como siempre los índices según el orden canónico, asumimos que contamos con una manera de codificar las gramáticas sensibles al contexto, y denotamos con L(G) al lenguaje generado por una gramática G.

Ejercicio. Proponer una codificación para las gramáticas sensibles al contexto.

Por un lado, se cumple que L es recursivo. La siguiente MT M lo decide:

Dada una entrada w, M determina su índice i tal que $w = w_i$, genera el código de la gramática sensible al contexto G_i, y acepta sii $w_i \notin L(G_i)$.

Por otro lado, se cumple que L no es sensible al contexto:

Suponiendo lo contrario, es decir que $L = L(G_i)$, siendo G_i una gramática sensible al contexto, llegamos a las siguientes contradicciones: si $w_i \in L$, entonces de acuerdo a la definición de L vale $w_i \notin L$, y si $w_i \notin L$, entonces por lo mismo vale $w_i \in L$.

Fin del ejemplo

Contradicciones como las de la prueba anterior son habituales en las diagonalizaciones. Observar, por otra parte, cómo se repite el esquema que desarrollamos en las otras pruebas. En este caso, las filas de la tabla en la que se basa la diagonalización representan todos los lenguajes sensibles al contexto.

Diagonalización que prueba que HP no es recursivo (prueba de Turing)

Mostramos a continuación una adaptación de la prueba de Turing de 1936:

Teorema 4.2. El lenguaje HP no es recursivo.

Prueba. Vamos a suponer que existe una MT M_{HP} que decide el lenguaje HP, y llegaremos a contradicciones del tipo de las que mencionamos recién. Sea M_1 una MT que, dada una entrada w, si w no es el código de una máquina de Turing la rechaza, y si en cambio $w = \langle M_2 \rangle$, hace:

1. Ejecuta M_{HP} a partir de $(\langle M_2 \rangle, \langle M_2 \rangle)$.
2. Si M_{HP} acepta, entonces entra en un bucle infinito.

   Si M_{HP} rechaza, entonces acepta.

La generación de un bucle infinito en la construcción de M_1 no reviste mayor dificultad. Intuitivamente, M_1 *le lleva la contra* a M_{HP} con respecto a cómo responder cuando se ejecuta una máquina de Turing a partir de su propio código. Veamos qué sucede en particular cuando la entrada de M_1 es su propio código $<M_1>$:

- Si M_1 no se detiene a partir de $<M_1>$, entonces por la definición de M_1 significa que M_{HP} acepta $(<M_1>, <M_1>)$. Pero entonces, por la definición de M_{HP}, significa que M_1 se detiene a partir de $<M_1>$ (contradicción).
- Si M_1 se detiene a partir de $<M_1>$, entonces por la definición de M_1 significa que M_{HP} rechaza $(<M_1>, <M_1>)$. Pero entonces, por la definición de M_{HP}, significa que M_1 no se detiene a partir de $<M_1>$ (contradicción).

Por lo tanto, M_1 no puede existir, y en consecuencia tampoco puede existir M_{HP} a partir de la cual M_1 se construyó, lo que quiere decir que el lenguaje HP no es recursivo.
Fin del teorema

La ejecución por parte de la MT M_1 de sí misma en la prueba anterior, su *autorreferencia*, permite llegar al resultado. Más específicamente, el teorema se prueba en base a un doble rol de M_1: hace las veces de máquina de Turing universal y al mismo tiempo de máquina de Turing común.

Observaciones finales

- La máquina de Turing universal se ajusta más que la máquina de Turing común al funcionamiento de una computadora. A los fines prácticos, es muy útil para las pruebas de indecibilidad.
- La construcción de máquinas de Turing no sirve para probar la no pertenencia de un lenguaje a un conjunto de lenguajes de la jerarquía de la computabilidad. Para este caso, los métodos de prueba fundamentales a emplear son la diagonalización y la reducción. En particular, el método de diagonalización es muy útil para obtener un lenguaje que separa dos conjuntos (pertenece sólo a uno de los dos).
- La no recursividad del lenguaje HP, representante del problema de la detención, significa que no existe un algoritmo para decidir si una máquina de Turing arbitraria se detiene a partir de una cadena de entrada arbitraria, lo que no implica que no existan algoritmos de decisión para máquinas de Turing y cadenas de entrada particulares. Esta diferenciación entre algoritmos con instancias arbitrarias e instancias particulares se puede generalizar a todos los lenguajes no recursivos.

Referencias

La máquina de Turing universal se describe en las secciones 6 y 7 del artículo de A. Turing de 1936. En la sección 8 del mismo se encuentra la prueba por diagonalización de la indecibilidad del problema de la detención.

Las pruebas presentadas para encontrar primeros lenguajes en RE – R y CO-RE – R son adaptaciones de las que se desarrollan en el capítulo 8 de (Hopcroft y Ullman, 1979).

A G. Cantor se lo considera una de las figuras más influyentes de las matemáticas. Se lo destaca sobre todo por haber sido el primero en abordar con rigor matemático el concepto del infinito. Su demostración por diagonalización para establecer la relación entre las cardinalidades de los conjuntos $\mathcal{R}$ y $\mathcal{N}$ es de 1892. En (Jourdain, 1955) se reúnen sus resultados más importantes.

Problemas sobre lenguajes, indecidibles con máquinas de Turing y gramáticas generales, pueden ser decidibles con máquinas de Turing y gramáticas particulares, por ejemplo con autómatas finitos y autómatas con pila y con gramáticas regulares y gramáticas libres de contexto (ver por ejemplo el capítulo 4 de (Sipser, 1997)).

El Ultimo Teorema de Fermat tardó más de 350 años para ser resuelto. P. de Fermat lo formuló en 1637, manifestando que tenía una demostración maravillosa del mismo, pero que no podía mostrarla en el libro donde lo presentaba porque *no cabía en el margen*. (Singh, 1997), uno de los libros de divulgación más conocidos de los últimos tiempos, relata cómo el matemático A. Wiles resolvió el teorema en 1993 al cabo de siete años de trabajo, acompañando la narración con un recorrido de la historia de las matemáticas a través de la vida de sus figuras más destacadas.

La autorreferencia es posible cuando se manejan distintos niveles discursivos. Por ejemplo, en la prueba de Turing de la no recursividad del lenguaje correspondiente al problema de la detención, una máquina de Turing actúa en un primer nivel de discurso como una máquina de Turing universal que ejecuta otra máquina de Turing, y en un segundo nivel de discurso como una máquina de Turing común que ejecuta un algoritmo determinado. Otro ejemplo es la demostración de K. Gödel de la incompletitud de la aritmética. En dicha prueba, Gödel utilizó fórmulas que expresan propiedades generales de la aritmética y al mismo tiempo relaciones entre números naturales. (Hofstadter, 1989) es una obra clásica sobre la autorreferencia y sobre los *bucles extraños* (como denomina el autor a las relaciones asociadas a las jerarquías de niveles discursivos).

Otras lecturas recomendadas:

(Piñeiro, 2013): dedicado al trabajo de Cantor sobre el infinito en las matemáticas.

(Goldstein, 2005): trata la vida y la obra de Gödel, incluyendo el análisis de su prueba de incompletitud de la aritmética con especial hincapié en el uso de la autorreferencia.

Ejercicios

4.1 Proponer una codificación para las máquinas de Turing con varias cintas.

4.2 Construir una máquina de Turing que genere los códigos de todas las máquinas de Turing que a partir de la cadena vacía se detienen en a lo sumo 100 pasos.

4.3 Construir una máquina de Turing que genere los códigos de todas las MT M_i que aceptan las cadenas w_i, considerando el orden canónico.

4.4 En el ejercicio 3.7 se pide explicar cómo una MT M_1 puede determinar si otra MT M_2 acepta al menos una cadena. Se trata del problema de decidir si una máquina de Turing reconoce un lenguaje $L \neq \emptyset$. Especificar el problema mediante un lenguaje y comprobar formalmente que el lenguaje es recursivamente enumerable.

4.5 Describir la tabla que se utiliza como base en la diagonalización del ejemplo 4.5, con la que se prueba que existen más lenguajes recursivos que lenguajes sensibles al contexto. Indicar qué representan las filas, las columnas, la diagonal y la diagonal modificada.

4.6 Lo mismo que el ejercicio anterior, pero con respecto a la prueba de Turing, por diagonalización, de la no recursividad del lenguaje HP (teorema 4.2).

4.7 Hemos probado, utilizando cardinalidades de conjuntos infinitos, que el conjunto $\mathcal{L}$ de todos los lenguajes no se puede enumerar (ejemplo 4.4). Probar lo mismo pero por el método de diagonalización. *Ayuda: sirve el esquema que utilizamos para demostrar que* $D^C \notin RE$.

4.8 Probar por diagonalización que $|2^X| > |X|$, para todo conjunto infinito X. *Ayuda: basarse en la prueba de que* $|2^{\mathcal{N}}| > |\mathcal{N}|$.

4.9 ¿Cómo se puede justificar que el conjunto $\mathcal{L}$ – (RE U CO-RE) no se puede enumerar?

4.10 Sea la función $f_{HP} : \Sigma^* \longrightarrow \{0, 1\}$ tal que:

$f_{HP}(v) = 1$, si v = (<M>, w) y M se detiene partir de w.
$f_{HP}(v) = 0$, si v = (<M>, w) y M no se detiene a partir de w o bien $v \neq$ (<M>, w).

Probar que no puede existir una máquina de Turing que calcule la función f_{HP} para toda cadena de Σ^*.

4.11 Probar que los siguientes lenguajes son recursivos (para simplificar, asumir que las MT M codificadas en los lenguajes tienen una cinta):

a. L = {<M> | M a partir de la cadena vacía escribe alguna vez un símbolo no blanco}.
b. L = {(<M>, w) | M se mueve siempre a la derecha y acepta la cadena w}.
c. L = {(<M>, w) | M nunca se mueve a la izquierda y acepta la cadena w}.
d. L = {(<M>, w) | el cabezal de M se mueve sólo a lo largo de las celdas que ocupa w}.

4.12 Justificar por qué se puede decidir, dada una MT M, si existe una entrada a partir de la cual M se detiene en a lo sumo 100 pasos. *Ayuda: ¿Hasta qué tamaño de cadenas hay que chequear?*

4.13 ¿Se puede decidir, dada una MT M, si existe una entrada de a lo sumo 100 símbolos a partir de la cual M se detiene?

CAPÍTULO 5
Las reducciones

Con este capítulo completamos la parte de computabilidad del libro. Describimos las *reducciones*, el método para caracterizar y relacionar lenguajes que nos falta tratar.

La noción de reducción es muy conocida en la algorítmica. La idea es, en vez de resolver de cero un problema nuevo, hacerlo con la ayuda de la resolución de un problema conocido (que en la programación se implementa con invocaciones a una subrutina), estando el problema conocido relacionado de algún modo con el problema nuevo. Existen dos clases de reducciones, las *m-reducciones* y las *Turing-reducciones*. Las Turing-reducciones son una generalización de las m-reducciones, pero las dos son necesarias, y se utilizan según el propósito buscado.

Las m-reducciones

Comenzamos describiendo las *m-reducciones*. Una m-reducción de un lenguaje L_1 a un lenguaje L_2 es una función $f : \Sigma^* \longrightarrow \Sigma^*$ que satisface dos condiciones, una condición de *computabilidad* y una condición de *correctitud*:

- Condición de computabilidad: existe una MT M_f que la computa. Es decir, para toda $w \in \Sigma^*$, $f(w) = M_f(w) \in \Sigma^*$. Por eso se dice que es *total computable* (o directamente *computable*).
- Condición de correctitud: asigna a toda cadena w que pertenece al lenguaje L_1 una cadena f(w) que pertenece al lenguaje L_2, y a toda cadena w que no pertenece a L_1 una cadena f(w) que no pertenece a L_2.

Las m-reducciones también se conocen como *funciones de reducción*. La figura 5.1 representa una m-reducción f, computada por una MT M_f:

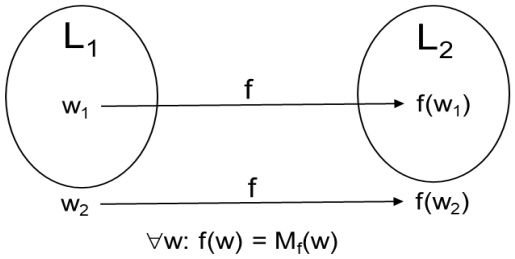

Figura 5.1. Representación de una m-reducción de un lenguaje L_1 a un lenguaje L_2.

La m de m-reducción proviene de *muchos a uno*, dado que en general no es una función inyectiva. Aparte de ser computable, lo único que se le exige con respecto a los lenguajes L_1 y L_2 que relaciona, entonces, es que asigne a toda cadena de L_1 una cadena de L_2 y a toda cadena de L_1^C una cadena de L_2^C. Intuitivamente, al ser dicha asignación de cadenas computable por una MT M_f, contando con una MT M_2 que reconozca L_2 se puede construir una MT M_1 que reconozca L_1, componiendo secuencialmente M_f con M_2 (M_1 reconoce L_1 recurriendo a M_2). Enseguida formalizamos esta idea.

Con $L_1 \leq_m L_2$ expresaremos que existe una m-reducción de L_1 a L_2. Vamos a utilizar de manera indistinta las expresiones $M_f(w)$ y f(w), y para simplificar, vamos a emplear directamente los términos *reducción* o *función de reducción* y el símbolo $\leq$ a menos que el contexto requiera mayor precisión.

El siguiente teorema formaliza la utilidad de las reducciones:

Teorema 5.1

a. Si $L_1 \leq L_2$ y $L_2 \in R$, entonces $L_1 \in R$.
b. Si $L_1 \leq L_2$ y $L_2 \in RE$, entonces $L_1 \in RE$.

Prueba. En palabras, los conjuntos R y RE son cerrados con respecto a las reducciones, es decir que reduciendo un lenguaje L_1 a un lenguaje L_2 de R (RE) se prueba que L_1 pertenece a R (RE). La demostración del teorema se obtiene de la definición de reducción. Mostramos la correspondiente a la parte (a):

Si existen una MT M_f que computa una reducción de L_1 a L_2 y una MT M_2 que decide L_2, entonces también existe una MT M_1 que decide L_1. Dada una entrada w, M_1 hace:

1. Ejecuta M_f y obtiene f(w).
2. Ejecuta M_2 a partir de f(w) y responde como M_2.

M_1 decide L_1 utilizando el algoritmo que ejecuta M_2 para decidir L_2. Formalmente, para toda cadena w se cumple: $w \in L_1$ sii $f(w) \in L_2$ sii M_2 acepta f(w) sii M_1 acepta w (figura 5.2).

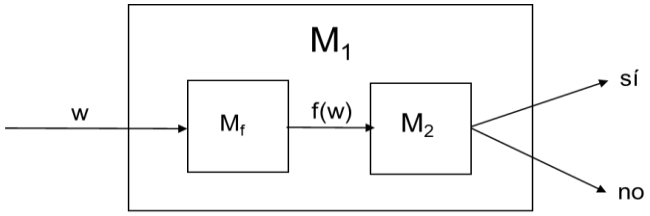

Figura 5.2. La MT M_1 acepta una cadena w sii la MT M_2 acepta la cadena $M_f(w) = f(w)$.

La demostración de la parte (b) es igual, salvo que hay que contemplar la posibilidad de que M_2 no se detenga.

Ejercicio. Probar la parte (b) del teorema.
Fin del teorema

Como se puede apreciar, el mecanismo de una reducción es tan simple como efectivo para poblar la jerarquía de la computabilidad, y para establecer relaciones entre sus lenguajes. En este último sentido, notar que el hecho de que exista una reducción de un lenguaje L_1 a un lenguaje L_2 implica que L_2 es *tan o más difícil* que L_1: no puede ser que L_2 sea recursivo y L_1 no lo sea, ni que L_2 sea recursivamente enumerable y L_1 no lo sea. Esto se puede formalizar con el siguiente corolario, aplicando el contrarrecíproco a las dos partes del teorema anterior:

Corolario 5.1

a. Si $L_1 \leq L_2$ y $L_1 \notin R$, entonces $L_2 \notin R$.
b. Si $L_1 \leq L_2$ y $L_1 \notin RE$, entonces $L_2 \notin RE$.

De este modo, las reducciones sirven para probar tanto la pertenencia como la no pertenencia de un lenguaje a R o a RE (en realidad, la pertenencia o no pertenencia a más conjuntos, como veremos en la parte de complejidad computacional). Otro corolario, derivado de la definición de reducción, es:

Corolario 5.2. $L_1 \leq L_2$ sii $L_1^C \leq L_2^C$.

Ejemplos

Los ejemplos que siguen se enfocan en pruebas de *no pertenencia* de un lenguaje a R o RE, que es en donde más se aprecia la utilidad de las reducciones en la computabilidad. Por resultar en general más simples, consideramos reducciones entre lenguajes que representan problemas sobre máquinas de Turing (en las notas adicionales consideramos reducciones entre lenguajes correspondientes a otros tipos de problemas).

Ejemplo 5.1. El primer ejemplo es una reducción del lenguaje diagonal $D = \{w_i \mid M_i \text{ acepta } w_i\}$, descripto en el capítulo anterior, al lenguaje $L_U = \{(<M>, w) \mid M \text{ acepta } w\}$, conocido como *lenguaje universal* porque representa el *problema de aceptación universal* (o directamente el *problema de aceptación*). El problema de aceptación generaliza el problema representado por D, y al igual que el problema de la detención, analizado previamente, identifica el grado de dificultad de la clase RE (lo formalizamos más adelante).

En base a la relación entre D y L_U proponemos la siguiente función de reducción f (en éste y en el resto de los casos, la definición de f debe entenderse para toda cadena de Σ^*):

$$f(w) = (<M_i>, w)$$

tal que i es el índice de la cadena w según el orden canónico. La figura 5.3 representa la reducción definida.

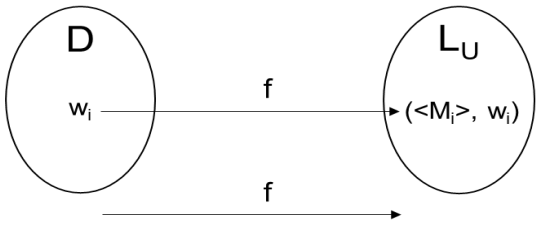

Figura 5.3. Reducción del lenguaje D al lenguaje L_U.

Veamos que f es efectivamente una reducción de D a L_U:

Por un lado, f satisface la condición de computabilidad. Existe una MT M_f que la computa. Dada una entrada w, M_f hace:

1. Encuentra el índice i tal que $w = w_i$ según el orden canónico.
2. Genera el código $<M_i>$ según el orden canónico.
3. Escribe en su cinta de salida el par $(<M_i>, w_i)$.

Por otro lado, f satisface la condición de correctitud. Si $w = w_i$ según el orden canónico:

- Si $w_i \in D$, entonces M_i acepta w_i, y por lo tanto $(<M_i>, w_i) \in L_U$.
- Si $w_i \notin D$, entonces M_i no acepta w_i, y por lo tanto $(<M_i>, w_i) \notin L_U$.

De este modo, se cumple $D \leq L_U$.

Veamos además cómo se puede caracterizar a L_U a partir de lo que sabemos de D:

$D \in RE - R$ (teorema 4.1).

De $D \notin R$, por el corolario 5.1 se cumple $L_U \notin R$.

Y de $D \in RE$, por el teorema 5.1 sólo se puede establecer que L_U es tan o más difícil que D, o sea que L_U puede tanto pertenecer como no pertenecer a RE. De todos modos, su pertenencia a RE se puede probar fácilmente construyendo una máquina de Turing que lo reconozca, y así llegamos a que $L_U \in RE - R$.

Ejercicio. Probar que $L_U \in RE$.

Fin del ejemplo

Presentada una primera reducción en detalle, en los siguientes ejemplos nos concentramos en aspectos particulares (e involucramos al lector en algunas pruebas). Nos interesa sobre todo mostrar cómo las reducciones permiten caracterizar y relacionar lenguajes.

Ejemplo 5.2. Vamos a definir dos reducciones entre el lenguaje HP = {(<M>, w) | M se detiene a partir de w}, representante del problema de la detención, y el lenguaje L_U, una en cada sentido. De esta forma probaremos, mediante reducciones, que cada lenguaje es tan o más difícil que el otro, es decir que tienen grado de dificultad equivalente, en el sentido de la computabilidad (ya sabemos de antes que los dos lenguajes pertenecen a RE – R).

Primero definimos una reducción de HP a L_U:

En base a las definiciones de HP y L_U, hay que asignar a todo par ($<M_1>$, w_1) tal que M_1 se detiene a partir de w_1, un par ($<M_2>$, w_2) tal que M_2 acepta w_2; a todo par ($<M_1>$, w_1) tal que M_1 no se detiene a partir de w_1, un par ($<M_2>$, w_2) tal que M_2 rechaza w_2; y a toda entrada sintácticamente incorrecta, es decir sin la forma (<M>, w), que por lo tanto no pertenece a HP, alguna cadena no perteneciente a L_U. Proponemos la siguiente función de reducción f. Dada una entrada v, si v es sintácticamente correcta (tiene la forma de un par ($<M_1>$, w)), establece:

$$f((<M_1>, w)) = (<M_2>, w)$$

siendo M_1 y M_2 iguales salvo que todo estado q_R de M_1 está reemplazado en M_2 por el estado q_A. Y si v no es sintácticamente correcta, establece:

$$f(v) = v$$

Ejercicio. Probar la computabilidad de f.

Veamos que se cumple la correctitud de f:

- Si $v = (<M_1>, w) \in HP$, entonces M_1 se detiene a partir de w. Así, M_2 acepta w, porque todo estado q_R de M_1 se reemplaza en M_2 por el estado q_A. Por lo tanto, $(<M_2>, w) \in L_U$.
- Si $v = (<M_1>, w) \notin HP$, entonces M_1 no se detiene a partir de w. Así, M_2 no se detiene a partir de w, porque si M_1 no se detiene a partir de w, los reemplazos definidos no evitan que M_2 no se detenga a partir de w. Por lo tanto, $(<M_2>, w) \notin L_U$.
- Si $v \neq (<M>, w) \notin HP$, entonces $f(v) = v \neq (<M>, w) \notin L_U$.

De este modo, se cumple $HP \leq L_U$.

Definimos ahora la reducción de L_U a HP (para simplificar, de acá en adelante vamos a ignorar las entradas sintácticamente incorrectas, que en todos los casos que veremos se pueden tratar sin mayores problemas):

Hay que asignar a todo par ($<M_1>$, w_1) tal que M_1 acepta w_1, un par ($<M_2>$, w_2) tal que M_2 se detiene a partir de w_2; y a todo par ($<M_1>$, w_1) tal que M_1 rechaza w_1, un par ($<M_2>$, w_2) tal que M_2 no se detiene a partir de w_2. Proponemos la siguiente función de reducción f:

$$f((<M_1>, w)) = (<M_2>, w)$$

siendo M_1 y M_2 iguales salvo que todo estado q_R de M_1 está reemplazado en M_2 por un estado q_{loop}, y además M_2 incluye un bucle infinito a partir de q_{loop}.

Ejercicio. Probar la computabilidad y la correctitud de f.

De este modo, se cumple $L_U \leq HP$.

Fin del ejemplo

Reflexión. Ya con el par de ejemplos presentados podemos vislumbrar en qué consiste una reducción. Por medio de manipulaciones *sintácticas*, más o menos complejas, se transforman, respectivamente, las instancias positivas y negativas de un problema en las instancias positivas y negativas de otro, de alguna manera relacionado con el primero. Por lo tanto, a pesar de su apariencia meramente sintáctica, las transformaciones llevadas a cabo son también *semánticas*, dado que tienen en cuenta la semántica de los problemas. Esto permite definir grupos de problemas por afinidad (ahora en el sentido de la computabilidad, después en el sentido de la complejidad computacional).

Fin de la reflexión

Ejemplo 5.3. Caracterizaremos por medio de reducciones al lenguaje $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$, que representa el problema de determinar si una máquina de Turing acepta todas las cadenas del conjunto Σ^*.

Empezamos reduciendo L_U a L_{Σ^*}. En base a las definiciones de los dos lenguajes, hay que asignar a todo par ($<M_1>$, w) tal que M_1 acepta w, un código $<M_2>$ tal que M_2 acepta todas las cadenas de Σ^*; y a todo par ($<M_1>$, w) tal que M_1 rechaza w, un código $<M_2>$ tal que M_2 rechaza al menos una cadena. Proponemos la siguiente función de reducción f:

$$f((<M_1>, w)) = <M_2>$$

siendo M_2 una MT que al comienzo reemplaza su entrada por la cadena w, y luego ejecuta M_1 y responde como M_1.

Ejercicio. Probar la computabilidad de f.

Veamos que se cumple la correctitud de f:

- Si (<M_1>, w) ∈ L_U, entonces M_1 acepta w. Así, M_2 acepta todas las cadenas de Σ*, es decir que reconoce el lenguaje Σ*, porque siempre ejecuta M_1 a partir de la misma entrada w. Por lo tanto, <M_2> ∈ L_{Σ^*}.
- Si (<M_1>, w) ∉ L_U, entonces M_1 rechaza w. Así, por la misma razón que antes, M_2 no acepta ninguna cadena de Σ*, es decir que reconoce el lenguaje vacío ∅, distinto del lenguaje Σ*. Por lo tanto, <M_2> ∉ L_{Σ^*}.

De este modo, se cumple $L_U \leq L_{\Sigma^*}$.

Dado que $L_U \notin R$, entonces por el corolario 5.1 se cumple $L_{\Sigma^*} \notin R$, mientras que por el teorema 5.1 no podemos determinar si pertenece a RE, como L_U, o es más difícil.

Intuitivamente, L_{Σ^*} no parece estar en RE, porque para estar en RE debería contar con una MT M_{Σ^*} capaz de aceptar una entrada <M> sii M acepta todas las cadenas de Σ*, es decir que M_{Σ^*} debería poder determinar en tiempo finito si M acepta infinitas cadenas, lo que no parece razonable. Y en efecto, se cumple $L_{\Sigma^*} \notin RE$. Lo vamos a probar con otra reducción, que describimos a continuación (por el corolario 5.1, debe ser desde un lenguaje que no esté en RE):

Vamos a recurrir a una reducción de $L_U{}^C$ a L_{Σ^*} ($L_U{}^C \notin RE$ porque $L_U \in RE - R$). De esta manera, las asignaciones de la función de reducción f tienen que ser las contrarias a las que definimos para reducir L_U a L_{Σ^*}. Es decir, ahora hay que asignar a todo par (<M_1>, w) tal que M_1 rechaza w, un código <M_2> tal que M_2 acepta todas las cadenas de Σ*, y a todo par (<M_1>, w) tal que M_1 acepta w, un código <M_2> tal que M_2 rechaza al menos una cadena. Un primer intento sería:

$$f_1((<M_1>, w)) = <M_2>$$

siendo M_2 una MT que reemplaza su entrada por la cadena w, ejecuta M_1 y responde *al revés* que M_1, con la idea de que si M_1 rechaza w entonces $L(M_2) = \Sigma^*$, y si en cambio M_1 acepta w entonces $L(M_2) = \emptyset \neq \Sigma^*$. Pero f_1 no es correcta, porque si M_1 no se detiene a partir de w, y por lo tanto rechaza, M_2 también va a rechazar. Así que proponemos esta otra función de reducción:

$$f_2((<M_1>, w)) = <M_2>$$

tal que M_2, a partir de su entrada v, ejecuta |v| pasos de M_1 a partir de w, y acepta sii M_1 no acepta. Veamos que ahora, acotando la cantidad de pasos de M_1, f_2 es efectivamente una reducción de $L_U{}^C$ a L_{Σ^*}.

Ejercicio. Probar la computabilidad de f_2.

Probamos a continuación la correctitud de f_2:

- Si (<M_1>, w) $\notin L_U$, entonces M_1 rechaza w. Así, M_2 siempre acepta su entrada v, dado que en |v| pasos M_1 no acepta w, cualquiera sea v, y entonces $L(M_2) = \Sigma^*$. Por lo tanto, se cumple <M_2> $\in L_{\Sigma^*}$.
- Si (<M_1>, w) $\in L_U$, entonces M_1 acepta w (en k pasos). Así, M_2 acepta su entrada v sii $|v| < k$, porque en $|v| \geq k$ pasos M_1 acepta w, cualquiera sea v, y entonces $L(M_2) = \{v \mid |v| < k\} \neq \Sigma^*$. Por lo tanto, se cumple <M_2> $\notin L_{\Sigma^*}$.

Así llegamos a que $L_{\Sigma^*} \notin$ RE. Y como $L_U \leq L_{\Sigma^*}$ (probado antes en este mismo ejemplo) implica $L_U{}^C \leq L_{\Sigma^*}{}^C$ (por el corolario 5.2), siendo $L_U{}^C \notin$ RE se cumple $L_{\Sigma^*}{}^C \notin$ RE (por el corolario 5.1). En definitiva, se cumple $L_{\Sigma^*} \notin$ RE y $L_{\Sigma^*}{}^C \notin$ RE, y así hemos encontrado formalmente un primer par de lenguajes que habitan el conjunto $\mathcal{L}$ – (RE ∪ CO-RE) de los lenguajes más difíciles de la jerarquía de la computabilidad, lo cual había quedado pendiente del capítulo anterior (en el capítulo 3 sólo habíamos mencionado como ejemplo el lenguaje que representa el problema de decisión en la aritmética y su complemento).

Fin del ejemplo

Ejemplo 5.4. Con una de las reducciones que describimos en el ejemplo anterior podemos probar que el lenguaje $L_\varnothing$ = {<M> | L(M) = ∅}, correspondiente al problema de determinar si una máquina de Turing no acepta ninguna cadena, pertenece al conjunto CO-RE – R.

La pertenencia del lenguaje $L_\varnothing$ a CO-RE (es decir, la pertenencia del lenguaje $L_\varnothing{}^C$ a RE) se puede probar sin mayor dificultad construyendo una máquina de Turing que reconozca $L_\varnothing{}^C$ (pedimos la prueba en el ejercicio 4.4). Y para probar que $L_\varnothing \notin$ R podemos utilizar nuevamente la reducción de L_U a L_{Σ^*}, porque la misma función de reducción f sirve para reducir L_U a $L_\varnothing{}^C$, o lo que es lo mismo, por el corolario 5.2, $L_U{}^C$ a $L_\varnothing$ (lo justificamos en el párrafo siguiente): a partir de $L_U{}^C \leq L_\varnothing$ y $L_U{}^C \notin$ RE obtenemos, por el corolario 5.1, $L_\varnothing \notin$ RE, y por lo tanto, $L_\varnothing \notin$ R.

Veamos que la función de reducción f de L_U a L_{Σ^*} que definimos en el ejemplo anterior sirve para reducir $L_U{}^C$ a $L_\varnothing$: asigna a todo par (<M_1>, w) un código <M_2>, tal que M_2 es una MT que al comienzo reemplaza su entrada por la cadena w, y luego ejecuta M_1 y responde como M_1. De esta forma:

- Si (<M_1>, w) $\notin L_U$, entonces M_1 rechaza w, por lo que $L(M_2) = \varnothing$, y así <M_2> $\in L_\varnothing$.
- Si (<M_1>, w) $\in L_U$, entonces M_1 acepta w, por lo que $L(M_2) = \Sigma^*$, y así <M_2> $\notin L_\varnothing$.

Fin del ejemplo

Ejemplo 5.5. En este último ejemplo (en las notas adicionales y en los ejercicios finales presentamos más), definimos una reducción que permite encontrar otro par de lenguajes en el conjunto $\mathcal{L}$ – (RE ∪ CO-RE).

Se trata del lenguaje $L_{EQ} = \{(<M_1>, <M_2>) \mid L(M_1) = L(M_2)\}$ y su complemento. El lenguaje L_{EQ} representa el problema de determinar si dos máquinas de Turing son equivalentes. Por lo tanto, este resultado, extrapolado al ámbito de la programación, significa que el problema de la equivalencia de programas es aún más difícil que el problema de la detención.

Intuitivamente, no parece razonable que $L_{EQ} \in$ RE:

Una MT M_{EQ} que reconozca L_{EQ}, dada una entrada $(<M_1>, <M_2>)$, debería ejecutar M_1 y M_2 a partir de cada una de las infinitas cadenas de Σ^*, y aceptar sii M_1 y M_2 siempre se comportan igual.

Formalmente, podemos aprovechar que $L_{\Sigma^*} \notin$ RE y $L_{\Sigma^*}{}^C \notin$ RE (probado en el ejemplo 5.3), para reducir uno a L_{EQ} y el otro a $L_{EQ}{}^C$, y así probar lo que buscamos, recurriendo al corolario 5.1. En realidad, por el corolario 5.2 alcanza con probar solamente $L_{\Sigma^*} \leq L_{EQ}$.

Para reducir L_{Σ^*} a L_{EQ} proponemos la siguiente función de reducción f:

$$f(<M>) = (<M>, <M_{\Sigma^*}>)$$

siendo M_{Σ^*} una MT que reconoce el lenguaje Σ^*.

Ejercicio. Probar la computabilidad y la correctitud de la función f.

Fin del ejemplo

En la figura 5.4 agrupamos los resultados obtenidos en los ejemplos anteriores, ubicando en la jerarquía de la computabilidad a cada uno de los lenguajes caracterizados.

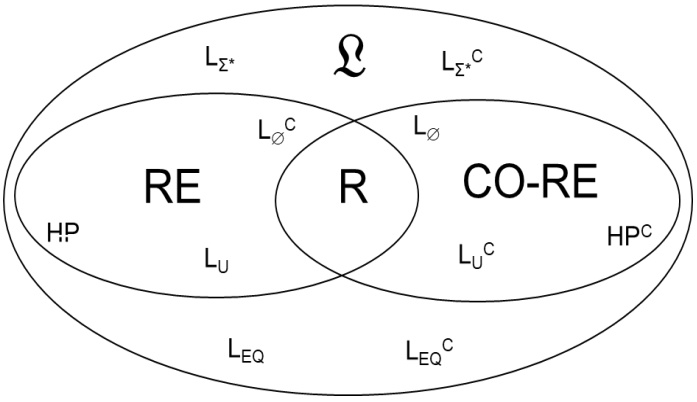

Figura 5.4. Los lenguajes HP, L_U, $L_\varnothing$, L_{Σ^*} y L_{EQ} en la jerarquía de la computabilidad.

Otras propiedades y usos

Por definición, una reducción no tiene por qué ser *inyectiva* ni *suryectiva*. Dos propiedades que sí cumple toda reducción son la *reflexividad* y la *transitividad*:

Teorema 5.2. Las reducciones son reflexivas y transitivas.

Prueba. Tanto la función *identidad* como la *composición* de dos reducciones son reducciones. En el ejercicio 5.8 pedimos comprobarlo formalmente.

Fin del teorema

Por otra parte, las reducciones no son *simétricas*: por lo probado antes sabemos que se cumple $L_U \leq L_{\Sigma^*}$, $L_U \in$ RE y $L_{\Sigma^*} \notin$ RE, y por lo tanto no puede cumplirse $L_{\Sigma^*} \leq L_U$. Tampoco son *antisimétricas*: vimos que se cumple HP $\leq L_U$ y $L_U \leq$ HP. La transitividad es muy útil porque permite utilizar un lenguaje intermedio cuando se dificulta encontrar una reducción directa entre dos lenguajes. Otra utilidad de las reducciones es la caracterización de clases de lenguajes a través de lenguajes representativos (esto lo notaremos mucho más en la complejidad computacional). Por ejemplo, todos los lenguajes recursivamente enumerables se reducen a L_U y a HP. Esto significa que L_U y HP son tan o más difíciles que cualquier otro lenguaje recursivamente enumerable, y en consecuencia, si fueran recursivos, se cumpliría RE = R. Se dice que HP y L_U son *RE-completos*, en el sentido de que representan el grado de dificultad de RE, de que resolverlos permite resolver cualquier lenguaje de RE. La RE-completitud de HP ya la referimos en el capítulo anterior, donde pedimos probarla construyendo una máquina de Turing. Con el siguiente teorema la probamos, al igual que la RE-completitud de L_U, por medio de reducciones:

Teorema 5.3. Los lenguajes L_U y HP son RE-completos.

Prueba. En el caso del lenguaje L_U, si L es un lenguaje cualquiera de RE y M es una MT que lo reconoce, entonces la función f(w) = (<M>, w) es una reducción de L a L_U. En el caso del lenguaje HP, como $L_U \leq$ HP, entonces por la transitividad de las reducciones se cumple, para todo L $\in$ RE, L $\leq$ HP. En la figura 5.5 ilustramos la prueba (f_i es una reducción de L_i a L_U y f es una reducción de L_U a HP).

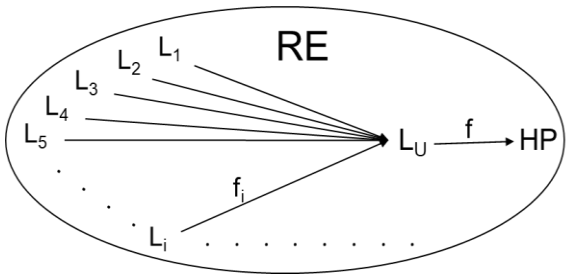

Figura 5.5. Todos los lenguajes recursivamente enumerables se reducen a L_U y HP.

Fin del teorema

Las Turing-reducciones y los oráculos

Pasamos a describir la otra clase de reducciones, las *Turing-reducciones*.

Para ello, antes tenemos que introducir las *máquinas de Turing con oráculo*. Una máquina de Turing M con un oráculo de un lenguaje L, identificada con M^L, es una máquina de Turing con un dispositivo adicional que le permite saber en cualquier momento de su ejecución si una cadena pertenece o no a L. Formalmente, una MT M^L:

- Además de las cintas y estados comunes, cuenta con una *cinta de pregunta*, un *estado de pregunta* $q_?$ y dos *estados de respuesta* q_S y q_N.
- Se ejecuta de la manera habitual, salvo que toda vez que el estado corriente es $q_?$ y la cinta de pregunta tiene una cadena v, en el paso siguiente pasa al estado q_S si $v \in L$ (respuesta positiva del oráculo) o al estado q_N si $v \notin L$ (respuesta negativa del oráculo).

La figura 5.6 representa una máquina de Turing con oráculo.

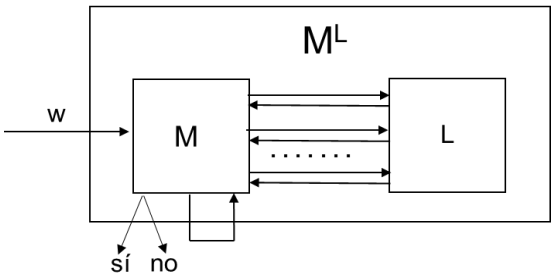

Figura 5.6. La MT M tiene un oráculo de un lenguaje L, al cual puede invocar varias veces.

No se establece ninguna restricción sobre el lenguaje con oráculo (puede pertenecer a cualquier conjunto de la jerarquía de la computabilidad). Por lo tanto, una máquina de Turing con oráculo no constituye un modelo real de una computadora. Debe entenderse como una abstracción para simplificar la descripción de una máquina que ejecuta otra máquina varias veces. La segunda máquina es capaz de decidir un lenguaje independientemente de su grado de dificultad, lo que permite relacionar lenguajes a partir de determinados supuestos.

Caracterizadas las máquinas de Turing con oráculo, ahora definimos las Turing-reducciones en términos de ellas:

Una Turing-reducción de un lenguaje L_1 a un lenguaje L_2 es una MT M^{L_2} que decide L_1. La notación $L_1 \leq_T L_2$ expresa que existe una Turing-reducción de L_1 a L_2.

De acuerdo a la definición, una Turing-reducción de un lenguaje L_1 a un lenguaje L_2 es una generalización de una m-reducción de L_1 a L_2, al poder invocarse repetidas veces el algoritmo que resuelve el lenguaje conocido. Esto hace que haya casos en que sólo existan Turing-reducciones entre dos lenguajes (mientras que se cumple que $L_1 \leq_m L_2$ implica $L_1 \leq_T L_2$, como pedimos probar en el ejercicio 5.10).

Ejemplo 5.6. La siguiente es una Turing-reducción de HP a L_U, es decir, una MT M^{Lu} que decide HP. Dada una entrada de la forma (<M_1>, w) (seguimos omitiendo las entradas sintácticamente incorrectas), M^{Lu} hace:

1. Ejecuta la MT M_f que computa la m-reducción de HP a L_U que describimos en el ejemplo 5.2, escribe la salida (<M_2>, w) correspondiente en su cinta de pregunta, y pasa al estado $q_?$.
2. Acepta o rechaza, según en el paso siguiente a la invocación al oráculo de L_U, el estado es q_S o q_N, respectivamente.

Ejercicio. Comprobar que M^{Lu} decide HP.
Fin del ejemplo

El siguiente teorema formaliza la utilidad de las Turing-reducciones:

Teorema 5.4. Si $L_1 \leq_T L_2$ y $L_2 \in R$, entonces $L_1 \in R$.

Prueba. El teorema establece lo mismo que el teorema 5.1 sobre las m-reducciones, pero considerando sólo los lenguajes recursivos (después justificamos esta restricción).

La idea general de la prueba es la siguiente. Dadas una MT M^{L2} que decide el lenguaje L_1 y una MT M_2 que decide el lenguaje L_2, una MT M_1 que se comporta como M^{L2}, pero reemplazando todas sus invocaciones al oráculo de L_2 por ejecuciones de M_2, decide L_1. Formalmente, dada una entrada w, M_1 hace:

1. Ejecuta M^{L2} hasta que alcanza el estado $q_?$ o termina (en este último caso termina y responde como M^{L2}).
2. Ejecuta M_2 a partir de la cadena de la cinta con la que simula la cinta de pregunta de M^{L2}.
3. Pasa al estado q_S o q_N según M_2 acepta o rechaza, respectivamente, y vuelve al bloque 1.

Fin del teorema

Aplicando el contrarrecíproco a lo que formula el teorema anterior, igual que hicimos con el teorema 5.1, obtenemos el siguiente corolario:

Corolario 5.3. Si $L_1 \leq_T L_2$ y $L_1 \notin R$, entonces $L_2 \notin R$.

De esta manera, las Turing-reducciones también permiten probar tanto la pertenencia como la no pertenencia de un lenguaje al conjunto R.

En lo que hace a la prueba de pertenencia a RE, en cambio, las Turing-reducciones no resultan útiles. $L_1 \leq_T L_2$ no asegura $L_2 \notin RE$ si $L_1 \notin RE$ (L_2 puede ser más fácil que L_1), consecuencia de que no se establece ninguna restricción sobre el tipo de lenguaje con oráculo.

Por ejemplo, se cumple $HP^C \leq_T HP$ (se prueba trivialmente), siendo $HP^C \notin RE$ y $HP \in RE$. También se cumple $L_\varnothing \leq_T L_U$ (lo probamos en una nota adicional), siendo $L_\varnothing \notin RE$ y $L_U \in RE$.

Como contrapartida, las Turing-reducciones permiten establecer relaciones entre lenguajes que no se pueden establecer utilizando m-reducciones (justamente debido a la falta de restricciones sobre los lenguajes con oráculo).

Por ejemplo, se cumple $L_\varnothing \leq_T L_U$ y $L_U \leq_T L_\varnothing$ (esto último lo probamos en la misma nota adicional referida antes), lo que quiere decir que si uno fuera recursivo también lo sería el otro. Se define en este caso que los lenguajes son *recursivamente equivalentes*. Así, con un criterio alternativo al que planteamos previamente, podemos interpretar que los grados de dificultad de $L_\varnothing$ y L_U son equivalentes.

Una manera particular de relacionar lenguajes con Turing-reducciones consiste en utilizar Turing-reducciones con oráculos que a su vez invocan a oráculos. Un ejemplo involucra a los lenguajes $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$ y $L_\varnothing^2 = \{<M> \mid L(M^{L_\varnothing}) = \varnothing\}$. $L_\varnothing^2$, como $L_\varnothing$, representa el problema de determinar si el lenguaje reconocido por una MT M es el vacío, pero con el agregado de que M cuenta con un oráculo del propio lenguaje $L_\varnothing$ (se pueden definir de la misma forma los lenguajes $L_\varnothing^3$, $L_\varnothing^4$, etc., cada uno considerando una máquina de Turing más potente que la anterior, y lo mismo se puede hacer con cualquier otro lenguaje). Se demuestra que $L_{\Sigma^*} \leq_T L_\varnothing^2$ y $L_\varnothing^2 \leq_T L_{\Sigma^*}$, es decir que L_{Σ^*} y $L_\varnothing^2$ son recursivamente equivalentes, lo que significa que no alcanza con una máquina de Turing con un oráculo de $L_\varnothing$ para decidir L_{Σ^*}, sino que se necesita una máquina con más poder computacional, de lo que se desprende que el grado de dificultad de L_{Σ^*} es mayor que el de $L_\varnothing$ (relación que ya determinamos utilizando m-reducciones).

Es interesante observar cómo las últimas relaciones entre lenguajes mencionadas se mantienen en escenarios más restringidos.

Por ejemplo, en el marco de los autómatas con pila, los lenguajes L_U y $L_\varnothing$ son recursivos y el lenguaje L_{Σ^*} no. Y en el marco de los autómatas finitos, los tres lenguajes son recursivos, pero mientras que L_U y $L_\varnothing$ se deciden en tiempo polinomial, para decidir L_{Σ^*} se requiere un tiempo mucho mayor, del orden exponencial.

Como en el caso de las m-reducciones, las Turing-reducciones son reflexivas y transitivas (ejercicio 5.8).

Notas adicionales

Agregamos ejemplos de m-reducciones y Turing-reducciones. En primer lugar, una m-reducción que permite probar la no recursividad de un tipo particular de lenguajes, los lenguajes que representan propiedades de los lenguajes recursivamente enumerables. Luego, dos Turing-reducciones mencionadas en el capítulo. Finalmente, dos m-reducciones con las que se puede demostrar la indecibilidad en la aritmética y la lógica de predicados.

El Teorema de Rice

El siguiente teorema, formulado por H. Rice, formaliza por medio de una m-reducción un esquema de prueba muy sencillo para demostrar que cierto tipo de lenguajes no son recursivos:

Teorema 5.5. Si Π es un conjunto de lenguajes recursivamente enumerables que cumple la condición $\emptyset \subset \Pi \subset RE$, entonces el lenguaje $L_\Pi = \{<M> \mid L(M) \in \Pi\}$ no es recursivo.

Prueba. Los conjuntos Π representan lenguajes recursivamente enumerables con una determinada propiedad (por ejemplo, ser recursivos, ser finitos, tener sólo palíndromos, etc). El Teorema de Rice establece la indecibilidad del problema de determinar, dado un conjunto de máquinas de Turing y una propiedad de los lenguajes de RE, si los lenguajes que reconocen las máquinas satisfacen dicha propiedad.

Ejercicio. ¿Por qué se plantea la condición $\emptyset \subset \Pi \subset RE$? *Ayuda: ¿se cumple la recursividad del lenguaje $L_\Pi = \{<M> \mid L(M) \in \Pi\}$ cuando $\Pi = \emptyset$ o $\Pi = RE$?*

Intuitivamente, L_Π no puede ser recursivo, porque para determinar, dada una entrada <M>, si L(M) satisface una propiedad específica, se deben procesar todas las cadenas de Σ*. Formalmente, el Teorema de Rice se puede probar mediante una m-reducción de L_U a L_Π (como $L_U \notin R$, entonces $L_\Pi \notin R$). La m-reducción es la siguiente:

Asumimos que Π no contiene el lenguaje vacío (de lo contrario, consideramos $\Pi^C = RE - \Pi$, probamos $L_{\Pi^C} \notin R$, y por lo tanto también $L_\Pi \notin R$). Sea $L \in \Pi$, con $L \neq \emptyset$, y sea M_L una MT que lo reconoce. Proponemos la siguiente función de reducción f de L_U a L_Π:

$$f((<M>, w)) = <M_{Lw}>$$

tal que la MT M_{Lw}, dada una entrada v, hace:

1. Ejecuta M a partir de w.
2. Si M acepta w, entonces ejecuta M_L a partir de v y acepta sii M_L acepta.
   Si M no acepta w, entonces no acepta.

Ejercicio. Probar la computabilidad de f.

Veamos que se cumple la correctitud de f:

- Si $(<M>, w) \in L_U$, entonces $L(M_{Lw}) = L$, y por lo tanto $<M_{Lw}> \in L_\Pi$.
- Si $(<M>, w) \notin L_U$, entonces $L(M_{Lw}) = \emptyset$, y por lo tanto $<M_{Lw}> \notin L_\Pi$.

Fin del teorema

El Teorema de Rice no puede aplicarse cuando los códigos <M> se definen no en términos de los lenguajes L(M) sino en términos de las propias MT M (por ejemplo, que se muevan sólo a la derecha, que el alfabeto sólo contenga los símbolos uno y cero, etc). En este último caso, las pruebas deben encararse de otra manera (ver los ejercicios finales).

Otros ejemplos de Turing-reducciones

En el ejemplo siguiente describimos un par de Turing-reducciones, una de L_U a $L_\varnothing$ y la otra de $L_\varnothing$ a L_U, que prueban que ambos lenguajes son recursivamente equivalentes (si uno fuera recursivo también lo sería el otro):

Ejemplo 5.7. Primero describimos la Turing-reducción de L_U a $L_\varnothing$. La siguiente MT $M^{L\varnothing}$ decide L_U. Dada una entrada (<M>, w), $M^{L\varnothing}$ hace:

1. Ejecuta la MT M_f que computa la m-reducción de L_U a $L_\varnothing^C$ que mostramos en el ejemplo 5.4, la cual asigna a (<M>, w) un código $<M_w>$ tal que $L(M_w) = \Sigma^*$ o $L(M_w) = \varnothing$ según M acepta o rechaza w, respectivamente.
2. Escribe $<M_w>$ en la cinta de pregunta y pasa al estado $q_?$.
3. Acepta o rechaza, según luego de la invocación al oráculo de $L_\varnothing$, el estado es q_N o q_S, respectivamente.

En el bloque 3, si el estado es q_N significa que $L(M_w) \neq \varnothing$, y así que (<M>, w) $\in L_U$. Y si el estado es q_S, significa que $L(M_w) = \varnothing$, y por lo tanto que (<M>, w) $\notin L_U$.

Ahora describimos la Turing-reducción de $L_\varnothing$ a L_U. La siguiente MT M^{Lu} decide $L_\varnothing$. Dada una entrada $<M_1>$, M^{Lu} hace:

1. Construye un código $<M_2>$, tal que M_2 reemplaza su entrada por $<M_1>$ y ejecuta una MT $M_\varnothing^C$ que reconoce el lenguaje $L_\varnothing^C$ (la construcción de $M_\varnothing^C$ la pedimos en el ejercicio 4.4), resultando así $L(M_2) = \Sigma^*$ o $L(M_2) = \varnothing$ según $L(M_1) \neq \varnothing$ o $L(M_1) = \varnothing$, respectivamente.
2. Escribe ($<M_2>$, λ) en la cinta de pregunta y pasa al estado $q_?$.
3. Acepta o rechaza, según luego de la invocación al oráculo de L_U, el estado es q_N o q_S, respectivamente.

En el bloque 3, si el estado es q_N significa que ($<M_2>$, λ) $\notin L_U$, y así que $L(M_1) = \varnothing$. Y si el estado es q_S, significa que ($<M_2>$, λ) $\in L_U$, y así que $L(M_1) \neq \varnothing$.

Fin del ejemplo

Otros ejemplos de m-reducciones

En esta última nota adicional mostramos la idea general de las m-reducciones que permiten probar la indecibilidad en la aritmética y la lógica de predicados.

Teorema 5.6. El problema de decisión en la aritmética es indecidible.

Prueba. La indecibilidad en la aritmética se deriva directamente del Primer Teorema de Incompletitud de K. Gödel, ya referido antes: establece que el lenguaje $L_{\mathcal{N}}$ = {ω | ω es un enunciado verdadero de la aritmética} no es recursivamente enumerable. Otra manera de probar lo mismo es mediante una m-reducción (Gödel formuló su teorema en 1931, cinco años antes del artículo de A. Turing con máquinas de Turing y m-reducciones). En efecto, existe una m-reducción de HP a $L_{\mathcal{N}}$, y como $HP \leq_m L_{\mathcal{N}}$ sii $HP^C \leq_m L_{\mathcal{N}}^C$, entonces $L_{\mathcal{N}}^C \notin RE$, y así también se cumple $L_{\mathcal{N}} \notin RE$ (ejercicio 3.9). En lo que sigue comentamos la idea general de la m-reducción:

- La computación de una máquina de Turing (con una cinta, sin perder generalidad) se puede representar por medio de una cadena de símbolos, más precisamente por medio de una secuencia de subcadenas que representan las distintas configuraciones de la computación, cada una con un estado, un contenido de la cinta y una posición del cabezal. En particular, la cadena es finita sii la máquina se detiene.
- Una cadena de símbolos se puede representar con un número natural, por medio de la técnica de *gödelización* creada y utilizada por Gödel en su teorema de incompletitud.

Teniendo en cuenta ambas observaciones, se define una función de reducción f que consiste en asignar a todo par (<M>, w) una fórmula aritmética ω que resulta verdadera sii la MT M se detiene a partir de la cadena w, donde ω expresa que existe una cadena de a lo sumo k símbolos que representa la computación de M a partir de w.

Fin del teorema

Teorema 5.7. El problema de decisión en la lógica de predicados es indecidible.

Prueba. Primero mostramos a grandes rasgos la m-reducción que presentó Turing en su artículo de 1936, y después, con algo más de detalle, otra que planteó A. Church, más sencilla.

La m-reducción de Turing parte del problema de determinar si una máquina de Turing imprime alguna vez el símbolo 0 (el lenguaje que representa el problema no es recursivo, lo que se puede demostrar mediante una m-reducción desde el lenguaje HP (ejercicio 5.4.h)).

La función de reducción f consiste en asignar a todo par (<M>, w), una fórmula ψ que incluye predicados relacionados con la computación de la MT M a partir de la cadena w, como: $R_s(x, y)$, que expresa que el símbolo de la celda *y* de la configuración x es s; I(x, y), que expresa que la

celda corriente de la configuración x es *y*; $K_q(x)$, que expresa que el estado corriente de la configuración x es q; y F(x, z), que expresa que la configuración z le sigue a la configuración x.

La asignación de ψ a (<M>, w) se hace de forma tal que ψ es demostrable en la lógica de predicados sii M a partir de w imprime alguna vez el símbolo 0. En consecuencia, la m-reducción prueba que el lenguaje L_{LP} = {ψ | ψ es un teorema de la lógica de predicados}, representante del problema de decisión en la lógica de predicados, no es recursivo, porque si lo fuera también lo sería por transitividad el lenguaje HP.

Describimos ahora la idea general de la m-reducción de Church. Parte del problema de correspondencia de Post, mencionado en el capítulo 3, que consiste en determinar, dado un conjunto finito de pares de cadenas C = {(u_1, v_1), …, (u_k, v_k)}, si hay una forma de ordenar los pares, permitiendo repeticiones, tal que la secuencia que resulte de concatenar las cadenas u_i coincida con la secuencia que resulte de concatenar las cadenas v_i. Se prueba que el lenguaje que representa dicho problema no es recursivo (se puede demostrar con una m-reducción desde el lenguaje L_U). De este modo, para probar que el lenguaje L_{LP} no es recursivo, se define una función de reducción f que asigna a un conjunto C de las características mencionadas, una fórmula ψ de la lógica de predicados, que es válida (es un teorema de la lógica de predicados) sii C tiene solución (existe un ordenamiento i_1, …, i_m tal que $u_{i1}…u_{im} = v_{i1}…v_{im}$).

La fórmula ψ utiliza la constante *e*, entendida como la cadena vacía; funciones $f_w(c)$, asociadas a la operación de concatenación de una cadena w al final de una cadena c (en realidad, f_w abrevia la composición de funciones f_s, siendo s un símbolo); y el predicado P(x, y), cuyo significado pretendido es que las cadenas x e *y* tienen, respectivamente, la forma $u_{i1}…u_{im}$ y $v_{i1}…v_{im}$, dado un determinado ordenamiento i_1, …, i_m. La forma de ψ es:

$$\psi_1 \wedge \psi_2 \rightarrow \psi_3$$

tal que:

$$\psi_1 = \bigwedge_{i=1,k} P(f_{ui}(e), f_{vi}(e))$$
$$\psi_2 = \forall x \forall y: P(x, y) \rightarrow \bigwedge_{i=1,k} P(f_{ui}(x), f_{vi}(y))$$
$$\psi_3 = \exists z: P(z, z)$$

usando más abreviaturas para simplificar la notación: $\bigwedge_{i=1,k}$ en vez de desarrollar una conjunción de k operandos, y → (implicación) en vez de plantear una disyunción con el antecedente negado.

Claramente, se cumple la computabilidad de f. También se cumple su correctitud:

- Si ψ es válida, cualquier interpretación la satisface, en particular la interpretación habitual de las cadenas de símbolos. Con dicha interpretación se comprueba que ψ_1 y ψ_2 son verdaderas, y por lo tanto también $\psi_3 = \exists z: P(z, z)$, lo que significa que C tiene solución.
- Si C tiene solución, existe un ordenamiento i_1, …, i_m tal que $u_{i1}…u_{im} = v_{i1}…v_{im}$. Con esto en consideración y partiendo de alguna interpretación, se comprueba que si ψ_1 y ψ_2 son verdaderas entonces $\psi_3 = \exists z: P(z, z)$ es verdadera, y así también ψ, lo que significa que ψ es válida porque es verdadera con cualquier interpretación.

Fin del teorema

Observaciones finales

- Existen dos clases de reducciones, las m-reducciones y las Turing-reducciones. Las Turing-reducciones son una generalización de las m-reducciones, pero las dos son de utilidad.
- Las reducciones permiten probar tanto la pertenencia como la no pertenencia de un lenguaje a R o a RE. Para probar la no pertenencia, en general es más fácil utilizar una reducción que una diagonalización, pero contando con un lenguaje ya caracterizado; de lo contrario, la diagonalización es el método de prueba a aplicar.
- Las reducciones también permiten agrupar lenguajes en conjuntos más específicos que los de la jerarquía de la computabilidad, de acuerdo a distintos criterios de afinidad, como veremos en la parte de complejidad computacional.

Referencias

Las m-reducciones y las Turing-reducciones incluidas en este capítulo se basan en las que se describen en el capítulo 8 de (Hopcroft y Ullman, 1979), el capítulo 5 de (Sipser, 1997) y el capítulo 5 de (Lewis y Papadimitriou, 1998).

Los oráculos en las máquinas de Turing los introdujo A. Turing en (Turing, 1939), para su tesis doctoral en Princeton sobre los sistemas lógicos basados en ordinales.

Otra alternativa para clasificar a los lenguajes por su grado de dificultad, en el sentido de la computabilidad, es por medio de *grados de Turing*, que son conjuntos de lenguajes recursivamente equivalentes, definidos en términos de máquinas de Turing con oráculo. Por ejemplo, los lenguajes recursivos conforman un grado de Turing, un grado de Turing de mayor dificultad incluye a los lenguajes HP, L_U y $L_\varnothing$., otro de mayor dificultad aún contiene los lenguajes recursivamente equivalentes al lenguaje HP^2 = {(<M>, w) | M^{HP} se detiene a partir de w}, que representa el problema de la detención de una máquina de Turing con un oráculo del propio lenguaje HP, y de la misma manera se puede continuar con grados de Turing con los lenguajes HP^3, HP^4, etc. (ver por ejemplo el capítulo 4 de (Aaronson, 2013)).

El Teorema de Rice no sólo permite probar la no recursividad de determinados lenguajes, sino que también se puede aplicar para demostrar la no pertenencia al conjunto de los lenguajes recursivamente enumerables. El trabajo completo de su autor, H. Rice, se describe en (Rice, 1953) y (Rice, 1956).

La prueba de Turing de la indecibilidad del problema de decisión en la lógica de predicados (el Entscheidungsproblem), que comentamos en una nota adicional, se desarrolla en la sección 11 de su artículo de 1936. Turing se enfocó en la *lógica canónica de primer orden* F_0 (también llamada *Engere Prädikatenkalkül* por D. Hilbert y *cálculo funcional puro de primer orden* por A. Church), teniendo en cuenta que si F_0 era decidible, también eran decidibles todas las lógicas de

primer orden. (Petzold, 2008) explica la prueba con mucho detalle. La prueba alternativa de Church que incluimos en la misma nota adicional referida antes se puede encontrar en el capítulo 2 de (Huth y Ryan, 2004).

Otras lecturas recomendadas:

El capítulo 6 de (Papadimitriou, 1994) y el capítulo 1 de (Arora y Barak, 2009): tratan la codificación de las computaciones de las máquinas de Turing.

El apéndice 1 de (Martínez y Piñeiro, 2009): presenta distintos ejemplos de teorías completas e incompletas.

Ejercicios

5.1 Sea w una cadena de unos y ceros y E(w) la cadena que se obtiene reemplazando en w los unos por ceros y los ceros por unos. Por ejemplo, E(100) = 011. Se define que un lenguaje L es un *lenguaje espejo* si toda cadena $w \in L$ distinta de la cadena vacía cumple que $E(w) \in L^C$. Probar que si L es un lenguaje espejo y no pertenece al conjunto RE, entonces el lenguaje L^C tampoco pertenece a RE.

5.2 Justificar por qué no puede existir una m-reducción del lenguaje $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$ al lenguaje $L_{\varnothing} = \{<M> \mid L(M) = \emptyset\}$.

5.3 Determinar a qué conjunto de la jerarquía de la computabilidad pertenece cada uno de los siguientes lenguajes. *Comentario: No se puede aplicar el Teorema de Rice.*

a. $L = \{<M> \mid M$ se detiene a partir de todas las cadenas$\}$.
b. $L = \{<M> \mid M$ se detiene a partir de alguna cadena$\}$.
c. $L = \{<M> \mid M$ no se detiene a partir de alguna cadena$\}$.
d. $L = \{<M> \mid M$ no se detiene a partir de ninguna cadena$\}$.

5.4 Lo mismo que el ejercicio anterior, considerando los siguientes lenguajes:

a. $L = \{<M> \mid M$ es una MT con un número impar de estados$\}$.
b. $L = \{<M> \mid L(M) \in RE\}$.
c. $L = \{<M> \mid L(M)$ incluye una cadena w tal que $|w| < 100\}$.
d. $L = \{<M> \mid L(M)$ es finito$\}$.
e. $L = \{(<M_1>, <M_2>) \mid L(M_1) \neq L(M_2)\}$.
f. $L = \{(<M_1>, <M_2>) \mid L(M_1) \subseteq L(M_2)\}$.
g. $L = \{((<M>, w, v) \mid M(w) = v\}$.
h. $L = \{(<M>, w) \mid M$ a partir de w imprime alguna vez el símbolo 0$\}$.

5.5 Con la siguiente función f se pretende definir una m-reducción del lenguaje $L_U{}^C$ al lenguaje $L_R = \{<M> \mid L(M) \in R\}$: dada una cadena con la forma (<M>, w), la función f le asigna un

código $<M_w>$, tal que M_w es una MT que a partir de una entrada v: ejecuta M a partir de w; si M no acepta, entonces no acepta; si M acepta, entonces ejecuta M_{Lu} (MT que reconoce L_U) a partir de v y acepta sii M_{Lu} acepta.

a. Probar que la función f es efectivamente una m-reducción de $L_U{}^C$ a L_R.
b. ¿Qué cadena podría asignar la función f a una entrada sintácticamente incorrecta?

5.6 Con esta otra función f se pretende definir una m-reducción del lenguaje $L_U{}^C$ al lenguaje $L_R{}^C = \{<M> \mid L(M) \notin R\}$: dada una cadena con la forma (<M>, w), la función f le asigna un código $<M_w>$, tal que M_w es una MT que a partir de una entrada v: ejecuta en paralelo M a partir de w y M_{Lu} (MT que reconoce L_U) a partir de v, y acepta sii alguna de las dos máquinas acepta.

a. Probar que la función f es efectivamente una m-reducción.
b. ¿Qué cadena podría asignar la función f a una entrada sintácticamente incorrecta?
c. ¿En qué conjunto de la jerarquía de la computabilidad están los lenguajes L_R y $L_R{}^C$ definidos en este ejercicio y el anterior?

5.7 En la lógica de predicados, los teoremas coinciden con las fórmulas válidas, es decir con las fórmulas verdaderas con cualquier interpretación. Así, por la indecibilidad del problema de decisión en la lógica de predicados, el lenguaje de las fórmulas válidas no es recursivo. Probar que entonces tampoco es recursivo el lenguaje de las fórmulas inválidas, es decir las fórmulas falsas con cualquier interpretación. *Ayuda: utilizar una m-reducción*.

5.8 Probar que las m-reducciones y las Turing-reducciones son reflexivas y transitivas.

5.9 Probar que se cumple $L_U{}^C \leq_T L_U$. ¿Se puede concluir algo sobre $L_U{}^C$ a partir de esta Turing-reducción?

5.10 Probar que $L_1 \leq_m L_2$ implica $L_1 \leq_T L_2$.

5.11 Probar que los siguientes enunciados son lógicamente equivalentes (es decir, se implican dos a dos):

a. A es Turing-reducible a B.
b. A^C es Turing-reducible a B.
c. A es Turing-reducible a B^C.
d. A^C es Turing-reducible a B^C.

5.12 Probar que si existen una m-reducción de L_1 a L_2 y una Turing-reducción de L_2 a L_3, entonces también existe una Turing-reducción de L_1 a L_3.

5.13 Determinar cuáles de los siguientes lenguajes son recursivos. *Comentario: se puede aplicar el Teorema de Rice cuando sea posible*.

a. L = {<M> | las cadenas de L(M) tienen la forma $a^n b^n$, con $n \geq 1$}.

b. L = {<M> | L(M) es un lenguaje libre de contexto}.

c. L = {<M> | M tiene un número par de estados}.

d. L = {<M> | L(M) $\in$ RE}.

e. L = {<M> | L(M) = L_{Σ^*}}.

CAPÍTULO 6
Tiempo polinomial y no polinomial

Hasta ahora nos hemos enfocado en la computabilidad de los problemas. Describimos varios problemas de interés computacional y nos interesamos principalmente en la propiedad de decidibilidad. Pero con saber que un problema es decidible no alcanza para establecer fehacientemente su grado de dificultad.

Por ejemplo, siempre se puede decidir si un enunciado sobre los números naturales es verdadero o falso cuando la aritmética considerada se restringe a la operación de suma (*aritmética de Presburger*), pero para lograrlo se necesita una máquina de Turing que tiene que ejecutar en general una cantidad de pasos tan grande, que hace que el problema de decisión correspondiente resulte *intratable*. Lo mismo se puede decir con respecto a la cantidad de celdas que la máquina debe recorrer.

Sobre esta segunda óptica de análisis, la del costo computacional para resolver los problemas decidibles, conocida como *complejidad computacional*, trataremos a partir de este capítulo y hasta el final del libro. Utilizaremos las dos métricas a las que hicimos referencia en el ejemplo anterior, que son las más habituales e intuitivas, expresadas en términos de máquinas de Turing: en primer lugar el *tiempo*, medido en cantidad de pasos ejecutados (métrica de *complejidad temporal*), y en segundo lugar el *espacio*, medido en cantidad de celdas ocupadas (métrica de *complejidad espacial*). En las referencias mencionamos otras métricas.

Partimos con la complejidad temporal. Como hicimos en la parte de computabilidad, comenzamos especificando el escenario de estudio. Planteamos las definiciones básicas, mostramos ejemplos representativos y presentamos una jerarquía de lenguajes, que en este caso iremos refinando a lo largo de los capítulos.

Seguimos presentando los temas en el marco de los problemas de decisión, y por lo tanto continuamos desarrollando las descripciones en términos de lenguajes, ahora solamente recursivos. En el último capítulo analizaremos otros tipos de problemas, y también recurriremos a otros modelos de computación.

Introducción a la complejidad temporal

Intuitivamente, un lenguaje L_1 es más difícil que un lenguaje L_2, en el sentido de la complejidad temporal, si para decidir L_1 es necesario ejecutar más pasos de una máquina de Turing que para decidir L_2. ¿Pero cualquier cantidad más grande de pasos determina un grado de dificultad

mayor? Además, ¿cuándo un lenguaje puede ser considerado tratable, teniendo en cuenta la cantidad de pasos que se requieren para decidirlo? Y en todo caso, ¿con qué modelo de máquina de Turing y con qué representación de las cadenas corresponde hacer las mediciones?

Evidentemente, para formalizar la noción de complejidad temporal de un lenguaje necesitamos acordar previamente varias convenciones, que presentamos a continuación:

Tiempo de ejecución con respecto al tamaño de las cadenas de entrada. Naturalmente, las máquinas de Turing *tardan* más a medida que procesan entradas más grandes, por lo que analizar cuántos pasos ejecutan en términos absolutos no tiene ningún sentido. Se utilizan en cambio *funciones temporales* T(n), siendo n el tamaño de las entradas, para especificar su crecimiento en relación al crecimiento de las entradas. El crecimiento de las funciones temporales puede ser, siendo k y c constantes: *lineal* como n, *cuadrático* como n^2, más en general *polinomial como* n^k, *cuasipolinomial* como $n^{\log_2 n}$, *subexponencial* como $c^{\sqrt{n}}$, *exponencial* como c^{n^k}, *doble exponencial* como 2 elevado a la c^{n^k}, etc. Utilizaremos a veces, para abreviar, la expresión *poly(n)* para referirnos a una función polinomial, y la expresión *exp(n)* para referirnos a una función exponencial (mejor dicho no polinomial, como aclararemos enseguida).

Formalmente, se define que una MT M se ejecuta en tiempo T(n) (o tarda tiempo T(n), o es de tiempo de ejecución T(n), etc.) sii para toda cadena de entrada w, con |w| = n, M a partir de w hace a lo sumo T(n) pasos. Se cumple $T(n) \geq n$, porque M tiene que leer mínimamente los n símbolos de w.

Tiempo de ejecución en el peor caso. De acuerdo a la definición anterior, el tiempo de ejecución de una máquina de Turing queda determinado por el tiempo *máximo* que puede tardar, considerando la población completa de las cadenas de entrada. Esto quiere decir que unas pocas entradas pueden castigar al conjunto. Por ejemplo, si a partir de la gran mayoría de sus entradas una MT M tarda tiempo polinomial, y unas pocas provocan ejecuciones de tiempo exponencial, entonces el tiempo de ejecución de M queda establecido como exponencial, el *peor caso*. Podría objetarse este criterio: ¿No es más lógico utilizar el tiempo *promedio*, considerando la distribución de las entradas, o en todo caso el tiempo *mínimo* producido cualquiera sea la entrada? La razón para no utilizar en general estos dos criterios es que los resultados correspondientes son difíciles de obtener. De hecho, actualmente a pocos lenguajes se les conoce el tiempo promedio o el tiempo mínimo que se necesita para decidirlos.

De esta manera, la complejidad temporal de un lenguaje queda determinada por la máquina de Turing conocida más rápida que lo decide, contemplando la cota temporal superior y todas las cadenas de Σ^*. Así se abre la posibilidad de mejora, es decir, la posibilidad de que se encuentre una máquina de Turing más rápida (lo que ha venido ocurriendo a lo largo de los años en distintos casos). Obviamente, para mejorar el tiempo también está la posibilidad de tratar sólo con un subconjunto de entradas, siempre y cuando el lenguaje, con dicha restricción, siga siendo de interés.

Tiempo de ejecución como orden de magnitud. En lugar de utilizar tiempos exactos T(n), en la complejidad temporal se manejan *órdenes de magnitud*, que se expresan con la *notación asintótica* O(T(n)), a la que ya recurrimos antes. Dicha expresión denota el conjunto de todas las funciones temporales acotadas superiormente por c**.**T(n), tal que c es una constante positiva.

Formalmente, se define que $T_1(n) = O(T_2(n))$, y se dice que la función $T_1(n)$ es *del orden de* la función $T_2(n)$, sii se cumple $T_1(n) \leq c.T_2(n)$ para todo n mayor o igual que un determinado n_0, con $c > 0$. Por ejemplo, $5n^2 + 3n + 8 = O(n^2)$, $n^2 = O(n^3)$, y $n^3 = O(2^n)$.

Ejercicio. Probar las tres igualdades.

El uso de órdenes de magnitud permite manejar distintos niveles de análisis. Por ejemplo, a veces puede interesar solamente distinguir entre tiempo polinomial y exponencial, mientras que otras veces puede requerirse una rigurosidad mayor, como conocer el grado del polinomio o la forma del exponente que definen un determinado tiempo de ejecución. En general se maneja un alto nivel de abstracción, que formalizaremos cuando presentemos la *jerarquía temporal*. Particularmente, los factores constantes de las funciones temporales T(n) se pueden eliminar, lo que se justifica por el Teorema de Aceleración Lineal (*Linear Speedup Theorem*): si existe una MT M_1 que se ejecuta en tiempo c**.**T(n), entonces también existe una MT M_2 equivalente que lo hace en tiempo T(n). La idea es que M_2, utilizando un alfabeto cuyos símbolos son secuencias de símbolos del alfabeto de M_1, puede procesar en un solo paso lo que a M_1 le lleva varios pasos. En los ejemplos utilizaremos en general la notación O(T(n)), teniendo en cuenta entonces que también podríamos utilizar directamente T(n)).

Antes de introducir las convenciones que faltan, mostramos algunos primeros ejemplos de cálculo de tiempo de ejecución. Refuerzan la fundamentación de las mismas, y permiten familiarizarnos con las definiciones que hemos formulado hasta el momento.

Ejemplo 6.1. En el capítulo 2 describimos dos máquinas de Turing, una MT M_1 con una cinta y una MT M_2 con dos cintas, para decidir el lenguaje de las cadenas a^kb^k, con $k \geq 1$ (ejemplos 2.1 y 2.2, respectivamente). Vamos a calcular el tiempo de ejecución de cada máquina.

Comenzamos por M_1. Dada una entrada w, M_1 hacía básicamente:

1. Leer una *a*, reemplazarla por una α y avanzar para buscar una b. Al encontrarla, reemplazarla por una β y retroceder para repetir el ciclo.
2. Agotadas las *a*, avanzar recorriendo todas las β hasta el final.

Para calcular el tiempo de ejecución de M_1 (en términos de $|w| = n$ y considerando el peor caso), tenemos que sumar los tiempos de ejecución de los bloques 1 y 2:

- Bloque 1: O(n/2) iteraciones, una por cada par de *a* y b, de O(n) pasos cada una, para avanzar desde una *a* hasta una b y luego retroceder hasta la siguiente *a*.

- Bloque 2: O(n/2) pasos, para avanzar sobre todas las β.
- Por lo tanto, el tiempo de ejecución de M_1 es $O(n^2/2) + O(n/2) = O(n^2)$, tiempo *cuadrático*.

Ejercicio. Justificar por qué también se puede establecer que el lenguaje de las cadenas $a^k b^k$, con $k \geq 1$, es decidible en tiempo n^2. *Ayuda: considerar el Teorema de Aceleración Lineal*.

Por su parte M_2, dada una entrada w, hacía básicamente:

1. Avanzar sobre las *a* en la cinta 1, y al mismo tiempo marcar una X por cada *a* en la cinta 2.
2. Avanzar sobre las b en la cinta 1, y al mismo tiempo retroceder sobre las X en la cinta 2.

En este caso, el tiempo de ejecución es menor. Los bloques 1 y 2 requieren, en conjunto, O(n) pasos, que es lo que lleva recorrer las *a* y b en la cinta 1 en simultáneo con la escritura y lectura de las X en la cinta 2. Por lo tanto, M_2 tarda tiempo O(n), tiempo *lineal*, y así cuadráticamente menor que el tiempo de ejecución de M_1, lo que es consistente con lo que describimos en el capítulo 2, cuando mostramos cómo simular una máquina de Turing con varias cintas con una máquina de Turing con una cinta (observación posterior al teorema 2.1).
Fin del ejemplo

Ejemplo 6.2. El lenguaje SAT = {φ | φ es una fórmula booleana sin cuantificadores satisfactible} representa el problema de satisfactibilidad. Como indicamos previamente, el problema consiste en determinar, dada una fórmula booleana sin cuantificadores, si existe alguna asignación de valores de verdad que la satisface, es decir que la hace verdadera. La sintaxis de las fórmulas booleanas sin cuantificadores la comentamos en el ejercicio 2.2 (asumiremos que tienen m variables, y recordar que hasta el capítulo 8 sólo utilizaremos fórmulas booleanas sin cuantificadores). Por ejemplo, la fórmula:

$$\varphi_1 = (x_1 \wedge x_2) \vee (\neg x_1 \vee \neg x_2)$$

es satisfactible, y en cambio la fórmula:

$$\varphi_2 = (x_1 \wedge x_2) \wedge (\neg x_1 \vee \neg x_2)$$

no lo es. Vamos a mostrar cómo decidir SAT de dos maneras, primero con una MT M_1 determinística y después con una MT M_2 no determinística, y cuál es el tiempo de ejecución requerido en cada caso.

Empezamos por la MTD M_1. Dada una entrada w, M_1 hace:

1. Si w no es una fórmula booleana φ, rechaza.
2. Genera una asignación de valores de verdad $\mathcal{A}$ para φ.

3. Evalúa φ con $\mathcal{A}$.

   Si la evaluación resulta verdadera, acepta.

   Si no, si no quedan asignaciones por probar, rechaza, y si quedan, vuelve al bloque 2.

M_1 decide SAT, porque si existe una asignación de valores que satisface φ, M_1 la encuentra. En cuanto al tiempo de ejecución de M_1, es el tiempo de ejecución del bloque 1 más el tiempo de ejecución del conjunto de iteraciones de los bloques 2 y 3:

- Bloques 1 y 3: se implementan con típicos algoritmos de análisis sintáctico y evaluación de fórmulas booleanas φ, basados en una pila, que requieren tiempo $O(|\varphi|^2) = O(n^2)$.
- Bloque 2: la generación de una asignación de valores de verdad a las m variables de una fórmula booleana φ requiere tiempo $O(m) = O(|\varphi|) = O(n)$.
- Los bloques 2 y 3 en conjunto se pueden iterar $2^m = O(2^{|\varphi|}) = O(2^n)$ veces, porque en el peor caso hay que probar con todas las asignaciones de valores de verdad posibles (*verdadero* y *falso* para cada una de las m variables).
- Por lo tanto, el tiempo de ejecución de M_1 es *exponencial*, determinado por la cantidad posiblemente exponencial de iteraciones que debe ejecutar.

Ahora mostramos la MTN M_2. A partir de una entrada w, M_2 hace:

1. Si w no es una fórmula booleana φ, rechaza.
2. Genera no determinísticamente una asignación de valores de verdad $\mathcal{A}$ para φ.
3. Evalúa φ con A, y acepta (rechaza) si la evaluación resulta verdadera (falsa).

Es decir, M_2 hace lo mismo que M_1, pero generando las asignaciones de valores de verdad de manera no determinística. Por definición, el tiempo de ejecución de una máquina de Turing no determinística es el tiempo de ejecución de la computación que tarda más (interpretación de ejecución paralela de todas las computaciones, que como comentamos en el capítulo 2 no es propia de un modelo real de una computadora). De este modo, a diferencia de la MTD M_1, el tiempo de ejecución de la MTN M_2 es *polinomial*, dado que los tres bloques, cualquiera sea la computación, tardan tiempo polinomial. Esto significa que M_2 tarda exponencialmente menos que M_1, lo que es consistente con lo que describimos en el capítulo 2, cuando mostramos cómo simular una máquina de Turing no determinística con una máquina de Turing determinística (observación posterior al teorema 2.2).

Fin del ejemplo

Ejemplo 6.3. Los dos ejemplos anteriores se relacionan con el modelo de máquina de Turing utilizado. En este último ejemplo introductorio nos enfocamos en la representación de las cadenas.

Sea el lenguaje DIV-3 = {N | N es un número natural que tiene un divisor que termina en 3}. Una MT M que, dado N, lo divide sucesivamente por 3, 13, 23, etc., hasta encontrar eventualmente un divisor, claramente decide DIV-3, ejecutando en el peor caso unas N/10 divisiones (M rechaza la entrada si no es un número). Por ejemplo, si N = 100, M prueba con 3, 13, 23, 33, 43, 53, 63, 73, 83 y 93.

De esta manera, M tarda lo que tardan O(N) divisiones (más la validación sintáctica inicial, de tiempo lineal). Una división, como toda operación aritmética elemental, es del orden polinomial. Así que para determinar el tiempo de M sólo queda expresarlo en términos de la longitud n de N, la cual depende de la codificación que elijamos para los números:

- Con la codificación unaria, n = N. Por lo tanto, el tiempo de ejecución de M es *polinomial*: O(n) divisiones.
- Con la codificación binaria, en cambio, $n = O(\log_2 N)$, y así $N = O(2^n)$. Por lo tanto, en este caso el tiempo de ejecución de M es *exponencial*: $O(2^n)$ divisiones. En otras palabras, utilizando codificación binaria (en realidad, cualquier codificación que no sea la unaria, como veremos enseguida), la cantidad de números que tiene que probar M en relación al tamaño de N es muy grande.

Fin del ejemplo

Continuamos con las últimas convenciones necesarias para el estudio de la complejidad temporal:

Lenguajes tratables e intratables. Los lenguajes decidibles con tiempos de ejecución muy grandes no pueden ser considerados tratables. En la complejidad temporal, concretamente, se toman como tratables sólo los lenguajes que se pueden decidir en tiempo polinomial. La fundamentación matemática de esta convención es que la diferencia entre el tiempo polinomial y el tiempo no polinomial se amplía bruscamente a medida que crece el tamaño de las cadenas. No menos importante, la fundamentación empírica es que al cabo de varias décadas de actividad algorítmica, el criterio adoptado se ha mantenido robusto.

Podría aducirse, y con razón, cierta arbitrariedad en la convención. Por ejemplo: ¿es tratable un lenguaje decidible en tiempo $O(n^{1000})$ e intratable uno decidible en tiempo $O(1{,}0000001^n)$? Pero en la práctica estos valores no se observan (en todo caso, los tiempos polinomiales de grado polinomial alto inicialmente suelen reducirse pronunciadamente a lo largo de los años, a través de distintas evoluciones algorítmicas).

Diremos que los lenguajes tratables cuentan con máquinas de Turing o algoritmos de decisión *eficientes*. Y para simplificar, nos basaremos en general en la división binaria entre el tiempo polinomial y el tiempo exponencial, considerando a este último como sinónimo del tiempo no polinomial, y así a los tiempos cuasipolinomiales, subexponenciales, doble exponenciales, etc., como tiempos exponenciales, salvo cuando por alguna razón amerite mayor precisión.

En la figura 6.1 mostramos una tabla que describe el comportamiento de algunas funciones polinomiales y exponenciales típicas, considerando distintos tamaños de cadenas (y una computadora determinada). Notar los saltos explosivos de las funciones exponenciales.

	10	20	30	40	50	60
n	0,00001 segundos	0,00002 segundos	0,00003 segundos	0,00004 segundos	0,00005 segundos	0,00006 segundos
n^2	0,0001 segundos	0,0004 segundos	0,0009 segundos	0,0016 segundos	0,0025 segundos	0,0036 segundos
n^3	0,001 segundos	0,008 segundos	0,027 segundos	0,064 segundos	0,125 segundos	0,216 segundos
n^5	0,1 segundos	3,2 segundos	24,3 segundos	1,7 minutos	5,2 minutos	13,0 minutos
2^n	0,001 segundos	1,0 segundos	17,9 minutos	12,7 días	35,7 años	366 siglos
3^n	0,059 segundos	58,0 minutos	6,5 años	3855 siglos	2.10^8 siglos	$1,3.10^{13}$ siglos

Figura 6.1. Comportamiento de algunas funciones polinomiales y exponenciales típicas.

Obviamente, que un lenguaje no sea decidible en tiempo polinomial no implica que no sea de interés computacional. Muchísimos lenguajes que no cuentan al día de hoy con algoritmos de decisión eficientes son muy tenidos en cuenta. Lo que se hace en estos casos, para lograr su tratabilidad, es recurrir a algún método de *remediación*, como restringir el dominio de las cadenas, aplicar algún algoritmo de aproximación (en el caso de la búsqueda de un óptimo), utilizar un modelo computacional alternativo, etc.

Modelo de máquina de Turing y representación de las cadenas. Las últimas dos convenciones que debemos acordar son el modelo computacional a utilizar y la forma de representar las cadenas, también determinantes para que el estudio de la complejidad temporal se desarrolle en un escenario robusto, es decir para que podamos identificar la dificultad intrínseca de los lenguajes.

El modelo computacional elegido es la máquina de Turing determinística con cualquier cantidad de cintas, el mismo que utilizamos en la parte de computabilidad. Es simple, permite un nivel de abstracción adecuado para los análisis, y sobre todo, según la Tesis Fuerte de Church-Turing, extensión de la Tesis de Church-Turing de la computabilidad, tiene la propiedad de simular la ejecución de cualquier otro modelo computacional *razonable*, es decir físicamente realizable, a lo sumo con un retardo polinomial (aunque esto, como ya comentamos, podría refutarse considerando las computadoras cuánticas). De esta manera, un lenguaje tratable o intratable con el modelo computacional elegido será tratable o intratable, respectivamente, con cualquier otro modelo computacional que sea razonable. Numerosas variantes de máquinas de

Turing determinísticas son modelos computacionales razonables. También lo son, entre otros modelos, los programas de cualquier típico lenguaje de programación y las máquinas RAM. Por el contrario, las máquinas de Turing no determinísticas son un claro ejemplo de modelo computacional no razonable. No son físicamente realizables, y así pueden alterar la robustez que necesitamos (como mostramos en el ejemplo 6.2).

El concepto de razonabilidad también tiene que considerarse en la representación de las cadenas, sobre todo en la de los números. En este sentido, lo razonable es excluir de las codificaciones de números a la *codificación unaria*. No sólo es prácticamente irrealizable (salvo para tamaños acotados de cadenas), sino que además contrasta con la robustez de las otras codificaciones: un algoritmo eficiente o ineficiente con una codificación no unaria sigue siendo eficiente o ineficiente, respectivamente, con otra codificación no unaria, lo que no sucede cuando una de ellas es la unaria (recordar el ejemplo 6.3). Más en detalle: un número N codificado en base $b_1 \neq 1$ ocupa $O(\log_{b1}N)$ dígitos, codificado en base $b_2 \neq 1$ ocupa $O(\log_{b2}N)$ dígitos, la relación entre ambas longitudes es constante, y por eso pasar de una codificación a la otra tarda tiempo polinomial, lo cual no ocurre cuando se incluye la codificación unaria. Para uniformar, usaremos siempre la *codificación binaria*. También la representación de las cadenas consideradas como un todo tiene que ser razonable. Por ejemplo, no es razonable representar un grafo con una cadena que sea la concatenación de todos sus caminos entre dos vértices i y j, ni tampoco representar un número con una cadena que sea la secuencia de todos sus divisores terminados en 3. Además de ser prácticamente irrealizables, representaciones de este tipo tergiversan la verdadera complejidad temporal de los lenguajes que las utilizan.

La jerarquía temporal

Completada la lista de convenciones que caracterizan el escenario de estudio de la complejidad temporal, pasamos a describirlo. Al igual que en la computabilidad, para analizar la complejidad temporal de los lenguajes se define una jerarquía, ahora en el marco del conjunto R de los lenguajes recursivos. Se la identifica como la *jerarquía temporal*, y se estructura básicamente en *clases temporales* TIME(T(n)). Una clase temporal TIME(T(n)) agrupa a todos los lenguajes que se pueden decidir en tiempo O(T(n)) (figura 6.2).

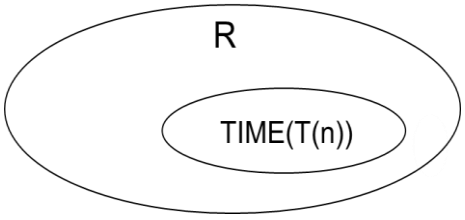

Figura 6.2. Una clase TIME(T(n)) de la jerarquía temporal.

La jerarquía temporal es *densa*: dada una clase temporal TIME($T_1(n)$), siempre existe otra clase temporal TIME($T_2(n)$) que la incluye de manera estricta.

Una segunda característica para destacar es que las funciones T(n) que definen las clases temporales son *tiempo-construibles*, lo que significa que son computables y que se computan en tiempo T(n). Más precisamente: una función T(n) es tiempo-construible sii existe una MT $M_{T(n)}$ tal que a partir de toda cadena w, con |w| = n, se ejecuta en T(n) pasos. El requerimiento de que las funciones temporales sean tiempo-construibles no representa una restricción, dado que todas las funciones temporales de uso habitual cumplen la propiedad (en el ejercicio 6.2 consideramos algunos ejemplos). Al ser tiempo-construibles, las funciones temporales se pueden utilizar como *relojes* para asegurar que la cantidad de pasos ejecutados por una máquina de Turing no supere un valor determinado. Otra razón para su uso es evitar efectos no deseados, como la posibilidad de que siempre se pueda construir una máquina de Turing más rápida que otra (Teorema de Aceleración, o *Speedup Theorem*), o la de que existan brechas arbitrariamente grandes entre dos clases temporales (Teorema de la Brecha, o *Gap Theorem*).

En una nota adicional profundizamos sobre dichas características de la jerarquía temporal (Teorema de la Jerarquía Temporal). En lo que sigue, comenzamos a describir sus clases más relevantes.

La clase P

La primera clase temporal que vamos a describir es la clase P. Consiste en la unión de todas las clases temporales TIME(n^k) con k constante (P es por *polinomial*). Por lo tanto, de acuerdo a lo que convinimos previamente, P es el conjunto de los lenguajes tratables. Se cumple:

$$P \subset R$$

lo que se puede probar por diagonalización, encontrando un lenguaje recursivo que requiere como mínimo tiempo exponencial para ser decidido (desarrollamos la prueba en otra nota adicional). Así llegamos a una primera versión de la jerarquía temporal (figura 6.3).

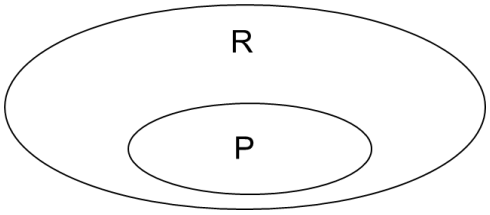

Figura 6.3. Primera versión de la jerarquía temporal.

También se prueba que P es cerrada con respecto al complemento, la intersección y la unión. Las pruebas son las mismas que hicimos para R (teoremas 3.1 y 3.2), basadas en la construcción de máquinas de Turing, salvo que ahora, adicionalmente, hay que tener en cuenta que las máquinas tienen que ejecutarse en tiempo polinomial.

Ejercicio. Probar las tres propiedades.

Ya mostramos un lenguaje de P, el de las cadenas $a^k b^k$ con $k \geq 1$. A continuación mostramos otros ejemplos, relacionados con problemas más interesantes.

Ejemplo 6.4. El lenguaje ACC = {(G, i, j) | G es un grafo no dirigido que tiene un camino del vértice i al vértice j} representa el problema de accesibilidad, ya definido previamente: consiste en determinar si un grafo no dirigido G tiene un camino del vértice i al vértice j (recordar que hasta el capítulo 8 sólo utilizaremos grafos no dirigidos). En el ejemplo 2.3 probamos que ACC es recursivo. Ahora vamos a probar que está en P, recurriendo a la misma máquina de Turing que construimos en dicho ejemplo.

Primero repasamos la representación que utilizamos. Un grafo G = (V, E) se representa concatenando la representación del conjunto V de vértices (secuencia de los números 1 a m, dado que por convención asumimos que los grafos siempre tienen m vértices), con la representación del conjunto E de arcos (secuencia de pares de números entre 1 y m, donde el orden de los vértices en cada par es irrelevante siendo el grafo no dirigido). La codificación de los números es la binaria, y los separadores utilizados son los paréntesis, las llaves y las comas. Por ejemplo, la siguiente cadena representa, con los números codificados en binario, el grafo de la figura 6.4, que ya mostramos en la figura 2.1, seguido de los vértices 3 y 5:

({1,2,3,4,5},{(1,2),(1,3),(2,3),(2,4),(4,5)}),3,5

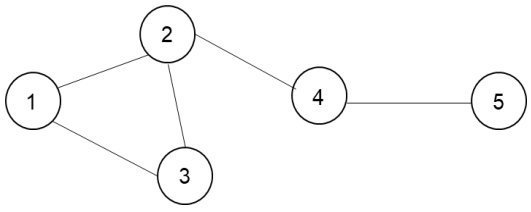

Figura 6.4. Ejemplo de un grafo no dirigido.

Repasamos ahora la MT M. El algoritmo se basa en el método DFS: se avanza desde el vértice i hacia el vértice j mientras se puede, y cuando no se puede seguir se retrocede para buscar otra alternativa. M, en la cinta 1, tiene la cadena de entrada; en la cinta 2, utilizada como pila, guarda los vértices que van determinando un camino del vértice i al vértice j; y en la cinta 3

marca los arcos procesados, para no pasar por ellos más de dos veces (cada celda corresponde a un arco del grafo). Formalmente, dada una entrada w, M hace:

1. Si w no tiene la forma (G, i, j), rechaza.
2. Hace v_1 := i y apila el vértice v_1 en la cinta 2.
3. Busca en la cinta 1 un arco (v_1, v_2) o (v_2, v_1) no marcado en la cinta 3.
   Si encuentra algún arco:
      Lo marca en la cinta 3 y apila el vértice v_2 en la cinta 2.
      Si v_2 = j, acepta.
      Si no, hace v_1 := v_2 y vuelve al bloque 3.
   Si no encuentra ningún arco:
      Desapila el vértice del tope de la cinta 2.
      Si la cinta 2 queda vacía, rechaza.
      Si no, si v_3 es el nuevo vértice del tope de la cinta 2, hace v_1 := v_3 y vuelve al bloque 3.

Probamos a continuación que M tarda tiempo polinomial:

- Bloque 1: Hay que validar que w empieza con V (vértices 1 a m), sigue con E (arcos definidos con vértices de V) y termina con dos vértices i y j, y que los separadores son paréntesis, llaves y comas ubicados adecuadamente. Así, básicamente, hay que recorrer primero |V| símbolos y luego |V| símbolos por cada vértice de cada arco de E: $O(|V|) + O(|E|.|V|) = O(|G|^2) = O(n^2)$ pasos.
- Bloque 2: Hay que asignar y apilar un vértice. Requiere $O(\log_2 m) = O(|V|) = O(n)$ pasos.
- Bloque 3: A lo sumo se recorre un arco dos veces, una vez cuando se avanza hacia el vértice j y otra eventualmente cuando se retrocede para buscar otra alternativa. Esto implica que el bloque se itera O(|E|) veces. En las iteraciones se recorren |E| símbolos para buscar un arco no marcado y marcarlo si se lo encuentra (O(|E|) pasos), y se efectúan operaciones de apilamiento o desapilamiento, asignación y comparación de vértices ($O(\log_2 m) = O(|V|)$ pasos). Queda: $O(|E|).(O(|E|) + O(|V|)) = O(|G|).(O(|G|) + O(|G|)) = O(|G|^2) = O(n^2)$ pasos.
- Por lo tanto, el tiempo de ejecución de M es $O(n^2) + O(n) + O(n^2) = O(n^2)$.

Fin del ejemplo

Ejemplo 6.5. Vamos a probar que el lenguaje PRI-REL = {(N_1, N_2) | N_1 y N_2 son dos números naturales *primos relativos*} pertenece a P. Dos números son primos relativos (también conocidos como coprimos) si tienen como máximo común divisor el número 1.

La idea es construir una MT M basada en el a*lgoritmo de Euclides* para el cálculo del máximo común divisor. Después de una serie de divisiones y de intercambios entre los números N_1 y N_2, en algún momento N_2 termina quedando con el valor 0 y N_1 con el máximo común divisor de los dos, a partir del cual M acepta o rechaza. Formalmente, dada una entrada w, M hace:

1. Si w no es un par (N_1, N_2), rechaza.
2. Hace $N_1 := N_1 \bmod N_2$ (la función *mod* devuelve el resto de la división entre N_1 y N_2).
3. Intercambia los valores de N_1 y N_2.

   Si $N_2 = 0$, acepta si $N_1 = 1$ y rechaza si $N_1 \neq 1$.

   Si $N_2 \neq 0$, vuelve al bloque 2.

La correctitud del algoritmo se puede comprobar sin dificultad. En cuanto al tiempo de ejecución de M, es el tiempo de ejecución del bloque 1 más el tiempo de ejecución del conjunto de iteraciones de los bloques 2 y 3:

- Bloque 1: $O(|N_1|) + O(|N_2|) = O(n)$ pasos, lo mismo que el bloque 3.
- Bloque 2: una división entre N_1 y N_2 requiere $O(|(N_1, N_2)|^2) = O(n^2)$ pasos.
- Después de ejecutarse el bloque 2, N_1 se reduce a menos de la mitad (salvo la primera vez, si inicialmente es menor que N_2). Así, M ejecuta a lo sumo $O(\log_2 N_1) = O(|N_1|) = O(n)$ iteraciones, y por lo tanto, el tiempo de ejecución de M es $O(n) + O(n).(O(n^2) + O(n)) = O(n^3)$.

Fin del ejemplo

Ejemplo 6.6. El lenguaje SAT no parece ser decidible en tiempo polinomial (ejemplo 6.2). Sin embargo, un sublenguaje de SAT lo es. Se trata de 2-SAT, compuesto por las fórmulas booleanas satisfactibles en *forma normal conjuntiva* (o *FNC*) con *cláusulas* de dos *literales*. Una fórmula booleana está en FNC si es una conjunción de cláusulas, una cláusula es una disyunción de literales, y un literal es una variable o una variable negada. Por ejemplo, la fórmula booleana:

$$\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_4)$$

pertenece al lenguaje 2-SAT.

La idea general de una MT M para decidir eficientemente 2-SAT es la siguiente:

Dada una fórmula booleana, M empieza asignando arbitrariamente el valor *verdadero* a alguna variable x_i, y completa consistentemente todas las asignaciones que puede. Si detecta insatisfactibilidad intenta con el valor *falso* para x_i, si vuelve a detectar insatisfactibilidad rechaza, y si no, elimina las cláusulas satisfechas y repite el proceso sobre las cláusulas restantes a partir de otra asignación arbitraria a alguna variable x_j. Así continúa hasta que eventualmente no quedan cláusulas por procesar, en cuyo caso M acepta.

Formalmente, dada una entrada w, M hace (l_i denota un literal con la variable x_i):

1. Si w no es una fórmula booleana φ con la sintaxis requerida por 2-SAT, rechaza.
2. Construye un conjunto C con las cláusulas de φ y marca a todas como *insatisfechas*.

   Construye un conjunto V con las variables de φ.

3. Si el conjunto V tiene variables:

   Dada alguna variable x_i de V, hace x_i := *verdadero* y primer-valor := *verdadero*.

4. Si C tiene una cláusula c = ($\ell_j \vee \ell_k$) marcada como *insatisfecha* con al menos un literal asignado:

   Si ℓ_j = *verdadero* o ℓ_k = *verdadero*, marca a la cláusula c como *satisfecha*.

   Si no: si ℓ_j = *falso*, ℓ_k = *falso* y primer-valor ≠ *verdadero*, rechaza.

   Si no: si ℓ_j = *falso*, ℓ_k = *falso* y primer-valor = *verdadero*:

      Marca a todas las cláusulas de C como *insatisfechas*.

      Borra todas las asignaciones a las variables de V.

      Hace x_i := *falso* y primer-valor := *falso.*

   Si no: si ℓ_j = *falso*, asigna un valor a x_k tal que ℓ_k sea *verdadero*.

   Si no: asigna un valor a x_j tal que ℓ_j sea *verdadero*.

   Vuelve al bloque 4.

Si no:

   Elimina de C todas las cláusulas marcadas como *satisfechas*.

   Elimina de V todas las variables asignadas.

   Vuelve al bloque 3.

Si no: acepta.

Ejercicio. Comprobar que el algoritmo es correcto y eficiente. *Ayuda: en lo que hace al tiempo de ejecución del algoritmo, el de los bloques 3 y 4, considerados en conjunto, queda determinado por la cantidad de ejecuciones de la iteración externa sobre las variables de V y la cantidad de ejecuciones de la iteración interna sobre las cláusulas de C*.

Fin del ejemplo

Reflexión. En lugar de los tres algoritmos anteriores (ejemplos 6.4 a 6.6), pudimos haber utilizado otros con una estructura mucho más simple, empleando la *fuerza bruta*, es decir probando con todas las posibles soluciones hasta encontrar eventualmente una: en el caso del algoritmo para decidir el lenguaje ACC, dada una terna (G. i, j), probando con todas las secuencias de vértices del grafo G iniciadas en i y terminadas en j; en el caso del algoritmo para decidir el lenguaje PRI-REL, dado un par (N_1, N_2), probando con todos los números naturales menores que el menor entre los números N_1 y N_2; y en el caso del algoritmo para decidir el lenguaje 2-SAT, dada una entrada φ, probando con todas las asignaciones de valores de verdad a sus variables. Pero no los consideramos, porque pueden llegar a demandar tiempo exponencial (los tres algoritmos, en el peor caso, deben recorrer una cantidad exponencial de posibles soluciones). Utilizamos en cambio algoritmos *analíticos*, de tiempo polinomial, aprovechando nuestro conocimiento profundo sobre los problemas correspondientes, y empleando en mayor o menor medida el ingenio. Esta particularidad, la de contar con algoritmos de decisión analíticos eficientes, es precisamente la que distingue a los lenguajes de la clase P del resto de los

lenguajes recursivos (la misma característica que destacamos, en el marco de la jerarquía de la computabilidad, de los lenguajes recursivos en comparación con el resto de los lenguajes, en dicho caso considerando únicamente el aspecto de la detención de los algoritmos).
Fin de la reflexión

Desafortunadamente, al día de hoy, para decidir muchísimos lenguajes recursivos de interés computacional contamos sólo con el método de la fuerza bruta referido en la reflexión anterior. En la siguiente sección analizamos una clase temporal con lenguajes de esta característica.

La clase NP

La segunda clase temporal que vamos a describir es la clase NP. Es otra clase relevante de la jerarquía temporal, por contener muchísimos lenguajes de interés computacional.

Todo lenguaje de NP tiene la siguiente propiedad: si una cadena le pertenece, entonces dicha pertenencia *se puede verificar en tiempo polinomial*, con la ayuda de otra cadena conocida como *certificado* o *prueba*.

Expresado de otra forma: podamos o no podamos decidir en tiempo polinomial si una cadena pertenece a un lenguaje de NP, lo que seguro podemos hacer es verificar en tiempo polinomial que la cadena pertenece al lenguaje, si es que efectivamente pertenece, contando con la ayuda de otra cadena que hace las veces de certificado o prueba (que se obtuvo de alguna manera, en un tiempo determinado).

Formalmente, $L \in NP$ sii existe un polinomio p y una MT M tal que para toda cadena w:

$$w \in L \text{ sii } \exists x: |x| \leq p(|w|) \text{ y M acepta } (w, x) \text{ en tiempo } poly(|w|)$$

La MT M se conoce como *verificador eficiente* de L, y la cadena x como *certificado sucinto* de w, sucinto porque su tamaño es polinomial con respecto al tamaño de w (obviamente, si fuera más grande, ya su sola lectura excedería el tiempo polinomial).

La denominación NP de la clase es por *no determinística polinomial*, debido a que se puede definir alternativamente que un lenguaje pertenece a NP si existe una máquina de Turing no determinística de tiempo de ejecución polinomial que lo decide (lo vemos al final de esta sección).

Hay todavía una tercera manera de caracterizar a los lenguajes de NP: para todo lenguaje L de NP, existen un *probador* P (una máquina de Turing con poder ilimitado) y un *verificador* V (una máquina de Turing de tiempo de ejecución polinomial), tales que para toda cadena w, si w pertenece a L, entonces P siempre puede convencer a V, a través de un certificado sucinto x, de que w pertenece a L (la interacción entre P y V es *completa*), y si w no pertenece a L, entonces P no tiene chance alguna de convencer a V de lo contrario, cualquiera sea el certificado con que lo intente (la interacción entre P y V es *sensata*).

Ejemplo 6.7. Los lenguajes SAT y DIV-3, descriptos previamente, pertenecen a la clase NP.

En el caso de SAT, un certificado sucinto de una fórmula booleana φ del lenguaje es una asignación de valores de verdad $\mathcal{A}$ a φ que la satisface: mide a lo sumo como φ y la evaluación de φ con $\mathcal{A}$ tarda tiempo polinomial.

En el caso de DIV-3, un certificado sucinto de un número natural N del lenguaje es otro número natural M, terminado en 3, que lo divide: lo mismo, mide a lo sumo como N y la división de N por M se ejecuta en tiempo polinomial.

Fin del ejemplo

De acuerdo a la definición de la clase temporal NP, se cumple:

$$P \subseteq NP$$

Ejercicio. Probar la inclusión. *Ayuda: el certificado sucinto de una cadena w de un lenguaje de NP mide entre 0 y poly(|w|) símbolos.*

La conjetura ampliamente aceptada es que P ≠ NP. Intuitivamente, obtener una solución es más difícil que verificarla (por ejemplo, se supone que demostrar un teorema es más difícil que verificar una prueba del teorema). Y en la práctica, la única opción con que se cuenta actualmente para decidir muchísimos lenguajes de NP es la fuerza bruta, explorando en el peor caso el espacio completo de las posibles soluciones de las instancias de los problemas representados, lo que puede provocar ejecuciones de tiempo exponencial, dado que una cadena de tamaño n tiene $c^{poly(n)}$ certificados sucintos posibles, con c constante. La conjetura tiene más de 50 años, y no parece que pueda comprobarse ni refutarse en el corto plazo (ver las referencias). Se la ha identificado como el problema abierto más importante de la teoría de la computación y las matemáticas. Más aún, en el año 2000, el prestigioso Instituto Clay de Matemáticas de los Estados Unidos incluyó al problema entre los *siete problemas matemáticos del milenio*, y ofreció un premio de un millón de dólares a quien logre resolverlo.

Por la definición de NP y lo observado previamente, un lenguaje de NP se puede decidir con un algoritmo que para toda cadena de tamaño n, verifica a lo sumo, cada vez en tiempo polinomial, $c^{poly(n)}$ certificados sucintos posibles, con c constante. Así, si EXP es la clase de los lenguajes decidibles en tiempo $O(c^{poly(n)})$ (o bien $O(2^{poly(n)})$ como suele expresarse), se cumple:

$$NP \subseteq EXP$$

También en este caso la conjetura ampliamente aceptada es que los conjuntos son distintos. Intuitivamente, por ejemplo volviendo al lenguaje SAT, el complemento de SAT está en EXP, ya que se puede decidir en tiempo $O(2^m)$ si una fórmula booleana de m variables es insatisfactible, y no parece estar en NP, porque no parece posible que exista un certificado sucinto para una fórmula booleana insatisfactible (el certificado debería incluir las 2^m asignaciones posibles de

valores de verdad). Lo mismo se puede decir del complemento del lenguaje DIV-3. Cabe observar que estos ejemplos sugieren adicionalmente que la clase NP no sería cerrada con respecto al complemento, reforzando la conjetura $P \neq NP$. En cambio, NP es cerrada con respecto a la intersección y la unión (ejercicio 6.8).

En definitiva, se cumplen las inclusiones:

$$P \subseteq NP \subseteq EXP$$

pareciera que ambas son estrictas, y al menos una inclusión es efectivamente estricta, porque como indicamos antes, se prueba que existe un lenguaje recursivo que requiere como mínimo tiempo exponencial para ser decidido, y así:

$$P \subset EXP$$

De esta manera llegamos a una segunda versión de la jerarquía temporal. Restringida a la clase EXP, y acompañada de la asunción $P \subset NP \subset EXP$, la mostramos en la figura 6.5.

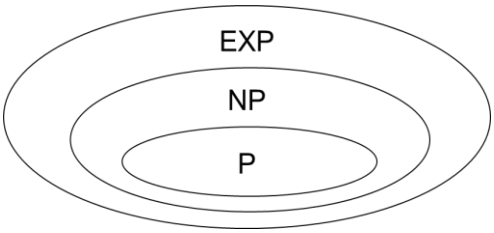

Figura 6.5. Segunda versión de la jerarquía temporal, asumiendo $P \subset NP \subset EXP$.

Completamos la descripción de la clase NP con más ejemplos. Luego de ellos nos referimos a la caracterización alternativa de NP por medio de máquinas de Turing no determinísticas.

Ejemplo 6.8. Sea el lenguaje SUM-SUB = {(S, K) | S es un conjunto de números naturales, K es un número natural, y existe un subconjunto de S cuyos elementos suman K}. El lenguaje se puede asociar a un *problema de planificación de tareas* (qué tareas se pueden ejecutar durante un tiempo determinado), y también a una versión simplificada del *problema de la mochila* (qué objetos se pueden incluir en una mochila hasta alcanzar un peso específico). El único algoritmo que se conoce para decidir SUM-SUB, dado un par (S, K), es probar con todos los subconjuntos de S y chequear cada vez si sus elementos suman K. De esta manera, en el peor caso el algoritmo ejecuta tantas iteraciones como subconjuntos tenga el conjunto S: $2^{|S|} = O(2^n)$, y por lo tanto, SUM-SUB no estaría en P. Por otro lado, con la siguiente MT M se prueba que SUM-SUB pertenece a NP:

Dados un par (S, K) y un conjunto de números C, con $|C| \leq |S|$, la MT M hace (como hicimos con las reducciones, de ahora en más, para simplificar, omitimos el tratamiento de las entradas incorrectas sintácticamente, eficiente en todos los casos que consideramos):

1. Verifica si $C \subseteq S$. Si no, rechaza.
2. Verifica si los números de C suman K, y acepta sii se cumple.

Todo par (S, K) perteneciente a SUM-SUB cuenta con un certificado sucinto C. Y M tarda tiempo polinomial: conjuntamente, los bloques 1 y 2 requieren $O(|S|^2) = O(n^2)$ pasos (la verificación del bloque 1 tarda tiempo $O(|S|^2)$, y cada suma del bloque 2 y la verificación final, tiempo $O(|S|)$).

Fin del ejemplo

Ejemplo 6.9. El lenguaje CH = {G | G es un grafo que tiene un *circuito de Hamilton*} representa el *problema del circuito hamiltoniano*. El problema consiste en determinar si un grafo G = (V, E) tiene un circuito de Hamilton, que es una permutación $(i_1, \ldots, i_m)$ de los m vértices de V (es decir, de (1, …, m)), tal que (i_1, i_2), (i_2, i_3), …, (i_{m-1}, i_m), (i_m, i_1) son arcos de E. En palabras, un grafo G tiene un circuito de Hamilton C si C recorre todos los vértices de G sin repetir ninguno salvo el primero al final. El único algoritmo conocido para decidir CH, dado un grafo G, es probar con todas las permutaciones de V y chequear cada vez si determinan un circuito de Hamilton de G, por lo que en el peor caso efectúa m! iteraciones: $m.(m-1)\ldots2.1 = O(m^m) = O(|G|^{|G|}) = O(n^n)$. De esta forma, CH no estaría en P. Veamos que CH pertenece a NP:

Sea la siguiente MT M. Dados un grafo G y una secuencia de vértices C, con $|C| \leq |V|$, M hace:

1. Verifica si C es una permutación de V (es decir, de (1, …, m)). Si no, rechaza.
2. Dada $C = (i_1, \ldots, i_m)$, verifica si (i_1, i_2), (i_2, i_3), …, (i_{m-1}, i_m), (i_m, i_1) son arcos de E, y acepta sii se cumple.

Todo grafo G perteneciente a CH cuenta con un certificado sucinto C. Y M tarda tiempo polinomial: en el bloque 1 ejecuta $O(|V|^2) = O(n^2)$ pasos, y en el bloque 2, $O(|V|.|E|) = O(n^2)$ pasos.

Fin del ejemplo

Ejemplo 6.10. El lenguaje CLIQUE = {(G, K) | G es un grafo que tiene un *clique* de tamaño K} representa el *problema del clique*, que consiste en determinar si un grafo tiene un clique de tamaño K, es decir un subgrafo completo de K vértices. El único algoritmo que se conoce para decidir CLIQUE, dado un grafo G = (V, E) con |V| = m, es probar con todos los subconjuntos de K vértices de V y chequear cada vez si los vértices de los mismos determinan un subgrafo completo de G. Así, las iteraciones del algoritmo pueden sumar combinaciones de m tomadas

de a K, es decir: $m!/((m - K)!.K!)) = m.(m - 1)\ldots(m - K + 1)/K! = O(m^m) = O(|G|^{|G|}) = O(n^n)$. Por lo tanto, CLIQUE no estaría en P. Probamos a continuación que CLIQUE pertenece a NP:

Sea la siguiente MT M. Dados un par (G, K) y un conjunto de vértices C, con $|C| \leq |V|$, la MT M hace:

1. Verifica si C es un subconjunto de K vértices de V. Si no, rechaza.
2. Verifica si todo par de vértices de C determina un arco de E, y acepta sii se cumple.

Todo par (G, K) de CLIQUE cuenta con un certificado sucinto C. Y M tarda tiempo polinomial: en el bloque 1 hace $O(|V|^2) = O(n^2)$ pasos, y en el bloque 2, $O(|V^2|.|E|) = O(n^3)$ pasos.

Fin del ejemplo

Ejercicio. El lenguaje $CLIQUE_K = \{G \mid G$ es un grafo que tiene un clique de tamaño K$\}$ representa una variante del problema del clique, en la que el tamaño del clique no forma parte de las instancias, es decir que es una constante. Probar que en este caso el lenguaje pertenece a P.

Como dijimos, la clase NP se puede definir alternativamente como la clase de los lenguajes decidibles en tiempo polinomial por máquinas de Turing no determinísticas. Antes de probar la equivalencia de esta definición con la definición original, mostramos un ejemplo de prueba de pertenencia a la clase NP empleándola (en realidad, en el ejemplo 6.2 ya probamos así la pertenencia a NP del lenguaje SAT):

Ejemplo 6.11. Volvemos al lenguaje CLIQUE del ejemplo 6.10. Vamos a construir una MT no determinística que lo decide eficientemente, y así demostraremos otra vez, de otro modo, que CLIQUE pertenece a NP. Sea la siguiente MTN M, que dado un par (G, K), con G = (V, E), hace:

1. Genera no determinísticamente un conjunto C de vértices, con $|C| \leq |V|$.
2. Verifica si C es un subconjunto de K vértices de V. Si no, rechaza.
3. Verifica si todo par de vértices de C define un arco de E, y acepta sii se cumple.

En la figura 6.6 ilustramos el comportamiento de la máquina construida.

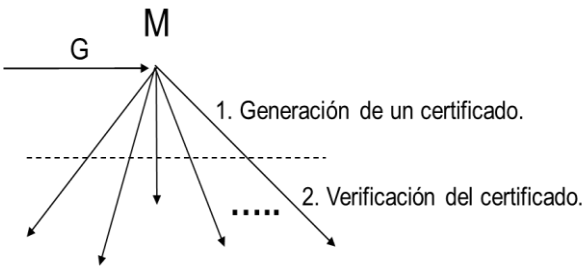

Figura 6.6. La MTN M decide el lenguaje CLIQUE en tiempo polinomial.

La MTN M decide CLIQUE: genera no determinísticamente un posible certificado sucinto C (que en el ejemplo 6.10 el verificador eficiente recibe como cadena de entrada adicional), y verifica si corresponde efectivamente a un clique de tamaño K del grafo de entrada G. Y lo hace en tiempo polinomial: el bloque 1 ejecuta O(n) pasos, y ya mostramos en el ejemplo 6.10 que los bloques 2 y 3 suman otros $O(n^3)$ pasos.

Fin del ejemplo

Ahora probamos la equivalencia de las dos definiciones de NP:

Teorema 6.1. Un lenguaje L pertenece a la clase NP sii existe una máquina de Turing no determinística que lo decide en tiempo polinomial.

Prueba. La implicación hacia la derecha, de la definición de NP con un verificador eficiente a su definición alternativa con una máquina de Turing no determinística que decide eficientemente, se prueba fácilmente construyendo una máquina que genera no determinísticamente un posible certificado sucinto y verifica si efectivamente lo es, como hicimos en el ejemplo 6.11.

Ejercicio. Probar la implicación hacia la derecha.

Por su parte, la prueba de la implicación recíproca se basa en la utilización de *discriminantes*, como describimos en el capítulo 2 para simular determinísticamente una máquina de Turing no determinística (teorema 2.2), pero ahora definidos en términos del recorrido DFS (en profundidad) en vez del recorrido BFS (a lo ancho) del árbol de computaciones de la máquina.

Formalmente, a partir de una MTN M_{NL} que decide el lenguaje L en tiempo polinomial, se puede construir una MTD M_{DL} que verifica la pertenencia de una cadena a L en tiempo polinomial: dada una entrada w, M_{DL} puede simular cualquier computación de M_{NL} a partir de w, con la ayuda de una cadena adicional x que indica las alternativas no determinísticas de dicha computación. Por ejemplo, utilizando la cadena x = 13214…, M_{DL} puede simular la computación de M_{NL} que primero elige la primera alternativa no determinística, después la tercera alternativa, después la segunda, etc. La cadena x es sucinta porque M_{NL} tarda tiempo polinomial, y M_{DL} acepta w (en tiempo polinomial) sólo si alguna computación de M_{NL} la acepta.

Fin del teorema

El problema P vs NP

Más allá de la imposibilidad, hasta el momento, de su demostración, la conjetura ampliamente aceptada es que P ≠ NP, reforzada por la existencia de miles de lenguajes de la clase NP que al día de hoy son decidibles únicamente con algoritmos de fuerza bruta, de tiempo exponencial. Resulta utópica la idea de que todos estos algoritmos puedan sustituirse alguna vez por otros

que sean eficientes (enormes árboles de decisión, propios de la inteligencia artificial, podrían recorrerse de manera exhaustiva eficientemente; contaríamos con demostradores automáticos de teoremas de tiempo polinomial para muchos sistemas axiomáticos de interés, etc).

Entre esos miles de lenguajes puede haber lenguajes de la clase P. La cuestión es encontrarlos, lo que requiere insoslayablemente conocer profundamente los problemas que representan, además de ingenio, experiencia y metodología (estrategias, métodos o técnicas del tipo *dividir y conquistar*, *programación dinámica*, *algoritmos codiciosos*, *problema dual*, *reducciones*, etc). Y es evidente que en la algorítmica, con un derrotero histórico aún breve, nos queda muchísimo por aprender, sobre todo entender por qué algunos problemas resultan más difíciles que otros. Podemos comprobarlo, sin ir más lejos, revisando algunos de sus numerosos resultados, cuanto menos curiosos:

- La dificultad de muchos problemas, en general difíciles, se reduce drásticamente cuando se *especializan*. Lo vimos en el problema de satisfactibilidad: el lenguaje SAT, con fórmulas booleanas arbitrarias, no estaría en P, pero en cambio sí lo está el lenguaje 2-SAT, con fórmulas booleanas en FNC con cláusulas con dos literales. Ya con cláusulas de tres literales, la tratabilidad volvería a perderse.
- Algo similar con los números dos y tres ocurre en el *problema de coloración de grafos*, que consiste en determinar si con K colores se pueden colorear los vértices de un grafo, de modo tal que nunca dos vértices adyacentes queden con el mismo color: cuando K = 2, el problema se decide en tiempo polinomial, y en cambio para K = 3, no se conoce algoritmo eficiente alguno. No obstante, a diferencia del caso anterior, a partir de K = 4, si los grafos son *planares*, es decir si pueden dibujarse en el plano sin que sus arcos se crucen, el problema siempre tiene solución (de acuerdo al Teorema de los Cuatro Colores, probado en 1976 por computadora, motivo por el cual generó no pocas controversias en el ambiente matemático).
- Otros problemas, al contrario de los anteriores, se tornan más difíciles cuando se especializan. Un caso es la *programación lineal*: decidir si un sistema de inecuaciones lineales tiene solución en el dominio de los números reales es eficiente, lo que no sucedería si se requiere que las soluciones sean exclusivamente de números enteros (variante conocida como *programación lineal entera*).
- Existen problemas con especificaciones muy similares, pero que se resuelven con algoritmos de eficiencia muy diferente. Un ejemplo lo constituyen el problema del circuito hamiltoniano y el problema del *circuito euleriano*: el primero, que ya analizamos, no tendría resolución de tiempo polinomial, mientras que el segundo, que consiste en determinar si un grafo tiene un *circuito de Euler* (recorrido de todos los arcos del grafo sin repeticiones, desde y hasta un mismo vértice), se resuelve eficientemente, verificando una condición que deben satisfacer los vértices. Otro ejemplo corresponde a los cálculos del *determinante* y la *permanente* de una matriz cuadrada: el cálculo del determinante se puede llevar a cabo en tiempo polinomial, en tanto que el de la permanente parecería ser mucho más difícil.

Frente al panorama actual de abundancia de problemas abiertos en la complejidad temporal (en realidad, en la complejidad computacional en general), que contrasta con las certezas de la computabilidad, se ha configurado un escenario de estudio con distintas herramientas complementarias, como el concepto de *completitud* y una variedad de aproximaciones más de naturaleza *estructural* que con foco en problemas particulares, definidas considerando distintas jerarquías y distintos modelos computacionales. Dichas herramientas permiten formular enunciados *condicionales*, con *conjeturas*. La idea es relacionar resultados, apuntando a esclarecer algunos a partir del esclarecimiento de otros. Este esquema de razonamiento ya podremos percibirlo en el capítulo siguiente.

Notas adicionales

Incluimos dos teoremas que caracterizan a la jerarquía temporal. El primero, conocido como Teorema de la Jerarquía Temporal, prueba que la jerarquía temporal es densa, y el segundo, que existen lenguajes recursivos decidibles en tiempo mínimamente exponencial. Los dos teoremas se demuestran por diagonalización.

El Teorema de la Jerarquía Temporal

El siguiente teorema establece que la jerarquía temporal es densa, es decir que para toda clase temporal de la jerarquía existe otra clase temporal que la incluye estrictamente (por el Teorema de Aceleración Lineal, el salto entre las funciones temporales asociadas a una y otra clase debe ser mayor que un factor constante).

Teorema 6.2. Dadas dos clases temporales TIME($T_1(n)$) y TIME($T_2(n)$) tales que $T_1(n)$ y $T_2(n)$ son funciones tiempo-construibles y $T_2(n) > \log_2 T_1(n) . T_1(n)$ cuando n tiende a infinito, se cumple que la clase temporal TIME(T(n)), siendo T(n) el máximo entre $T_1(n)$ y $T_2(n)$ para todo n, incluye estrictamente a TIME($T_1(n)$).

Prueba. Vamos a construir una máquina de Turing de tiempo de ejecución $T_2(n)$ que decide un lenguaje que no decide ninguna máquina de Turing de tiempo de ejecución $T_1(n)$. Así probaremos que existe un lenguaje en TIME($T_2(n)$) – TIME($T_1(n)$), y por lo tanto, de acuerdo a la definición de la función T(n), que la clase TIME(T(n)) incluye estrictamente a TIME($T_1(n)$).

Proponemos la siguiente MT M: dada una entrada w, con |w| = n, M la asume como el código $<M_w>$ de una MT M_w (si la cadena w no es un código válido la rechaza), ejecuta $T_2(n)$ pasos de M_w a partir de $<M_w>$, lo que es posible porque $T_2(n)$ es una función tiempo-construible, y acepta sólo si M_w se ejecuta hasta el final y rechaza.

Veamos que M es la MT que buscamos. Dado que M ejecuta $T_2(n)$ pasos de M_w, si el tiempo de ejecución de M_w es $T_1(n)$, siendo $T_2(n) > \log_2 T_1(n) \cdot T_1(n)$ M puede simular M_w completamente y responder lo contrario. Y como esto vale para toda entrada w, entonces el lenguaje que decide M difiere de todos los lenguajes decidibles por máquinas de Turing de tiempo de ejecución $T_1(n)$, lo que significa que el lenguaje pertenece a $TIME(T_2(n)) - TIME(T_1(n))$.

Para completar la prueba faltan hacer tres aclaraciones:

- No alcanza con $T_2(n) > T_1(n)$, sino que se requiere $T_2(n) > \log_2 T_1(n) \cdot T_1(n)$, porque las MT M_w que tiene que simular M pueden tener cualquier cantidad de cintas. Sabemos que M lo puede hacer con dos cintas, según comentamos luego del teorema 2.1, pero al precio de un retardo del orden $\log_2 T_1(n) \cdot T_1(n)$, lo que explica la relación requerida (también comentamos que la simulación se puede hacer con una sola cinta pero a un precio mayor, un retardo del orden cuadrático).
- M tiene que contemplar además que el alfabeto de las MT M_w puede tener cualquier cantidad de símbolos. En este caso alcanza con un factor constante c que depende de M_w (un símbolo del alfabeto de M_w se puede representar con c símbolos del alfabeto de M).
- Puede suceder que hasta un cierto n se cumpla $T_2(n) \leq c \cdot \log_2 T_1(n) \cdot T_1(n)$, pero esto no invalida la prueba, porque los códigos de máquinas de Turing pueden estar precedidos por cualquier cantidad de ceros, tal como lo señalamos en el capítulo 4 cuando describimos la codificación de las máquinas de Turing, y así M siempre va a encontrar algún código $<M_w>$ de toda MT M_w de tiempo de ejecución $T_1(n)$ lo suficientemente grande que satisfaga la relación $T_2(|<M_w>|) > c \cdot \log_2 T_1(|<M_w>|) \cdot T_1(|<M_w>|)$.

Fin del teorema

Prueba de que existe un lenguaje recursivo no perteneciente a la clase P

Por la densidad de la jerarquía temporal, es obvio que deben existir lenguajes recursivos fuera de la clase P. El siguiente teorema define un lenguaje concreto con estas características.

Teorema 6.3. El lenguaje $L = \{<M> \mid M \text{ acepta } <M> \text{ en } 2^{|<M>|} \text{ pasos}\}$ pertenece a R – P.

Prueba. Intuitivamente, decidir si un código <M> pertenece al lenguaje L no puede llevar en general menos de $2^{|<M>|}$ pasos. La prueba de su recursividad es muy sencilla.

Ejercicio. Probar que el lenguaje L es recursivo.

Para probar que $L \notin P$ supondremos lo contrario y llegaremos a un par de contradicciones. Partimos entonces de que se cumple $L \in P$. Como la clase P es cerrada con respecto al complemento, también se cumple $L^C \in P$, siendo $L^C = \{<M> \mid M \text{ no acepta } <M> \text{ en } 2^{|<M>|} \text{ pasos}\}$.

Sea M^C una MT que decide L^C en tiempo polinomial. Veamos qué sucede cuando relacionamos a la MT M^C con su propio código $<M^C>$ en el lenguaje L^C:

- Si $<M^C> \in L^C$, por la suposición M^C acepta $<M^C>$ en $|<M^C>|^k$ pasos, con k constante, y así, M^C acepta $<M^C>$ en $2^{|<M^C>|}$ pasos (si $|<M^C>|$ es lo suficientemente grande, lo que siempre se puede lograr anteponiéndole ceros, se cumple $|<M^C>|^k < 2^{|<M^C>|}$). Pero entonces, de acuerdo a la definición de L^C, $<M^C> \notin L^C$ (contradicción).
- Si $<M^C> \notin L^C$, por la suposición M^C no acepta $<M^C>$, y por lo tanto no acepta $<M^C>$ en $2^{|<M^C>|}$ pasos. Pero entonces, de acuerdo a la definición de L^C, $<M^C> \in L^C$ (contradicción).

De esta manera, la MT M^C no puede existir, lo que significa que $L^C \notin P$, y por lo tanto $L \notin P$.
Fin del teorema

Observaciones finales

- La máquina de Turing determinística con cualquier cantidad de cintas, en conjunto con una representación de cadenas basada en una codificación de números distinta de la codificación unaria, conforman un escenario robusto para el estudio de la complejidad temporal.
- La clase P es la clase de los lenguajes decidibles en tiempo polinomial. Se la considera la clase de los lenguajes tratables de la jerarquía temporal.
- La clase NP es la clase de los lenguajes que cuentan con verificadores eficientes. También se define como la clase de los lenguajes decidibles en tiempo polinomial con máquinas de Turing no determinísticas.
- NP incluye miles de lenguajes de interés computacional, que al día de hoy sólo pueden ser decididos por medio de algoritmos de fuerza bruta, en tiempo exponencial.
- La conjetura ampliamente aceptada es que $P \neq NP$. Intuitivamente, obtener una solución es más difícil que verificarla. La conjetura tiene más de 50 años, y no parece que pueda demostrarse o refutarse en el corto plazo.

Referencias

La complejidad computacional comenzó a tratarse en los años 1950. En ese entonces, fundamentalmente A. Grzegorczyk, M. Rabin y algunos investigadores de la Unión Soviética empezaron a analizar clases de funciones recursivas desde el punto de vista de la dificultad para calcularlas. Ya hacían referencia al tiempo polinomial y a los algoritmos de fuerza bruta.

De esa misma época es la famosa carta que K. Gödel le envió a J. von Neumann (carta que trascendió muchos años más tarde), en la que le preguntaba en qué tiempo una máquina de

Turing podía decidir la existencia de una prueba con una longitud determinada para demostrar una fórmula del cálculo de predicados. Gödel observaba que si el tiempo resultaba polinomial (lineal o cuadrático) con respecto a la longitud fijada, entonces no era tan grave la indecidibilidad del Entscheidungsproblem, dado que su versión finita podía resolverse eficientemente.

La sistematización del estudio de la complejidad computacional llegó una década más tarde. En el caso particular de la complejidad temporal, la publicación que se considera fundacional es (Hartmanis y Stearns, 1965). Trabajos relevantes simultáneos fueron (Cobham, 1964), (Edmonds, 1965), (Edmonds, 1966-1967a) y (Edmonds, 1966-1967b), que instalaron el concepto del tiempo polinomial como propiedad deseable de los algoritmos.

El tiempo de ejecución y el espacio ocupado por una máquina de Turing son las métricas de complejidad computacional más habituales e intuitivas. Otras métricas relacionadas con las máquinas de Turing son, por el lado de las métricas dinámicas, la cantidad de cambios de dirección de sus cabezales, y por el lado de las métricas estáticas, su complejidad estructural (tamaño del alfabeto, cantidad de estados, cantidad de cintas, tipos de movimientos, tipos de cintas, etc). Ver por ejemplo el capítulo 12 de (Hopcroft y Ullman, 1979). En dicho capítulo se presentan además distintos teoremas sobre la jerarquía temporal.

Las pruebas de pertenencia a las clases P y NP que presentamos son adaptaciones de las que se describen en el capítulo 4 de (Bovet y Crescenzi, 1994), el capítulo 7 de (Sipser, 1997) y el capítulo 6 de (Lewis y Papadimitriou, 1998). En algunos casos, al igual que hicimos en la parte de computabiidad, para simplificar omitimos el análisis sintáctico de las cadenas de entrada de las máquinas de Turing, en general sencillo. Aún si fuera complejo, una primera aproximación sólo con cadenas sintácticamente correctas facilita la construcción de los algoritmos. Así restringidos, los problemas se conocen como *problemas de promesa* (*promise problems*). El capítulo 2 de (Goldreich, 2008) incluye una sección sobre ellos.

A lo largo de los años se lograron importantes avances con respecto al entendimiento de la dificultad, en términos de la complejidad temporal, de varios problemas de interés computacional, como el producto de matrices, el isomorfismo de grafos y el problema de primalidad. En el prefacio y el capítulo 5 de (Aaronson, 2013) se profundiza sobre esto.

El problema de primalidad en particular, consistente en determinar si un número es primo, tuvo una evolución muy interesante: en 1975 se probó su pertenencia a la clase NP, entre 1977 y 1992 se encontraron algoritmos probabilísticos eficientes para resolverlo, y finalmente en 2002 se demostró que pertenece a la clase P (Agrawal, Kayal y Saxena, 2004).

También, y sobre todo en los últimos años, se avanzó sobremanera en la elaboración de distintas alternativas metodológicas para encarar el problema P vs NP. Un ejemplo para destacar es la *complejidad descriptiva* (Immerman, 1999), la cual permite definir clases de lenguajes sin apelar a la noción de máquina, y refleja la estrecha relación que existe entre la complejidad computacional y la lógica. Entre sus aportes más relevantes figura la identificación de la clase NP con la clase de los problemas expresables con lógica existencial de segundo orden (Teorema de Fagin). Esto quiere decir que si se encontrara una lógica menos expresiva para especificar los lenguajes de la clase P, se probaría la conjetura $P \neq NP$.

Otras lecturas recomendadas:

La serie de artículos (Gasarch, 2002), (Gasarch, 2012) y (Gasarch, 2019): describen tres encuestas a referentes de la complejidad computacional, realizadas en un espacio de casi veinte años, acerca del problema P vs NP. Las preguntas se centraron especialmente en la igualdad o desigualdad de las clases, y en la técnica utilizada y la fecha estimada para la resolución del problema, en uno u otro sentido. En la encuesta más reciente, en lo que hace a la franja de los expertos, el 99% votó a favor de $P \neq NP$, el 40% indicó que el problema se resolverá con una técnica matemática hoy inexistente, y el 55% predijo que su resolución será antes del año 2100.

(Sipser, 1992): sobre la historia del problema P vs NP.

Los capítulos 3 y 6 de (Moore y Mertens, 2011): en el capítulo 3 se describen varias estrategias de construcción de algoritmos, y se analiza por qué funcionan bien para algunos problemas y no para otros, y en el capítulo 6 se tratan distintos aspectos de la complejidad temporal, como la facilidad para encontrar cotas temporales superiores, la dificultad para encontrar cotas temporales inferiores, y la utilidad de la diagonalización.

Ejercicios

6.1 Probar que si $T_1(n) = O(T_2(n))$, entonces $TIME(T_1(n)) \subseteq TIME(T_2(n))$.

6.2 Probar que las siguientes funciones son tiempo-construibles:

a. $T(n) = n$.
b. $T(n) = n^2$.
c. $T(n) = 2^n$.

6.3 Probar que si $T_1(n)$ y $T_2(n)$ son funciones tiempo-construibles, también lo son las funciones $T_s(n) = T_1(n) + T_2(n)$ y $T_p(n) = T_1(n).T_2(n)$.

6.4 Indicar en qué casos se cumple $TIME(T_1(n)) \subset TIME(T_2(n))$:

a. $T_1(n) = n^k$ y $T_2(n) = n^{k+1}$, con $k \geq 0$.
b. $T_1(n) = 2^n$ y $T_2(n) = 2^{n+1}$.
c. $T_1(n) = n^k$ y $T_2(n) = 2^n$, con $k \geq 0$.

Comentario: la relación entre las clases $TIME(T_1(n))$ y $TIME(T_2(n))$ según la relación entre las funciones $T_1(n)$ y $T_2(n)$, se describe en una nota adicional.

6.5 Construir una MT M que genere los códigos de todas las MT M_i, según el orden canónico, tales que a partir de $<M_i>$ tardan tiempo $T(|<M_i>|)$, siendo T(n) una función tiempo-construible.

6.6 Sea f una función computable en tiempo polinomial, y sea L un lenguaje perteneciente a la clase P. Probar que $f^{-1}(L) = \{w \mid f(w) \text{ pertenece a } L\}$ también pertenece a P. *Ayuda: si una función f se computa en tiempo polinomial, entonces* $|f(w)| \leq poly(|w|)$ *para todo w*.

6.7 Determinar si los siguientes lenguajes de fórmulas booleanas, con las restricciones que se indican, pertenecen a la clase P:

a. Las fórmulas booleanas están en *forma normal disyuntiva* (o *FND*) y son satisfactibles. Una fórmula booleana está en FND si es una disyunción de conjunciones de literales.
b. Las fórmulas booleanas están en FNC y son satisfactibles con a lo sumo diez variables con el valor de verdad *verdadero*.

6.8 Probar que la clase NP es cerrada con respecto a la intersección y la unión. *Comentario: se puede utilizar la definición de NP con máquinas de Turing no determinísticas*.

6.9 Probar que la clase EXP es cerrada con respecto al complemento.

6.10 Probar que los siguientes lenguajes pertenecen a la clase NP, utilizando la definición de NP con verificadores eficientes:

a. DOM = {(G, K) | G es un grafo que tiene un *conjunto dominante* de K vértices}. Un subconjunto de vértices C de un grafo G es un conjunto dominante de G sii todo otro vértice de G es adyacente a algún vértice de C.
b. ISO = {(G_1, G_2) | G_1 y G_2 son grafos *isomorfos*}. Dos grafos son isomorfos sii existe una biyección entre sus vértices que preserva su relación de adyacencia. En otras palabras, los grafos son iguales salvo por la denominación de sus arcos.
c. PVC = {(G, B) | G es un grafo completo ponderado que tiene un circuito de Hamilton tal que la suma de los valores asociados a sus arcos es menor o igual que B}, El nombre PVC se debe a que el lenguaje representa el *problema del viajante de comercio*.
d. IND = {(G, K) | G es un grafo que tiene un *conjunto independiente* de K vértices}. Un subconjunto de vértices C de un grafo G es un conjunto independiente de G sii todo par de vértices i y j de C cumplen que (i, j) no es un arco de G.
e. CV = {(G, K) | G es un grafo que tiene un *cubrimiento de vértices* de tamaño K}. Un subconjunto de vértices C de un grafo G es un cubrimiento de vértices de G sii al menos un extremo de cada arco de G está en C.
f. PARTICIÓN = {T | T es un conjunto de números naturales que se puede particionar en dos subconjuntos, de manera tal que las sumas de sus elementos coincidan}.

6.11 Comentar por qué los lenguajes del ejercicio anterior no estarían en P, ni sus complementos en NP.

6.12 El lenguaje FACT = {(N, M_1, M_2) | N, M_1 y M_2 son números naturales y N tiene un factor primo en el intervalo [M_1, M_2]} representa el *problema de factorización*. Comentar por qué FACT no estaría en P, y probar que tanto FACT como $FACT^C$ pertenecen a NP. *Ayuda para el caso de $FACT^C$: por el Teorema Fundamental de la Aritmética, todo número natural N mayor que 1 tiene una única forma de expresarse como producto de números primos, y la cantidad de números primos de la expresión es logarítmica con respecto a N*.

CAPÍTULO 7
NP-completitud

La diagonalización y la reducción son dos métodos fundamentales en la teoría de la computación. Ya lo comprobamos en la computabilidad, y comenzamos a hacerlo en la complejidad computacional con la diagonalización en el capítulo anterior.

A partir de este capítulo vamos a comprobar la importancia en la complejidad computacional de las reducciones, más precisamente de las reducciones *polinomiales*, es decir las reducciones computables en tiempo polinomial, que son las que se necesitan en este marco. Recurrimos a las reducciones polinomiales para estudiar un conjunto particular de lenguajes de la clase NP, los lenguajes *NP-completos*. Estos lenguajes tienen la propiedad de que todos los lenguajes de NP se reducen polinomialmente a ellos, lo que hace que constituyan un tema central en la complejidad temporal: si se encontrara un lenguaje NP-completo perteneciente a la clase P, entonces se estaría demostrando la igualdad P = NP. Es decir, P ≠ NP implica que un lenguaje NP-completo no pertenece a P. De este modo, la sospecha de la inexistencia de un algoritmo eficiente para decidir un lenguaje de NP, respaldada empíricamente por numerosos intentos infructuosos para obtenerlo, se refuerza sobremanera cuando se demuestra que el lenguaje es NP-completo (en la práctica, como la conjetura ampliamente aceptada es que P ≠ NP, el lenguaje queda *condenado* a no estar en P).

El esquema de razonamiento descripto recuerda el que mostramos en la parte de computabilidad del libro, cuando tratamos la RE-completitud de los lenguajes HP y L_U: todos los lenguajes de la clase RE se reducen a HP y L_U, por lo que si éstos fueran recursivos se cumpliría R = RE. La diferencia, claro está, es que la no pertenencia de ambos lenguajes a la clase R no depende de ninguna conjetura, sino que se prueba directamente (no existe un problema abierto R vs RE análogo al problema P vs NP).

La completitud es un concepto esencial. Permite identificar el grado de dificultad de toda una clase de lenguajes. En la complejidad computacional, en particular, se considera que una clase sin lenguajes completos no tiene razón de existir. En los capítulos siguientes analizaremos, en este sentido, distintas clases y jerarquías.

Pasamos a desarrollar las nociones mencionadas, comenzando por la descripción de las reducciones polinomiales. En este capítulo y el siguiente nos enfocamos en las m-reducciones polinomiales (o *Karp-reducciones*, o *reducciones poly-time*), postergando hasta el capítulo final la descripción de las Turing-reducciones polinomiales (o *Cook-reducciones*). Para simplificar, mientras no sea necesaria la distinción, llamaremos a las m-reducciones polinomiales directamente reducciones polinomiales.

Las reducciones polinomiales

Las reducciones polinomiales se definen como las reducciones generales, pero con el agregado de que se computan en tiempo polinomial.

Formalmente, se define que una *reducción polinomial* de un lenguaje L_1 a un lenguaje L_2 es una función $f : \Sigma^* \longrightarrow \Sigma^*$ que satisface dos condiciones, una condición de *complejidad temporal* y una condición de *correctitud*:

- Condición de complejidad temporal: existe una máquina de Turing M_f que la computa en tiempo polinomial.
- Condición de correctitud: asigna a toda cadena w que pertenece al lenguaje L_1 una cadena f(w) que pertenece al lenguaje L_2, y a toda cadena w que no pertenece a L_1 una cadena f(w) que no pertenece a L_2.

La notación $L_1 \leq_p L_2$ expresa que existe una reducción polinomial de L_1 a L_2.

El siguiente teorema, análogo al teorema 5.1 de las reducciones generales, formaliza la utilidad de las reducciones polinomiales:

Teorema 7.1

a. Si $L_1 \leq_p L_2$ y $L_2 \in P$, entonces $L_1 \in P$.
b. Si $L_1 \leq_p L_2$ y $L_2 \in NP$, entonces $L_1 \in NP$.

Prueba. En palabras, las clases P y NP son cerradas con respecto a las reducciones polinomiales, es decir que reduciendo polinomialmente un lenguaje L_1 a un lenguaje L_2 de P (NP), se prueba que L_1 pertenece a P (NP). Probamos la parte (a):

Si existen una MT M_f que reduce L_1 a L_2 en tiempo polinomial y una MT M_2 que decide L_2 en tiempo polinomial, entonces también existe una MT M_1 que decide L_1 en tiempo polinomial: dada una entrada w, M_1 primero ejecuta M_f y obtiene f(w), y luego ejecuta M_2 a partir de f(w) y responde como M_2. Ya demostramos previamente que M_1 decide L_1 (teorema 5.1). Veamos que lo hace en tiempo polinomial:

- A partir de w, la MT M_f ejecuta poly(|w|) pasos y obtiene f(w), que cumple $|f(w)| \leq poly(|w|)$, ya que en k pasos una máquina de Turing no recorre más que k celdas.
- Luego, a partir de f(w), la MT M_2 ejecuta poly(|f(w)|) pasos, que de acuerdo a lo anterior no son más que poly(|w|) pasos.
- En suma, a partir de w, la MT M_1 tarda tiempo polinomial.

La parte (b) se prueba de manera similar.

Ejercicio. Probar la parte (b) del teorema.

Fin del teorema

Aplicando el contrarrecíproco a las dos partes del teorema anterior, como hicimos con el teorema 5.1, se obtiene el siguiente corolario:

Corolario 7.1

a. Si $L_1 \leq L_2$ y $L_1 \notin P$, entonces $L_2 \notin P$.

b. Si $L_1 \leq L_2$ y $L_1 \notin NP$, entonces $L_2 \notin NP$.

El corolario 7.1 establece que en una reducción polinomial de un lenguaje L_1 a un lenguaje L_2, L_2 es *tan o más difícil* que L_1, en el sentido de la complejidad temporal. Concretamente, no puede ser que L_2 pertenezca a P y L_1 no, ni que L_2 pertenezca a NP y L_1 no.

Otra analogía con las reducciones generales es que también las reducciones polinomiales son reflexivas, transitivas y no simétricas (ejercicio 7.1).

Introducidas las reducciones polinomiales (mostramos varios ejemplos más adelante), presentamos a continuación los lenguajes NP-completos, el tema central del capítulo.

Los lenguajes NP-completos

Un lenguaje es *NP-completo* si satisface dos condiciones:

- Pertenece a la clase NP.
- Todos los lenguajes de NP se reducen polinomialmente a él. Se dice en este caso que el lenguaje es *NP-difícil*.

La figura 7.1 ilustra las dos condiciones que cumple un lenguaje NP-completo (f_i es una reducción polinomial de L_i a L).

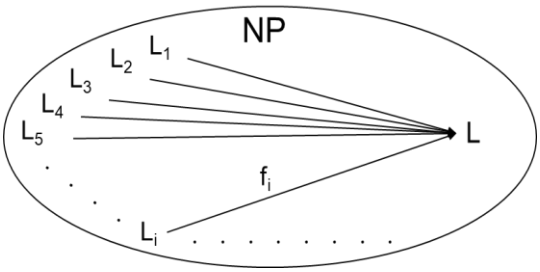

Figura 7.1. L es NP-completo: está en NP y todo lenguaje de NP se reduce polinomialmente a él.

La clase de los lenguajes NP-completos se denomina NPC. Notar que de acuerdo a su definición y al corolario 7.1, un lenguaje NP-completo es tan o más difícil que cualquier lenguaje de la clase NP, y como también pertenece a NP, identifica el grado de dificultad de toda la clase. Existen miles de lenguajes de este tipo, representantes de problemas sobre grafos, fórmulas booleanas, números, conjuntos, ecuaciones, autómatas, etc., todos de mucho interés computacional. Comenzamos a caracterizarlos con tres teoremas.

El primer teorema revela la importancia del concepto de NP-completitud en la complejidad temporal, que anticipamos en la introducción del capítulo:

Teorema 7.2. Sea L un lenguaje NP-completo. Si $L \in P$, entonces P = NP.

Prueba. El teorema establece que un lenguaje NP-completo no puede decidirse en tiempo polinomial a menos que se cumpla P = NP. La prueba es la siguiente:

Sean L un lenguaje NP-completo y L_1 un lenguaje cualquiera de NP. Por definición, se cumple $L_1 \leq_p L$. Si $L \in P$, entonces por propiedad de las reducciones polinomiales (teorema 7.1) se cumple también $L_1 \in P$. Y como esto vale para cualquier lenguaje L_1 de NP, llegamos a que $NP \subseteq P$, y por lo tanto P = NP.

Fin del teorema

De acuerdo al teorema anterior, asumiendo $P \neq NP$ podemos plantear una versión más detallada de la clase NP, incorporando a la clase NPC. La mostramos en la figura 7.2.

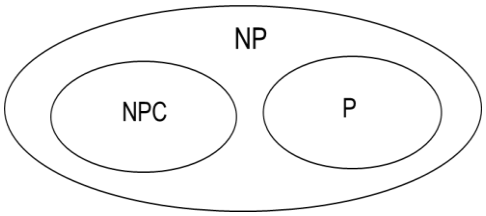

Figura 7.2. La clase NP y sus subclases P y NPC, asumiendo $P \neq NP$.

La región restante de NP que muestra la figura 7.2 reúne, por lo tanto, a sus lenguajes de *dificultad intermedia* (ni *los más fáciles* que son los de P, ni *los más difíciles* que son los de NPC). Asumiendo $P \neq NP$, se prueba que existe. La analizaremos en el próximo capítulo.

El segundo teorema para caracterizar a los lenguajes NP-completos describe cómo poblar la clase NPC (mecanismo que vamos a ejemplificar en la próxima sección):

Teorema 7.3. Sea L_1 un lenguaje de NP. Si existe un lenguaje NP-completo L tal que $L \leq_p L_1$, entonces L_1 también es NP-completo.

Prueba. El teorema establece que encontrando una reducción polinomial de un lenguaje NP-completo L a un lenguaje L_1 de NP, se prueba que también L_1 es NP-completo. La prueba es la siguiente:

Sean L un lenguaje NP-completo, L_1 un lenguaje de NP tal que $L \leq_p L_1$, y L_2 un lenguaje cualquiera de NP. Por definición, se cumple $L_2 \leq_p L$, y por la transitividad de las reducciones polinomiales, se cumple $L_2 \leq_p L_1$. Como esto vale para cualquier lenguaje L_2 de NP, llegamos a que todo lenguaje de NP se reduce polinomialmente a L_1, que sumado a que L_1 pertenece a NP significa que L_1 es NP-completo.

La figura 7.3 ilustra el mecanismo de poblamiento de NPC definido en el teorema (f_i es una reducción poilinomial de L_i a L y f es una reducción polinomial de L a L_1).

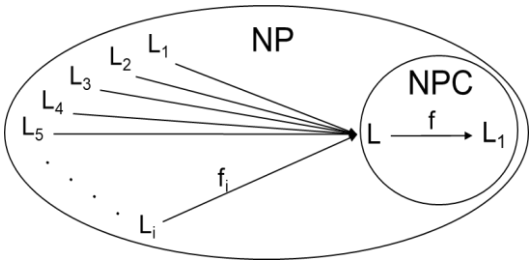

Figura 7.3. Si $L \in NPC$, $L_1 \in NP$ y $L \leq_p L_1$, entonces $L_1 \in NPC$.

Fin del teorema

Finalmente, el tercer teorema permite asegurar que efectivamente existen lenguajes NP-completos. Mencionamos que los hay por miles, describimos cómo poblar la clase NPC, pero todavía no encontramos formalmente ninguno. El teorema establece concretamente que el lenguaje SAT, representante del problema de satisfactibilidad, es NP-completo. Fue el primer lenguaje NP-completo que se encontró. La prueba la presentaron, a comienzos de la década de 1970, de manera independiente y prácticamente en simultáneo, S. Cook en los Estados Unidos y L. Levin en la Unión Soviética (curiosamente como A. Church y A. Turing probaron en 1936 la indecibilidad del Entscheidungsproblem):

Teorema 7.4. El lenguaje SAT es NP-completo.

Prueba. En los ejemplos 6.2 y 6.7 mostramos de dos maneras distintas que SAT pertenece a NP. Presentamos a continuación la idea general de una reducción polinomial, variante de las que se definieron en las pruebas originales, que prueba que SAT es NP-difícil, y así NP-completo (en una nota adicional la detallamos):

Al no haber ningún lenguaje NP-completo específico desde el cual reducir polinomialmente a SAT, se define una función de reducción f desde un lenguaje genérico L de NP, decidible en tiempo polinomial p(n) por una MTN M con una cinta (sin perder generalidad). La función f asigna a una cadena w una fórmula booleana φ_w:

$$f(w) = \varphi_w$$

tal que φ_w expresa una computación de M a partir de w. La idea es que φ_w sea satisfactible sólo si M acepta w (idea similar a la que empleó A. Turing en su prueba de indecibilidad del Entscheidungsproblem, que comentamos en el teorema 5.7). La figura 7.4 representa la función de reducción f propuesta.

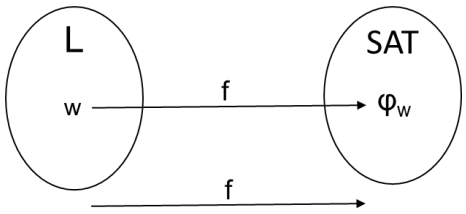

Figura 7.4. L es un lenguaje de NP, y w ∈ L sii la fórmula booleana φ_w es satisfactible.

Dado que el tiempo de ejecución de M es p(n), φ_w contempla una computación de a lo sumo p(n) + 1 configuraciones de p(n) símbolos cada una. φ_w expresa que la primera configuración contiene la entrada w, que una configuración sucede a la anterior según la relación de transición de M, y que la última configuración contiene el estado de aceptación. Para ello utiliza variables c_{ix}. La semántica de una variable c_{ix} es que es verdadera o falsa según en la posición i (una celda determinada de una configuración determinada) se encuentra o no el símbolo x, respectivamente. Se comprueba que f efectivamente reduce polinomialmente L a SAT:

Por un lado, f es computable en tiempo polinomial (f satisface la condición de complejidad temporal). Dada una entrada w con |w| = n, alcanzan $O(p^2(n))$ pasos para generar una fórmula booleana que exprese una computación de a lo sumo p(n) + 1 configuraciones de p(n) símbolos cada una.

Por otro lado, w ∈ L sii f(w) = φ_w ∈ SAT (f satisface la condición de correctitud). Si w ∈ L, entonces existe una computación de M que acepta w, y por lo tanto φ_w, con valores de verdad consistentes con las características de dicha computación, resulta verdadera, lo que significa que φ_w ∈ SAT. Y si φ_w ∈ SAT, entonces existe una asignación de valores de verdad que satisface φ_w, y de esta manera, la computación de M a partir de w que expresa φ_w con dicha asignación es de aceptación, lo que significa que w ∈ L.

Fin del teorema

Así, SAT está entre los lenguajes más difíciles de NP. Es lo máximo que podemos afirmar. Lo ideal sería diagonalizar y probar que SAT no está en P (con lo que P ≠ NP), como cuando encontramos por diagonalización un lenguaje recursivo L decidible en tiempo mínimamente

exponencial y probamos que P ≠ R (teorema 6.3), pero lamentablemente la diagonalización, útil para tratar lenguajes difíciles como L, no lo es para tratar los lenguajes NP-completos.

Ejemplos de lenguajes NP-completos

Probada la NP-completitud de SAT, la clase NPC comenzó a poblarse por medio de reducciones polinomiales, primero desde SAT y después desde otros lenguajes NP-completos. En los ejemplos siguientes describimos algunas reducciones polinomiales clásicas, basadas en distintas heurísticas.

Ejemplo 7.1. El lenguaje CSAT y el lenguaje 3-SAT son dos sublenguajes del lenguaje SAT también NP-completos. CSAT contiene las fórmulas booleanas de SAT en FNC, y 3-SAT, las fórmulas booleanas de CSAT con tres literales por cláusula. Las pruebas de pertenencia a NP de CSAT y 3-SAT difieren de la prueba de pertenencia a NP de SAT sólo en el análisis sintáctico inicial de las cadenas de entrada. Se puede probar que CSAT es NP-difícil con una reducción polinomial desde SAT. En este ejemplo probamos que 3-SAT es NP-difícil con una reducción polinomial desde CSAT.

Proponemos la siguiente función de reducción f (seguimos omitiendo, para simplificar, el análisis sintáctico de las cadenas de entrada):

$$f(\varphi) = \varphi^*$$

tal que $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_k$ es una fórmula booleana en FNC, y $\varphi^* = \varphi^*_1 \wedge \ldots \wedge \varphi^*_k$ es una fórmula booleana en FNC con tres literales por cláusula, que cumplen:

- Si φ_i tiene un solo literal l, entonces $\varphi^*_i = (l \vee l \vee l)$.
- Si φ_i tiene dos literales l_1 y l_2, entonces $\varphi^*_i = (l_1 \vee l_2 \vee l_1)$.
- Si φ_i tiene tres literales l_1, l_2 y l_3, entonces $\varphi^*_i = (l_1 \vee l_2 \vee l_3)$.
- Si φ_i tiene m > 3 literales l_1, …, l_m, entonces φ^*_i es una conjunción de cláusulas con m – 3 variables nuevas x_1, …, x_{m-3} que tiene la siguiente forma:

$$\varphi^*_i = (l_1 \vee l_2 \vee x_1) \wedge (l_3 \vee \neg x_1 \vee x_2) \wedge (l_4 \vee \neg x_2 \vee x_3) \wedge \ldots \wedge (l_{m-2} \vee \neg x_{m-4} \vee x_{m-3}) \wedge (l_{m-1} \vee l_m \vee \neg x_{m-3})$$

La función f es efectivamente una reducción polinomial de CSAT a 3-SAT. En tiempo polinomial, toda cláusula de φ de m literales se puede transformar en O(m) cláusulas de φ* de tres literales. Y se cumple $\varphi \in$ CSAT sii $\varphi^* \in$ 3-SAT:

- Si φ ∈ CSAT y $\mathcal{A}$ es una asignación de valores de verdad que satisface φ, se verifica que la siguiente asignación $\mathcal{A}$*, extensión de $\mathcal{A}$ sobre φ*, satisface φ*: $\mathcal{A}$* repite las asignaciones de $\mathcal{A}$ a los literales originales, y en cuanto a las variables nuevas surgidas de cada cláusula ($\ell_1 \vee \ldots$ v ℓ_m) con m > 3, si de izquierda a derecha ℓ_i es el primer literal con valor *verdadero* según $\mathcal{A}$, entonces $\mathcal{A}$* asigna el valor *verdadero* a las variables $x_1, \ldots, x_{i-2}$ y el valor *falso* a las variables restantes. De esta manera, llegamos a φ* ∈ 3-SAT.
- Si φ* ∈ 3-SAT y $\mathcal{A}^*$ es una asignación de valores de verdad que satisface φ*, la asignación $\mathcal{A}$ que resulta de restringir $\mathcal{A}^*$ a φ satisface φ. Asumiendo lo contrario llegamos a una contradicción: tomando el conjunto de cláusulas obtenido por la reducción a partir de cualquier cláusula ($\zeta_1 \vee \ldots$ v ζ_m) con m > 3, si dichos literales son falsos según $\mathcal{A}$* entonces las variables correspondientes $x_1, x_2, \ldots, x_{m-3}$ tienen que ser verdaderas según $\mathcal{A}$*, y también $\neg x_{m-3}$ (contradicción). De esta manera, llegamos a φ ∈ CSAT.

Fin del ejemplo

La reducción polinomial que acabamos de describir pertenece a una familia de reducciones polinomiales basadas en *reemplazos locales*. Se identifican componentes específicos de las cadenas del lenguaje NP-completo conocido y se los modifica para producir cadenas del lenguaje NP-completo nuevo. La reducción polinomial del próximo ejemplo pertenece a una familia de reducciones polinomiales más complejas. También se parte de componentes específicos de las cadenas del lenguaje desde el que se lleva a cabo la reducción, pero en lugar de transformarlos se *diseñan* componentes nuevos que se *interconectan* adecuadamente.

Ejemplo 7.2. El lenguaje 3-SAT es muy útil para probar la NP-completitud de numerosos lenguajes (como lo son en la computabilidad los lenguajes HP y L_U para probar la no recursividad). En este ejemplo recurrimos a 3-SAT para probar la NP-completitud del lenguaje CV = {(G, K) | G es un grafo que tiene un cubrimiento de vértices de tamaño K}, que representa el *problema del cubrimiento de vértices*, consistente en determinar si un grafo G tiene un cubrimiento de vértices de tamaño K, es decir un subconjunto C de K vértices tal que todo arco de G tiene un extremo en C (en el ejercicio 6.10.e pedimos probar que CV pertenece a NP). La siguiente reducción de 3-SAT a CV permite probar que CV es NP-difícil, y así que es NP-completo.

Proponemos la siguiente función de reducción f:

$$f(\varphi) = (G, 2K)$$

tal que φ es una fórmula booleana en FNC con tres literales por cláusula, K es la cantidad de cláusulas de φ, y G es un grafo que se construye de la siguiente forma (todavía consideramos sólo grafos no dirigidos):

- Por cada literal de cada cláusula de φ se crea un vértice en G.
- Todo par de vértices creados a partir de dos literales de una misma cláusula se unen por un arco (identificado como *enlace de tipo 1*). De esta manera, toda cláusula determina un triángulo con enlaces de tipo 1.
- Todo par de vértices creados a partir de dos literales x_i y $\neg x_i$ de distintas cláusulas también se unen por un arco (identificado como *enlace de tipo 2*).

Por ejemplo, en la figura 7.5 mostramos el grafo G asignado por la función de reducción f a la fórmula booleana $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$.

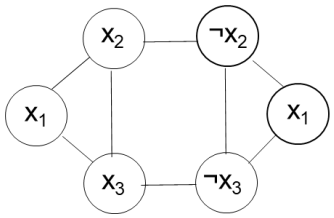

Figura 7.5. Grafo G asignado por la función f a la fórmula $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$.

En la figura 7.5, el conjunto de vértices $\{x_2, x_3, \neg x_2, \neg x_3\}$ es un cubrimiento de vértices de tamaño 4 del grafo G, el doble de la cantidad de cláusulas de φ, tal como se requiere.

La función f es efectivamente una reducción polinomial de 3-SAT a CV. En tiempo polinomial, por cada una de las K cláusulas se puede crear un triángulo, y por cada literal de cada cláusula se pueden crear O(K) arcos. Y se cumple $\varphi \in$ 3-SAT sii $(G, 2K) \in$ CV:

- Sean $\varphi \in$ 3-SAT, $\mathcal{A}$ una asignación de valores de verdad que satisface φ, y el siguiente conjunto C de 2K vértices del grafo G: incluye a todos los vértices asociados a los literales falsos de φ según $\mathcal{A}$, y si con esto no es suficiente para que tenga dos vértices de cada triángulo (por existir cláusulas de φ con dos o tres literales verdaderos según $\mathcal{A}$), incluye más vértices hasta completar el par. Se cumple que C es un cubrimiento de vértices de G: todo enlace de tipo 1 está cubierto porque C tiene dos vértices de cada triángulo formado con dichos enlaces, y todo enlace de tipo 2 también está cubierto porque uno de sus vértices está asociado a un literal falso según $\mathcal{A}$, el cual está en C. Así, $(G, 2K) \in$ CV.
- Sean $(G, 2K) \in$ CV, C un cubrimiento de vértices de tamaño 2K del grafo G, y la siguiente asignación de valores de verdad $\mathcal{A}$ a φ: a los literales asociados a los vértices que no están en C les asigna el valor *verdadero*, y al resto de los literales les asigna valores consistentes cualesquiera. Se cumple que $\mathcal{A}$ satisface φ: C está compuesto necesariamente por dos vértices de cada uno de los triángulos, y por lo tanto en toda cláusula hay un literal verdadero según $\mathcal{A}$. Además, $\mathcal{A}$ no tiene inconsistencias, porque si dos literales x_i y $\neg x_i$ de dos cláusulas

distintas son verdaderos según $\mathcal{A}$, significa que el enlace de tipo 2 asociado no está cubierto por C (contradicción). Así, φ ∈ 3-SAT.

Fin del ejemplo

La siguiente reducción polinomial es más sencilla que las anteriores. Pertenece a una tercera familia de reducciones polinomiales, que relacionan lenguajes que representan *problemas muy similares*.

Ejemplo 7.3. Vamos a definir una reducción polinomial del lenguaje CV, que probamos recién que es NP-completo, al lenguaje CLIQUE = {(G, K) | G es un grafo que tiene un clique de tamaño K}, correspondiente al *problema del clique*, el problema de determinar si un grafo tiene un clique de tamaño K, es decir un subgrafo completo de K vértices (en el ejemplo 6.10 demostramos que CLIQUE pertenece a NP). Con esta reducción demostraremos que CLIQUE es NP-difícil, y así que es NP-completo. Proponemos la siguiente función de reducción f:

$$f(G, K) = (G^C, m – K)$$

tal que G^C es el grafo complementario de G (tiene los vértices de G y los arcos que no tiene G), y m es la cantidad de vértices de G. Mostramos un ejemplo en la figura 7.6.

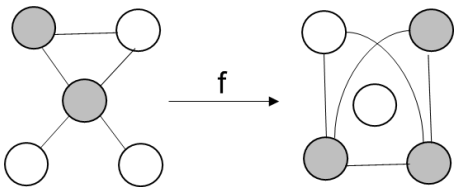

Figura 7.6. f asigna a un grafo G con un cubrimiento de tamaño 2, el grafo G^C con un clique de tamaño 3.

En la figura 7.6, G tiene m = 5 vértices y un cubrimiento de vértices de tamaño K = 2, y G^C tiene un clique de tamaño m – K = 3, tal como se requiere.

La función f es efectivamente una reducción polinomial de CV a CLIQUE.

Ejercicio. Comprobar que f se computa en tiempo polinomial.

Veamos que se cumple (G, K) ∈ CV sii (G^C, m – K) ∈ CLIQUE. Probamos sólo la implicación hacia la derecha, la prueba de la implicación recíproca es muy similar:

Dado un par (G, K) ∈ CV, con G = (V, E) de m vértices, si C es un cubrimiento de vértices de tamaño K de G, entonces V – C constituye un clique de tamaño m – K de G^C:

- El tamaño de V – C es m – K.

- V – C constituye un clique de G^C: Supondremos que no y llegaremos a una contradicción. Si V – C no constituye un clique de G^C, entonces existen dos vértices i y j de V – C tales que el arco (i, j) no está en G^C, o lo que es lo mismo, que está en G. Por lo tanto, G tiene un arco cuyos dos extremos no están en C, y así C no es un cubrimiento de vértices de G (contradicción). En consecuencia, V – C constituye un clique de G^C.

Ejercicio. Probar la implicación recíproca: si $(G^C, m - K) \in$ CLIQUE entonces $(G, K) \in$ CV.
Fin del ejemplo

Para definir la última reducción polinomial de esta sección recurrimos a una cuarta heurística (en las notas adicionales y los ejercicios finales presentamos más ejemplos).

Ejemplo 7.4. El lenguaje SUB-ISO = {(G_1, G_2) | G_1 y G_2 son dos grafos y G_1 es isomorfo a un subgrafo de G_2} representa una variante del *problema del isomorfismo de grafos*, problema este último que consiste en determinar si dos grafos son isomorfos, es decir si existe una biyección entre sus vértices que preserva su relación de adyacencia, o dicho de otra forma, si los grafos son iguales salvo por la denominación de sus arcos. SUB-ISO es NP-completo:

La prueba de que SUB-ISO pertenece a NP es prácticamente la misma que la prueba de que ISO pertenece a NP (la cual pedimos en el ejercicio 6.10.b). Para probar que SUB-ISO es NP-difícil, y así que es NP-completo, vamos a definir una reducción polinomial desde el lenguaje CLIQUE, cuya NP-completitud probamos en el ejemplo anterior.

Proponemos la siguiente función de reducción f:

$$f(G, K) = (G_K, G)$$

tal que G_K es un grafo completo de K vértices.

La función f es efectivamente una reducción polinomial de CLIQUE a SUB-ISO:

En tiempo polinomial se puede generar un grafo completo de K vértices y se puede copiar G.

Y se cumple $(G, K) \in$ CLIQUE sii $(G_K, G) \in$ SUB-ISO: $(G, K) \in$ CLIQUE sii G tiene un subgrafo completo de K vértices sii G tiene un subgrafo isomorfo a G_K sii $(G_K, G) \in$ SUB-ISO.
Fin del ejemplo

La reducción polinomial del último ejemplo pertenece a una cuarta familia, basada en la heurística de *restricción*. La idea es definir las cadenas del lenguaje NP-completo nuevo restringidas a las características de las cadenas del lenguaje NP-completo conocido (en el ejemplo, los isomorfismos que se obtienen son entre cliques).

Otro aspecto para destacar del último ejemplo es que curiosamente, a diferencia del lenguaje SUB-ISO, el lenguaje ISO no sería NP-completo. En efecto, al día de hoy no se ha podido reducir polinomialmente ningún lenguaje NP-completo a ISO (ni tampoco se ha podido encontrar ningún algoritmo polinomial que lo decida, y así ISO sería uno de los lenguajes de NP de dificultad

intermedia que mencionamos antes). Intuitivamente, SUB-ISO sería más difícil que ISO por su *redundancia de información*: modificando de varias maneras el grafo G en un par (G_K, G) de SUB-ISO, G_K puede seguir siendo isomorfo a un subgrafo de G, pero en cambio cualquier alteración no trivial a uno de los grafos de un par (G_1, G_2) de ISO tiene impacto. El tamaño de un lenguaje, como veremos enseguida, estaría directamente relacionado con su grado de dificultad, en el sentido de la complejidad temporal.

Reflexión. Una idea fascinante que podría plantearse considerando los miles de ejemplos que existen como los que acabamos de mostrar, es que tras la enorme cantidad de lenguajes NP-completos en realidad se esconden muchos menos, o expresado de otro modo, que en el ámbito de la NP-completitud existen pocas familias de lenguajes, cada una con muchos lenguajes que son en verdad un *único lenguaje* pero especificado de múltiples maneras y en múltiples disciplinas (teoría de grafos, lógica, aritmética, teoría de conjuntos, álgebra, teoría de autómatas, etc). Una observación que refuerza esta interpretación es que tomando dos lenguajes cualesquiera de gran parte de los lenguajes NP-completos conocidos, se cumple que los certificados de las cadenas de uno se pueden transformar eficientemente en los certificados de las cadenas del otro. En la siguiente sección y el último capítulo mencionamos otras observaciones de similares características.
Fin de la reflexión

Dos propiedades distintivas de los lenguajes NP-completos

En los miles de lenguajes NP-completos que se conocen, se observan dos características que refuerzan la conjetura P ≠ NP:

La primera característica es que entre todo par de lenguajes NP-completos conocidos existe una biyección, tal que tanto ella como su inversa son computables en tiempo polinomial. Por definición, entre dos lenguajes NP-completos cualesquiera L_1 y L_2 existen una reducción polinomial de L_1 a L_2 y una reducción polinomial de L_2 a L_1, pero dichas reducciones no tienen por qué ser una la inversa de la otra. Sin embargo, cualquiera sea el par de lenguajes NP-completos conocidos que se consideren, se puede construir una biyección con la característica mencionada. Se dice en este caso que los lenguajes son *p-isomorfos* (la p es por polinomial). La Conjetura de Berman-Hartmanis establece que los lenguajes NP-completos son p-isomorfos dos a dos. Es una conjetura muy importante, ya que si se cumple significa que P ≠ NP: por el contrarrecíproco, si P = NP entonces P = NPC (la inclusión P $\subseteq$ NPC se obtiene a partir del ejercicio 7.2), y así, NPC incluye lenguajes finitos e infinitos, lo que invalida la conjetura de Berman-Hartmanis dado que un lenguaje finito no puede ser p-isomorfo a un lenguaje infinito. Existe también una contraconjetura, que plantea la posibilidad de que existan lenguajes NP-completos no p-isomorfos al lenguaje SAT.

La segunda característica observada es que todos los lenguajes NP-completos conocidos son *densos*, tienen *muchas cadenas*. Intuitivamente, a más cadenas en un lenguaje L más tiempo para determinar si una cadena w está entre las cadenas de L, o mejor dicho, más tiempo para encontrar un certificado de w entre todos los certificados posibles (mientras que en la computabilidad, como dijimos antes, más que el tamaño de un lenguaje es su especificación el componente que mejor identifica su grado de dificultad, que asociamos gráficamente al contorno de la figura con la que se puede representar en el plano). Formalmente, se define que un lenguaje L es denso si su *función censo* $c_L(n)$, la cual determina la cantidad de cadenas de L de tamaño a lo sumo n, crece exponencialmente, y es *disperso* si el crecimiento de $c_L(n)$ es polinomial. Se prueba que si existe un lenguaje NP-completo disperso entonces P = NP, por lo que el hecho de que todos los lenguajes NP-completos conocidos sean densos es otra evidencia de que P ≠ NP.

El problema P vs NP revisitado

En la práctica, cuando se prueba que un lenguaje es NP-completo se encara alguna alternativa de *remediación*, para poder procesarlo en un tiempo razonable (se asume P ≠ NP, y por lo tanto que el lenguaje no puede decidirse en tiempo polinomial). Entre las remediaciones se encuentran:

- *Restricción a cadenas de cierto tipo*. Vimos por ejemplo el lenguaje 2-SAT, con conjunciones de cláusulas de dos literales; mientras que el lenguaje SAT es NP-completo, 2-SAT pertenece a la clase P. También mencionamos el lenguaje CLIQUE_K, en el que el tamaño de los cliques no forma parte de las cadenas, sino que es una constante K; el lenguaje CLIQUE es NP-completo, pero CLIQUE_K pertenece a P (si bien podría discutirse su tratabilidad cuando K es muy grande). Otro ejemplo de restricción de cadenas en lenguajes con grafos corresponde al uso exclusivo de árboles: lenguajes NP-completos con grafos sin restricciones son decidibles eficientemente con árboles, entre ellos los que representan los problemas del clique, el circuito hamiltoniano, el conjunto independiente de vértices y el cubrimiento de vértices.
- *Método de retroceso* (*backtracking*). Se puede explicar fácilmente en términos de la definición de un lenguaje de NP mediante una MTN M: en lugar de procesar secuencialmente las computaciones de M, la idea es analizarlas en paralelo, avanzando y retrocediendo según la evolución en cada caso. Por ejemplo, para decidir si una fórmula booleana es satisfactible, se puede elegir en base a una heurística determinada una primera variable, asignarle en una computación el valor *verdadero* y en otra el valor *falso*, simplificar la fórmula en cada computación, seguir con una segunda variable, decidir eventualmente descartar una computación cuando se detecta que la misma no conduce a una solución, etc.
- *Aproximaciones polinomiales*. En este caso, los lenguajes corresponden a problemas de búsqueda, específicamente la búsqueda de un óptimo, máximo o mínimo. Los algoritmos que

se construyen consisten en obtener, en tiempo polinomial, una solución relativamente buena, cercana al óptimo.

Las alternativas de remediación parten de la conjetura P ≠ NP, ampliamente aceptada, pero en la complejidad temporal no se descartan otras posibilidades.

Una posibilidad es que se encuentren algoritmos que decidan los lenguajes NP-completos en tiempo del orden polinomial, pero con un orden muy alto, del estilo $O(n^{100})$. Un mundo en estas condiciones podría no ser utópico como el que caracterizamos en la sección sobre el problema P vs NP del capítulo anterior (en un mundo así, el costo para obtener una solución seguiría siendo significativamente mayor que el costo para verificarla).

Una segunda posibilidad es que se demuestre la pertenencia de un lenguaje NP-completo a la clase P pero con una prueba no constructiva. El problema en este caso sería que aún sabiendo que existe un algoritmo polinomial para decidir el lenguaje, no tendríamos idea alguna sobre su estructura.

Todavía queda una tercera posibilidad, y más problemática que la anterior (que de todos modos la opinión generalizada descarta): que la lógica que manejamos no sea lo suficientemente potente como para permitirnos probar la relación entre P y NP. Dicho de otra manera, que las dos conjeturas, P ≠ NP y P = NP, sean *independientes* en el marco lógico existente. En esta situación, podría suceder que contemos con un algoritmo que decida eficientemente un lenguaje NP-completo, pero que no lo podamos comprobar.

Notas adicionales

Comenzamos detallando la reducción polinomial de la prueba de NP-completitud de SAT, cuya idea general presentamos en el teorema 7.4. Después describimos otras tres reducciones polinomiales. La primera también permite encontrar un primer lenguaje NP-completo (de una manera mucho más fácil que en el teorema 7.4), la segunda es un ejemplo en el área de la teoría de conjuntos, y la última permite demostrar la NP-completitud de uno de los problemas de grafos más estudiados, ya formulado en el siglo XIX: el problema del viajante de comercio.

Detalle de la reducción polinomial para probar que SAT es NP-completo

Nos enfocamos en la forma de la fórmula booleana generada, que en el teorema 7.4 sólo se presentó en general. Primero repasamos la idea de la reducción y agregamos algunos detalles técnicos:

- Se define una función f para reducir un lenguaje L cualquiera de NP al lenguaje SAT. Se asume que existe una MTN M que decide L en tiempo polinomial p(n), con una cinta (sin

perder generalidad), un conjunto de estados Q_M y un alfabeto Γ_M. La idea de f es generar, dada una cadena w, una fórmula booleana φ_w satisfactible sii w pertenece a L, tal que φ_w expresa una computación de M a partir de w que termina en el estado de aceptación.

- La forma de φ_w se basa en la representación de una computación de M a partir de w con una cadena $\#\beta_0\#\beta_1\#\ldots\#\beta_{p(n)}$, con n = |w| y β_k, de p(n) símbolos, la representación de la configuración k-ésima. En la representación de β_k, el estado corriente, el símbolo corriente y la selección no determinística del próximo paso (número natural entre 1 y K si el grado de no determinismo de M es K) forman un solo símbolo compuesto (de esta manera, existen símbolos simples de $\Gamma_M \cup \{\#\}$ y símbolos compuestos con un estado de Q_M, un símbolo de $\Gamma_M \cup \{\#\}$ y un número entre 1 y K). Si β_k corresponde a una configuración final y k < p(n), se hace $\beta_k = \ldots = \beta_{p(n)}$.
- Por cada símbolo de la cadena $\#\beta_0\#\beta_1\#\ldots\#\beta_{p(n)}$, φ_w tiene una variable booleana c_{ix}. Los subíndices i varían a lo largo de todas las posiciones posibles de la cadena, entre 0, la primera posición de la configuración inicial, y $(p(n) + 1)^2 - 1$, la última posición de la última configuración. Los subíndices x son todos los símbolos posibles. La idea es que c_{ix} sea verdadera o falsa según en la i-ésima posición se encuentre o no el símbolo x, respectivamente.
- Para simplificar la escritura, conjunciones como $c_{1x} \wedge c_{2x} \wedge \ldots$ se abrevian con $\bigwedge_i c_{ix}$, y disyunciones como $c_{1x} \vee c_{2x} \vee \ldots$ con $\bigvee_i c_{ix}$ (el rango de i se especifica en cada caso, y el de x debe entenderse como que varía entre todos los símbolos, salvo que se indique alguna restricción).

Teniendo en cuenta estas consideraciones, la fórmula booleana φ_w que genera la función de reducción f consiste en la conjunción de cuatro subfórmulas:

$$\varphi_w = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$$

La subfórmula φ_1 expresa que en una posición i no hay dos símbolos x e *y* distintos:

$$\varphi_1 = \bigwedge_i [(\bigvee_x c_{ix}) \wedge \neg(\bigvee_{x \neq y} c_{ix} \wedge c_{iy})]$$

siendo i cualquier posición. La subfórmula φ_2 expresa la configuración inicial de M, que contiene la entrada $w = w_1 \ldots w_n$:

$\varphi_2 = \varphi_{21} \wedge \varphi_{22} \wedge \varphi_{23} \wedge \varphi_{24}$, tal que:

$\varphi_{21} = c_{0,\#} \wedge c_{p(n)+1,\#}$

$\varphi_{22} = c_{1,x1} \vee c_{1,x2} \vee \ldots \vee c_{1,xm}$

$\varphi_{23} = c_{2,w2} \wedge c_{3,w3} \wedge \ldots \wedge c_{n,wn}$

$\varphi_{24} = c_{n+1,B} \wedge c_{n+2,B} \wedge \ldots \wedge c_{p(n),B}$

donde los x_i son todos los símbolos compuestos que pueden representar el estado inicial de M, el primer símbolo de w y un posible próximo paso de M. La subfórmula φ_3 expresa que la última configuración tiene el estado final q_A:

$$\varphi_3 = V_i (V_x c_{ix})$$

tal que i varía a lo largo de la última configuración y x incluye el estado q_A. Finalmente, la subfórmula φ_4 expresa la condición que debe cumplir una configuración para suceder a otra, según la relación de transición de M y la selección no determinística del paso anterior:

$$\varphi_4 = \Lambda_i (V_{v,x,y,z}\ c_{i-p(n)-2,v} \wedge c_{i-p(n)-1,x} \wedge c_{i-p(n),y} \wedge c_{iz})$$

tal que i varía entre la primera posición de la segunda configuración y la última posición de la última configuración, y z puede estar en la posición i de una configuración cuando los símbolos v, x e *y* están en las posiciones i – 1, i, i + 1 de la configuración anterior (considerando que una configuración con un estado final es igual a la siguiente).

Reducción polinomial alternativa para encontrar un primer lenguaje NP-completo

La siguiente reducción polinomial permite encontrar un primer lenguaje NP-completo de una manera mucho más fácil que la que mostramos antes (pero que no representa un problema natural como el problema de satisfactibilidad):

Ejemplo 7.5. Vamos a probar que el lenguaje $L_{U\text{-}TK}$ = {(<M>, w, 1^K) | M es una MTN con una cinta que acepta w en a lo sumo K pasos} es NP-completo. El lenguaje $L_{U\text{-}TK}$ es la versión del lenguaje universal L_U de la clase RE en el marco de la clase NP, y como tal representa el problema de aceptación, en este caso acotada por un número de pasos.

Primero probamos que $L_{U\text{-}TK} \in$ NP:

La siguiente MTN $M_{U\text{-}TK}$ decide $L_{U\text{-}TK}$ en tiempo polinomial: dada una terna (<M>, w, 1^K), genera no determinísticamente una cadena C de $(K + 1)^2$ símbolos, y acepta sii C representa una computación de M de a lo sumo K pasos que acepta w, considerando la representación de una computación que describimos en la nota adicional anterior.

Ejercicio. Comprobar que efectivamente la MTN $M_{U\text{-}TK}$ decide $L_{U\text{-}TK}$ en tiempo polinomial. ¿Es relevante que K esté codificado en unario?

Ahora probamos que $L_{U\text{-}TK}$ es NP-difícil, y por lo tanto NP-completo:

Naturalmente, la prueba es similar a la de la RE-completitud del lenguaje L_U (teorema 5.3). Si L es un lenguaje cualquiera de NP y M es una MTN con una cinta que lo decide en tiempo polinomial p(n), la función f que asigna a una cadena w la terna (<M>, w, $1^{p(|w|)}$), es una reducción polinomial de L a $L_{U\text{-}TK}$: se computa en tiempo polinomial y cumple, para toda cadena w, w ∈ L sii (<M>, w, $1^{p(|w|)}$) ∈ $L_{U\text{-}TK}$.

Fin del ejemplo

Otras reducciones polinomiales

Presentamos dos reducciones polinomiales más, una del área de la teoría de conjuntos (hasta ahora hemos considerado solamente lenguajes con grafos y fórmulas booleanas), y la otra relacionada con uno de los problemas de grafos más conocidos, el problema del viajante de comercio.

Ejemplo 7.6. Vamos a definir una reducción polinomial del lenguaje SUM-SUB = {(S, K) | S es un conjunto de números naturales, K es un número natural, y existe un subconjunto de S cuyos elementos suman K} al lenguaje PARTICIÓN = {T | T es un conjunto de números naturales que se puede particionar en dos subconjuntos, de manera tal que las sumas de sus elementos coincidan}.

El lenguaje SUM-SUB representa un *problema de planificación de tareas* y también una versión simplificada del *problema de la mochila*. En el ejemplo 6.8 probamos que pertenece a NP, y se puede probar que es NP-difícil, y así NP-completo, con una reducción polinomial desde el lenguaje NP-completo que representa el *problema del cubrimiento exacto*, otro lenguaje del área de la teoría de conjuntos. Por su parte, el lenguaje PARTICIÓN también representa un problema de planificación de tareas. En el ejercicio 6.10.f pedimos probar que pertenece a NP, y con esta reducción completaremos la prueba de su NP-completitud.

Proponemos la siguiente función de reducción f:

$$f(S, K) = T$$

tal que S es un conjunto $\{s_1, \ldots, s_m\}$ de números naturales, K es un número natural, y T es el conjunto $\{s_1, \ldots, s_m, t_1, t_2\}$, siendo $t_1 = 2H + 2K$ y $t_2 = 4H$, con $H = (\sum_S s_i) / 2$.

Ejercicio. Probar que f es computable en tiempo polinomial.

Veamos que (S, K) ∈ SUB-SUM sii T ∈ PARTICIÓN:

- Si T ∈ PARTICIÓN, entonces T se puede particionar en dos subconjuntos cuyos elementos suman lo mismo. Ninguno de los dos subconjuntos puede incluir juntos a t_1 y t_2, porque la

suma de ambos es mayor que la suma de los elementos restantes. De esta manera, se cumple:

$$4H + \sum_P s_i = 2H + 2K + \sum_{S-P} s_i$$

para algún subconjunto P de S. Sumando $\sum_P s_i$ en ambos miembros de la igualdad, queda:

$$4H + 2\sum_P s_i = 4H + 2K$$

y por lo tanto, $\sum_P s_i = K$, lo que significa que (S, K) ∈ SUM-SUB.

- Si (S, K) ∈ SUM-SUB, entonces existe un subconjunto P de S que cumple $\sum_P s_i = K$. A partir de esta igualdad, podemos llegar fácilmente a:

$$\sum_P s_i + t_2 = \sum_{S-P} s_i + t_1$$

y así, los subconjuntos P ∪ $\{t_2\}$ y (S – P) ∪ $\{t_1\}$ forman una partición de T con elementos que suman lo mismo, lo que significa que T ∈ PARTICIÓN.

Fin del ejemplo

Ejemplo 7.7. El *problema del viajante de comercio* plantea determinar si un vendedor puede visitar un conjunto de ciudades una sola vez y volver al punto de partida, sin recorrer más que una cierta distancia. En el ejemplo 6.10.c lo representamos con el lenguaje PVC = {(G, B) | G es un grafo completo ponderado que tiene un circuito de Hamilton tal que la suma de los valores asociados a sus arcos es menor o igual que B}, y pedimos probar la pertenencia del lenguaje a NP. En lo que sigue probaremos que PVC es NP-difícil, y por lo tanto que es NP-completo.

Vamos a definir una reducción polinomial a PVC desde el lenguaje CH = {G | G tiene un circuito de Hamilton}, representante del *problema del circuito Hamiltoniano*. En ejemplo 6.9 probamos que pertenece a NP, y se puede probar que es NP-difícil, y así NP-completo, con una reducción polinomial desde el lenguaje NP-completo CV correspondiente al problema del cubrimiento de vértices (probamos su NP-completitud en el ejemplo 7.2).

Proponemos la siguiente función de reducción f:

$$f(G) = (G_{12}, m)$$

tal que G es un grafo con m vértices, y G_{12} es un grafo completo cuyos vértices son los mismos de G y cuyos arcos tienen asociado el valor 1 o 2 según estén o no en G, respectivamente.

La función f es efectivamente una reducción polinomial de CH a PVC. Dado G, con m vértices, en tiempo polinomial se puede generar (G_{12}, m), y además se cumple G ∈ CH sii (G_{12}, m) ∈ PVC:

$G \in CH$ sii G tiene un circuito de Hamilton sii G_{12} tiene un circuito de Hamilton cuyos valores asociados a sus arcos suman m sii $(G_{12}, m) \in PVC$.

Fin del ejemplo

Observaciones finales

- En la práctica, probar que un lenguaje es NP-completo significa *condenarlo* a no pertenecer a la clase P.
- Para probar que un lenguaje de la clase NP es NP-completo, hay que encontrar una reducción polinomial de un lenguaje NP-completo a dicho lenguaje.
- Al probarse que un lenguaje es NP-completo, se plantea alguna alternativa de remediación para poder procesarlo en un tiempo razonable.
- La completitud es un concepto que trasciende la clase NP, es fundamental en toda la complejidad computacional. Un lenguaje completo de una clase permite identificar el grado de dificultad de toda la clase.

Referencias

S. Cook creó el concepto de NP-completitud y encontró el primer lenguaje NP-completo, el lenguaje SAT (Cook, 1971). Un poco después, R. Karp probó mediante reducciones polinomiales la NP-completitud de 21 lenguajes de interés computacional (Karp, 1972), y L. Levin publicó en la Unión Soviética un trabajo similar al de Cook (Levin, 1973), pero enfocado en los problemas de búsqueda. Más allá de las fechas de sus publicaciones, se considera que el trabajo de Levin fue simultáneo al de Cook, y por eso el teorema que establece la NP-completitud de SAT se conoce como Teorema de Cook-Levin. La simultaneidad de estos trabajos no fue casual: a uno y otro lado de la Cortina de Hierro se estaban analizando desde hacía tiempo varios problemas, de la investigación operativa, la lógica, la aritmética, la combinatoria, etc., que se distinguían por su grado de dificultad y su mutuo relacionamiento. De todos modos, se puede observar de acuerdo a los trabajos de esa época, que en general la comunidad científica de entonces no captó la real dimensión del descubrimiento de la NP-completitud en lo que hace a su relación con el problema P vs NP.

La prueba que presentamos de que SAT es NP-completo se basa en la que aparece en el capítulo 13 de (Hopcroft y Ullman, 1979). En dicho capítulo también se prueba la NP-completitud de los lenguajes de fórmulas booleanas CSAT y 3-SAT, y del lenguaje de grafos CV representante del problema del cubrimiento de vértices. El capítulo 7 de (Sipser, 1997) y el capítulo 7 de (Lewis y Papadimitriou, 1998) incluyen varias pruebas de NP-completitud, considerando lenguajes de distintas áreas.

(Garey y Johnson, 1979) es una excelente guía para estudiar los lenguajes NP-completos. Describe distintas heurísticas para la construcción de reducciones polinomiales (mencionamos algunas en este capítulo), y presenta una extensa lista de lenguajes NP-completos y NP-difíciles agrupados por área (teoría de grafos, diseño de redes, conjuntos y particiones, almacenamiento y recuperación de datos, secuenciamiento y planificación de tareas, programación matemática, álgebra y teoría de números, juegos, lógica, autómatas y teoría de lenguajes, optimización de programas, y misceláneos), indicando por cada lenguaje qué reducción polinomial se puede emplear para demostrar su NP-dificultad.

La construcción de p-isomorfismos entre lenguajes NP-completos se basa en la técnica de *relleno* (*padding*), la cual consiste en concatenar al final de las cadenas de un lenguaje subcadenas de fácil reconocimiento, lo que aumenta artificialmente las longitudes de las cadenas y en consecuencia reduce el grado de dificultad del lenguaje, en el sentido de la complejidad temporal. Para profundizar sobre dicha técnica se puede consultar el capítulo 3 de (Balcázar, Díaz y Gabarró, 1995), y sobre los p-isomorfismos en la clase NPC, el capítulo 5 de (Bovet y Crescenzi, 1994). En este capítulo también se trata la densidad de los lenguajes NP-completos.

Otras lecturas recomendadas:

(Kapron, 2023): libro de reciente aparición dedicado al trabajo de Cook.

Los artículos (Aaronson, 2003) y (Aaronson, 2005): en el primero se discute sobre la posibilidad de que las conjeturas $P \neq NP$ y $P = NP$ sean independientes, y en el segundo se analizan distintos dispositivos computacionales no tradicionales para resolver los problemas NP-completos.

(Fichte, Le Berre, Hecher y Szeider, 2023): analiza la significativa evolución durante los últimos, en lo que hace a su eficiencia, de los *solucionadores SAT* (*SAT solvers*), aplicativos de la industria informática que resuelven el problema de satisfactibilidad,

Ejercicios

7.1 Probar que las reducciones polinomiales son reflexivas, transitivas y no simétricas. *Ayuda: basarse en la prueba del teorema 5.2 y el comentario que le sigue*.

7.2 Probar que se pueden reducir polinomialmente entre sí todos los lenguajes de la clase P distintos de $\emptyset$ y Σ^*. *Ayuda: dichos lenguajes se deciden en tiempo polinomial, y tienen al menos una cadena pero no tienen todas.*

7.3 Dados dos lenguajes L_1 y L_2 distintos de $\emptyset$ y Σ^*, y además $L_2 \in P$, definir:

a. Una reducción polinomial de $L_1 \cap L_2$ a L_1.

b. Una reducción polinomial de $L_1 \cup L_2$ a L_1.

7.4 Definir reducciones polinomiales:

a. De 2-COLOR a 2-SAT, tal que 2-COLOR = {G | G es un grafo y sus vértices pueden colorearse con dos colores, de manera tal que ningún par de vértices adyacentes reciban el mismo color} (el lenguaje 2-COLOR representa el problema de coloración de grafos con dos colores). *Ayuda: intentar con una función que genere por cada arco (i, j) de un grafo, cláusulas de la forma* $(x_i \vee x_j)$ *y* $(\neg x_i \vee \neg x_j)$.
b. De 3-SAT a CLIQUE. *Ayuda: intentar con una función que asigne a una fórmula booleana φ, un par (G, K), siendo K la cantidad de cláusulas de φ, y G un grafo con un vértice por cada literal de cada cláusula de φ y arcos que conectan a todos sus vértices entre sí, salvo cuando los vértices corresponden a literales de una misma cláusula o a un literal y su negación*.
c. De DCH a CH, tal que DCH es el lenguaje que representa el problema del circuito hamiltoniano en grafos dirigidos. *Ayuda: intentar con una función que asigne a un grafo* G_1 *un grafo* G_2*, que incluya por cada vértice v de* G_1*, vértices* v_0*,* v_1*,* v_2*, arcos* (v_0, v_1) *y* (v_1, v_2)*, y el arco* (v_2, w_0) *sólo en caso de que (v, w) sea un arco de* G_1*.*
d. De PARTICIÓN a SUM-SUB. *Ayuda: si T pertenece a PARTICIÓN, entonces incluye un subconjunto cuyos números suman la mitad de la suma de los números de T*.

7.5 Probar que los lenguajes HP y L_U, representantes de los problemas de la detención y aceptación, respectivamente, son NP-difíciles. ¿Son NP-completos?

7.6 Probar que los complementos de todos los lenguajes de la clase NP se reducen polinomialmente al lenguaje SAT^C.

7.7 Probar que si $L_1 \leq_p L_2$, $L_2 \leq_p L_1$ y $L_1 \in$ NPC, entonces $L_2 \in$ NPC.

7.8 Dados dos lenguajes L_1 y L_2 distintos de $\emptyset$ y Σ^*, y además $L_1 \in$ NP y $L_2 \in$ P, determinar si se cumple:
a. Si $L_1 \cap L_2 \in$ NPC, entonces $L_1 \in$ NPC.
b. Si $L_1 \cup L_2 \in$ NPC, entonces $L_1 \in$ NPC.

7.9 Probar la NP-completitud de los siguientes lenguajes:
a. 2-CH = {G | G es un grafo y tiene dos circuitos de Hamilton}. *Ayuda: intentar con una reducción polinomial desde el lenguaje CH*.
b. SH = {(G, i, j) | G es un grafo y tiene un *camino de Hamilton* del vértice i al vértice j}. Un camino de Hamilton tiene las mismas características que un circuito de Hamilton, salvo que el vértice inicial no se repite al final. *Ayuda: intentar también con una reducción polinomial desde el lenguaje CH*.
c. El lenguaje IND, representante del *problema del conjunto independiente de vértices* (definido en el ejercicio 6.10.d). *Ayuda: intentar con una reducción polinomial desde el lenguaje 3-SAT, y basarse en la reducción que sugerimos en el ejercicio 7.4.b*.

d. El lenguaje PROG-LIN-ENT-01, representante del *problema de la programación lineal entera* con la restricción de que los coeficientes de las variables son sólo 0 y 1. *Ayuda: intentar con una reducción polinomial desde el lenguaje CSAT, que asigne a una fórmula booleana, un sistema de desigualdades en base a la siguiente idea: a cláusulas como* $x_i \vee x_j \vee x_k$*, asociarles desigualdades de la forma* $x_i + x_j + x_k \geq 1$*, y a cláusulas como* $x_i \vee \neg x_j \vee \neg x_k$*, desigualdades de la forma* $x_i + (1 - x_j) + (1 - x_k) \geq 1$.

CAPÍTULO 8
La jerarquía espacio-temporal

Asumiendo la conjetura ampliamente aceptada P ≠ NP, se prueba que además de P y NPC, NP incluye una tercera clase de lenguajes, la clase NPI (la I es por su condición de clase *intermedia* entre P y NPC, teniendo en cuenta el grado de dificultad de sus lenguajes, en el sentido de la complejidad temporal). Otra conjetura ampliamente aceptada es que NP no es cerrada con respecto al complemento, lo que refuerza la conjetura P ≠ NP. La clase que reúne a los complementos de los lenguajes de NP se denomina CO-NP. En NPI y CO-NP se identifican lenguajes de mucho interés computacional. Comenzamos este capítulo describiendo ambas clases, con las que completamos la presentación de la jerarquía temporal.

Después nos enfocamos en la métrica de complejidad computacional que nos falta tratar, la métrica de *complejidad espacial*. Del mismo modo que hicimos con la complejidad temporal, formulamos el escenario de estudio y las definiciones más relevantes de la complejidad espacial, mostramos ejemplos representativos, e introducimos una jerarquía de lenguajes, incluyendo lenguajes completos (generalizando el concepto de completitud, que hasta ahora utilizamos solamente en el ámbito de la clase NP). La jerarquía espacial la presentamos combinada con la jerarquía temporal en una única jerarquía espacio-temporal, para reflejar la relación que existe entre las dos métricas.

Al final hacemos una breve referencia a la región de la jerarquía espacio-temporal en la que habitan los lenguajes efectivamente intratables, es decir los lenguajes que se prueba que son intratables sin utilizar ninguna conjetura.

La clase NPI

Empezamos describiendo la clase NPI. Se define como el conjunto NP – (P ∪ NPC). Su existencia, asumiendo P ≠ NP, se puede probar recurriendo a la técnica de *relleno* que mencionamos en las referencias del capítulo anterior. La idea es concatenar secuencias de unos, de una forma determinada, a las fórmulas booleanas del lenguaje SAT. De este modo, las cadenas del lenguaje L obtenido son más grandes y más fácilmente reconocibles que las originales, lo que provoca que L tenga un grado de dificultad menor que SAT, pero sin llegar al de un lenguaje de la clase P.

Otra manera de probar que NPI no es el conjunto vacío si P ≠ NP, es aplicando el siguiente teorema, conocido como Teorema de Ladner:

Teorema 8.1

Si $P \neq NP$, entonces dado un lenguaje recursivo $L_1 \notin P$, existe un lenguaje $L_2 \in P$ tal que:

a. $L_1 \cap L_2 \notin P$.
b. $L_1 \cap L_2$ se reduce polinomialmente a L_1.
c. L_1 no se reduce polinomialmente a $L_1 \cap L_2$.

Prueba. No vamos a probar el teorema. En lugar de ello, mostramos en lo que sigue cómo a partir de su enunciado podemos encontrar un lenguaje concreto de NPI. Sea $L_1 \in NPC$. Si se cumple $P \neq NP$ entonces $L_1 \notin P$, por lo que según los tres incisos planteados existe un lenguaje $L_2 \in P$ que satisface:

- Por (a), $L_1 \cap L_2 \notin P$.
- Por (b), $L_1 \cap L_2$ se reduce polinomialmente a L_1, y como $L_1 \in NP$, vale $L_1 \cap L_2 \in NP$.
- Por (c), L_1 no se reduce polinomialmente a $L_1 \cap L_2$, y como $L_1 \in NP$, vale $L_1 \cap L_2 \notin NPC$.

Así llegamos a que $L_1 \cap L_2 \in NP - (P \cup NPC) = NPI$ (figura 8.1).

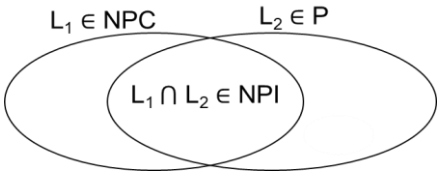

Figura 8.1. Si $P \neq NP$ y $L_1 \in NPC$, entonces existe $L_2 \in P$ tal que $L_1 \cap L_2 \in NP - (P \cup NPC) = NPI$.

Por ejemplo, asumiendo $P \neq NP$ y tomando el lenguaje NP-completo CH correspondiente al problema del circuito hamiltoniano, de acuerdo al Teorema de Ladner existe un lenguaje de grafos L de la clase P tal que CH, restringido a los grafos de L, pertenece a la clase NPI.

La construcción del lenguaje $L_1 \cap L_2$ recuerda la relación entre la dificultad y el tamaño de un lenguaje referida en el capítulo anterior:

Quitando de un lenguaje L_1 NP-completo, que asumiendo $P \neq NP$ no pertenece a P, todas sus cadenas no pertenecientes a un determinado lenguaje L_2 de P, se obtiene un tercer lenguaje no tan difícil como L_1 ni tan fácil como L_2. O más en general, quitando de un lenguaje NP-completo una cantidad suficiente de cadenas, ni tantas ni tan pocas, se obtiene otro lenguaje con un grado de dificultad menor; no se sacan tantas cadenas como para que se alcance un lenguaje de P, ni tan pocas como para que se preserve la NP-completitud.

Fin del teorema

De esta manera, podemos plantear una nueva versión de la clase NP, asumiendo P ≠ NP, que mostramos en la figura 8.2.

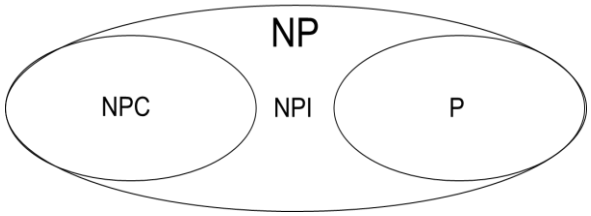

Figura 8.2. La clase NP y sus subclases P, NPI y NPC, asumiendo P ≠ NP.

Teniendo en cuenta cómo se puede obtener, aplicando el Teorema de Ladner, un lenguaje de NPI a partir de uno de NPC, es fácil comprobar que NPI incluye sucesiones infinitas de lenguajes con grado de dificultad cada vez menor, pero sin llegar al de un lenguaje de P.

Ejercicio. Comprobar la observación anterior.

También se cumple que NPI incluye pares de lenguajes incomparables, es decir pares de lenguajes entre los que no existe una reducción polinomial en ninguno de los dos sentidos, lo que contrasta con lo que ocurre en P y NPC. Otra característica distintiva de NPI con respecto a NPC es que podría incluir lenguajes dispersos.

Dos lenguajes de NP candidatos a estar en NPI son los que representan el *problema del isomorfismo de grafos* y el *problema de factorización*. Al día de hoy no se ha podido probar su pertenencia a la clase P ni a la clase NPC. Volvemos a ellos enseguida.

La clase CO-NP

Describimos ahora la clase CO-NP, la clase de los complementos de los lenguajes de NP, y con ella finalizamos la presentación de la jerarquía temporal.

La conjetura ampliamente aceptada es que NP ≠ CO-NP. Intuitivamente, no parece posible que existan certificados sucintos para las cadenas de lenguajes como SAT^C, $CLIQUE^C$, etc. Por ejemplo, en el caso de SAT^C, un certificado con una asignación de valores de verdad no alcanza para determinar si una fórmula booleana con m variables es insatisfactible, sino que debe incluir las 2^m asignaciones posibles. Lo mismo se puede decir sobre $CLIQUE^C$ y otros complementos de lenguajes de NP que hemos descripto previamente. Más aún, la conjetura NP ≠ CO-NP es más fuerte que la conjetura P ≠ NP, la primera implica la segunda: por el contrarrecíproco, si se cumple P = NP, entonces como P es cerrada con respecto al complemento también lo es NP, y por lo tanto NP = CO-NP.

De la clausura de P con respecto al complemento también se deriva P ⊆ CO-NP, y en relación a P y NP ∩ CO-NP sólo se puede decir que P ⊆ NP ∩ CO-NP. Por otro lado, CO-NP ⊆ EXP,

porque para decidir la pertenencia de una cadena de tamaño n al complemento de un lenguaje L de NP, alcanza con chequear sus $c^{poly(n)}$ certificados sucintos posibles, con c constante, los mismos que se requieren en el peor caso para decidir su pertenencia a L. Luego mostramos una nueva versión de la jerarquía temporal con estas relaciones.

La definición formal de la clase CO-NP es la siguiente: $L \in$ CO-NP sii existe un polinomio p y una MT M tal que para toda cadena w:

$$w \in L \text{ sii } \forall x: |x| \leq p(|w|), M \text{ acepta } (w, x) \text{ en tiempo } poly(|w|)$$

Notar que a diferencia de la definición de NP, los certificados sucintos se cuantifican universalmente (no basta con considerar uno, sino que se requiere chequear todos los posibles).

Ejemplo 8.1. El *problema de la validez de las fórmulas booleanas sin cuantificadores* consiste en determinar si una fórmula booleana sin cuantificadores φ es satisfactible por todas las asignaciones posibles de valores de verdad, es decir si φ es una *tautología*. El lenguaje que representa el problema es TAUT = {φ | φ es una tautología}.

Se cumple que TAUT pertenece a CO-NP. Sea la siguiente MT M, que a partir de todo par de la forma (φ, $\mathcal{A}$), con $|\mathcal{A}| \leq |\varphi|$, hace (como antes, para simplificar, omitimos el análisis sintáctico de las entradas):

1. Verifica si $\mathcal{A}$ es una asignación de valores de verdad a φ. Si no, rechaza.
2. Acepta sii $\mathcal{A}$ satisface φ.

φ ∈ TAUT sii M acepta (φ, $\mathcal{A}$) para toda asignación de valores de verdad $\mathcal{A}$ a φ, y además en todos los casos M tarda tiempo polinomial (el bloque 1 se ejecuta en tiempo lineal y el bloque 2 en tiempo cuadrático), por lo que TAUT pertenece a CO-NP.
Fin del ejemplo

Ejercicio. Indicar qué certificado sucinto x sirve para verificar eficientemente que una fórmula booleana *no* es una tautología (se dice en este caso que x es un *descalificador sucinto*).

La *CO-NP-completitud* se define de la misma manera que la NP-completitud: un lenguaje L es *CO-NP-completo* (o pertenece a la clase CO-NPC) sii L pertenece a CO-NP y todos los lenguajes de CO-NP se reducen polinomialmente a él. Obviamente, el complemento de un lenguaje NP-completo es CO-NP-completo (ejercicio 8.3). Por ejemplo, el lenguaje SAT^C es CO-NP-completo. También lo es el lenguaje TAUT.

Ejercicio. Probar que el lenguaje TAUT es CO-NP-completo. *Ayuda: existe una reducción polinomial de SAT^C a TAUT*.

Así como los lenguajes NP-completos son los más difíciles de la clase NP, los lenguajes CO-NP-completos son los más difíciles de la clase CO-NP.

Asumiendo NP ≠ CO-NP, se demuestra que el conjunto NP ∩ CO-NP no incluye lenguajes NP-completos ni CO-NP-completos. Intuitivamente, los lenguajes de NP ∩ CO-NP son más fáciles que los otros lenguajes de NP ∪ CO-NP, porque a diferencia de éstos, cumplen que tanto ellos como sus complementos cuentan con verificadores eficientes.

La relación entre NP ∩ CO-NP y NPC se formaliza con el siguiente teorema:

Teorema 8.2. Sea L un lenguaje NP-completo. Si L pertenece a CO-NP, entonces NP = CO-NP.

Prueba. Formulado por el contrarrecíproco, el teorema establece que si NP ≠ CO-NP, que es la conjetura ampliamente aceptada, entonces las clases NPC y CO-NP son disjuntas.

Partiendo de un lenguaje L de NPC ∩ CO-NP, probamos sólo la inclusión NP ⊆ CO-NP (la inclusión CO-NP ⊆ NP se prueba de manera similar):

Si L_1 es un lenguaje cualquiera de NP, entonces como L es NP-completo se cumple $L_1 \leq_p L$, y por lo tanto también se cumple $L_1^C \leq_p L^C$. Como además $L^C \in$ NP, entonces por propiedad de las reducciones polinomiales se obtiene $L_1^C \in$ NP, o lo que es lo mismo, $L_1 \in$ CO-NP.

Ejercicio. Probar la otra inclusión, es decir CO-NP ⊆ NP.
Fin del teorema

Así llegamos a una última versión de la jerarquía temporal en el marco de la clase EXP. La mostramos en la figura 8.3, asumiendo las conjeturas más aceptadas.

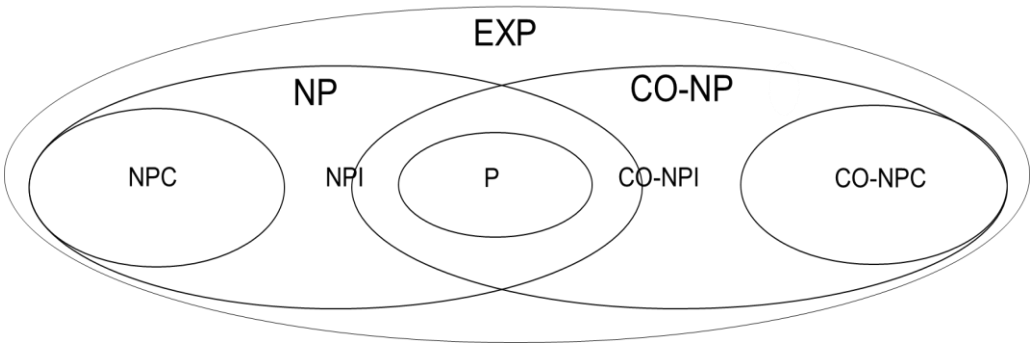

Figura 8.3. Versión definitiva de la jerarquía temporal, asumiendo las conjeturas más aceptadas.

Como en la jerarquía de la computabilidad, en la jerarquía temporal también podemos establecer un orden sobre las clases, de acuerdo al grado de dificultad creciente de sus lenguajes:

- *La clase P*. Es la clase de los lenguajes decidibles en tiempo polinomial.

- *La clase (NP ∩ CO-NP) – P*. Contiene los lenguajes de NPI tales que sus complementos también están en NPI.
- *La clase NPI – (NP ∩ CO-NP).* Contiene los lenguajes restantes de NPI.
- *La clase NPC*. Es la clase de los lenguajes NP-completos.
- *La clase CO-NPI – (NP ∩ CO-NP).* Contiene los complementos de los lenguajes de la clase NPI – (NP ∩ CO-NP).
- *La clase CO-NPC*. Es la clase de los lenguajes CO-NP-completos.

Ya anticipamos que dos lenguajes de mucho interés computacional, candidatos a pertenecer a NPI, son el que corresponde al problema del isomorfismo de grafos y el que corresponde al problema de factorización.

El primer lenguaje, ISO, tiene como elementos pares (G_1, G_2), siendo G_1 y G_2 grafos de m vértices que cumplen que existe una permutación π de (1, …, m), que podemos expresar con (π(1), …, π(m)), tal que (i, j) es un arco de G_1 sii (π(i), π(j)) es un arco de G_2 (dicha permutación hace las veces de certificado sucinto). No parece posible que existan certificados sucintos para las cadenas de ISO^C, por lo que ISO estaría en la clase NPI – (NP ∩ CO-NP) de la jerarquía temporal. En el capítulo siguiente seguiremos analizando el lenguaje con otros modelos computacionales.

Por otro lado, los elementos del segundo lenguaje, FACT, tienen la forma (N, M_1, M_2), tal que N, M_1 y M_2 son números naturales y N tiene un factor primo en el intervalo [M_1, M_2] (ésta es la forma habitual de especificar el problema de decisión). A diferencia de ISO, FACT estaría en la clase (NP ∩ CO-NP) – P (ejercicio 6.12), es decir que sería menos difícil que ISO, lo que podría justificarse por el hecho de que mientras las instancias del problema del isomorfismo de grafos pueden no tener solución o tener varias, las instancias del problema de factorización siempre tienen solución y la solución es única (por el Teorema Fundamental de la Aritmética, que establece que todo número natural mayor que uno se puede expresar como un producto de números primos de una sola manera). Curiosamente, el *problema de primalidad*, que plantea determinar si un número es primo, tiene resolución polinomial (ver las referencias del capítulo 6). Otra curiosidad es que existe un *algoritmo cuántico* que resuelve la factorización eficientemente (lo comentaremos en el próximo capítulo).

El lenguaje FACT es un muy buen ejemplo de cómo sacar provecho de un problema difícil. Su supuesta dificultad permite implementar un esquema de seguridad informática de uso muy extendido, el *sistema criptográfico de clave pública*, cuya variante más popular es el *Sistema RSA* (las siglas corresponden a sus autores, R. Rivest, A. Shamir y L. Adleman). Descripto muy simplificadamente, el sistema se basa en la hipótesis de que la multiplicación es una *función de un solo sentido* (*one-way function*), lo que significa que es fácil de computar (tiempo polinomial) pero en general difícil de invertir (tiempo no polinomial). Por ejemplo, calcular $N = N_1.N_2$, siendo N_1 y N_2 dos números primos muy grandes, es fácil, pero obtener N_1 y N_2 de N es difícil. Tomando esto en consideración se utilizan dos claves, una *clave pública e* conocida por todos los emisores de mensajes, que incluye un número como N, y una *clave privada* d conocida sólo por el receptor

de los mensajes, que incluye dos números como N_1 y N_2. El mecanismo de emisión y recepción de mensajes es el siguiente: el emisor le envía al receptor un mensaje encriptado v, obtenido mediante un algoritmo de encriptación E de tiempo polinomial aplicado sobre un par (w, e), siendo w el mensaje original; y el receptor recibe el mensaje encriptado v y lo desencripta mediante un algoritmo de desencriptación D también de tiempo polinomial, aplicado sobre el par (v, d), obteniendo el mensaje original w. Así, la única forma que tiene un *hacker* de interceptar el mensaje w en tiempo polinomial es conociendo la clave privada d.

Introducción a la complejidad espacial

Pasamos a estudiar la complejidad espacial. El espacio requerido para decidir un lenguaje es otra métrica de complejidad computacional ampliamente utilizada. Como sucede con el tiempo, con más espacio pueden decidirse más lenguajes. La analogía con la complejidad temporal también se manifiesta en las convenciones utilizadas: el espacio se mide con respecto al tamaño de las cadenas de entrada; se considera el peor caso; se utilizan órdenes de magnitud, ahora mediante *funciones espaciales* S(n); el modelo computacional es el de las máquinas de Turing determinísticas con varias cintas; y la codificación de los números es cualquiera que no sea la unaria (seguiremos utilizando la codificación binaria).

Formalmente, se define que una MT M se ejecuta en espacio S(n) (u ocupa espacio S(n), o es de espacio S(n), etc.) sii para toda entrada w de tamaño n, la ejecución de M a partir de w ocupa a lo sumo S(n) celdas en cualquiera de sus cintas, *sin considerar su cinta de entrada*, que es de *sólo lectura*, es decir que su cabezal únicamente se mueve a lo largo de la cadena de entrada w (más los dos símbolos blancos que la delimitan a izquierda y derecha) y los símbolos de w no pueden modificarse. La cinta de entrada se excluye de las mediciones para lograr espacio logarítmico, necesario como justificamos más adelante, porque de lo contrario ya con la entrada w la MT M ocuparía espacio lineal. El tratamiento de la *cinta de salida*, si fuese necesario utilizarla, también es especial: la misma es de *sólo escritura*, es decir que sobre ella únicamente se puede escribir un símbolo y avanzar un lugar a la derecha. Estas características de las cintas de entrada y salida tienen su correlato en las computadoras reales, en las que se distingue la memoria externa de la memoria interna.

Toda función espacial S(n) satisface la relación $S(n) \geq \log_2 n$, para que mínimamente puedan almacenarse índices de celdas. Además, como una máquina de Turing con una cinta puede simular una máquina de Turing con varias cintas en el mismo espacio (el teorema 2.1 establece un retardo cuadrático, pero también que el espacio ocupado no se altera), para el estudio de la complejidad espacial son suficientes máquinas de Turing con una cinta de entrada de sólo lectura y una cinta de entrada/salida o *cinta de trabajo* (utilizaremos este modelo por defecto).

Ejercicio. La complejidad espacial, como la complejidad temporal, trata únicamente con lenguajes recursivos, pero en la definición anterior nos referimos solamente al espacio ocupado

por una máquina de Turing, sin puntualizar su detención. ¿Por qué no hace falta modificar la definición? *Ayuda: ver el ejercicio 3.8 y el ejemplo 4.3*.

El escenario de estudio de la complejidad espacial se estructura en *clases espaciales* SPACE(S(n). Cada una agrupa a todos los lenguajes que se pueden decidir en espacio O(S(n)), y en conjunto forman la *jerarquía espacial*.

Como en la jerarquía temporal, los factores constantes de las funciones espaciales se pueden eliminar, de acuerdo al Teorema de Compresión de Cintas (*Tape Compression Theorem*), análogo al Teorema de Aceleración Lineal de la complejidad temporal y probado del mismo modo. En los ejemplos utilizaremos en general la notación O(S(n)), teniendo en cuenta entonces que también podríamos utilizar directamente S(n).

También como la jerarquía temporal, la jerarquía espacial es densa, siempre hay una clase espacial que incluye estrictamente a otra (lo formalizamos en una nota adicional). Otra similitud es que las funciones que se utilizan son construibles, en este caso *espacio-construibles*. Una función S(n) es espacio-construible sii existe una MT $M_{S(n)}$ tal que a partir de toda entrada w, con |w| = n, su ejecución ocupa S(n) celdas. De esta manera, utilizando funciones espacio-construibles S(n) se pueden escribir cadenas de tamaño S(n) en espacio S(n) (ejercicio 8.5), lo que resulta de mucha utilidad para demostrar algunos teoremas. Todas las funciones habitualmente utilizadas en la complejidad espacial son espacio-construibles (en el ejercicio 8.6 consideramos algunos ejemplos).

En contraste con estas analogías, el efecto del no determinismo en el espacio es muy diferente a su efecto en el tiempo. Concretamente, por el Teorema de Savitch, la simulación determinística de una máquina de Turing no determinística, que temporalmente produce un retardo exponencial, espacialmente sólo provoca un aumento cuadrático. Otro resultado en este sentido, aún más contraintuitivo que el anterior, proviene del Teorema de Immerman, que establece que en el mismo espacio en el que se puede decidir no determinísticamente un lenguaje, se puede decidir no determinísticamente su complemento (cuando en la complejidad temporal, considerando por ejemplo las clases NP y CO-NP, la conjetura más aceptada es que son distintas). En otra nota adicional nos referimos a ambos teoremas.

Los lenguajes decidibles en espacio polinomial conforman la clase PSPACE, y los lenguajes decidibles en espacio logarítmico, la clase LOGSPACE. De la misma forma, pero en términos de máquinas de Turing no determinísticas, se definen las clases NPSPACE y NLOGSPACE, respectivamente. Una máquina de Turing no determinística ocupa espacio S(n) sii todas sus computaciones ocupan espacio S(n). NPSPACE y NLOGSPACE también se pueden definir, alternativamente, en términos de verificadores que hacen uso de certificados (máquinas de Turing determinísticas), igual que definimos NP en la complejidad temporal (en la complejidad espacial, al revés que en la complejidad temporal, es más habitual recurrir a las máquinas de Turing no determinísticas que deciden lenguajes que a las máquinas de Turing determinísticas que los verifican).

Como una MT M que tarda tiempo T(n) no ocupa más que espacio T(n), porque en T(n) pasos se recorren a lo sumo T(n) celdas, instanciando T(n) con poly(n) se obtiene:

$$P \subseteq PSPACE$$

lo que significa que los lenguajes tratables se pueden decidir en espacio polinomial.

Por otro lado, una MT M que ocupa espacio S(n) puede tardar mucho más que tiempo S(n), tanto como exp(S(n)), producto de la gran cantidad de configuraciones distintas que puede recorrer en dicho espacio: |Q| estados, |Γ| símbolos, una cinta de entrada de sólo lectura y una cinta de trabajo, determinan en total $(n + 2).S(n).|Q|.|\Gamma|^{S(n)}$ configuraciones distintas, por n + 2 posiciones del cabezal de la cinta de entrada, S(n) posiciones del cabezal de la cinta de trabajo, |Q| estados y $|\Gamma|^{S(n)}$ contenidos de la cinta de trabajo (la cinta de entrada tiene un solo contenido). Se prueba que $(n + 2).S(n).|Q|.|\Gamma|^{S(n)} \leq (4.|Q|.|\Gamma|)^{S(n)}$, por lo que para simplificar, se suele utilizar dicha cota, o directamente $c^{S(n)}$, siendo c una constante que depende de M. De este modo, espacio S(n) implica tiempo a lo sumo $c^{S(n)}$, y por lo tanto, en espacio poly(n) el tiempo puede llegar a ser $c^{poly(n)}$, con lo que se cumple:

$$PSPACE \subseteq EXP$$

y en espacio $\log_2 n$ el tiempo puede llegar a ser $c^{\log_2 n} = n^{\log_2 c}$, es decir poly(n), y así:

$$LOGSPACE \subseteq P$$

lo que significa que los lenguajes de LOGSPACE son tratables.

En lo que hace a la relación de la clase temporal NP con la jerarquía espacial, una característica fundamental de la complejidad espacial la explica: *el espacio se puede reutilizar*. Podemos verlo por ejemplo con el lenguaje SAT. SAT no podría decidirse en tiempo polinomial (probar la pertenencia al lenguaje puede requerir que se verifiquen $O(2^n)$ certificados sucintos posibles), pero sí puede decidirse en espacio polinomial: el espacio requerido para verificar un certificado sucinto posible es polinomial, y se puede reutilizar en la verificación de otro. De esta capacidad de iterar sobre todos los certificados sucintos posibles, reutilizando cada vez el mismo espacio polinomial, se deriva la inclusión:

$$NP \subseteq PSPACE$$

Ejercicio. Otra manera de probar $NP \subseteq PSPACE$ es mostrando que una máquina de Turing no determinística que tarda tiempo polinomial se puede simular determinísticamente en espacio polinomial. ¿Cómo sería la simulación?

Así llegamos a las siguientes inclusiones entre las clases de lenguajes mencionadas:

$$\text{LOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP}$$

y al menos una inclusión entre las clases LOGSPACE y PSPACE es estricta, porque se cumple LOGSPACE $\subset$ PSPACE (ejercicio 8.7.a). Lo mismo sucede con las inclusiones entre las clases P y EXP, porque se cumple P $\subset$ EXP (teorema 6.3). La conjetura más aceptada es que todas las inclusiones son estrictas. Describimos a continuación las clases espaciales que definimos.

Las clases LOGSPACE y NLOGSPACE

Empezamos por las clases de los lenguajes decidibles en espacio logarítmico. El siguiente es un ejemplo de lenguaje de la clase LOGSPACE muy simple, pero muy representativo de lo que significa pertenecer a dicha clase:

Ejemplo 8.2. En el ejemplo 6.1 probamos que el lenguaje de las cadenas $a^k b^k$, con $k \geq 1$, se puede decidir en tiempo polinomial, y por lo tanto en espacio polinomial. Pero dicho lenguaje se puede decidir en menos espacio, en espacio logarítmico.

La idea general para probarlo es construir una MT M que utilice dos índices para apuntar a los símbolos de la cadena de entrada w, un índice para los símbolos de su parte izquierda y un índice para los símbolos de su parte derecha, e iterativamente copie en memoria sólo un par de símbolos para validarlos, reutilizando cada vez el mismo espacio. El acceso a un símbolo específico de w se puede hacer moviendo el cabezal de la cinta de entrada hasta el símbolo apuntado por el índice correspondiente.

Formalmente, M tiene cuatro cintas de trabajo. En las cintas de trabajo 1 y 2 mantiene dos índices, i y j, y en las cintas de trabajo 3 y 4 valida cada vez los símbolos w_i y w_j de w apuntados por los índices respectivos, reutilizando el espacio ocupado en la validación anterior. Dada una entrada w, con |w| = n, M hace:

1. Si n no es par, rechaza.
2. Hace i := 1 en la cinta 1.
3. Hace j := n en la cinta 2.
4. Si i < j:
   Copia w_i en la cinta 3.
   Copia w_j en la cinta 4.
   Si $w_i \neq a$ o $w_j \neq b$, rechaza.
   Hace i := i + 1 en la cinta 1.
   Hace j := j – 1 en la cinta 2.
   Vuelve al bloque 4.
5. Acepta.

M decide el lenguaje porque acepta w sii |w| = n es par, sus símbolos w_1, …, $x_{n/2}$ son *a*, y sus símbolos $w_{n/2+1}$, …, w_n son b. Con respecto al espacio que ocupa:

- La validación relacionada con n, lo mismo que las operaciones sobre los índices i y j, ocupan espacio $O(\log_2 n)$,
- las copias y validaciones de los símbolos de w ocupan espacio constante,
- y para ejecutar una iteración se puede reutilizar el espacio ocupado por la iteración anterior,

lo que da como resultado un total de $O(\log_2 n)$ celdas.
Fin del ejemplo

Ejercicio. Justificar por qué también se puede establecer que el lenguaje de las cadenas $a^k b^k$, con $k \geq 1$, es decidible en espacio $\log_2 n$. *Ayuda: considerar el Teorema de Compresión de Cintas.*

La técnica utilizada en el ejemplo anterior es típica para probar que un lenguaje pertenece a la clase LOGSPACE. Se emplean índices para el posicionamiento de los cabezales, se manipulan pequeños fragmentos de la cadena de entrada, y se reutiliza espacio. Con esto alcanza, porque los algoritmos correspondientes consisten básicamente en validaciones y operaciones sobre pequeños componentes (análisis sintácticos y operaciones aritméticas elementales).

Un lenguaje de LOGSPACE de mucho interés computacional es el lenguaje ACC, representante del problema de la accesibilidad en los grafos no dirigidos que ya tratamos en distintos capítulos. Su pertenencia a LOGSPACE la probó O. Reingold en 2005, hito muy relevante en la historia de la complejidad espacial.

Obviamente, el mismo problema pero planteado en grafos dirigidos también es de mucho interés computacional. Al lenguaje que lo representa lo vamos a identificar con D-ACC. Curiosamente, a diferencia de ACC, D-ACC no estaría en LOGSPACE, pareciera que con grafos dirigidos el problema de accesibilidad es más difícil. Pero sí está en NLOGSPACE, es decir que se puede decidir en espacio logarítmico por medio de una máquina de Turing no determinística, como mostramos a continuación:

Ejemplo 8.3. La siguiente MTN M decide el lenguaje D-ACC = {(G, i, j) | G es un grafo dirigido que tiene un camino del vértice i al vértice j} en espacio logarítmico.

La idea general de M es ir armando un camino arco por arco, a partir del vértice i, pero guardando sólo el último arco procesado, para no superar el espacio logarítmico. Como un camino de un grafo de m vértices que no repite ninguno (camino simple) tiene a lo sumo m – 1 arcos, la cantidad de iteraciones de cada computación de M se puede acotar con dicho valor (un contador c lleva la cuenta). Formalmente, dada una entrada (G, i, j), básicamente con tres cintas de trabajo (cintas 1, 2 y 3), M hace:

1. Hace c := 1 en la cinta 3.
2. Hace v_1 := i en la cinta 1.
3. Escribe no determinísticamente un vértice v_2 de G en la cinta 2.
4. Si (v_1, v_2) no es un arco de G, rechaza.
5. Si v_2 = j, acepta.
6. Hace c := c + 1 en la cinta 3.
7. Si c = m, rechaza.
8. Hace v_1 := v_2 en la cinta 2.
9. Vuelve al bloque 3.

Ejercicio. Probar que efectivamente la MTN M decide D-ACC en espacio logarítmico.
Fin del ejemplo

Como dijimos, la clase NLOGSPACE también se puede definir en términos de verificadores basados en certificados (máquinas de Turing determinísticas). Formalmente, su definición alternativa es la siguiente: un lenguaje L pertenece a NLOGSPACE sii existe un polinomio p y una MT M tal que para toda cadena w:

$$w \in L \text{ sii } \exists x: |x| \leq p(|w|) \text{ y M acepta (w, x) en espacio } O(\log_2|w|)$$

con la salvedad de que la cadena x, que hace las veces de certificado sucinto, se encuentra en otra cinta especial de M, la cual es de sólo lectura como la cinta de entrada, y además tiene la restricción de que su cabezal se mueve únicamente a la derecha. Tal restricción se debe a que el procesamiento de x tiene que hacerse en simultáneo con su lectura, símbolo a símbolo, porque de lo contrario excedería el espacio logarítmico, dado que su tamaño es polinomial con respecto al de la cadena de entrada (en el ejemplo 8.3 se aprecia claramente este esquema). La equivalencia de las dos definiciones de la clase NLOGPSACE se puede demostrar sin mayor dificultad (ejercicio 8.9).

Naturalmente, se cumple:

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE}$$

porque una máquina de Turing determinística es un caso particular de una máquina de Turing no determinística. La conjetura más aceptada es que la inclusión es estricta. En este sentido, el lenguaje D-ACC en la práctica queda *condenado* a no pertenecer a LOGSPACE, porque se prueba que está entre los lenguajes más difíciles de NLOGSPACE (mecanismo de razonamiento que recuerda al que vimos con los lenguajes NP-completos, y de hecho D-ACC es un lenguaje *NLOGSPACE-completo*, definición que formalizamos más adelante). Además, pertenece a P (el

ejemplo 6.4 que prueba la pertenencia de ACC a P también sirve para probar la pertenencia de D-ACC a P, adaptando ligeramente la máquina de Turing utilizada), lo que sugiere que todos los lenguajes de NLOGSPACE pertenecen a P, es decir:

$$\text{NLOGSPACE} \subseteq \text{P}$$

que en efecto se cumple. En consecuencia, también los lenguajes de NLOGSPACE son tratables. Una manera de probar dicha inclusión es mostrando que toda máquina de Turing no determinística que ocupa espacio logarítmico se puede simular determinísticamente en tiempo polinomial (ejercicio 8.10). Otra manera, más general y que es la que presentamos a continuación, se basa en un método que permite probar varias propiedades de la jerarquía espacial (después volvemos a utilizarlo):

Teorema 8.3. $\text{NLOGSPACE} \subseteq \text{P}$

Prueba. El método referido se denomina *método de accesibilidad*. Se basa en la construcción de un *grafo de configuraciones*, que se forma con las configuraciones de una máquina de Turing. Para probar el teorema instanciamos el método a una máquina de Turing no determinística que ocupa espacio logarítmico.

Dados un lenguaje L de NLOGSPACE y una MTN M_{NL} con una cinta de trabajo que lo decide en espacio logarítmico, la idea es construir una MTD M_{DL} de tiempo polinomial que se comporte de la siguiente forma. Por cada cadena de entrada w, la MTD M_{DL}:

- Primero construye un grafo dirigido G, con vértices que representan todas las configuraciones que se pueden formar teniendo en cuenta M_{NL} y w, y con arcos que unen los vértices que representan configuraciones consecutivas. El objetivo es que G tenga un camino de su vértice inicial, representante de la configuración inicial de M_{NL}, a su vértice final, representante de una configuración de aceptación de M_{NL}, sólo si w pertenece a L.
- Y después decide si G tiene un camino del vértice inicial al vértice final, es decir ejecuta una máquina de Turing que decide D-ACC (por ejemplo la del ejemplo 6.4 adaptada) a partir de G, su vértice inicial y su vértice final.

Formalmente, dada una entrada w, M_{DL} hace:

1. *Generación de los vértices del grafo G*. Primero escribe la configuración inicial de M_{NL}, que identificamos con INI (cabezales apuntando a las primeras celdas de cada cinta, estado inicial y contenido de la cinta de trabajo con todos símbolos blancos). Luego escribe las configuraciones restantes, en el orden canónico. Finalmente escribe un vértice especial, que identificamos con FIN.

2. *Generación de los arcos del grafo G*. Escribe en el orden canónico todas las configuraciones (las identificamos con C_i), pero haciendo luego de cada escritura lo siguiente: si C_i no es una configuración final (es decir, no tiene un estado final), obtiene las configuraciones C_j alcanzables desde C_i en un paso y escribe los arcos (C_i, C_j); y si C_i es una configuración de aceptación (configuración con un estado de aceptación), escribe el arco (C_i, FIN).
3. Finalmente, vuelve a escribir los vértices INI y FIN para completar la terna (G, INI, FIN), y ejecuta a partir de ella una máquina de Turing que decide el lenguaje D-ACC.

Claramente, M_{DL} decide el lenguaje L. Y lo hace en tiempo polinomial:

Por un lado, generar la terna (G, INI, FIN) se puede hacer eficientemente. La MTN M_{NL} a partir de w, con |w| = n, puede pasar a lo sumo por poly(n) configuraciones distintas (espacio logarítmico implica tiempo polinomial). Cada configuración se puede representar en espacio $O(\log_2 n)$: la posición del cabezal de la cinta de entrada en espacio $O(\log_2 n)$, la posición del cabezal de la cinta de trabajo en espacio $O(\log_2(\log_2 n))$, el estado en espacio constante, y el contenido de la cinta de trabajo en espacio $O(\log_2 n)$ (el contenido de la cinta de entrada de sólo lectura no se considera). Y además, cada configuración se puede generar en espacio $O(\log_2 n)$ (y así en tiempo polinomial), porque la función $\log_2 n$ es espacio-construible (ejercicio 8.6.a). En síntesis, M_{DL} escribe primero poly(n) vértices, después poly(n) arcos, y finalmente los vértices INI y FIN, y la escritura de cada vértice le lleva tiempo poly(n), por lo que en total, para generar la terna (G, INI, FIN), tarda tiempo polinomial.

Por otro lado, D-ACC se puede decidir en tiempo polinomial a partir de (G, INI, FIN), de tamaño polinomial con respecto a w.

Así, cualquier lenguaje decidible no determinísticamente en espacio logarítmico se puede decidir determinísticamente en tiempo polinomial, es decir, NLOGSPACE $\subseteq$ P.

Fin del teorema

Incorporando a la clase NLOGSPACE en la sucesión anterior de inclusiones, queda:

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP}$$

y la conjetura más aceptada sigue siendo que todas las inclusiones son estrictas.

Dos últimas características de la clase NLOGSPACE para destacar tienen que ver con los efectos del no determinismo en el espacio, ya mencionados antes en general. Una, de acuerdo al Teorema de Savitch, es que los lenguajes de la clase NLOGSPACE se pueden decidir determinísticamente en espacio $O((\log_2 n)^2)$. La otra, de acuerdo al Teorema de Immerman, es que si un lenguaje está en NLOGSPACE, también está en NLOGSPACE su complemento. Por ejemplo, considerando el lenguaje D-ACC, se puede afirmar que el espacio $O((\log_2|G|)^2)$ alcanza para decidir determinísticamente si un grafo dirigido G tiene un camino del vértice i al vértice j, lo que no resulta intuitivo, y también que el espacio $O(\log_2|G|)$ es suficiente para decidir no

determinísticamente (o verificar determinísticamente) si un grafo dirigido G no tiene un camino del vértice i al vértice j, resultado aún menos intuitivo.

Identificando con CO-NLOGSPACE a la clase de los lenguajes que son complementos de los lenguajes de NLOGSPACE, entonces de acuerdo al Teorema de Immerman se cumple:

$$\text{NLOGSPACE} = \text{CO-NLOGSPACE}$$

Las clases PSPACE Y NPSPACE

Analizamos ahora las clases de los lenguajes decidibles en espacio polinomial. La conjetura más aceptada es que P $\subset$ PSPACE, es decir que PSPACE incluiría lenguajes no tratables, conjetura reforzada por el hecho de que NP $\subseteq$ PSPACE.

Un lenguaje representativo de la clase PSPACE es QSAT, la versión de SAT con fórmulas booleanas totalmente cuantificadas (*quantified SAT*). También se lo conoce como QBF (*quantified boolean formulas*). El lenguaje se define de la siguiente manera: QSAT = {θ | θ es una fórmula booleana totalmente cuantificada, en *forma normal prenexa*, y es verdadera}. Una fórmula booleana totalmente cuantificada no tiene variables libres, y está en forma normal prenexa (o *FNP*) si todos sus cuantificadores se encuentran al inicio. Por ejemplo:

$$\theta_1 = \forall x_1 \exists x_2 \colon (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

es una fórmula booleana que pertenece a QSAT (es verdadera), y en cambio la fórmula:

$$\theta_2 = \exists x_1 \forall x_2 \colon (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

no pertenece a QSAT (es falsa). El requerimiento de que las fórmulas estén en FNP corresponde a la definición clásica del lenguaje. En todo caso, toda fórmula booleana con cuantificadores se puede convertir a una fórmula equivalente en FNP en tiempo (y espacio) polinomial. Suele usarse la notación $Q_1x_1 \ldots Q_mx_m$: $\varphi(x_1, \ldots, x_m)$, tal que Q_i es un cuantificador existencial $\exists$ o un cuantificador universal $\forall$, y $\varphi(x_1, \ldots, x_m)$ es una fórmula booleana sin cuantificadores. También se usa la forma más abreviada $Q_1x_1 \ldots Q_mx_m$: φ.

Notar que $\varphi(x_1, \ldots, x_m)$ es satisfactible sii $\exists x_1,,,\exists x_m$: $\varphi(x_1, \ldots, x_m)$ es verdadera, por lo que QSAT, al igual que SAT, es NP-difícil. Pero a diferencia de SAT, QSAT no estaría en NP. En realidad, ni siquiera estaría en CO-NP. Observar, por ejemplo, que un certificado para verificar si la fórmula $\theta = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \colon x_1 \wedge x_2 \wedge x_3 \wedge x_4$ es verdadera es un *árbol de valores de verdad*, compuesto por algún valor de x_1, los dos valores de x_2, algún valor de x_3 y los dos valores de x_4. Con respecto al espacio, se cumple que QSAT pertenece a PSPACE:

Ejemplo 8.4. Mostramos a continuación una MT M que decide QSAT en espacio polinomial.

M invoca a una función recursiva Eval con una fórmula booleana θ totalmente cuantificada y en FNP, y acepta sii Eval(θ) devuelve el valor *verdadero*. La función Eval tiene la siguiente forma:

Eval(θ):
Si $\theta = \exists x: \theta_1$:
  Si Eval(θ_1) = *verdadero* con x = *verdadero*, devolver *verdadero*.
  Si Eval(θ_1) = *verdadero* con x = *falso*, devolver *verdadero*.
  Devolver *falso*.
Si $\theta = \forall x: \theta_1$:
  Si Eval(θ_1) = *falso* con x = *verdadero*, devolver *falso*.
  Si Eval(θ_1) = *falso* con x = *falso*, devolver *falso*.
  Devolver *verdadero*.
Si $\theta = \varphi$: devolver $\text{Eval}_c(\varphi)$.

La condición θ = φ se da cuando θ tiene sólo constantes lógicas (la función Eval_c corresponde a uno de los típicos algoritmos de evaluación a los que nos referimos previamente).

Ejercicio. Probar que M decide QSAT.

Se cumple además que M ocupa espacio polinomial: tanto la profundidad de la recursión como el espacio requerido en cada una de sus instancias es O(n), y en consecuencia utiliza en total $O(n^2)$ celdas.
Fin del ejemplo

QSAT está entre los lenguajes más difíciles de la clase PSPACE (es *PSPACE-completo*, definición que formalizamos después), y representa un problema que pertenece a una familia de problemas que capturan la esencia de la clase. Se trata de los problemas que consisten en determinar si en un juego de dos jugadores, con información perfecta, el primer jugador cuenta con una estrategia ganadora. Más precisamente, un problema de dicha familia se plantea en los siguientes términos: dos jugadores J_1 y J_2 hacen sus jugadas alternadamente sobre un tablero u otra estructura, visible en todo momento por ambos (por eso se dice que la información es perfecta), y hay que determinar si existe una jugada 1 de J_1, tal que para toda jugada 1 de J_2, existe una jugada 2 de J_1, tal que para toda jugada 2 de J_2, …, J_1 gana. Ejemplos de juegos de esta naturaleza son el ajedrez, las damas, el go, el juego del hexágono y el juego de geografía, entre otros, pero para lograr el tratamiento asintótico que se necesita, el tamaño del tablero o de la estructura que se utilice se debe expresar en términos de n.

En el caso particular de QSAT, los jugadores son ∃ y ∀, la estructura que se utiliza es una fórmula booleana φ sin cuantificadores con variables x_1, x_2, x_3, x_4, …, y el juego consiste en que

$\exists$ y $\forall$ asignen alternadamente un valor de verdad a una variable de φ, primero $\exists$ a x_1, después $\forall$ a x_2, después $\exists$ a x_3, después $\forall$ a x_4, etc., ganando $\exists$ si φ resulta verdadera o ganando $\forall$ si φ resulta falsa. Así, para que haya una estrategia ganadora para $\exists$, tiene que haber alguna asignación a x_1, tal que para toda asignación a x_2, exista alguna asignación a x_3, tal que para toda asignación a x_4, …, la fórmula φ resulte verdadera, lo que significa que $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots$: φ tiene que pertenecer a QSAT.

En cualquier caso, para determinar si existe una estrategia ganadora para el primer jugador, se deben considerar todas las posibles alternativas a partir de su primera jugada, lo que recuerda las búsquedas exhaustivas en NP, pero ahora con certificados no sucintos.

En lo que hace a la clase NPSPACE, como según el Teorema de Savitch un lenguaje decidible no determinísticamente en espacio $O(n^k)$ es decidible determinísticamente en espacio $O(n^{2k})$, con k constante, se cumple:

$$PSPACE = NPSPACE$$

es decir que en espacio polinomial el no determinismo no tiene impacto, no amplía el conjunto de lenguajes que se pueden decidir. En términos de la definición alternativa de NPSPACE: si en espacio polinomial se puede verificar la pertenencia de una cadena w a un lenguaje L (leyendo y procesando en simultáneo un certificado posible de w como describimos en la sección anterior), en espacio polinomial directamente se la puede decidir determinísticamente.

Como la clase PSPACE es cerrada con respecto al complemento, identificando con CO-NPSPACE a la clase de los lenguajes que son complementos de los lenguajes de NPSPACE, se obtiene:

$$NPSPACE = CO\text{-}NPSPACE$$

que en términos de verificadores significa que si en espacio polinomial se puede verificar un posible certificado, en el mismo espacio se puede verificar su inexistencia. La igualdad NPSPACE = CO-NPSPACE también se obtiene del Teorema de Immerman.

Lenguajes completos en la complejidad espacial

Ya dicho, la completitud es un concepto fundamental de la complejidad computacional. Permite identificar el grado de dificultad de una clase de lenguajes, temporal o espacial, a través de lenguajes representativos, *completos* de dicha clase. Hasta ahora sólo consideramos la completitud en la clase temporal NP. En esta sección, con la que finalizamos nuestro breve estudio sobre la complejidad espacial (agregamos luego un par de notas adicionales en las que profundizamos en la jerarquía espacial), nos enfocamos en la completitud en dos clases

espaciales que presentamos, NLOGSPACE y PSPACE. Primero formalizamos algunos conceptos.

Vimos que si un lenguaje es NP-completo, es decir si pertenece a NP y todos los lenguajes de NP se reducen polinomialmente a él, entonces se encuentra entre los lenguajes más difíciles de NP, no pertenece a P a menos que P = NP, y así identifica el grado de dificultad de NP. Este mismo mecanismo se puede generalizar a todas las clases, con la salvedad de que en las clases relacionadas con el espacio logarítmico (LOGSPACE y NLOGSPACE), las reducciones polinomiales utilizadas no pueden exceder el espacio logarítmico. Consecuentemente, dichas reducciones, que vamos a llamar *log-space*, tienen que manejarse de una manera particular. Las describimos a continuación (usaremos $L_1 \leq_{log} L_2$ para indicar que existe una reducción log-space de L_1 a L_2, y de ahora en más llamaremos *poly-time* a las reducciones polinomiales generales, es decir las que no necesariamente son de espacio logarítmico):

Por un lado, por definición, en la medición del espacio de las máquinas de Turing que las computan no se consideran sus cintas de entrada de sólo lectura ni sus cintas de salida de sólo escritura.

Por otro lado, para que se cumplan las propiedades requeridas por el mecanismo mencionado, que son:

- si $L_1 \leq_{log} L_2$ y $L_2 \in$ LOGSPACE, entonces $L_1 \in$ LOGSPACE, y
- si $L_1 \leq_{log} L_2$ y $L_2 \in$ NLOGSPACE, entonces $L_1 \in$ NLOGSPACE,

o en palabras, que LOGSPACE y NLOGSPACE sean cerradas con respecto a las reducciones log-space, las MT M_1 y M_2 de espacio logarítmico involucradas, M_1 para computar la reducción de L_1 a L_2 y M_2 para decidir L_2, no se ejecutan secuencialmente como en la prueba del teorema 7.1 con reducciones poly-time. La razón es que M_1 puede generar una cadena de tamaño polinomial con respecto al tamaño de su entrada, por lo que si se la envía entera a M_2 se estaría violando el espacio logarítmico. Lo que se hace en cambio es que M_1 le pasa a M_2 cada vez sólo el símbolo que M_2 necesita; si M_2 se mueve a la derecha sobre su entrada, M_1 le pasa el símbolo siguiente de su salida, y si M_2 se mueve a la izquierda sobre su entrada, M_1 se vuelve a ejecutar desde el principio, obtiene de su salida el símbolo que M_2 necesita y se lo pasa.

Con estas consideraciones, la completitud en cualquier clase de lenguajes y el mecanismo de prueba asociado se definen de la siguiente manera:

Un lenguaje L es *C-difícil* con respecto a las reducciones poly-time (log-space) sii para todo lenguaje L_1 de la clase C se cumple $L_1 \leq_p L$ ($L_1 \leq_{log} L$), y si además pertenece a la clase C, entonces es *C-completo* con respecto a las reducciones correspondientes. Así, dadas dos clases de lenguajes C_1 y C_2 tales que $C_1 \subseteq C_2$, si un lenguaje L es C_2-completo con respecto a las reducciones correspondientes, entonces $L \notin C_1$ a menos que se cumpla $C_1 = C_2$.

Instanciando estas definiciones a clases de lenguajes que hemos descripto, tanto de la jerarquía espacial como de la jerarquía temporal, podemos establecer:

- Si un lenguaje L es NLOGSPACE-completo con respecto a las reducciones log-space, entonces se cumple L ∉ LOGSPACE a menos que LOGSPACE = NLOGSPACE.
- Si un lenguaje L es P-completo con respecto a las reducciones log-space, entonces se cumple L ∉ NLOGSPACE a menos que NLOGSPACE = P.
- Si un lenguaje L es NP-completo con respecto a las reducciones poly-time, entonces se cumple L ∉ P a menos que P = NP (visto en el capítulo anterior).
- Si un lenguaje L es PSPACE-completo con respecto a las reducciones poly-time, entonces se cumple L ∉ NP a menos que NP = PSPACE.

Para establecer la completitud en la clase LOGSPACE, en cambio, no alcanza ni siquiera con las reducciones log-space, se debe recurrir a otra alternativa, porque todos los lenguajes de LOGSPACE se pueden reducir entre sí con reducciones log-space, lo que se prueba de la misma manera que se prueba que todos los lenguajes de P se pueden reducir entre sí con reducciones poly-time (ejercicio 7.2).

Ejercicio. Probar que existe una reducción log-space entre cualquier par de lenguajes de LOGSPACE distintos de ∅ y Σ*. *Ayuda: considerar la reducción que resuelve el ejercicio 7.2.*

Hechas las formalizaciones que necesitábamos, ahora sí probamos, completando la sección, que el lenguaje D-ACC es NLOGSPACE-completo con respecto a las reducciones log-space, y que el lenguaje QSAT es PSPACE-completo con respecto a las reducciones poly-time.

Teorema 8.4. El lenguaje D-ACC es NLOGSPACE-completo con respecto a las reducciones log-space.

Prueba. Ya probamos que D-ACC pertenece a NLOGSPACE (ejemplo 8.3), y en realidad ya desarrollamos gran parte de la prueba de que D-ACC es NLOGSPACE-difícil, empleando el método de accesibilidad (teorema 8.3):

Partiendo de un lenguaje L cualquiera de NLOGSPACE y una MTN M_{NL} con una cinta de trabajo que lo decide en espacio logarítmico, hemos construido una MTD M_{DL} que en tiempo polinomial, a partir de una cadena w, genera una terna formada por un grafo dirigido G, su vértice inicial INI y su vértice final FIN, tal que:

- Los vértices de G representan todas las configuraciones por las que puede pasar M_{NL}, sus arcos conectan los pares de vértices que representan configuraciones consecutivas, INI representa la configuración inicial, y FIN representa una configuración de aceptación.
- M_{DL} escribe sobre su cinta de salida primero los vértices de G, luego sus arcos, y finalmente los vértices INI y FIN, reutilizando toda vez el mismo espacio.

De este modo, como $w \in L$ sii G tiene un camino del vértice INI al vértice FIN, M_{DL} no es sino una MTD que computa una reducción poly-time de L a D-ACC. Así que lo único que queda por probar es que dicha reducción es log-space. Lo hacemos a continuación:

La cinta de salida de M_{DL} no cuenta en el cálculo del espacio, los vértices y arcos de G ocupan $O(\log_2|w|)$ celdas, se pueden generar en dicho espacio porque la función $\log_2 n$ es espacio-construible, y cada generación y escritura de un vértice reutiliza el espacio ocupado por la generación y escritura del vértice anterior.

Fin del teorema

Teorema 8.5. El lenguaje QSAT es PSPACE-completo con respecto a las reducciones poly-time.

Prueba. Ya probamos que QSAT pertenece a PSPACE (ejemplo 8.4). Comentamos a continuación sólo la idea general de la prueba de que QSAT es PSPACE-difícil, la cual guarda similitud con la prueba de la NP-completitud de SAT (teorema 7.4).

Dado un lenguaje L de PSPACE y una MT M con una cinta que lo decide en espacio polinomial p(n), y así en $c^{p(n)}$ pasos, siendo c una constante que depende de M, la prueba consiste en definir una reducción poly-time que asigne a toda cadena w, con $|w| = n$, una fórmula booleana θ_w totalmente cuantificada, que exprese una computación de M a partir de w de espacio p(n), y que sea verdadera sólo si $w \in L$.

Vamos a mostrar principalmente cómo se puede generar una fórmula θ_w de dichas características en tiempo polinomial (a pesar de que la computación expresada por θ_w puede tener una cantidad exponencial de configuraciones):

- θ_w tiene variables c_{ix}, tales que i varía entre 1 y p(n), y x es un símbolo del alfabeto de M o un par con un símbolo del alfabeto de M y un estado de M. La semántica de c_{ix} es que es verdadera sii en la posición i de la computación de M a partir de w está el símbolo x.
- θ_w expresa la configuración inicial y una configuración de aceptación de M a partir de w, y la condición que debe cumplir una configuración para ser la siguiente de otra.
- Si $F_k(C_1, C_2)$ expresa que M puede pasar de una configuración C_1 a una configuración C_2 en a lo sumo 2^k pasos, INI(C) expresa que C es la configuración inicial, y FIN(C) expresa que C es una configuración de aceptación, entonces:

$$\theta_w = \exists C_0 \exists C_f : F_{p(n).\log 2c}(C_0, C_f) \wedge INI(C_0) \wedge FIN(C_f)$$

  siendo $\exists C_0$ y $\exists C_f$ conjuntos de variables c_{ix} cuantificadas existencialmente, expresa que M acepta w en a lo sumo $2^{p(n).\log 2c} = c^{p(n)}$ pasos. θ_w sería la fórmula buscada si no fuera porque su tamaño no permite que pueda escribirse en tiempo polinomial: $\exists C_0$, $\exists C_f$, $INI(C_0)$ y $FIN(C_f)$ ocupan espacio polinomial, pero $F_{p(n).\log 2c}(C_0, C_f)$ involucra $2^{p(n).\log 2c}$ configuraciones.
- Una primera redefinición de θ_w podría consistir en desdoblar F_k de la siguiente manera:

$$F_k(C_1, C_2) = \exists C: F_{k-1}(C_1, C) \wedge F_{k-1}(C, C_2)$$

pero así el tamaño de θ_w sigue resultando exponencial.

- La siguiente redefinición, en cambio, logra lo que necesitamos:

$$F_k(C_1, C_2) = \exists S\, \exists T\, \exists C: ((S = C_1 \wedge T = C) \vee (S = C \wedge T = C_2)) \longrightarrow F_{k-1}(S, T)$$

Escribimos F_k como una implicación para que sea más legible, pero su forma correcta, equivalente, es la de una disyunción con el antecedente negado.

Notar que ahora F_k se expresa en términos de *una sola subfórmula* F_{k-1}, y por lo tanto, partiendo de $k = p(n).\log_2 c$, θ_w queda con una cantidad polinomial de símbolos, por lo que se puede escribir en tiempo polinomial.

Fin del teorema

Reflexión. La prueba de la PSPACE-completitud del lenguaje QSAT refuerza la evidencia de la íntima relación que existe entre la complejidad computacional y la lógica.

Por ejemplo, antes mostramos que los lenguajes de la clase NP se pueden especificar por medio de fórmulas lógicas existenciales que expresan la búsqueda de certificados, y que el grado de dificultad del lenguaje SAT identifica al de todos los lenguajes de NP.

Y en este capítulo mostramos que el grado de dificultad de QSAT identifica al de todos los lenguajes de la clase PSPACE, y que dichos lenguajes se pueden especificar en términos de fórmulas booleanas totalmente cuantificadas que expresan la búsqueda de estrategias ganadoras para el primer jugador en juegos de dos jugadores con información perfecta.

Estos y varios ejemplos más comprueban la relevancia de la lógica en la complejidad computacional, entre otras cosas como alternativa válida para resolver algunos de sus problemas abiertos.

Fin de la reflexión

En la figura 8.4 combinamos en una única jerarquía espacio-temporal, dentro de la clase EXP, las jerarquías temporal y espacial que hemos descripto, asumiendo las conjeturas más aceptadas.

Más allá de la clase PSPACE

Cerramos este capítulo dando una rápida mirada a la región que en la figura 8.4 se extiende más allá de la frontera de la clase PSPACE.

La clase NEXP, como la clase EXP, es la clase de los lenguajes decidibles en tiempo $O(c^{poly(n)})$, con c constante, pero con máquinas de Turing no determinísticas.

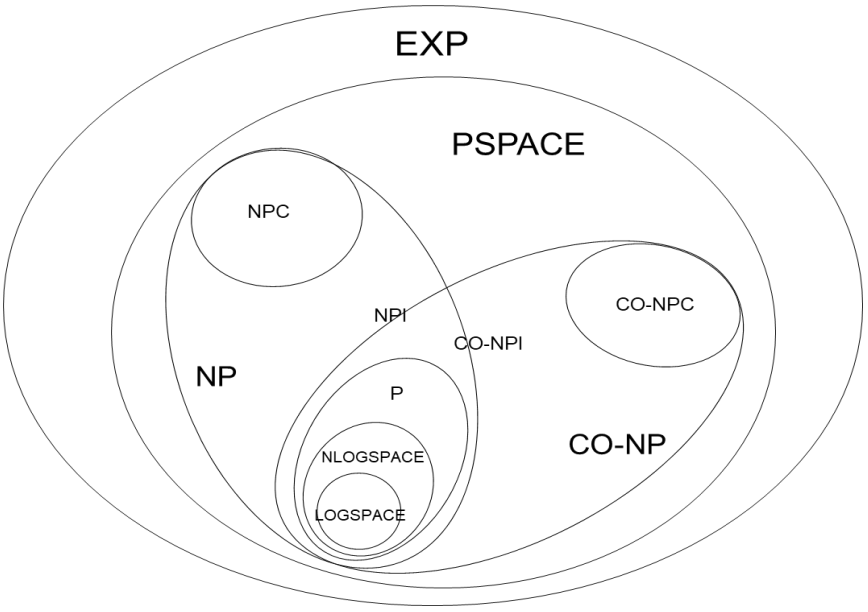

Figura 8.4. La jerarquía espacio-temporal dentro de EXP, asumiendo las conjeturas más aceptadas.

NEXP incluiría estrictamente a EXP, lo mismo que la clase CO-NEXP de los complementos de los lenguajes de NEXP (se repite el mismo tipo de conjeturas que mencionamos en relación a las clases P, NP y CO-NP).

A su vez, incluyendo a NEXP y CO-NEXP estaría la clase EXPSPACE de los lenguajes decidibles en espacio $O(c^{poly(n)})$, con c constante. Y luego seguirían las clases 2-EXP (tiempo doble exponencial), 3-EXP (tiempo triple exponencial), 4-EXP (tiempo cuádruple exponencial), etc., sin solución de continuidad hasta la misma frontera de la clase R de los lenguajes recursivos.

Mientras que al día de hoy podemos decir, de acuerdo a lo que fuimos definiendo y probando, que dentro de PSPACE los lenguajes NP-completos, CO-NP-completos y PSPACE-completos *serían intratables*, en lo que hace al resto de la jerarquía espacio-temporal podemos afirmar que la habitan lenguajes *efectivamente intratables*, es decir que su intratabilidad se puede probar directamente, sin apelar a ninguna conjetura: como la clase EXP incluye estrictamente a la clase P (teorema 6.3), los lenguajes por lo menos EXP-completos no pueden ser decididos en tiempo polinomial. Entre ellos hay dos grupos que se destacan:

Un primer grupo es el de los lenguajes que corresponden a problemas sobre grafos y fórmulas booleanas que se representan sucintamente. En particular, los problemas con grafos sucintos son de sumo interés en el área del diseño de circuitos integrados, donde se requieren sofisticadas técnicas de codificación. Un ejemplo es el lenguaje asociado al problema del circuito hamiltoniano en un grafo sucinto, codificado de una manera determinada, el cual se prueba que es NEXP-completo. Otro ejemplo de lenguaje NEXP-completo relacionado con esta clase de representaciones, ahora con fórmulas booleanas, es el del problema de satisfactibilidad de las fórmulas booleanas codificadas mediante circuitos sucintos.

El segundo grupo de lenguajes intratables corresponde a problemas de decisión en sistemas axiomáticos (lógica de predicados, aritmética, teoría de conjuntos, etc). Por ejemplo, el lenguaje que representa el problema de decisión en el fragmento decidible de la lógica de predicados con

fórmulas de Schönfinkel-Bernays (fórmulas en FNP, con los cuantificadores existenciales primero, sin símbolos de función ni de igualdad), es NEXP-completo. Otro ejemplo es el lenguaje correspondiente al problema de decisión en la teoría de los números reales con la suma (sin la multiplicación), que es NEXP-difícil. También el lenguaje asociado al problema de decisión en el fragmento decidible de la aritmética conocido como *aritmética de Presburger* (con la suma y sin la multiplicación), tiene un grado de dificultad muy grande; en este caso, para decidirlo se requiere como mínimo tiempo doble exponencial no determinístico.

Notas adicionales

Incluimos tres teoremas que caracterizan a la jerarquía espacial. El primero, conocido como Teorema de la Jerarquía Espacial, prueba que la jerarquía espacial es densa. Los dos teoremas restantes, el Teorema de Savitch y el Teorema de Immerman, se refieren al impacto del no determinismo en el espacio ocupado por los algoritmos.

El Teorema de la Jerarquía Espacial

Al igual que la jerarquía temporal, la jerarquía espacial es densa, dada una clase espacial siempre hay otra clase espacial que la incluye estrictamente (por el Teorema de Compresión de Cintas, el salto entre las funciones espaciales que definen las clases debe superar un factor constante). Demostramos a continuación dicha característica de la jerarquía espacial.

Teorema 8.6. Si $S_1(n)$ y $S_2(n)$ son dos funciones espacio-construibles mayores o iguales que $\log_2 n$, y $S_2(n)$ es mayor que $S_1(n)$ cuando n tiende a infinito, entonces se cumple que la clase espacial SPACE(S(n)), siendo S(n) el máximo entre $S_1(n)$ y $S_2(n)$ para todo n, incluye estrictamente a la clase espacial SPACE($S_1(n)$).

Prueba. La idea general de la prueba es básicamente la misma que la del Teorema de la Jerarquía Temporal (teorema 6.2), por diagonalización:

Dadas dos funciones $S_1(n)$ y $S_2(n)$ con las características enunciadas, una MT M_2 de espacio $S_2(n)$, con una cinta de trabajo, puede ejecutar hasta el final cada MT M_1 de espacio $S_1(n)$ a partir de su código $<M_1>$, por lo que aceptando sii M_1 rechaza, puede decidir un lenguaje no perteneciente a la clase SPACE($S_1(n)$). Si $S_2(n) \leq S_1(n)$ hasta un determinado n, la prueba sigue siendo válida porque permitiendo códigos de máquinas de Turing precedidos por cualquier cantidad de ceros, siempre habrá códigos $<M_1>$ lo suficientemente grandes como para que valga $S_2(|M_1|) > c.S_1(|M_1|)$ (el factor c relaciona los tamaños de los alfabetos de las dos máquinas).

De esta manera se prueba que existe un lenguaje en SPACE($S_2(n)$) – SPACE($S_1(n)$), y por la definición de la función S(n), que la clase SPACE(S(n)) incluye estrictamente a SPACE($S_1(n)$).

Notar que a diferencia del Teorema de la Jerarquía Temporal, en el que se agrega un factor logarítmico a la función $S_1(n)$ para contemplar el retardo provocado por la simulación de máquinas de Turing con cualquier cantidad de cintas por medio de una máquina de Turing con sólo dos cintas, en este teorema no se agrega ningún factor (una máquina de Turing con varias cintas de trabajo de espacio S(n) se puede simular con una máquina de Turing con una sola cinta de trabajo en el mismo espacio S(n)).

Fin del teorema

El Teorema de Savitch y el Teorema de Immerman

Empezamos demostrando el Teorema de Savitch, que relaciona los espacios ocupados por la ejecución de una máquina de Turing no determinística y su simulación determinística.

Teorema 8.7. Una MTN M_1 de espacio S(n), tal que S(n) es una función espacio-construible mayor o igual que $\log_2 n$, se puede simular con una MTD M_2 de espacio $O(S^2(n))$.

Prueba. El teorema se prueba aplicando el método de accesibilidad que describimos antes. La idea general es la siguiente. Dada una MTN M_1 de espacio S(n), para simularla determinísticamente una MTD M_2 no necesita mantener en memoria las $c^{S(n)}$ configuraciones por las que puede pasar M_1, siendo c una constante que depende de M_1, sino que en todo momento le alcanza con O(S(n)) configuraciones, empleando un algoritmo recursivo y reutilizando espacio. De este modo, como una configuración puede representarse en O(S(n)) celdas, y generarse en O(S(n)) celdas por ser S(n) espacio-construible, M_2 resulta de espacio $O(S^2(n))$. Formalmente:

Dada una entrada w, M_2 genera primero la configuración inicial C_0 y luego iterativamente todas las posibles configuraciones de aceptación C_{f1}, C_{f2}, … de M_1 a partir de w, verificando en cada iteración si alguna configuración C_{fi} es alcanzable desde w en a lo sumo $c^{S(n)}$ pasos, en cuyo caso acepta (si al cabo de todas las iteraciones no encuentra ninguna, rechaza). En una iteración, M_2 ejecuta un algoritmo recursivo y reutiliza el espacio ocupado en la iteración anterior. Utilizando la notación $C_1 \vdash_k C_2$ para expresar que M_1 puede pasar de su configuración C_1 a su configuración C_2 en a lo sumo 2^k pasos, el algoritmo recursivo se define de este modo:

En toda instancia invocada, con parámetros C_1 y C_2, M_2 genera en el orden canónico todas las posibles configuraciones C de M_1 a partir de w, verifica si existe o no alguna configuración C que cumpla $C_1 \vdash_{k-1} C$ y $C \vdash_{k-1} C_2$, y devuelve el valor *verdadero* o *falso*, respectivamente, a la instancia invocante. Ocupa así cada vez, básicamente, lo que ocupan las configuraciones C_1, C_2 y C, más lo que ocupa verificar $C_1 \vdash_{k-1} C$ o $C \vdash_{k-1} C_2$ (es una disyunción porque el espacio se reutiliza).

De esta manera, M_2 acepta w sii M_1 la acepta. Además, verificar si $C_0 \vdash_{S(n),\log 2c} C_{fi}$, es decir si M_1 a partir de su entrada w puede pasar de su configuración inicial a una de sus configuraciones de aceptación en a lo sumo $2^{S(n).\log 2c} = c^{S(n)}$ pasos, ocupa $O(S^2(n))$:

Por un lado, la profundidad de la recursión nunca supera el espacio O(S(n)).

Por otro lado, las configuraciones de M_1 se pueden representar en espacio O(S(n)) (y generar en espacio O(S(n)) porque S(n) es una función espacio-construible): con O(S(n)) celdas se puede representar una posición del cabezal de la cinta de entrada, una posición del cabezal de la cinta de trabajo (asumimos sin perder generalidad que es única), un estado y un contenido de la cinta de trabajo.

Notar la similitud de esta prueba con la de la PSPACE-completitud del lenguaje QSAT (teorema 8.5), en lo que concierne al uso de la recursión y la reutilización del espacio. Pero esta prueba es más simple, ya que no se requiere generar eficientemente una fórmula booleana que exprese una computación de tiempo exponencial.

Fin del teorema

Demostramos ahora el Teorema de Immerman, que relaciona los espacios requeridos para decidir no determinísticamente un lenguaje y su complemento.

Teorema 8.8. Si M es una MTN que decide un lenguaje L en espacio S(n), tal que S(n) es una función espacio-construible mayor o igual que $\log_2 n$, entonces existe una MTN M^C que decide el lenguaje L^C en el mismo espacio.

Prueba. La prueba de este teorema también se hace en base al método de accesibilidad. La idea general es la siguiente. Dada una MTN M que decide un lenguaje L en espacio S(n), contando para toda cadena w con la cantidad de configuraciones de M alcanzables desde w, se puede decidir el lenguaje L^C en espacio O(S(n)). Con esta consideración, la prueba se divide en dos partes:

- Parte 1: Se construye una MTN M_1 de espacio O(S(n)) que dada una entrada w, calcula la cantidad N de configuraciones de M alcanzables desde w. M_1 calcula el mismo valor N en todas sus computaciones de aceptación.
- Parte 2: Se construye una MTN M_2 de espacio O(S(n)) que dada una entrada w y el valor N relacionado, acepta w sii M la rechaza.

Claramente, componiendo secuencialmente las MTN M_1 y M_2 se obtiene una MTN M^C que decide el lenguaje L^C en espacio O(S(n)). Desarrollamos a continuación las dos partes de la prueba:

Comenzamos por la parte 2, la construcción de la MTN M_2. La idea es que M_2 genere en el orden canónico todas las configuraciones posibles de M a partir de w, y acepte sii ninguna de las configuraciones alcanzables desde w es de aceptación. Formalmente, dada una entrada w y el valor N relacionado, M_2 hace:

1. i := 0.
2. Genera la primera configuración C.
3. Ejecuta M. Si encuentra C, si C es de aceptación, rechaza, y si no, hace i := i + 1.
4. Si i = N, acepta.

5. Si C es la última configuración, rechaza, y si no, genera la siguiente configuración C y vuelve al bloque 3.

M_2 ocupa O(S(n)) celdas: las configuraciones de M a partir de w se pueden representar en espacio O(S(n)), se pueden generar en espacio O(S(n)) porque S(n) es espacio-construible, y como a lo sumo son $c^{S(n)}$, siendo c una constante que depende de M, el contador i en base c no excede las S(n) celdas, lo mismo que N. Veamos que M_2 además decide el lenguaje L^C:

- Si $w \in L$, entonces existen a lo sumo N – 1 configuraciones de M alcanzables desde w que no son de aceptación. Así, M_2 rechaza en el bloque 3 por encontrar una configuración de aceptación, o rechaza en el bloque 5 por no encontrar las N configuraciones alcanzables (nunca acepta, porque nunca puede ocurrir i = N). Por lo tanto, M_2 rechaza w.
- Si $w \notin L$, entonces existen N configuraciones de M alcanzables desde w que no son de aceptación. Así, M_2 acepta en el bloque 4 por encontrar las N configuraciones alcanzables, o rechaza en el bloque 5 por no encontrarlas (pero siempre acepta en alguna computación, porque en alguna computación seguro que encuentra las N configuraciones alcanzables). Por lo tanto, M_2 acepta w.

Ahora completamos la prueba con la parte 1, la construcción de la MTN M_1 (describimos sólo la idea general). Identificando con N_k a la cantidad de configuraciones de M alcanzables desde w en a lo sumo k pasos, la idea es que M_1 calcule iterativamente N_1, N_2, N_3, etc., hasta llegar a obtener dos valores N_k y N_{k+1} iguales, que serán el valor N buscado. Precisando, dada una entrada w, M_1 primero calcula N_1 directamente a partir de M y w, y luego calcula iterativamente N_{k+1} a partir de N_k de la siguiente forma:

Genera en el orden canónico todas las configuraciones posibles C de M a partir de w. Por cada configuración C, ejecuta a lo sumo k pasos de M a partir de w, y por cada una de las N_k configuraciones D de M alcanzables desde w que encuentra durante la ejecución, chequea si es igual a C o si C es alcanzable desde D en un paso, en cuyo caso hace $N_{k+1} := N_{k+1} + 1$. Procesadas todas las configuraciones C, si M_1 no detectó las N_k configuraciones, rechaza, y si las detectó, si $N_{k+1} = N_k$ hace $N := N_{k+1}$, escribe N en su cinta de salida y acepta.

Es fácil comprobar que M_1 ocupa espacio O(S(n)) y obtiene el valor N definido.

Fin del teorema

Observaciones finales

- NPI es la clase de los lenguajes de NP de grado de dificultad intermedio entre los grados de dificultad de los lenguajes de P y NPC. Su existencia se prueba si $P \neq NP$. Dos lenguajes de mucho interés computacional que estarían en NPI son los que representan los problemas del isomorfismo de grafos y la factorización.

- La conjetura NP ≠ CO-NP es más fuerte que la conjetura P ≠ NP. Si se cumple, no existen lenguajes NP-completos en la clase CO-NP.
- Las clases pertenecientes a la jerarquía espacial con lenguajes tratables son LOGSPACE y NLOGSPACE. Lenguajes muy representativos de estas clases son los correspondientes al problema de accesibilidad, en el primer caso en grafos no dirigidos y en el segundo caso en grafos dirigidos. En particular, el segundo lenguaje, el lenguaje D-ACC, es NLOGSPACE-completo.
- El grado de dificultad de la clase PSPACE se identifica con el de una familia de lenguajes que representan problemas de búsqueda de una estrategia ganadora para el primer jugador en un juego entre dos jugadores con información perfecta. Un lenguaje de esta familia, PSPACE-completo, es QSAT, la versión de SAT con fórmulas booleanas totalmente cuantificadas.
- El impacto del no determinismo en el espacio es mucho menos pronunciado que en el tiempo. En términos de búsquedas y verificaciones de soluciones, la búsqueda de una solución sólo es cuadráticamente más costosa que su verificación, y el costo de la verificación de una solución coincide con el costo de la verificación de su inexistencia.

Referencias

La existencia de la clase NPI si se cumple P ≠ NP se deriva de un conjunto de resultados presentados por R. Ladner en (Ladner, 1975), relacionados con las reducciones polinomiales y su utilización para clasificar problemas.

El problema del isomorfismo de grafos mereció especial atención recientemente, cuando L. Babai anunció que había encontrado un algoritmo de tiempo cuasipolinomial para resolverlo (Babai, 2016). Más atrás en el tiempo fue noticia otro problema también relacionado con la clase NPI, cuando P. Shor presentó un algoritmo cuántico de tiempo polinomial para factorizar números (Shor, 1994), a partir de lo cual la comunidad científica se esperanzó con la posibilidad de que los algoritmos cuánticos decidieran eficientemente los lenguajes NP-completos. Sobre el aprovechamiento en la criptografía de la dificultad para factorizar, en particular la implementación del sistema criptográfico de clave pública RSA, se puede consultar el capítulo 12 de (Papadimitriou, 1994).

El estudio sistemático de la complejidad espacial comenzó con las publicaciones de (Hartmanis, Lewis y Stearns, 1965) y (Lewis, Stearns y Hartmanis, 1965).

Entre los hitos relevantes de la evolución de la complejidad espacial figuran: el Teorema de Savitch (Savitch, 1970); la prueba de la PSPACE-completitud del lenguaje QSAT (Stockmeyer y Meyer, 1973); la formalización de la relación entre la clase PSPACE y los problemas de búsqueda de una estrategia ganadora del primer jugador en juegos de dos jugadores con información perfecta (Even y Tarjan, 1976); el Teorema de Immerman (Immerman, 1988), también conocido como Teorema de Immerman-Szelepcsényi por haber sido presentado independientemente por ambos investigadores; y la prueba de O. Reingold de que el lenguaje ACC, asociado al problema

de la accesibilidad en los grafos no dirigidos, pertenece a la clase LOGSPACE (Reingold, 2005). Para este último problema ya existía un algorítmico probabilístico con esta complejidad espacial. La prueba de Reingold significó que el no determinismo en las máquinas de Turing simétricas, máquinas en las que todo paso se puede ejecutar también en sentido contrario, se puede simular determinísticamente en el mismo espacio.

En (Meyer y Stockmeyer, 1972) se presentó el primer lenguaje correspondiente a un problema natural decidible en tiempo mínimamente exponencial, lenguaje relacionado con las expresiones regulares que mencionamos en las referencias del capítulo 3.

La complejidad computacional de los problemas de decisión en la teoría de los números reales con la suma y en la aritmética de Presburger, que comentamos en el capítulo, proviene de (Fischer y Rabin, 1974). La teoría de los números reales es decidible incluso con la multiplicación, lo que resulta contraintuitivo teniendo en cuenta la indecibilidad en la aritmética con la multiplicación, como lo establece el Primer Teorema de Incompletitud, y se explica por el hecho de que con los enunciados de primer orden no se puede expresar la propiedad de *ser un número natural*.

Otras lecturas recomendadas:

El capítulo 4 de (Arora y Barak, 2009) y el capítulo 8 de (Moore y Mertens, 2011): contienen abundantes conceptos y técnicas que se relacionan con la complejidad espacial, como la verificación de certificados, el método de accesibilidad, los problemas sobre estrategias ganadoras en juegos de dos jugadores, y los algoritmos recursivos para aprovechar la posibilidad de reutilizar espacio.

Los capítulos 19 y 20 de (Papadimitriou, 1994): en el capítulo 19 se analizan el juego de geografía, el go generalizado y los *juegos contra la naturaleza* (en estos últimos, las jugadas del segundo jugador son aleatorias), y en el capítulo 20 se describen algunos lenguajes completos de la jerarquía espacio-temporal más allá de la clase PSPACE.

Ejercicios

8.1 De acuerdo al Teorema de Ladner, si $P \neq NP$ entonces existe un lenguaje de grafos L en P, tal que el lenguaje NP-completo CH correspondiente al problema del circuito hamiltoniano restringido a los grafos de L pertenece a NPI. Plantear un resultado análogo considerando el lenguaje SAT en lugar del lenguaje CH.

8.2 Sea el lenguaje $L = \{(G_1, G_2) \mid G_1$ es un grafo que tiene un circuito de Hamilton y G_2 es un grafo que no tiene un circuito de Hamilton$\}$. ¿L pertenece a CO-NP?

8.3 Probar que si un lenguaje es NP-completo, su complemento es CO-NP-completo.

8.4 Justificar:

a. Si una función espacial S(n) cumple para toda cadena w que $S(|w|) \geq |w|$, entonces no hace falta recurrir al modelo de máquinas de Turing con una cinta de entrada de sólo lectura.
b. Las máquinas de Turing con una cinta de entrada de sólo lectura pueden ser también las máquinas de Turing estándar en el marco de la complejidad temporal.
c. Si $S_1(n) = O(S_2(n))$, entonces $SPACE(S_1(n)) \subseteq SPACE(S_2(n))$.

8.5 Dada una función S(n) espacio-construible, construir una máquina de Turing de espacio S(n) que a partir de una cadena w genere una cadena de S(|w|) marcas X.

8.6 Probar que las siguientes funciones son espacio-construibles:
a. $S(n) = \log_2 n$.
b. $S(n) = n$.
c. $S(n) = n^2$.
d. $S(n) = 2^n$.
e. $S_1(n) + S_2(n)$, si $S_1(n)$ y $S_2(n)$ son espacio-construibles.
f. $S_1(n).S_2(n)$, si $S_1(n)$ y $S_2(n)$ son espacio-construibles.

8.7 Indicar en qué casos se cumple $SPACE(S_1(n)) \subset SPACE(S_2(n))$:
a. $S_1(n) = \log_2 n$ y $S_2(n) = n^k$, con $k \geq 1$.
b. $S_1(n) = n^k$ y $S_2(n) = n^{k+1}$, con $k \geq 0$.
c. $S_1(n) = n^k$ y $S_2(n) = 2^n$, con $k \geq 0$.
d. $S_1(n) = 2^n$ y $S_2(n) = 2^{n+1}$.

Comentario: la relación entre las clases $SPACE(S_1(n))$ y $SPACE(S_2(n))$ según la relación entre las funciones $S_1(n)$ y $S_2(n)$, se describe en una nota adicional.

8.8 Una cadena de paréntesis está balanceada si tiene igual número de paréntesis de apertura que de cierre, y nunca tiene más paréntesis de cierre que de apertura contando desde su inicio. Por ejemplo, la cadena (()()) está balanceada, pero la cadena ())(() no. Probar que el lenguaje de las cadenas de paréntesis balanceadas pertenece a LOGSPACE.

8.9 Probar la equivalencia de las dos definiciones que formulamos de la clase NLOGSPACE, una en términos de máquinas de Turing no determinísticas que deciden lenguajes y la otra en términos de máquinas de Turing determinísticas que los verifican.

8.10 Probar que $NLOGSPACE \subseteq P$, mostrando que una máquina de Turing determinística de tiempo polinomial puede simular una máquina de Turing no determinística de espacio logarítmico.

8.11 Sea f: $\Sigma^* \longrightarrow \mathcal{N}$ una función asociada a una MTN M de tiempo polinomial, tal que a toda cadena w le asigna la cantidad de computaciones de aceptación de M a partir de w. Probar que f se puede computar en espacio polinomial.

8.12 Dijimos que la inclusión NP $\subseteq$ PSPACE se puede probar construyendo una máquina de Turing que itere sobre todos los posibles certificados sucintos de una cadena, y también construyendo una máquina de Turing que simule determinísticamente una máquina de Turing no determinística. Plantear una tercera alternativa de prueba, sabiendo que todos los lenguajes de NP se reducen polinomialmente a SAT. *Ayuda: SAT $\in$ PSPACE*.

8.13 Probar que las reducciones log-space son reflexivas, transitivas y no simétricas.

8.14 Probar que LOGSPACE, NLOGSPACE, P, NP y PSPACE son cerradas con respecto a las reducciones log-space.

8.15 Probar:

a. Si un lenguaje es NLOGSPACE-completo con respecto a las reducciones log-space y está en LOGSPACE, entonces LOGSPACE = NLOGSPACE.
b. Si un lenguaje es P-completo con respecto a las reducciones log-space y está en LOGSPACE (NLOGSPACE), entonces LOGSPACE = P (NLOGSPACE = P).
c. Si un lenguaje es PSPACE-completo con respecto a las reducciones poly-time y está en P (NP), entonces P = PSPACE (NP = PSPACE).

8.16 El lenguaje $L_{U\text{-}SK}$ = {(<M>, w, 1^K) | M es una MT con una cinta de trabajo que acepta w utilizando a lo sumo K celdas} representa el problema de aceptación en un espacio acotado. Probar que $L_{U\text{-}SK}$ es PSPACE-completo. *Ayuda: ver el ejemplo 7.5*.

CAPÍTULO 9
Temas avanzados de complejidad computacional

En este último capítulo presentamos, de una manera muy introductoria, algunos temas avanzados de la complejidad computacional.

En las primeras dos secciones tratamos la complejidad temporal de los *problemas de búsqueda* y los *problemas de conteo*. En particular, en el contexto de los primeros analizamos los *problemas de optimización* y las *aproximaciones polinomiales* planteadas para resolverlos.

En las dos secciones que siguen volvemos a los problemas de decisión y a los lenguajes que los representan. Describimos un par de jerarquías temporales que permiten profundizar en dos regiones de la jerarquía espacio-temporal: la *jerarquía polinomial*, y la *jerarquía NC* de los lenguajes decidibles por *algoritmos paralelos eficientes*. Para ello utilizamos otros modelos computacionales, como los *circuitos booleanos*.

Finalmente, las últimas tres secciones las dedicamos a la complejidad temporal de los algoritmos pertenecientes a dos paradigmas alternativos al paradigma determinístico que utilizamos a lo largo de todo el libro. Se trata de los *algoritmos probabilísticos* (utilizamos las *máquinas de Turing probabilísticas* y los *sistemas de pruebas interactivas*) y los *algoritmos cuánticos* (utilizamos los *circuitos cuánticos*). Al final de la última sección ilustramos cómo se relaciona la jerarquía espacio-temporal con las que se consideran las clases probabilística y cuántica de los lenguajes tratables.

La complejidad temporal de los problemas de búsqueda y los problemas de conteo

Desarrollar los capítulos anteriores en términos de lenguajes, o lo que es lo mismo en términos de problemas de decisión tal como definimos en el capítulo 2, facilitó la presentación de los temas, pero sin perder generalidad en relación a los aspectos que nos interesaban. En esta sección, enfocándonos fundamentalmente en los lenguajes NP-completos, ilustramos dicha no pérdida de generalidad con algunas extrapolaciones a los *problemas de búsqueda* y los *problemas de conteo* asociados.

Comenzamos por los problemas de búsqueda, que consisten en obtener, dada una instancia, una solución de la misma. Utilizamos la convención de nomenclatura de anteponer a los nombres de los problemas de búsqueda una F, como FACC, FSAT, etc., lo mismo que a los nombres de

las clases correspondientes, como FP, FNP, etc. (la F es por *problema de función* o *function problem*, como también se identifica a un problema de búsqueda).

Es obvio que si un problema de búsqueda FA puede resolverse en tiempo polinomial, también puede decidirse en tiempo polinomial el lenguaje A asociado: dada una MT M_{FA} de tiempo polinomial que resuelve FA, una MT M_A que a partir de una entrada w ejecuta M_{FA} y acepta sii M_{FA} obtiene una solución, decide A en tiempo polinomial. Así, si A es NP-completo, FA es intratable a menos que P = NP.

Otro resultado, menos intuitivo, es que si un lenguaje NP-completo A fuera tratable (y por lo tanto P = NP), entonces también lo sería el problema de búsqueda FA correspondiente. Antes de mostrar un ejemplo y de comentar la prueba de dicho resultado, vamos a introducir las *Cook-reducciones*, herramienta muy útil en este caso.

Una Cook-reducción es una Turing-reducción (definida en el capítulo 5) con dos particularidades: es de tiempo polinomial, y se puede definir entre problemas de decisión y/o de búsqueda. Formalmente, una Cook-reducción de un problema de decisión o de búsqueda A a un problema de decisión o de búsqueda B es una MT M^B, es decir una MT M con un oráculo de B, que en tiempo polinomial resuelve A, decidiendo la existencia de soluciones u obteniéndolas según el caso. Con $A \leq_C B$ expresaremos que existe una Cook-reducción de A a B. Notar que de acuerdo a esta definición, el oráculo B puede devolver una cadena.

De esta manera, las Cook-reducciones son una generalización, por un lado de las reducciones poly-time (o m-reducciones polinomiales), planteadas entre los problemas de decisión, y por otro lado de las reducciones polinomiales entre los problemas de búsqueda, que no vimos antes. Estas últimas se denominan *Levin-reducciones*, y se definen de la siguiente forma: una Levin-reducción de un problema de búsqueda FA a un problema de búsqueda FB es un par de funciones f y g computables en tiempo polinomial, que cumplen, para toda instancia w: $w \in FA$ sii $f(w) \in FB$, y v es una solución de f(w) sii g(v) es una solución de w.

En el ejemplo siguiente mostramos, por medio de una Cook-reducción, que si el lenguaje SAT fuera decidible en tiempo polinomial, entonces el problema de búsqueda FSAT asociado también podría resolverse en tiempo polinomial:

Ejemplo 9.1. Se cumple $FSAT \leq_C SAT$. La siguiente MT M^{SAT} resuelve FSAT en tiempo polinomial. Dada una fórmula booleana φ con variables $x_1, \ldots, x_m$, M^{SAT} hace (seguimos omitiendo la validación sintáctica de las entradas, y volvemos a asumir por defecto que las fórmulas booleanas no tienen cuantificadores):

1. Invoca al oráculo de SAT con φ. Si el oráculo responde negativamente, entonces responde que no hay solución.
2. Invoca al oráculo de SAT con φ, tal que x_1 = verdadero. Si el oráculo responde positivamente, fija como *verdadero* el valor de x_1. y si no, lo fija como *falso*.
3. Invoca al oráculo de SAT con φ, tal que x_1 tiene el valor obtenido antes y x_2 = verdadero. Si el oráculo responde positivamente, fija como *verdadero* el valor de x_2. y si no, lo fija como *falso*.

4. Y así sigue de la misma manera con la variable x_3, considerando los valores de x_1 y x_2 obtenidos previamente, con la variable x_4, considerando los valores de x_1, x_2 y x_3, etc., hasta llegar a la variable x_m. obteniendo una asignación de valores de verdad que satisface φ.

La figura 9.1 muestra la Cook-reducción definida.

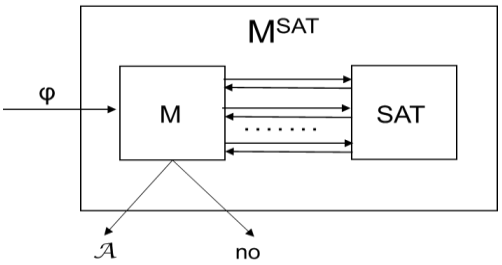

Figura 9.1. M^{SAT} es una Cook-reducción de FSAT a SAT.

Claramente, M^{SAT} resuelve FSAT, y lo hace eficientemente, porque las O(|φ|) iteraciones que efectúa, con asignaciones de valores de verdad a variables booleanas e invocaciones al oráculo de SAT, tardan tiempo polinomial. Notar que asumimos que el oráculo de SAT admite fórmulas booleanas con constantes; en caso contrario, antes de cada invocación M^{SAT} debe llevar a cabo una simplificación para eliminarlas, lo que también tarda tiempo polinomial y así la prueba sigue siendo válida.

De este modo, asumiendo la tratabilidad de SAT, FSAT también resulta tratable.

Ejercicio. Probar la última observación.
Fin del ejemplo

En definitiva, FSAT es tratable si SAT es tratable, y por lo dicho antes, también se cumple la implicación recíproca. Se dice en este caso que FSAT y SAT son *polinomialmente equivalentes*. La propiedad de un problema de búsqueda FA de ser Cook-reducible al lenguaje A asociado se conoce como *autorreducibilidad*. Por lo tanto, FSAT es *autorreducible*. Se prueba que todos los problemas de búsqueda correspondientes a lenguajes NP-completos son autorreducibles, y así, que son polinomialmente equivalentes a dichos lenguajes. La prueba se basa, como en el ejemplo 9.1, en la construcción de una máquina de Turing con oráculo que a lo largo de sucesivas invocaciones va extendiendo un prefijo que termina siendo una solución de su entrada.

Un tercer resultado refuerza aún más la no pérdida de generalidad mencionada al principio de la sección, porque es aplicable a toda la clase NP. Lo formalizamos a continuación:

Teorema 9.1. Asumiendo P = NP, todo problema de búsqueda correspondiente a un lenguaje de NP se puede resolver en tiempo polinomial.

Prueba. En otras palabras, el teorema establece que P = NP implica FP = FNP.

La demostración se basa en la reducción poly-time f utilizada para probar la NP-completitud del lenguaje SAT (teorema 7.4), aprovechando que es en realidad una Levin-reducción. En efecto, f no sólo es una función de tiempo polinomial que cumple $w \in L$ sii $f(w) \in SAT$ para toda cadena w y todo lenguaje L de NP, sino que también permite transformar eficientemente una solución de w en una asignación de valores de verdad que satisface la fórmula booleana definida por f(w) y viceversa. Incluso, con una pequeña modificación de f, se puede lograr una biyección entre las soluciones de L y SAT (las reducciones de este tipo se denominan *parsimoniosas*, y se utilizan en la mayoría de las pruebas de NP-completitud).

En consecuencia, asumiendo P = NP, y por lo tanto que SAT pertenece a P, ejecutando combinadamente la Levin-reducción referida y el algoritmo que prueba la autorreducibilidad de FSAT (ejemplo 9.1), se puede resolver eficientemente cualquier problema de FNP.

Fin del teorema

Ejercicio. Utilizando el mecanismo descripto en la prueba del teorema anterior, probar que si se cumple P = NP, entonces el problema de búsqueda del isomorfismo de grafos se puede resolver en tiempo polinomial.

Por su parte, los problemas de conteo consisten en calcular, dada una instancia, la cantidad de sus soluciones. Un resultado obvio, análogo al que formulamos con los problemas de búsqueda, es que si un problema de conteo puede resolverse en tiempo polinomial, entonces también puede decidirse en tiempo polinomial el lenguaje asociado.

Ejercicio. Justificar el resultado mencionado.

Por lo tanto, si un lenguaje es NP-completo, el problema de conteo correspondiente es intratable a menos que P = NP.

En el marco ampliado de todos los lenguajes de la clase NP, para el análisis de su relación con los problemas de conteo se utiliza la clase #P, integrada por funciones $f_L: \Sigma^* \longrightarrow \mathcal{N}$, cada una definida para un lenguaje L de NP, de modo tal que para toda cadena w de L, $f_L(w)$ calcula su número de soluciones. Por ejemplo, f_{SAT} es la función definida para el lenguaje SAT, que calcula, dada una fórmula booleana, el número de asignaciones de valores de verdad que la satisfacen, y corresponde al problema de conteo #SAT (por convención de nomenclatura, se antepone un # a los nombres de los problemas de conteo de #P).

A su vez, las relaciones temporales entre los miembros de #P se establecen mediante reducciones definidas como las Levin-reducciones, salvo que la función g, en lugar de asignar soluciones, asigna cantidades de soluciones (resulta conveniente utilizar en este ámbito reducciones parsimoniosas). En términos de dichas reducciones se define la #P-completitud, con la semántica habitual. Se prueba que #SAT es #P-completo.

De esta manera, de acuerdo al resultado mencionado previamente, si #SAT o cualquier otra función #P-completa fuera computable en tiempo polinomial, se cumpliría P = NP.

No se sabe si la implicación recíproca es cierta, lo que marca una diferencia con la clase FNP (teorema 9.1). El solo hecho de que un problema de decisión sea tratable no significa que el problema de conteo asociado lo sea. Un ejemplo es el *problema del emparejamiento perfecto en un grafo bipartito* (*perfect matching in a bipartite graph)*. Un grafo es bipartito si sus vértices se pueden particionar en dos subconjuntos V_1 y V_2 independientes (es decir, con vértices no adyacentes dos a dos), y tiene un emparejamiento perfecto si existe un subconjunto de arcos no adyacentes que conectan a todos los vértices de V_1 con todos los vértices de V_2. Mientras que se puede decidir en tiempo polinomial si un grafo bipartito tiene un emparejamiento perfecto, no parece que se pueda contar en tiempo polinomial cuántos emparejamientos perfectos tiene, dado que la cuenta consiste en calcular la *permanente* de la *matriz de adyacencia* del grafo (otra representación habitual de los grafos, con celdas [i, j] con 1 o 0 según los vértices i y j sean o no adyacentes, respectivamente), cálculo que como indicamos anteriormente no sería eficiente (de hecho, la función asociada es #P-completa).

Los problemas de optimización y las aproximaciones polinomiales

Los *problemas de optimización*, casos especiales de los problemas de búsqueda, han sido una de las motivaciones principales para el desarrollo de la complejidad temporal en general y de la NP-completitud en particular.

Un problema de optimización es un problema de búsqueda que consiste en obtener, dada una instancia, una *solución óptima* según una medida o costo que se asocia a las soluciones, de valor máximo (*problema de maximización*) o mínimo (*problema de minimización*), como por ejemplo una asignación de valores de verdad que satisfaga el máximo número de cláusulas de una fórmula booleana en FNC, un cubrimiento de vértices de tamaño mínimo de un grafo, etc.

Al igual que un problema de búsqueda general, si un problema de optimización tiene resolución eficiente entonces el lenguaje asociado también la tiene. Por ejemplo, sea OCV el problema de búsqueda de un cubrimiento de vértices de tamaño mínimo de un grafo (como convención de nomenclatura, antepondremos una O a los nombres de los problemas de optimización y de las clases correspondientes). Se cumple que si el problema OCV fuera tratable, entonces el lenguaje asociado CV = {(G, K) | G es un grafo y tiene un cubrimiento de vértices de tamaño K} también lo sería: dada una MT M_{OCV} de tiempo polinomial que resuelve OCV, una MT M_{CV} que a partir de un par (G, K) ejecuta M_{OCV} sobre G y acepta sii M_{OCV} devuelve un cubrimiento de vértices de tamaño a lo sumo K, decide CV en tiempo polinomial.

Ejercicio. Justificar la última observación.

De esta manera, un problema de optimización cuyo lenguaje asociado sea NP-completo no puede resolverse eficientemente a menos que P = NP (por ejemplo el problema OCV, dado que el lenguaje CV es NP-completo tal como probamos en el ejemplo 7.2). Y entonces, a los fines prácticos, para resolver un problema de optimización con un lenguaje asociado NP-completo se suele recurrir, cuando existe, a una *aproximación polinomial*, que es un algoritmo de tiempo polinomial que obtiene para toda instancia una solución *cercana* a la óptima, de acuerdo a un criterio determinado. Formalmente:

Dado un problema de optimización OA y una MT M de tiempo polinomial que resuelve el problema de búsqueda correspondiente, llamando opt(w) a la medida de una solución óptima de una instancia w y m(M(w)) a la medida de la solución obtenida por M al ejecutarse a partir de w, se define que M es una *Ɛ-aproximación polinomial* para el problema OA sii para toda instancia w se cumple:

$$\frac{|m(M(w)) - opt(w)|}{\max(m(M(w)), opt(w))} \leq Ɛ$$

De la definición se obtiene que Ɛ, conocido como *error relativo*, varía entre 0 y 1 (lo que se pretende, obviamente, es que esté lo más cerca posible del 0), y que según el problema sea de maximización o de minimización, m(M(w)) nunca resulta menor que opt(w)**.**(1 – Ɛ) ni mayor que opt(w)/(1 – Ɛ), respectivamente.

En el ejemplo siguiente describimos una aproximación polinomial para el problema de optimización OCV del cubrimiento de vértices de un grafo (volvemos a asumir por defecto que los grafos son no dirigidos):

Ejemplo 9.2. Mostramos una Ɛ-aproximación polinomial M_{ACV} para el problema OCV (enseguida indicamos el valor del error relativo Ɛ). Dado un grafo G = (V, E), la MT M_{ACV} que presentamos a continuación obtiene en tiempo polinomial un cubrimiento de vértices V_c de G:

1. Hace $V_c := \emptyset$ y $E_c := E$.
2. Si $E_c = \emptyset$, acepta.
3. Hace $E_c := E_c - \{(i, j)\}$, siendo (i, j) algún arco de E_c.
4. Si $i \notin V_c$ y $j \notin V_c$, hace $V_c := V_c \cup \{i, j\}$.
5. Vuelve al bloque 2.

El conjunto V_c generado por la MT M_{ACV} es un cubrimiento de vértices del grafo G: tiene los dos vértices de todos los arcos de un conjunto A de arcos no adyacentes de G tal que cualquier otro arco de G es adyacente a algún arco de A. En la figura 9.2 mostramos un ejemplo de cubrimiento V_c generado por M_{ACV} (marcamos los arcos del conjunto A correspondiente).

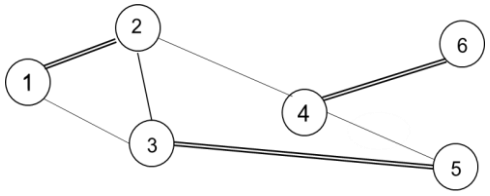

Figura 9.2. Ejemplo de un cubrimiento de vértices V_c generado por la MT M_{ACV}.

En el ejemplo, el cubrimiento V_c es todo el conjunto V (un cubrimiento mínimo es {2, 3, 4}). Como $|A| = |V_c| / 2$, y todo cubrimiento de vértices C de G debe incluir al menos un vértice de todo arco de A, se cumple $|C| \geq |A| = |V_c| / 2$, lo que significa que el tamaño de V_c es a lo sumo el doble del óptimo. En cuanto al tiempo de ejecución de M_{ACV}, el mismo es polinomial, porque M_{ACV} itera |E| veces y en cada iteración recorre dos veces V_c, siendo $|V_c| \leq |V|$. En definitiva, el algoritmo construido es una 1/2-aproximación polinomial para el problema OCV.

Fin del ejemplo

Se prueba que si $P \neq NP$, existen problemas de optimización que no son aproximables polinomialmente, es decir problemas de optimización que no cuentan con Ɛ-aproximaciones polinomiales para ningún Ɛ. En lugar de desarrollar la prueba, presentamos en lo que sigue un ejemplo que refleja su idea general.

Ejemplo 9.3. Vamos a probar que asumiendo $P \neq NP$, el problema de optimización OPVC del viajante de comercio, que plantea encontrar un circuito de Hamilton de longitud mínima de un grafo completo ponderado, no es aproximable polinomialmente (en el ejemplo 7.7 demostramos que el lenguaje asociado PVC es NP-completo).

La idea es probar, asumiendo $P \neq NP$, que si existe una Ɛ-aproximación polinomial para OPVC, entonces el lenguaje CH del problema del circuito hamiltoniano se puede decidir en tiempo polinomial, lo que no es posible porque CH es NP-completo según lo mencionado en el ejemplo 7.7. Formalmente, si M_{APVC} es una Ɛ-aproximación polinomial para OPVC, la siguiente MT M_{CH} decide CH en tiempo polinomial. Dado un grafo $G_1 = (V, E)$, M_{CH} hace:

1. Genera un grafo G_2 completando el grafo G_1 y ponderándolo con el valor 1 en todos los arcos que pertenecen a G_1 y con el valor $|V|/(1 – Ɛ)$ en los arcos restantes.
2. Ejecuta M_{APVC} y acepta sii M_{APVC} devuelve un circuito de Hamilton ponderado con una longitud igual a |V|.

Claramente, M_{CH} tarda tiempo polinomial. Veamos que además decide el lenguaje CH:

- Si el circuito de Hamilton ponderado que devuelve M_{APVC} mide |V|, quiere decir que el grafo de entrada G_1 tiene un circuito de Hamilton.

- En caso contrario, significa que el circuito de Hamilton ponderado devuelto tiene al menos un arco que mide $|V|/(1 - \varepsilon)$, y así su longitud es estrictamente mayor que $|V|/(1 - \varepsilon)$. Y como M_{APVC} es una ε-aproximación polinomial para un problema de minimización, la longitud de un circuito de Hamilton ponderado óptimo no puede ser menor que $(1 - \varepsilon)$ veces la longitud del circuito de Hamilton ponderado devuelto, lo que implica que un circuito de Hamilton ponderado óptimo mide más que $|V|$, y así G_1 no tiene un circuito de Hamilton.

Fin del ejemplo

La reducción empleada en el ejemplo anterior es típica de esta clase de demostraciones. Las instancias que genera son tales que la brecha entre la medida de una solución óptima cuando la instancia original pertenece al lenguaje de referencia, y la medida de una solución óptima cuando la instancia original no pertenece al lenguaje de referencia, es muy grande.

La clase de los problemas de optimización aproximables polinomialmente, es decir los problemas de optimización que cuentan con ε-aproximaciones polinomiales para algún ε, se denomina APX.

Algunos problemas de optimización de APX cuentan con ε-aproximaciones polinomiales para todo ε. Dichos problemas conforman la clase PAS de los *esquemas de aproximación polinomial* (*polynomial approximation schemes*). Un ejemplo de un problema de optimización de PAS es el problema de maximización correspondiente a una variante del problema de la mochila que mencionamos en los ejemplos 6.8 y 7.6. El problema, cuyo lenguaje asociado es NP-completo, consiste en obtener, dado un conjunto C de m pares (v_i, p_i) que representan el volumen y el peso de m objetos, un subconjunto de C que maximice la suma de los valores v_i en tanto la suma de los valores p_i correspondientes no supere una cota K determinada (representante del peso máximo admisible de una mochila).

Siendo OP y ONP las clases de los problemas de optimización correspondientes a las clases P y NP, respectivamente, si $P \neq NP$ se cumple:

$$OP \subset PAS \subset APX \subset ONP$$

Las inclusiones estrictas $OP \subset PAS \subset APX$ se prueban de la misma manera que se prueba la inclusión estricta $APX \subset ONP$ que acabamos de ejemplificar.

Como en las jerarquías que estudiamos previamente, para poblar esta jerarquía también se recurre a las reducciones y al concepto de completitud. En este caso se utilizan reducciones que preservan la propiedad de *aproximabilidad polinomial*, o *APX-reducciones* (existen varios tipos), que por lo tanto permiten probar la no aproximabilidad polinomial de los problemas de optimización de ONP que sean completos con respecto a las mismas, a menos que P = NP.

En comparación con los lenguajes NP-completos, los problemas ONP-completos con respecto a las APX-reducciones que se conocen al día de hoy son escasos, lo que se justifica por la mayor restrictividad de dichas reducciones, y porque los problemas de optimización asociados a muchos lenguajes NP-completos son aproximables polinomialmente. Un camino

alternativo para obtener resultados en este ámbito es aplicando el Teorema PCP, que mencionamos en una nota adicional.

La jerarquía polinomial

Volviendo a los problemas de decisión, en esta sección describimos una jerarquía temporal que se solapa con la jerarquía espacio-temporal. Se trata de la *jerarquía polinomial* o PH (*polynomial hierarchy*), cuyas clases de lenguajes cubren la franja que se extiende desde la clase P hasta la clase PSPACE, permitiendo caracterizar a muchos lenguajes de interés computacional que estarían fuera de las fronteras de NP y CO-NP.

Los siguientes dos lenguajes sirven para introducir las definiciones que necesitamos:

- MAX-IND = {(G, K) | el grafo G tiene un conjunto independiente de K vértices, y no tiene conjuntos independientes de vértices más grandes}. El lenguaje MAX-IND representa el problema de determinar, dada una entrada (G, K), si el conjunto independiente de vértices más grande que tiene el grafo G, es decir su subconjunto de vértices no adyacentes más grande, tiene tamaño K.
- MIN-FORM = {(φ, K) | la fórmula booleana φ tiene K símbolos, y no existe ninguna fórmula booleana equivalente más chica}. El lenguaje MIN-FORM representa el problema de determinar, dada una entrada (φ, K), si la fórmula booleana φ, de K símbolos, es la más chica del conjunto de todas las fórmulas booleanas equivalentes a ella, siendo dos fórmulas booleanas equivalentes si toda asignación de valores de verdad las evalúa igual.

Si caracterizamos a ambos lenguajes con el formato que utilizamos para caracterizar a los lenguajes de NP y CO-NP, es decir con certificados de tamaño polinomial p y verificadores eficientes M, podemos especificar el lenguaje MAX-IND de la siguiente forma:

$$w \in \text{MAX-IND sii } (\exists x_1: |x_1| \leq p(|w|), \forall x_2: |x_2| \leq p(|w|), M \text{ acepta } (w, x_1, x_2))$$

donde w es un par (G, K) formado por un grafo G y un número natural K, x_1 es un conjunto de K vértices de G, y x_2 es un conjunto de más de K vértices de G, mientras que el lenguaje MIN-FORM se puede especificar así:

$$w \in \text{MIN-FORM sii } (\forall x_1: |x_1| \leq p(|w|), \exists x_2: |x_2| \leq p(|w|), M \text{ acepta } (w, x_1, x_2))$$

donde w es un par (φ, K) formado por una fórmula booleana φ de K símbolos y un número natural K, x_1 es una fórmula booleana de menos de K símbolos, y x_2 es una asignación de valores de verdad.

Ejercicio. Describir el comportamiento de las MT M en cada caso.

Como podemos apreciar, dichas caracterizaciones no son las de un lenguaje de NP, porque no utilizan un único cuantificador existencial, ni las de un lenguaje de CO-NP, porque no utilizan un único cuantificador universal, sino que se basan en la combinación de los dos tipos de cuantificadores, y pareciera que no hay otra manera de formularlas, lo que significa que MAX-IND y MIN-FORM no estarían en NP ni CO-NP.

Teniendo en cuenta estas consideraciones, se definen la clase Σ_2P, con lenguajes como MAX-IND, y la clase Π_2P, con lenguajes como MIN-FORM. Y generalizando con más cuantificadores alternados, se definen las clases Σ_3P, Σ_4P, etc., y las clases Π_3P, Π_4P, etc. Formalmente:

Un lenguaje L pertenece a la clase Σ_iP, con $i \geq 0$, si existen un polinomio p y una MT M de tiempo polinomial que satisfacen, para toda cadena w:

$$w \in L \text{ sii } (\exists x_1: |x_1| \leq p(|w|, \forall x_2: |x_2| \leq p(|w|), \ldots, Q_i x_i: |x_i| \leq p(|w|, M \text{ acepta } (w, x_1, x_2, x_3, \ldots, x_i))$$

donde Q_i es el cuantificador $\exists$ o el cuantificador $\forall$ según i sea impar o par, respectivamente. Y un lenguaje L pertenece a la clase Π_iP si existen un polinomio p y una MT M de tiempo polinomial que satisfacen, para toda cadena w:

$$w \in L \text{ sii } (\forall x_1: |x_1| \leq p(|w|, \exists x_2: |x_2| \leq p(|w|), \ldots, Q_i x_i: |x_i| \leq p(|w|, M \text{ acepta } (w, x_1, x_2, x_3, \ldots, x_i))$$

donde Q_i es el cuantificador $\forall$ o el cuantificador $\exists$ según i sea impar o par, respectivamente. Notar que la clase Π_iP tiene los complementos de la clase Σ_iP.

Ejercicio. Probar la última afirmación. *Ayuda: se deriva directamente de cómo se definen dichas clases.*

Así llegamos a la definición de la jerarquía polinomial o PH: es la unión infinita de las clases Σ_iP, con $i \geq 0$. Se la puede definir también como la unión infinita de las clases Π_iP. PH se estructura por niveles:

- Nivel 0: las clases $\Sigma_0P = \Pi_0P = P$.
- Nivel 1: las clases $\Sigma_1P = NP$ y $\Pi_1P = CO\text{-}NP$.
- Nivel 2: las clases Σ_2P y Π_2P.
- Etc.

Se cumple, para todo $i \geq 0$, que $\Sigma_{i+1}P$ y $\Pi_{i+1}P$ incluyen a Σ_iP y Π_iP. La conjetura más aceptada es que las inclusiones son estrictas, o como se dice habitualmente, que PH *no colapsa*. Precisando, se dice que PH *colapsa en el nivel i* si existe algún i tal que $\Sigma_iP = PH$, es decir, si a partir de un determinado i se cumple $\Sigma_iP = \Sigma_{i+1}P = \ldots$ Se prueba que si P = NP, PH colapsa en el

nivel 0, es decir, P = PH (aun valiendo P ≠ NP, podría suceder que PH colapse). Se prueba además que PH $\subseteq$ PSPACE (ejercicio 9.11), y también en este caso la conjetura más aceptada es que la inclusión es estricta. En la figura 9.3 representamos la jerarquía polinomial, con las conjeturas más aceptadas.

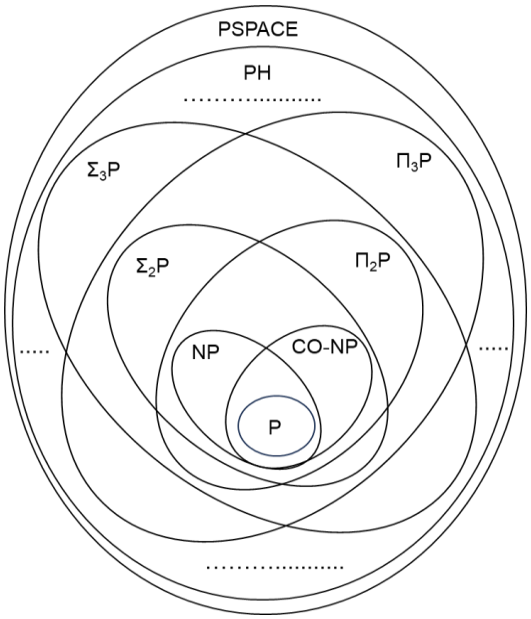

Figura 9.3 La jerarquía polinomial, asumiendo las conjeturas más aceptadas.

La inclusión estricta PH $\subset$ PSPACE se fundamenta en el hecho de que la clase PSPACE tiene lenguajes completos, como por ejemplo el lenguaje QSAT (teorema 8.5), mientras que PH no tendría ninguno, lo que justificamos en lo que sigue (tanto en PSPACE como en PH consideramos la completitud con respecto a las reducciones poly-time):

Teorema 9.2. Si existe un lenguaje PH-completo, entonces PH colapsa.

Prueba. Si L es un lenguaje PH-completo, entonces todo lenguaje L_1 de PH cumple $L_1 \leq_p L$, y como L pertenece a una clase $\Sigma_i P$, también L_1 pertenece a $\Sigma_i P$ (se demuestra fácilmente). En consecuencia, PH $\subseteq \Sigma_i P$, lo que quiere decir que PH colapsa en el nivel i.
Fin del teorema

Al contrario de PH, todas las clases $\Sigma_i P$ y $\Pi_i P$ tienen lenguajes completos. Por ejemplo, todo lenguaje $QSAT_i$, compuesto por las fórmulas booleanas totalmente cuantificadas verdaderas de la forma $\exists x_1 \forall x_2 \exists x_3 \ldots Q_i x_i$: φ, siendo Q_i el cuantificador $\exists$ o el cuantificador $\forall$ según i sea impar o par, respectivamente, es $\Sigma_i P$-completo.

Ejercicio. Todo lenguaje $QSAT_i$ es como el lenguaje QSAT, salvo que sus fórmulas booleanas tienen un número limitado de alternancias entre los cuantificadores existenciales y universales. Vimos que el lenguaje QSAT se puede asociar al problema de determinar, dado un juego entre

dos jugadores con información perfecta, si existe una estrategia ganadora para el primer jugador. ¿A qué problema entonces se podría asociar el lenguaje $QSAT_i$?

Existe una manera alternativa de definir la jerarquía polinomial, utilizando *máquinas de Turing alternantes*, las cuales generalizan las máquinas de Turing no determinísticas:

A diferencia de una MT no determinística, cuyo árbol de computaciones se interpreta exclusivamente con estados de tipo *existencial*, en el sentido de que se evalúan mediante la disyunción de las informaciones aportadas por sus estados hijos, en una MT alternante su árbol de computaciones se interpreta con estados de tipo tanto existencial como *universal*, siendo estos últimos evaluados aplicando una conjunción en lugar de una disyunción. Y en cuanto a su tiempo de ejecución, se define teniendo en cuenta no sólo la cantidad de pasos sino también la cantidad de alternancias entre los estados de tipo existencial y de tipo universal de sus computaciones. Se puede probar fácilmente que para todo $i \geq 1$, la clase $\Sigma_i P$ (respectivamente $\Pi_i P$) de PH es la clase de los lenguajes decididos por MT alternantes de tiempo polinomial con a lo sumo $i - 1$ alternancias entre estados de tipo existencial y de tipo universal, partiendo de un estado de tipo existencial (respectivamente universal), y así, la definición de PH en términos de MT alternantes generaliza la definición de la clase NP en términos de MT no determinísticas. Por ejemplo, el lenguaje MIN-FORM perteneciente a la clase $\Pi_2 P$ se puede decidir con una MT alternante de tiempo polinomial con una sola alternancia, partiendo de un estado de tipo universal (ejercicio 9.12).

Hay todavía una tercera forma de definir la jerarquía polinomial, históricamente la primera, que utiliza máquinas de Turing con oráculo:

Si NP^{SAT} es la clase de los lenguajes decidibles en tiempo polinomial por MT no determinísticas con un oráculo del lenguaje SAT, se prueba que:

- $\Sigma_2 P = NP^{SAT}$ y $\Pi_2 P = CO\text{-}NP^{SAT}$.
- $\Sigma_3 P = NP^{\Sigma_2 P}$ y $\Pi_3 P = CO\text{-}NP^{\Sigma_2 P}$.
- $\Sigma_4 P = NP^{\Sigma_3 P}$ y $\Pi_4 P = CO\text{-}NP^{\Sigma_3 P}$.
- Etc.

equivalencias no tan intuitivas como en el caso de las máquinas de Turing alternantes. Por ejemplo, el lenguaje MAX-IND de la clase $\Sigma_2 P$ se puede decidir con una MT no determinística con un oráculo de SAT (ejercicio 9.13). En realidad, las clases se pueden definir utilizando cualquier lenguaje NP-completo como oráculo, no solamente SAT, y por eso una manera habitual de especificarlas es mediante las expresiones NP^{NP} y $CO\text{-}NP^{NP}$.

Los lenguajes decidibles por algoritmos paralelos eficientes

Otra jerarquía temporal de lenguajes que se solapa con la jerarquía espacio-temporal, esta vez dentro de la clase P, es la jerarquía NC (*Nick's Class*, por Nicholas Pippenger, quien la

definió), creada para estudiar los lenguajes decidibles por *algoritmos paralelos* eficientes, es decir, los lenguajes de la clase P con algoritmos paralelos significativamente más rápidos que los algoritmos secuenciales con los que cuentan, de acuerdo a un criterio determinado.

El modelo computacional más sencillo y extendido empleado para dicho estudio es el *circuito booleano*, o directamente el *circuito*. Otro modelo habitual es la versión paralela de la máquina RAM que presentamos en una nota adicional del capítulo 2, la *máquina paralela de acceso aleatorio* o *PRAM* (*parallel random access machine*), a la que también dedicamos una nota adicional en este capítulo.

Formalmente, un circuito es un grafo dirigido sin ciclos, con vértices que representan variables booleanas x_1, …, x_n, que pueden adoptar los valores uno o cero, o puertas lógicas, que pueden corresponder a las operaciones lógicas *and* ($\wedge$), *or* ($\vee$) o *not* ($\neg$). Los primeros vértices constituyen la entrada del circuito, sin arcos de entrada, y entre los otros vértices, con uno o dos arcos de entrada según sea la operación lógica que representen, se encuentra el que constituye la salida del circuito, sin arcos de salida. Un circuito C acepta una cadena de entrada w de unos y ceros (cada dígito asignado a una variable booleana) sii luego de aplicar a partir de ella las operaciones de sus puertas lógicas obtiene como salida el valor 1, que se expresa con C(w) = 1 (nos enfocamos en la decisión de lenguajes, pero los circuitos, como las máquinas de Turing, pueden también devolver cadenas, para lo cual necesitan varios vértices de salida).

Ejemplo 9.4. En la figura 9.4 representamos un circuito que calcula el *or exclusivo* de sus dos variables de entrada. De esta manera, el circuito acepta las cadenas 01 y 10.

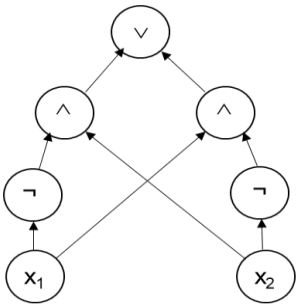

Figura 9.4. Circuito que calcula el *or exclusivo* de x_1 y x_2.

Fin del ejemplo

La cantidad de vértices de un circuito determina su *tamaño*, y la longitud del camino más largo desde uno de sus vértices de entrada a su vértice de salida determina su *profundidad*. Intuitivamente, el tamaño representa el número de procesadores de la máquina paralela modelizada, y la profundidad, el tiempo de ejecución de la misma. Así, sólo los *circuitos polinomiales*, es decir los circuitos de tamaño polinomial, son físicamente realizables, que entonces no pueden decidir eficientemente ningún lenguaje cuyo algoritmo secuencial de decisión más rápido sea del orden exponencial (en particular ningún lenguaje NP-completo, a menos que P = NP).

Ejercicio. Justificar la última observación.

Como los circuitos tienen una cantidad fija de variables de entrada, para poder decidir lenguajes, que en general incluyen cadenas de cualquier tamaño, se utilizan *familias de circuitos*. Una familia de circuitos, denotada con $\{C_n\}_{n\geq 1}$, es una secuencia infinita de circuitos C_n, cada uno de n vértices de entrada, que decide un lenguaje L sii todo circuito C_n decide todas las cadenas de L de tamaño n. Si además existe una máquina de Turing de espacio logarítmico que para todo n construye la descripción del circuito C_n, la familia de circuitos se dice *uniforme*. Dicha condición de constructividad eficiente evita que puedan decidirse lenguajes muy difíciles con circuitos muy simples, incluso indecidibles (ejercicio 9.15), y por otra parte asegura que todos los circuitos implementen en conjunto un único algoritmo, como lo hace una máquina de Turing. La descripción estándar de un circuito consiste en una secuencia de ternas ordenadas e identificadas adecuadamente, cada una con un operador lógico y a lo sumo dos operandos.

La clase de los lenguajes decidibles por *familias de circuitos polinomiales* se denomina $P_{/poly}$. Y la clase de los lenguajes decidibles por *familias uniformes de circuitos polinomiales* no es otra que la clase P, como mostramos a continuación:

Teorema 9.3. Un lenguaje L pertenece a la clase P sii existe una familia uniforme de circuitos polinomiales $\{C_n\}_{n\geq 1}$ que lo decide.

Prueba. Por un lado, si $\{C_n\}_{n\geq 1}$ es una familia uniforme de circuitos polinomiales que decide un lenguaje L, entonces la siguiente MT M decide L en tiempo polinomial: dada una entrada w de tamaño n, primero construye en espacio logarítmico, y así en tiempo polinomial, la descripción del circuito polinomial C_n (posible porque $\{C_n\}_{n\geq 1}$ es uniforme), luego ejecuta C_n a partir de w en tiempo polinomial (porque C_n es polinomial), y finalmente acepta sii $C_n(w) = 1$.

Por otro lado, se demuestra que dada una MT M de tiempo polinomial p(n) que decide un lenguaje L, se puede generar en espacio logarítmico la descripción de un circuito polinomial C_n que acepta una cadena w de tamaño n sii M la acepta. Describimos en lo que sigue las características generales de dicha generación:

- El circuito C_n tiene p(n) + 1 capas de O(p(n)) vértices cada una.
- La capa i de C_n, con $1 \leq i \leq p(n)$, simula el paso i de M, es decir, el pasaje de la configuración producida por el paso i – 1 de M (o la configuración inicial si i = 1) a la configuración producida por el paso i de M. Los vértices de la capa, representantes de las celdas de M, implementan la simulación, constituyendo circuitos de profundidad constante, y teniendo en cuenta que el contenido de la celda en la posición j de la configuración producida en el paso i depende exclusivamente de la función de transición de M y de los contenidos de las celdas en las posiciones j – 1, j y j + 1 de la configuración anterior (los símbolos del alfabeto Γ y los estados del conjunto Q de M se pueden representar con $O(\log_2|\Gamma|)$ y $O(\log_2|Q|)$ dígitos binarios, respectivamente).

- Finalmente, la capa p(n) + 1 simula la respuesta de M. Sus vértices también constituyen circuitos de profundidad constante. La salida de C_n será uno o cero, según la representación del estado final alcanzado por la simulación corresponda a la de un estado de aceptación o de rechazo, respectivamente.

El circuito tiene $O(p(n)^2)$ vértices (es de tamaño polinomial), y es fácil comprobar que se puede generar en espacio logarítmico.
Fin del teorema

Ejercicio. Probar que $P \subseteq P_{/poly}$. *Ayuda: basarse en la prueba del teorema anterior*.

Introducidas las familias uniformes de circuitos polinomiales, ya podemos caracterizar formalmente a los lenguajes con algoritmos paralelos eficientes en los que nos enfocamos en esta sección: son los lenguajes decidibles mediante familias uniformes de circuitos polinomiales que tienen profundidad $O((\log_2 n)^k)$, con $k \geq 1$. Como dijimos al comienzo, la jerarquía que los reúne se denomina NC. La integran así infinitas clases NC^1, NC^2, …, con lenguajes decidibles por familias uniformes de circuitos polinomiales de profundidad $O(\log_2 n)$, $O((\log_2 n)^2)$, … La definición de NC, en lo que hace a su frontera dentro de la clase P, no está tan aceptada. No obstante, al igual que en otras jerarquías, en NC interesan principalmente los lenguajes de sus primeras clases.

Ejemplo 9.5. Mostramos que el lenguaje PARIDAD = {w | w es una cadena de unos y ceros con una cantidad impar de unos} pertenece a la clase NC^1.

El siguiente circuito C_n, en forma de árbol binario, acepta todas las cadenas de unos y ceros de tamaño n con una cantidad impar de unos. Dada una entrada w:

C_n va calculando el resultado desde sus hojas, por medio de la aplicación de puertas lógicas que implementan el *or exclusivo* (como describimos en el ejemplo 9.4). Al final, cuando llega a su nodo raíz, aplica un último *or exclusivo* entre el dígito que recibe de su subárbol izquierdo, correspondiente a los primeros |w|/2 dígitos, y el dígito que recibe de su subárbol derecho, correspondiente a los dígitos restantes de w, y acepta sii obtiene el valor 1.

Ejercicio. Probar que C_n tiene tamaño polinomial, profundidad $O(\log_2 n)$, su descripción se puede generar en espacio logarítmico, y acepta las cadenas del lenguaje PARIDAD de tamaño n.
Fin del ejemplo

Otro ejemplo de problema con algoritmo paralelo eficiente es la suma de dos números. El algoritmo aprendido en la escuela es secuencial, de tiempo lineal, consistente en sumar desde los dígitos menos significativos y mantener un dígito de acarreo a lo largo de todo el cálculo. Pero existe un algoritmo más eficiente, paralelo, de tiempo $O(\log_2 n)$, con O(n) procesos, cada uno dedicado a una posición de dígito, y con comunicaciones que se ocupan de los acarreos.

También cuentan con algoritmos paralelos eficientes, entre otros problemas: las operaciones aritméticas elementales restantes; los ordenamientos y las búsquedas en arreglos; la multiplicación, el determinante y la inversa de matrices; y la clausura transitiva y la búsqueda del camino mínimo en grafos.

Del teorema 9.3 se deriva la inclusión:

$$NC \subseteq P$$

y la conjetura más aceptada es que la inclusión es estricta, es decir que no todos los lenguajes de la clase P cuentan con algoritmos paralelos eficientes. De esta manera, como NC es cerrada con respecto a las reducciones log-space (ejercicio 9.16), si un lenguaje es P-completo con respecto a dichas reducciones, entonces en la práctica se asume que no pertenece a NC (ejercicio 9.17).

Un ejemplo de lenguaje P-completo con respecto a las reducciones log-space es el lenguaje correspondiente al *problema del máximo flujo en una red*, que consiste en maximizar, dada una red con un único punto de entrada y un único punto de salida, el flujo que la puede recorrer. La red se representa por un grafo dirigido ponderado (los números de los arcos representan capacidades), con un vértice inicial sin arcos de entrada y un vértice final sin arcos de salida. El problema es muy representativo de los problemas con resolución eminentemente secuencial: definible su resolución necesariamente en etapas, si bien éstas pueden paralelizarse satisfactoriamente, pueden llegar a sumar una cantidad mayor que un valor polilogarítmico.

Otro lenguaje P-completo con respecto a las reducciones log-space es el lenguaje asociado al *problema de evaluación de circuitos*: EVAL-CIRCUIT = {(C, w) | C es la descripción de un circuito C de n vértices de entrada, w es una cadena de n dígitos binarios, y C(w) = 1}. Su P-completitud también se deriva del teorema 9.3.

Dentro de NC, se destacan las inclusiones:

$$NC^1 \subseteq LOGSPACE \subseteq NLOGSPACE \subseteq NC^2$$

lo que significa que los lenguajes P-completos no sólo no admitirían resoluciones eficientes paralelas con respecto al tiempo, sino tampoco resoluciones eficientes secuenciales con respecto al espacio (como ya comentamos en el capítulo anterior). La inclusión intermedia se cumple por definición. Las otras dos las probamos a continuación:

Teorema 9.4. $NC^1 \subseteq LOGSPACE$ y $NLOGSPACE \subseteq NC^2$.

Prueba. $NC^1 \subseteq LOGSPACE$: Dadas la descripción de un circuito polinomial C_n de profundidad $O(\log_2 n)$ y una cadena w de n dígitos binarios, una máquina de Turing puede evaluar recursivamente C_n a partir de w desde su puerta de salida, utilizando una pila de $O(\log_2 n)$ elementos de tamaño constante.

NLOGSPACE $\subseteq$ NC^2: La idea general de la prueba es la siguiente. Dados un lenguaje L de NLOGSPACE, una MTN M de espacio logarítmico que decide L, y una cadena w de n dígitos binarios, M acepta w sii el grafo de configuraciones G de M a partir de w, tal como lo definimos en el teorema 8.3, tiene un camino de su primer vértice al último. Una manera alternativa a la que vimos previamente para construir G, de m vértices, y decidir si G tiene un camino del vértice 1 al vértice m, es: representar G con una matriz de adyacencia A de m x m; obtener la clausura transitiva de G, la cual se puede calcular haciendo $B = A^m$ (B[i, j] = 1 sii existe un camino del vértice i al vértice j); y finalmente chequear si B[1, m] = 1. En consecuencia, como se demuestra que A y B se pueden construir con un circuito C_n de tamaño polinomial y profundidad $O((\log_2 n)^2)$, siendo la descripción de C_n construible mediante una máquina de Turing de espacio logarítmico, llegamos a que L pertenece a NC^2.

Fin del teorema

En la figura 9.5 mostramos la jerarquía NC y sus relaciones con las clases LOGSPACE y NLOGSPACE dentro de la clase P, teniendo en cuenta las conjeturas más aceptadas.

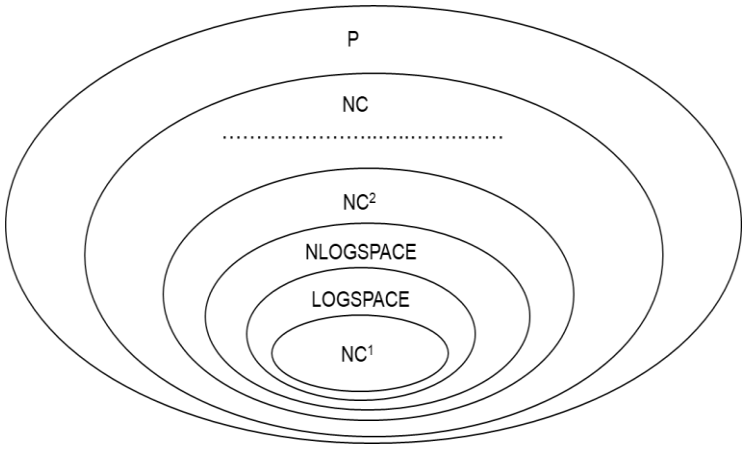

Figura 9.5. NC, LOGSPACE y NLOGSPACE, asumiendo las conjeturas más aceptadas.

Relaciones como las anteriores se pueden generalizar a distintos modelos paralelos con adecuadas restricciones (uniformidad, cantidad de procesadores, etc.), mediante la Tesis de la Computación Paralela, que sostiene que el tiempo paralelo se relaciona polinomialmente con el espacio secuencial. Más precisamente:

$$PTIME(T(n)) \subseteq SPACE(p_1(T(n))) \text{ y } SPACE(S(n)) \subseteq PTIME(p_2(S(n)))$$

siendo p_1 y p_2 dos polinomios, y PTIME(T(n)) una clase de lenguajes decidibles en tiempo paralelo O(T(n)) por medio de un determinado modelo computacional. El modelo de las familias uniformes de circuitos polinomiales satisface la tesis.

Los circuitos también se consideran útiles para encarar la resolución del problema P vs NP. La razón es que la gran mayoría de las funciones que van de las cadenas de unos y ceros al

conjunto {1, 0} no pueden ser computadas por circuitos de tamaño polinomial. Esto significa que encontrando algún lenguaje NP-completo asociado a una función de estas características, se estaría probando P ≠ NP (de acuerdo al teorema 9.3). Lamentablemente, hasta el momento no se han logrado resultados importantes en este sentido, pero el camino de demostración referido sigue siendo de interés, máxime que se conjetura que los lenguajes NP-completos no pueden ser decididos por familias de circuitos polinomiales ni siquiera no uniformes, es decir que no pertenecerían a la clase $P_{/poly}$ (de hecho, se prueba que si SAT perteneciera a $P_{/poly}$, la jerarquía polinomial colapsaría en el segundo nivel).

Los algoritmos probabilísticos en la complejidad temporal

Los *algoritmos probabilísticos* constituyen un área de estudio muy importante en la complejidad computacional. Uno de los motivos, sobre el que nos concentramos en esta sección, es que resuelven muchos problemas de interés computacional de una manera más eficiente o al menos más simple que los algoritmos determinísticos (es decir no probabilísticos) existentes. Y si bien pueden equivocarse y dependen de una *verdadera aleatoriedad* siempre cuestionada, en la práctica su uso, que ya lleva varias décadas, viene resultando sumamente satisfactorio.

Lo que caracteriza a un algoritmo probabilístico es que en ciertos pasos efectúa una *elección aleatoria*, que determina una continuación entre varias alternativas posibles. Esto hace que con una misma entrada, en distintas ejecuciones pueda generar distintas salidas. Dado este comportamiento, sus aceptaciones y rechazos (seguimos utilizando lenguajes) se determinan en base a un análisis probabilístico sobre la distribución de sus respuestas (análisis distinto al que mencionamos en el capítulo 6, basado en la distribución de las entradas, con otro propósito que es el de calcular la complejidad computacional del caso promedio).

El modelo computacional que se utiliza es la *máquina de Turing probabilística* (abreviada con MTP). Formalmente, una MT probabilística:

- A partir de la cadena de entrada, avanza paso a paso siempre eligiendo aleatoriamente una entre dos continuaciones posibles, más allá de que pueda hacer lo mismo en las dos continuaciones.
- Toda elección aleatoria es independiente de la anterior y tiene probabilidad 1/2 (por eso se la refiere comúnmente como un *lanzamiento de moneda* o *coin flip*). Así, su árbol de computaciones posibles asociado es un árbol binario perfecto.
- Puede terminar aceptando o rechazando.

El criterio que se adopta para la aceptación de una cadena depende de la cantidad de computaciones de aceptación (precisaremos esto enseguida, cuando describamos MT probabilísticas con cuatro comportamientos distintos, cada una determinando una clase diferente de lenguajes).

Por lo tanto, de acuerdo a la definición, la sintaxis de una MT probabilística es la misma que la de una MT no determinística (incluyendo la restricción de las dos continuaciones, dado que para toda MTN M_1 existe una MTN M_2 equivalente de grado de no determinismo 2, como pedimos probar en el ejercicio 2.5), pero su semántica es bien distinta: sus bifurcaciones son equiprobables, y sus aceptaciones se basan en un criterio probabilístico.

Otra característica que diferencia a la MT probabilística de la MT no determinística es que la MT probabilística constituye un modelo computacional físicamente realizable (de hecho, todo típico lenguaje de programación tiene un *generador aleatorio* de números).

Dada una MTP M y una entrada w, vamos a identificar con α(M, w) a la relación entre la cantidad de computaciones en las que M acepta w y la cantidad total de computaciones, es decir la *probabilidad de que M acepte w*, y con β(M, w) a la relación entre la cantidad de computaciones en las que M rechaza w y la cantidad total de computaciones, es decir la *probabilidad de que M rechace w*. El tiempo que tarda y el espacio que ocupa una MT probabilística se definen como en el modelo de la MT determinística, incluyendo el criterio del peor caso. En lo que sigue nos enfocaremos en las MT probabilísticas de tiempo polinomial.

Ejemplo 9.6. Como primer ejemplo de MT probabilística describimos una MTP M muy simple, para decidir el lenguaje MAY-SAT de las fórmulas booleanas satisfactibles por más de la mitad de las asignaciones de valores de verdad posibles. Dada una fórmula booleana φ, M hace:

1. Genera aleatoriamente una asignación de valores de verdad $\mathcal{A}$ (a cada una de las variables de φ le asigna aleatoriamente el valor *verdadero* o el valor *falso*).
2. Evalúa φ con $\mathcal{A}$ y acepta sii el resultado es *verdadero*.

Claramente, φ ∈ MAY-SAT sii M acepta φ en más de la mitad de sus computaciones, y M tarda tiempo polinomial p(n), dado un polinomio p (mientras que SAT es NP-completo y #SAT es #P-completo). Con respecto a la probabilidad de error de M:

- La probabilidad de que rechace mal es a lo sumo $1/2 - 1/2^{p(n)}$
- La probabilidad de que acepte mal es a lo sumo 1/2.

Ejercicio. Justificar las probabilidades de error indicadas.
Fin del ejemplo

El ejemplo 9.6 sirve para introducir la primera de las cuatro clases de lenguajes referidas previamente, la clase PP (*tiempo polinomial probabilístico* o *probabilistic polynomial time*). Un lenguaje L pertenece a PP sii existe una MTP M de tiempo polinomial que cumple, para toda cadena w:

- Si w ∈ L entonces α(M, w) > 1/2.

- Si $w \notin L$ entonces $\beta(M, w) \geq 1/2$.

Queda claro que el lenguaje MAY-SAT del ejemplo 9.6 pertenece a PP, la MTP M que construimos satisface las condiciones de la definición anterior, y por lo tanto decide el lenguaje. También podemos concluir que la clase PP no identifica a los lenguajes con algoritmos probabilísticos eficientes. En efecto, la probabilidad de error de máquinas como M puede llegar a ser muy grande, y si bien se puede reducir con varias ejecuciones independientes, adoptando como respuesta la respuesta mayoritaria, para llegar a un valor razonable se necesita una cantidad de iteraciones del orden exponencial (de acuerdo a la *cota de Chernoff* de la teoría de las probabilidades). Otra razón que refuerza esta conclusión es que $NP \subseteq PP$ (ejercicio 9.19).

Una segunda clase de lenguajes, que describimos a continuación, es la que se toma como la clase de los lenguajes tratables en el ámbito de los algoritmos probabilísticos. Se trata de la clase BPP (*tiempo polinomial probabilístico acotado* o *bounded probabilistic polynomial time*), considerada entonces la *clase P probabilística*. Un lenguaje L pertenece a BPP sii existe una MTP M de tiempo polinomial que cumple, para toda cadena w:

- Si $w \in L$ entonces $\alpha(M, w) \geq 2/3$.
- Si $w \notin L$ entonces $\beta(M, w) \geq 2/3$.

Ahora, como se aprecia, la idea es que las aceptaciones y rechazos sean claramente mayoritarias. No obstante, la cota 2/3 se puede reemplazar por otras, mayores que 1/2, por la posibilidad que mencionamos de reducir la probabilidad de error efectuando varias ejecuciones independientes y quedándonos con la respuesta mayoritaria (en este caso, a diferencia de lo que observamos con las máquinas correspondientes a la clase PP, es suficiente una cantidad polinomial de ejecuciones). Esta robustez en la definición de la clase BPP recuerda la de la definición de la clase P.

Ejemplo 9.7. Probamos a continuación que el lenguaje COMP de los números compuestos, es decir los números que no son primos, pertenece a la clase BPP, construyendo una MTP M que lo decide según las características que especificamos recién.

M se basa en una propiedad de los números compuestos, según la cual todo número compuesto c tiene al menos $(c - 1) / 2$ *certificados de composicionalidad*, números que varían entre 1 y $c - 1$ que permiten establecer mediante un algoritmo determinístico de tiempo polinomial que c es compuesto. La propiedad se deriva de una caracterización de los números primos formulada por P. de Fermat, que es que si p es un número primo, entonces todo número *a* entre 1 y $p - 1$ cumple $a^{p-1} = 1 \pmod p$. Aprovechando dicha propiedad de los números compuestos, dado un número N, la máquina M hace:

1. Si N es par, acepta.
2. Hace r := *random*(1, N – 1).

3. Acepta sii r es un certificado de composicionalidad de N.

La función *random*(1, N – 1) devuelve aleatoriamente un número entre 1 y N – 1 (función implementable mediante un generador aleatorio de dígitos 1 y 0).

Los tres bloques del algoritmo tardan tiempo polinomial. Además, si N es par, con seguridad es compuesto, y si N es impar, de acuerdo a la propiedad utilizada de los números compuestos: si N es compuesto, M lo acepta en al menos la mitad de sus computaciones; y si N no es compuesto, M lo rechaza siempre (dado que en ninguna de las computaciones encuentra un certificado de composicionalidad). Por lo tanto, la probabilidad de que M rechace mal es a lo sumo 1/2, y de que acepte mal es 0. Así, con apenas dos ejecuciones independientes, se logra la probabilidad de error requerida.

Ejercicio. Explicar la última afirmación.

La probabilidad de error de M se puede reducir de manera significativa ejecutándola más veces. Por ejemplo, después de 1000 iteraciones independientes, su probabilidad de error es a lo sumo $1/2^{1000}$.

Fin del ejemplo

Entre otros algoritmos probabilísticos de este tipo se encuentran los que resuelven la búsqueda de la mediana de un conjunto de números (de tiempo lineal como los algoritmos determinísticos más rápidos conocidos pero más simple), el testeo de igualdad de dos polinomios multivariados (no se conoce resolución determinística eficiente), y la determinación de si un grafo bipartito tiene un emparejamiento perfecto (de igual tiempo que los algoritmos determinísticos existentes pero con la ventaja de que cuenta con una implementación paralela eficiente).

Las dos clases de lenguajes que nos quedan por presentar están incluidas en la clase BPP. Así, también se relacionan con las resoluciones probabilísticas eficientes, pero por sus especificidades amerita que las tratemos aparte.

Comenzamos por la clase RP (*tiempo polinomial aleatorio* o *randomized polynomial time*). Sus lenguajes se caracterizan por ser decidibles por MT probabilísticas que nunca aceptan mal (algoritmos necesarios, por ejemplo, en el área de la verificación de pruebas, donde puede admitirse rechazar erróneamente pero no aceptar erróneamente). Dichas máquinas se conocen como máquinas *de error unilateral* (*one-side error*), a diferencia de las que describimos antes, que son máquinas *de error bilateral* (*two-side error*). Formalmente, un lenguaje L pertenece a RP sii existe una MTP M de tiempo polinomial que cumple, para toda cadena w:

- Si $w \in L$ entonces $\alpha(M, w) \geq 1/2$.
- Si $w \notin L$ entonces $\beta(M, w) = 1$

Como en la definición de la clase BPP, la cota de la definición de RP, en este caso 1/2, se puede reemplazar por otras, ahora incluso menores que 1/2. Un ejemplo de lenguaje de RP es

el mismo lenguaje COMP de los números compuestos que analizamos en el ejemplo 9.7. También su complemento, el lenguaje PRIMOS, que representa el problema de determinar si un número es primo, cuenta con un algoritmo de este tipo.

Finalmente, la cuarta clase de lenguajes es la clase ZPP (*tiempo polinomial probabilístico de error cero* o *zero-error probabilistic polynomial time*). Su nombre se debe a que las MT probabilísticas correspondientes no se equivocan nunca, responden apropiadamente o de una manera indefinida (cuentan con un tercer estado final, *no sé* o *don´t know*). Estos algoritmos pertenecen a la familia de los *algoritmos de Las Vegas* (los anteriores, que se pueden equivocar, son de la familia de los *algoritmos de Monte Carlo*). Formalmente, un lenguaje L pertenece a ZPP sii existe una MTP M de tiempo polinomial que cumple, para toda cadena w:

- Si $w \in L$ entonces $\alpha(M, w) \geq 1/2$ y $\beta(M, w) = 0$.
- Si $w \notin L$ entonces $\beta(M, w) \geq 1/2$ y $\alpha(M, w) = 0$.

Se cumple ZPP = RP ∩ CO-RP, siendo CO-RP la clase de los complementos de RP (es decir que los lenguajes de CO-RP son decidibles por MT probabilísticas que nunca rechazan mal). La prueba de la inclusión ZPP $\subseteq$ RP ∩ CO-RP no reviste mayor dificultad (se obtiene resolviendo los ejercicios 9.18 y 9.21). Con respecto a la inclusión recíproca, RP ∩ CO-RP $\subseteq$ ZPP, en lugar de probarla mostramos directamente un ejemplo que ilustra cómo hacerlo:

Ejemplo 9.8. Que se cumpla RP ∩ CO-RP $\subseteq$ ZPP significa que si dos lenguajes L y L^C cuentan con un algoritmo correspondiente a la clase RP, es decir un algoritmo que nunca acepta mal y puede rechazar mal con probabilidad a lo sumo 1/2, entonces L cuenta también con un algoritmo correspondiente a la clase ZPP, es decir un algoritmo que nunca se equivoca y puede no responder nada con probabilidad a lo sumo 1/2. Tomando los lenguajes PRIMOS y COMP de RP, vamos a probar que PRIMOS también pertenece a ZPP (lo mismo puede probarse para COMP).

Dadas una MTP M_1 que decide PRIMOS y una MTP M_2 que decide COMP, las cuales nunca aceptan mal y pueden rechazar mal con probabilidad a lo sumo 1/2, construimos la siguiente MTP M para decidir PRIMOS de acuerdo a la definición de la clase ZPP. Dado un número N, M hace:

1. Ejecuta M_1. Si acepta, acepta (N es primo porque M_1 nunca acepta mal).
2. Ejecuta M_2. Si acepta, rechaza (N es compuesto porque M_2 nunca acepta mal).
3. Responde *no sé*.

M tarda tiempo polinomial. Además, la probabilidad de que se equivoque es 0, y de que no responda nada es a lo sumo 1/2, porque es la probabilidad de que M_1 y M_2 rechacen, lo que sucede cuando alguna de las dos se equivoca (sólo una se puede equivocar, no puede ocurrir que las dos rechacen mal, porque N es un número primo o es un número compuesto). Por lo

tanto, el lenguaje PRIMOS pertenece a ZPP. Adicionalmente, llevando a cabo varias ejecuciones independientes de M, hasta un límite polinomial, la probabilidad de que M no responda nada se reduce significativamente.

Fin del ejemplo

Las clases descriptas se relacionan entre sí y con P y NP de la siguiente forma:

$$P \subseteq ZPP \subseteq RP \subseteq BPP \subseteq PP$$
$$RP \subseteq NP \subseteq PP$$

Todas las inclusiones se prueban fácilmente (ejercicios 9.18 y 9.19), y no se sabe si son estrictas. En particular, en lo que hace a la relación entre las clases P y BPP, la conjetura más aceptada, sobre todo en los últimos años, es que son iguales (y así también iguales a las clases ZPP y RP). En efecto, de investigaciones recientes se ha adquirido mucha evidencia, considerando ciertas asunciones, de que hay un modo de *desaleatorizar*, es decir transformar en un algoritmo determinístico, todo algoritmo probabilístico correspondiente a la clase BPP, con un retardo sólo polinomial.

El componente central de dichas investigaciones es el *generador pseudoaleatorio* (*pseudorandom generator*), algoritmo determinístico que genera cadenas que *parecen* aleatorias, básicamente amplificando pequeñas cadenas aleatorias llamadas *semillas*. Como lograr verdadera aleatoriedad es en general difícil, en los algoritmos probabilísticos es común utilizar generadores pseudoaleatorios como fuentes aleatorias sustitutas. Se han obtenido generadores pseudoaleatorios muy sofisticados, capaces de producir, asumiendo la existencia de las funciones de un solo sentido que mencionamos en el capítulo anterior, cadenas indistinguibles de las cadenas aleatorias frente a testeos de tiempo polinomial.

Asumiendo P = NP se prueba que BPP = P, dado que se cumple $BPP \subseteq PH$, y como indicamos en la sección sobre la jerarquía polinomial, P = NP implica que PH colapsa en P. No se sabe si $BPP \subseteq NP$, pero sí que $BPP \subseteq P_{/poly}$, así que $SAT \notin BPP$ a menos que PH colapse, según lo que establecimos en la sección sobre los circuitos booleanos (PH colapsa si se cumple $SAT \in P_{/poly}$). También se cumple:

$$BPP \subseteq EXP$$

porque BPP se puede definir alternativamente en términos de MT determinísticas que reciben una cadena adicional (como en la definición de la clase NP), en este caso con dígitos 1 y 0 con los que pueden simular lanzamientos de moneda consecutivos, y de esta manera, como en tiempo polinomial p(n) hay que considerar $2^{p(n)}$ cadenas de este tipo, toda MT probabilística que decide un lenguaje de BPP puede simularse en tiempo exponencial con una MT determinística de las características mencionadas.

No se conocen lenguajes completos en BPP (obviamente existen si BPP = P). La dificultad para encontrarlos se debe a que la propiedad que caracteriza a las máquinas relacionadas con la clase, de aceptar o rechazar con probabilidad al menos 2/3, es una propiedad *semántica* (e indecidible), a diferencia, por ejemplo, de las MT no determinísticas, que se caracterizan *sintácticamente* (simplemente como códigos válidos de máquinas de Turing).

La misma noción de aleatoriedad que se aplica a una máquina de Turing que decide un lenguaje se puede aplicar a una máquina de Turing que reduce un lenguaje a otro. Así se define, con las mismas cotas requeridas para las MT probabilísticas relacionadas con la clase BPP, una *reducción aleatoria* entre dos lenguajes L_1 y L_2 ($L_1 \leq_r L_2$ expresa que existe una reducción aleatoria de L_1 a L_2). Utilizando reducciones aleatorias se facilita el poblamiento de BPP, dado que BPP es cerrada con respecto a dichas reducciones.

Los sistemas de pruebas interactivas

La caracterización de un lenguaje de la clase NP mostrada en el capítulo 6, basada en la existencia de un certificado verificable en tiempo polinomial para cada una de sus cadenas, resalta la importancia del chequeo eficiente de una prueba independientemente del esfuerzo requerido para construirla. En particular, la hemos asociado a una interacción eficiente entre un *probador* P (máquina de Turing con poder ilimitado) y un *verificador* V (máquina de Turing de tiempo polinomial), en la que para toda cadena w, si w pertenece a L, entonces P puede convencer a V de dicha pertenencia enviándole un certificado (la interacción es *completa*), y si w no pertenece a L, entonces P no puede convencer a V cualquiera sea el certificado que le envíe (la interacción es *sensata*).

Intuitivamente, una interacción de este tipo pero con varios intercambios debería permitir decidir más lenguajes que el esquema de envío completo y posterior lectura completa de una prueba. En el plano real, un ejemplo es el de un profesor que debe demostrar un teorema a sus alumnos: si en lugar de desarrollar la prueba y luego responder preguntas permite interrupciones durante su exposición para aclarar dudas, se evita tener que pensar por adelantado todas las respuestas a todas las preguntas posibles, y así asegura más la comprensión de su explicación. Sin embargo, esto no ocurre: un lenguaje de NP también se puede caracterizar mediante un modelo interactivo. Formalmente, el modelo se denomina *sistema de pruebas interactivas determinísticas* (o DIP, por *deterministic interactive proofs*), y se especifica de la siguiente forma:

- Está compuesto por un par de máquinas de Turing que se ejecutan conjuntamente, una MT P, el probador, de poder ilimitado, y una MT V, el verificador, de tiempo polinomial. Las dos máquinas comparten una cinta de entrada de sólo lectura.
- En toda ejecución a partir de una entrada w, P y V pueden ejecutar tanto acciones internas como intercambios de mensajes. En todos los mensajes se puede considerar la información

de w y de los mensajes anteriores. La cantidad y tamaño de los mensajes es polinomial con respecto a |w|.

- V es quien termina las ejecuciones, aceptando o rechazando, en cuyo caso se dice que el sistema, que se denota con el par (P, V), acepta o rechaza, respectivamente.

Se define que un lenguaje L *admite* una *prueba interactiva determinística* sii existe un verificador V tal que:

1. Existe un probador P con el que (P, V) acepta todas las cadenas de L.
2. Para todo probador P se cumple que (P, V) rechaza todas las cadenas que no están en L.

Las condiciones (1) y (2) corresponden, respectivamente, a las propiedades de completitud y sensatez antes mencionadas.

Así, un lenguaje de la clase NP se puede definir alternativamente como un lenguaje que admite una prueba interactiva determinística. La equivalencia entre esta definición y la definición con certificados que formulamos en el capítulo 6 se demuestra fácilmente (ejercicio 9.22).

Una pregunta razonable para hacerse, considerando lo estudiado en la sección anterior sobre los algoritmos probabilísticos, es qué sucede si se potencia al verificador con elecciones aleatorias, o en otras palabras, si se recurre a *pruebas interactivas probabilísticas* en vez de determinísticas. ¿Nos mantenemos dentro de la clase NP, o saltamos a una clase más amplía? Y si se cumple la segunda alternativa, ¿de qué magnitud es el salto? La respuesta es ciertamente no intuitiva: no sólo se produce un salto, sino que el salto es mayúsculo. En efecto, la clase de lenguajes obtenida llega a ser la clase PSPACE. A pesar de que todo algoritmo probabilístico eficiente podría sustituirse por un algoritmo determinístico eficiente según la conjetura más aceptada, y de que los sistemas interactivos descriptos recién no agregan lenguajes a la clase NP, cuando se combinan los modelos computacionales probabilístico e interactivo se obtendría uno mucho más potente. En lo que resta de la sección lo analizamos sucintamente (para simplificar la nomenclatura, llamaremos por lo general directamente *pruebas interactivas* a las que sean probabilísticas, y *sistemas de pruebas interactivas* a los sistemas de pruebas correspondientes). Comenzamos con un ejemplo, relacionado con el problema del isomorfismo de grafos. Dicho problema, al que ya nos referimos previamente varias veces, nos servirá para describir distintas características del modelo.

Ejemplo 9.9. El lenguaje del problema del isomorfismo de grafos, ISO = {(G_1, G_2) | G_1 y G_2 son grafos isomorfos}, pertenece a la clase NP, y pareciera que su complemento ISO^C no (el certificado natural para verificar que dos grafos con vértices 1, …, m no son isomorfos es la secuencia de las m! permutaciones posibles de los vértices, de tamaño exponencial con respecto a |(G_1, G_2)|). En consecuencia, ISO^C no admitiría una prueba interactiva determinística. En cambio, sí admite una prueba interactiva probabilística, que mostramos a continuación.

Definimos el siguiente sistema de pruebas interactivas (P, V). Dada una entrada (G_1, G_2), tal que G_1 y G_2 son grafos con vértices 1, …, m, el sistema (P, V) hace:

1. V elige aleatoriamente un número i entre 1 y 2, y una permutación π de (1, …, m), que expresamos con (π(1), …, π(m)).
2. V obtiene el grafo H = $\pi(G_i)$, tal que $\pi(G_i)$ es el grafo que resulta de reemplazar en G_i cada arco (a, b) por el arco (π(a), π(b)), por lo que H es isomorfo a G_i.
3. V le envía a P el grafo H.
4. P obtiene el número j entre 1 y 2, según H sea isomorfo a G_1 o a G_2, respectivamente.
5. P le envía a V el número j.
6. V acepta sii i = j.

Se cumple que los bloques 1, 2, 3 y 6, que son los que importan en lo que hace al tiempo de ejecución de (P, V), tardan tiempo polinomial con respecto a |(G_1, G_2)|, los mensajes intercambiados son dos, y son de tamaño polinomial con respecto a la entrada.

Por otro lado, si los grafos no son isomorfos, V acepta con probabilidad 1, y si son isomorfos, lo hace con probabilidad a lo sumo 1/2 (con el probador P definido o con cualquier otro).

Ejercicio. Justificar la última observación.

De la misma forma que vimos en la sección anterior, después de k ejecuciones independientes de (P, V), con k polinomial con respecto a |(G_1, G_2)|, la probabilidad de error pasa a ser a lo sumo $1/2^k$.

Fin del ejemplo

Toda cadena de ISO^C, entonces, cuenta con un certificado sucinto que permite determinar su pertenencia al lenguaje en tiempo probabilístico polinomial con una probabilidad de error muy baja. Esta es la característica de todos los lenguajes que admiten pruebas interactivas.

Formalmente, la clase IP (*interactive proofs*) es la clase de los lenguajes que admiten pruebas interactivas con probabilidad de error a lo sumo 1/3. O sea que en el caso de un lenguaje L de IP, la propiedad de completitud establece que toda cadena de L debe aceptarse con una probabilidad al menos 2/3 (con algún probador), y la propiedad de sensatez, que toda cadena que no pertenezca a L debe rechazarse con la misma probabilidad (con cualquier probador). Como en la definición de la clase BPP, la cota 2/3 se puede reemplazar por otra mientras sea mayor que 1/2.

Ejercicio. Probar que el lenguaje ISO^C pertenece a la clase IP.

Así como a BPP se la considera la *clase P probabilística*, a IP se la considera la *clase NP probabilística* (en este caso, la conjetura más aceptada es que IP incluye estrictamente a NP).

De su estudio se han derivado importantes aportes a la criptografía y las aproximaciones polinomiales, entre otras áreas.

Como anticipamos, se cumple:

$$IP = PSPACE$$

igualdad para nada intuitiva. Por ejemplo, IP incluye al lenguaje $QSAT^C$, cuando SAT^C no estaría ni siquiera en la clase NP. La prueba de la inclusión $IP \subseteq PSPACE$ se prueba sin mayor dificultad (ejercicio 9.24). La prueba de la inclusión recíproca, $PSPACE \subseteq IP$, consiste en demostrar que QSAT pertenece a IP, teniendo en cuenta que QSAT es PSPACE-completo. Se utiliza una técnica llamada *aritmetización*, basada en reducir el problema de satisfactibilidad al problema de determinar si un polinomio admite un valor diferente de cero. Esta técnica es muy importante para la complejidad computacional, dado que no *relativiza* (en una nota adicional nos referimos a esta propiedad).

Revisando la prueba interactiva del ejemplo 9.9, se puede apreciar que se basa en el hecho de que el probador P no sabe qué número i entre 1 y 2 elige el verificador V (de lo contrario podría enviarle siempre un número j tal que el sistema (P, V) nunca responda mal). Las pruebas interactivas en las que P no tiene acceso a los dígitos aleatorios de V se conocen como pruebas interactivas de *monedas privadas* (*private coins*). Si en cambio P tiene acceso a dichos dígitos, y V sólo puede enviar mensajes con dígitos aleatorios y no puede utilizar ningún otro dígito aleatorio que no esté en sus mensajes, las pruebas interactivas se identifican como pruebas interactivas de *monedas públicas* (*public coins*), y también como *protocolos Arturo-Merín* (*Arthur-Merlin protocols*). Tampoco intuitivo, se demuestra que todo lenguaje que admite una prueba interactiva de monedas privadas admite una prueba interactiva de monedas públicas. La idea general de la demostración consiste en transformar una prueba interactiva de un tipo en el otro, encarando el problema en cuestión con una visión más cuantitativa. Por ejemplo, en el caso de los grafos no isomorfos, se prueba que se puede construir una prueba interactiva de monedas públicas, en la que el probador es capaz de convencer al verificador de que el tamaño del conjunto de los grafos isomorfos al grafo G_1 o al grafo G_2 es el doble tanto del tamaño del conjunto de los grafos isomorfos a G_1 como del tamaño del conjunto de los grafos isomorfos a G_2. Un resultado particular muy importante obtenido considerando pruebas interactivas de monedas públicas, a partir de una relación que se puede establecer entre ellas y la jerarquía polinomial, es que el lenguaje ISO del problema del isomorfismo de grafos no es NP-completo a menos que la jerarquía polinomial colapse.

Otra característica para analizar en una prueba interactiva es cuánto *aprende* de ella el verificador, o en otras palabras, qué información adicional adquiere además de la pertenencia o no pertenencia de una cadena a un lenguaje. Un caso extremo en este sentido es el de las pruebas interactivas determinísticas, en las que el verificador directamente recibe la solución de la instancia procesada (una permutación de vértices, una asignación de valores de verdad, etc). Y en el otro extremo están las pruebas interactivas en las que el verificador no aprende

absolutamente nada, lo que resulta sumamente relevante en áreas como la criptografía (por ejemplo, en el contexto de los procesos de autenticación). Las pruebas interactivas con esta última característica se conocen como *pruebas de conocimiento cero* (*zero knowdlege proofs*).

Ejemplo 9.10. Mostramos un sistema (P, V) con una prueba de conocimiento cero para el lenguaje ISO del problema de los grafos isomorfos. Dado un par de grafos G_1 y G_2, con vértices 1, …, m, el sistema (P, V) hace:

1. P obtiene: una permutación π_{12} de (1, …, m), que si los grafos G_1 y G_2 son isomorfos cumple $\pi_{12}(G_1) = G_2$; otra permutación cualquiera π_2 de (1, …, m); y el grafo $H = \pi_2(G_2)$.
2. P le envía a V el grafo H.
3. V elige aleatoriamente un número i entre 1 y 2.
4. V le envía a P el número i.
5. Si i = 1, P le envía a V la permutación $\pi = \pi_2 \circ \pi_{12}$, donde $\pi_2 \circ \pi_{12}(G) = \pi_2(\pi_{12}(G))$.
   Si i = 2, P le envía a V la permutación $\pi = \pi_2$.
6. V acepta sii $H = \pi(G_i)$.

Los bloques 3, 4 y 6, correspondientes al verificador V, tardan tiempo polinomial con respecto a $|(G_1, G_2)|$, y los mensajes intercambiados son tres y de tamaño polinomial con respecto a la entrada. Además:

- Si G_1 y G_2 son isomorfos, V acepta con probabilidad 1, porque elija i = 1 o 2 en el bloque 3, en el bloque 6 siempre compara dos grafos iguales: cuando i = 1, compara $\pi_2(G_2)$ con $\pi_2(\pi_{12}(G_1))$, siendo $\pi_{12}(G_1) = G_2$, y cuando i = 2, compara $\pi_2(G_2)$ con $\pi_2(G_2)$.
- Se puede comprobar que si G_1 y G_2 no son isomorfos, V acepta con probabilidad a lo sumo 1/2 (con el probador P definido o con cualquier otro). Considerando la estrategia de prueba del ejemplo: si V elige 1, compara $\pi_2(G_2)$ con $\pi_2(\pi_{12}(G_1))$, que son grafos distintos, y si elige 2, compara $\pi_2(G_2)$ con $\pi_2(G_2)$, que son grafos iguales.

Notar que con dos ejecuciones independientes de (P, V) se alcanza la cota requerida (con más ejecuciones hasta un límite polinomial, la probabilidad de error se reduce significativamente), y que del grafo y la permutación que recibe, V no aprende nada.
Fin del ejemplo

Demostrar la correctitud de una prueba interactiva de conocimiento cero consiste en verificar, además de las propiedades de completitud y sensatez, una tercera propiedad, de *conocimiento cero*, que establece que lo que el verificador puede aprender participando en la interacción con el probador también lo puede aprender sin participar en ella. Dicho de otra manera, que lo mismo que se puede obtener con alguna estrategia V* de verificación interactiva de tiempo polinomial probabilístico, se siga o no el protocolo establecido en la prueba, se puede obtener con algún *algoritmo de simulación* S* no interactivo del mismo tiempo de ejecución. Asumiendo la existencia

de las funciones de un solo sentido, y relajando la propiedad de conocimiento cero de modo tal que las respuestas de cualquier sistema interactivo de pruebas (P, V*) y del algoritmo no interactivo S*, según los especificamos recién, puedan diferir pero de un modo indistinguible frente a chequeos de tiempo polinomial, se cumple que todo lenguaje de NP cuenta con una prueba interactiva de conocimiento cero (la propiedad se conoce como *propiedad de conocimiento cero computacional*).

Todavía existe otra variante de los sistemas de verificaciones probabilísticas eficientes que caracterizan a la clase NP. Son las *pruebas chequeables probabilísticamente* o *PCP* (*probabilistically checkable proofs*). En dicha variante, que presentamos en una nota adicional, se vuelve a la idea de la exposición completa de una prueba desde el principio, pero ahora con la particularidad de que sólo pueden leerse algunas de sus partes, y de modo probabilístico. Su uso es muy relevante en la complejidad computacional, porque permite obtener resultados relacionados con la aproximabilidad polinomial de varios problemas de optimización, por lo general difíciles de lograr como comentamos en la sección correspondiente de este capítulo.

Los algoritmos cuánticos en la complejidad temporal

Completamos este capítulo presentando un último modelo computacional, la *máquina cuántica*, basada en los principios de la mecánica cuántica (*superposición*, *entrelazamiento*, etc). La máquina cuántica podría refutar la Tesis Fuerte de Church-Turing mencionada en el capítulo 6, dado que permite resolver eficientemente algunos problemas, como la factorización, que al día de hoy los mejores algoritmos clásicos resuelven en tiempo mínimamente exponencial.

Entre las características de la mecánica cuántica que adopta este modelo se destaca la de que todo parámetro físico de una partícula elemental como el electrón (posición, energía, etc.), *en tanto no se mida* no tiene como valor un simple número sino una *onda de probabilidad*, interpretada como la *superposición* de todos los valores posibles. Recién al medirse el parámetro adquiere un valor definitivo, lo que se conoce como *colapso de la onda de probabilidad*.

En realidad, las ondas de probabilidad se asocian no a partículas aisladas sino a conjuntos de partículas, que pueden alcanzar cantidades exorbitantes. Aún en este caso se considera que la máquina cuántica es físicamente realizable, asumiendo que se la puede aislar suficientemente para evitar interferencias indeseables, y complementar con mecanismos de corrección de errores adecuados.

En la máquina cuántica, el objeto análogo al dígito binario o *bit* de la máquina clásica es el *dígito binario cuántico* o *cúbit* (*qubit*). Como los bits, los cubits pueden estar en el estado básico 0 o en el estado básico 1, pero a diferencia de los bits, los cubits también pueden estar en un *estado de superposición* de los dos estados básicos, o dicho de otra forma, pueden estar en el estado 0 *y* en el estado 1 al mismo tiempo, con una determinada probabilidad. Los estados básicos de un cúbit se denotan con $|0\rangle$ y $|1\rangle$, y los estados de superposición con $\alpha_0|0\rangle + \alpha_1|1\rangle$, combinación lineal en la que las α_i son *amplitudes*, números complejos que satisfacen la ecuación

$|\alpha_0|^2 + |\alpha_1|^2 = 1$. De este modo, el estado de un cúbit se puede representar con un *vector de amplitudes* (α_0, α_1). En concordancia con lo dicho previamente, un cúbit permanece en un estado de superposición hasta que es medido, momento en el que pasa al estado $|0\rangle$ con probabilidad $|\alpha_0|^2$ o al estado $|1\rangle$ con probabilidad $|\alpha_1|^2$, y las amplitudes resultantes quedan, según el caso, con los valores $\alpha_0 = (1, 0)$ y $\alpha_1 = (0, 0)$ o $\alpha_0 = (0, 0)$ y $\alpha_1 = (1, 0)$.

Ya con amplitudes reales se observan los efectos cuánticos. En adelante, para simplificar la presentación, las consideramos exclusivamente.

Ejemplo 9.11. Un cúbit en el estado $(1/\sqrt{2})|0\rangle + (1/\sqrt{2})|1\rangle$, al ser medido, pasa al estado $|0\rangle$ con probabilidad 1/2 o al estado $|1\rangle$ con probabilidad 1/2. Y un cúbit en el estado $(1/\sqrt{2})|0\rangle - (1/\sqrt{2})|1\rangle$, al ser medido, se comporta de la misma manera.

Sin embargo, los dos estados se pueden distinguir por medio de *operaciones cuánticas*, que describimos enseguida.

Fin del ejemplo

Los cubits se agrupan en *registros cuánticos*. De lo anterior se deriva que un registro cuántico de m cubits puede estar en una superposición de 2^m estados básicos. Por ejemplo, un registro cuántico de dos cubits puede estar en un estado básico $|00\rangle$, $|01\rangle$, $|10\rangle$ o $|11\rangle$, o en un estado de superposición $\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$, con $\alpha_{00}^2 + \alpha_{01}^2 + \alpha_{10}^2 + \alpha_{11}^2 = 1$. En este último caso, el estado de superposición del registro se representa mediante el vector de amplitudes $(\alpha_{00}, \alpha_{01}, \alpha_{10}, \alpha_{11})$, y cuando el registro es medido pasa con probabilidad α_{d1d2}^2 al estado $|d_1d_2\rangle$, siendo $d_i = 0$ o 1, quedando la amplitud correspondiente en 1 y las restantes en 0. Lo mismo sucede con los registros cuánticos de tres cubits, cuatro cubits, etc. En todos los casos, las coordenadas de los vectores de amplitudes se disponen según el orden lexicográfico, y a toda medición le sigue el colapso de la superposición.

Ejercicio. ¿Cuántos registros clásicos de m bits se necesitan para almacenar la información que puede tener un registro cuántico de m cubits?

Los registros cuánticos pasan de un estado a otro por medio de *operaciones cuánticas*, que se aplican sobre uno o más cubits. Una manera de representar las operaciones cuánticas es mediante matrices, de modo tal que la aplicación de una operación cuántica sobre un registro cuántico consiste en multiplicar la matriz que representa la operación por el vector que representa el estado del registro, dando como resultado otro vector que representa el nuevo estado del registro. Las matrices tienen que ser *unitarias*, es decir que al multiplicarlas por sus transpuestas (transpuestas conjugadas en el caso más general de los números complejos) se debe obtener la matriz identidad. Esta restricción es inevitable para que se cumpla la propiedad de que los cuadrados de las amplitudes sumen 1, y hace *reversibles* a las operaciones: aplicando una operación y luego la operación correspondiente a la matriz transpuesta de la primera, se vuelve al estado original. Esto también puede suceder a veces repitiendo una misma operación cuántica.

Ejemplo 9.12. La operación cuántica conocida como *puerta de Hadamard* se aplica sobre un solo cúbit y se representa por la siguiente matriz:

$$\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

Aplicando una vez la puerta de Hadamard sobre un cúbit en el estado básico |0>, el cúbit pasa al estado de superposición $(1/\sqrt{2})|0> + (1/\sqrt{2})|1>$:

$$\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$$

Y aplicando nuevamente la misma operación sobre el cúbit en su nuevo estado, el cúbit vuelve al estado básico |0>:

$$\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \cdot \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

De esta manera, después de aplicar al cúbit en un estado de superposición la puerta de Hadamard, cuyo efecto es pasarlo a su vez a un estado de superposición, el cúbit queda en un estado básico, lo que se explica por lo siguiente:

A partir de $(1/\sqrt{2})|0> + (1/\sqrt{2})|1>$, existen dos caminos que conducen a |0> y dos caminos que conducen a |1>. Pero mientras que en los dos caminos que conducen a |0> las amplitudes son positivas, lo que hace que los caminos se interfieran constructivamente, en los dos caminos que conducen a |1> una amplitud es positiva y la otra es negativa, produciéndose una interferencia destructiva que cancela a ambos (la representamos en la figura 9.6).

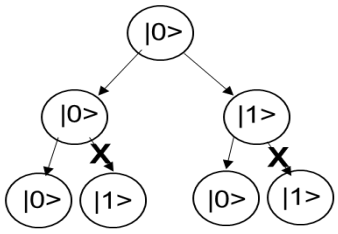

Figura 9.6. Ejemplo de computación cuántica con interferencias constructivas y destructivas.

Fin del ejemplo

Interferencias destructivas como las del ejemplo anterior, provocadas por amplitudes negativas, son las que constituyen la *real* diferencia entre la mecánica cuántica y la teoría clásica

de las probabilidades (porque en verdad, el estado de un registro clásico de m bits de un algoritmo probabilístico también se puede representar con un vector de dimensión 2^m, asociando cada coordenada a la probabilidad de que el registro incluya una cadena determinada). Esta característica de las máquinas cuánticas es la que les proveería una capacidad computacional superior a la de las máquinas clásicas.

Las operaciones cuánticas también se conocen como *puertas cuánticas* (*quantum gates*), como observamos en el ejemplo anterior, porque se pueden representar alternativamente mediante *circuitos cuánticos*. Se puede definir una gran variedad de operaciones cuánticas, que por razones de factibilidad técnica sólo se consideran sobre conjuntos de hasta tres cubits. Por lo tanto, se representan con matrices de 2 x 2, 4 x 4 u 8 x 8.

Ejemplo 9.13. La operación cuántica conocida como *puerta de Toffoli* se aplica sobre un conjunto de tres cubits, y su efecto es el siguiente: a partir de $|d_1d_2d_3>$, con $d_i = 0$ o 1, cambia d_3 de 0 a 1 y de 1 a 0 sólo si se cumple $d_1 = d_2 = 1$. Es decir: $|000> \longrightarrow |000>$, $|001> \longrightarrow |001>$, $|010> \longrightarrow |010>$, $|011> \longrightarrow |011>$, $|100> \longrightarrow |100>$, $|101> \longrightarrow |101>$, $|110> \longrightarrow |111>$, $|111> \longrightarrow |110>$. La matriz que representa la operación es:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Ejercicio. Comprobar que la matriz representa efectivamente la operación cuántica definida.
Fin del ejemplo

En los ejercicios finales mostramos más operaciones cuánticas, y en una nota adicional describimos un algoritmo cuántico completo, que refleja una característica distintiva de la computación cuántica, el *paralelismo cuántico*, el cual se puede resumir como la capacidad de evaluar una función simultáneamente sobre diferentes valores.

Se define que un lenguaje L es decidible en tiempo cuántico T(n) sii existe una MT M de tiempo polinomial que para todo n, a partir del par $(1^n, 1^{T(n)})$, genera la descripción de un conjunto de puertas cuánticas $\{F_1, \ldots, F_k\}$, que dada una entrada w de tamaño n, acepta w si $w \in L$ y rechaza w si $w \notin L$ con probabilidad al menos 2/3, por medio del siguiente proceso:

1. Inicializar un registro cuántico de m cubits de la forma $|w0^{m-n}>$, es decir con entrada w complementada con ceros, siendo $m \leq T(n)$.
2. Aplicar sobre el registro T(n) veces, una tras otra, alguna puerta cuántica de $\{F_1, \ldots, F_k\}$.
3. Medir el registro.

El resultado queda en el cúbit que se designa para tal fin, y es de aceptación o rechazo si el estado de dicho cúbit es |1> o |0>, respectivamente.

Los lenguajes decidibles en tiempo cuántico polinomial conforman la clase BQP (*tiempo polinomial cuántico con error acotado* o *bounded-error quantified polynomial time*), considerada la *clase P cuántica*. La definición se puede generalizar a resultados con más de un cúbit, y la cota 2/3 se puede sustituir por otra por la posibilidad de repetir ejecuciones independientes, como en el caso de los algoritmos probabilísticos. La MT M de tiempo polinomial se requiere por la misma necesidad de uniformidad que mencionamos cuando definimos la clase NC de los lenguajes decidibles por algoritmos paralelos eficientes. Justamente, otra forma de definir BQP es mediante familias uniformes de circuitos cuánticos polinomiales. Formalmente, se define alternativamente que un lenguaje L pertenece a BQP sii existe una familia uniforme de circuitos cuánticos de tamaño polinomial $\{C_n\}_{n\geq 1}$, tal que para toda entrada w de tamaño n, si $w \in L$ entonces C_n acepta |w0…0> con probabilidad al menos 2/3, y si $w \notin L$ entonces C_n rechaza |w0…0> con probabilidad al menos 2/3. Ahora las puertas de los circuitos no son lógicas sino cuánticas, y la cantidad de arcos que llegan a una puerta coincide con la cantidad de arcos que salen de la puerta (a lo sumo tres). En cuanto a la variedad de las puertas cuánticas, se demuestra que un grupo reducido es suficiente para describir cualquier algoritmo cuántico. Por ejemplo, las puertas de Hadamard y Toffoli descriptas en los ejemplos 9.12 y 9.13 alcanzan, de la misma manera que alcanzan las puertas *and*, *or* y *not* en el caso de los circuitos booleanos (en verdad, son suficientes las puertas *and* y *not*, y también las puertas *or* y *not*). Se cumple:

$$P \subseteq BQP$$

porque las máquinas de Turing de tiempo polinomial se pueden simular con circuitos booleanos de tamaño polinomial, que a su vez se pueden simular con circuitos cuánticos de tamaño polinomial. También se cumple:

$$BPP \subseteq BQP$$

porque los lanzamientos de moneda se pueden simular con aplicaciones de la puerta de Hadamard.

No se cumpliría la inclusión recíproca $BQP \subseteq BPP$, teniendo en cuenta que no se ha encontrado hasta el momento ningún algoritmo probabilístico eficiente para la factorización. Otra inclusión que se prueba es:

$$BQP \subseteq PSPACE$$

recurriendo como en demostraciones anteriores a un procedimiento recursivo y la reutilización de espacio.

Y con respecto a la relación entre las clases BQP y NP, la conjetura más aceptada es que son incomparables: los algoritmos cuánticos sólo podrían acelerar cuadráticamente los algoritmos clásicos más rápidos correspondientes a los lenguajes NP-completos, al tiempo que existe un lenguaje en BQP que no estaría en la jerarquía polinomial.

La figura 9.7 muestra cómo se relaciona la clase BQP con las clases P y BPP (pudiendo ser P = BPP). Incluimos la jerarquía espacio-temporal completa para ilustrar también cómo se relacionan sus clases con BQP y BPP.

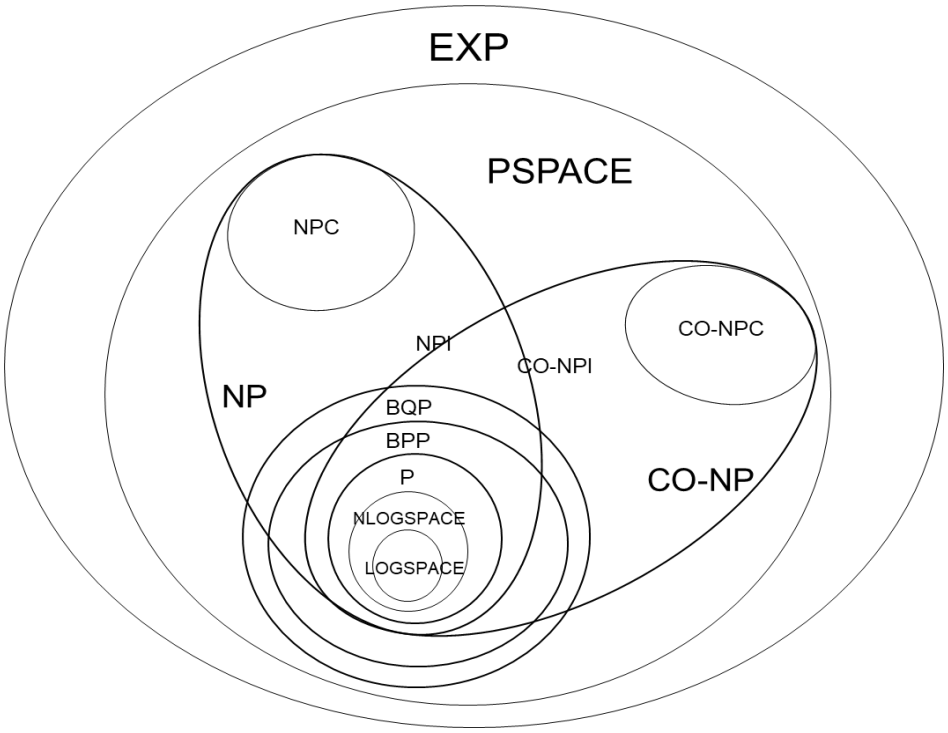

Figura 9.7. Las clases P, BPP y BQP en el marco de la jerarquía espacio-temporal.

Actualmente se conocen pocos algoritmos cuánticos mejores que sus contrapartes clásicas, en el sentido de la complejidad temporal, lo que puede explicarse por la dificultad para diseñarlos, y también por nuestra intuición mucho más cercana a los algoritmos clásicos.

Se destacan básicamente dos grupos de algoritmos. Un primer grupo reúne a los algoritmos de búsqueda, basados en el *algoritmo de Grover*, que aceleran cuadráticamente los algoritmos de búsqueda clásicos más veloces. Y en el segundo grupo se ubican los algoritmos que utilizan la *transformada cuántica de Fourier*, como por ejemplo los *algoritmos de Shor* para factorizar y para calcular el logaritmo discreto, en este caso con mejoras temporales más sustanciales, del orden exponencial.

Notas adicionales

Comenzamos con otra prueba de autorreducibilidad en el ámbito de los problemas de búsqueda, ahora de un problema de optimización: el problema de optimización del viajante de

comercio (OPVC). En la siguiente sección comentamos otro uso de los oráculos en la complejidad computacional, que permite identificar técnicas útiles para resolver el problema P vs NP. Continuamos con la descripción de la máquina paralela de acceso aleatorio o PRAM, modelo computacional paralelo alternativo al del circuito booleano, y nos referimos también a la jerarquía de lenguajes AC asociada al modelo. Luego introducimos las pruebas chequeables probabilísticamente, otra variante de los sistemas de verificaciones probabilísticas eficientes, destacando en particular las que permiten caracterizar a los lenguajes de la clase NP. Finalmente, mostramos un ejemplo de algoritmo cuántico, que ilustra un aspecto distintivo del paradigma.

Prueba de autorreducibilidad del problema de optimización del viajante de comercio

Vimos que todo problema de búsqueda correspondiente a un lenguaje NP-completo es autorreducible (y polinomialmente equivalente al lenguaje), y mostramos como ejemplo una Cook-reducción de FSAT a SAT. En esta sección mostramos otro ejemplo, ahora para probar la autorreducibilidad de un problema de optimización, que refleja un esquema de Cook-reducción más complicado, lo que es consistente con el hecho de que un problema de optimización supone mayor dificultad que un problema de búsqueda general, en el sentido de la complejidad temporal.

Ejemplo 9.14. Vamos a probar la autorreducibilidad del problema OPVC del viajante de comercio, que como definimos en el ejemplo 9.3, consiste en encontrar un circuito de Hamilton de longitud mínima de un grafo completo ponderado.

Construiremos una Cook-reducción de OPVC al lenguaje asociado PVC = {(G, B) | G es un grafo completo ponderado y tiene un circuito de Hamilton de tamaño a lo sumo B}, lenguaje NP-completo tal como demostramos en el ejemplo 7.7. Es decir, vamos a definir una MT M^{PVC} que resuelve OPVC en tiempo polinomial. Dado un grafo G = (V, E), M^{PVC} hace:

1. Primero obtiene la longitud K de un circuito de Hamilton de longitud mínima de G, invocando al oráculo de PVC varias veces. Como K varía entre 1 y 2^n, con n = |G|, utilizando búsqueda binaria puede obtener K luego de O(n) invocaciones (podría empezar invocando al oráculo, por ejemplo, con K = 2^n / 2).
2. Luego obtiene un circuito de Hamilton de longitud mínima de G, invocando al oráculo de PVC también varias veces. La idea es la siguiente. En toda invocación, modifica el número asociado a un arco distinto de G, asignándole el valor K + 1. Si el oráculo acepta, significa que dicho arco no forma parte de un circuito de Hamilton de longitud mínima de G, y por lo tanto no hace nada. Si en cambio el oráculo rechaza, significa que el arco procesado sí forma parte de un circuito de Hamilton de longitud mínima de G, por lo que en este caso lo marca (el arco

formará parte de la solución), y le restituye su número asociado original. En consecuencia, al cabo de O(|E|) = O(n) iteraciones, completa la segunda parte del algoritmo.

De esta forma, M^{PVC} resuelve OPVC, y lo hace en tiempo polinomial. La Cook-reducción establece que si se pudiera decidir eficientemente si un grafo G tiene un circuito de Hamilton de hasta cierta longitud, entonces también se podría obtener eficientemente un circuito de Hamilton de G de longitud mínima.

Fin del ejemplo

Generalizando, la prueba de la autorreducibilidad de los problemas de optimización se basa en Cook-reducciones estructuradas en dos etapas. En la primera se obtiene la medida de una solución óptima, y en la segunda, una solución óptima propiamente dicha.

El problema P vs NP relativizado

Los oráculos sirven para establecer relaciones entre lenguajes, tanto en la computabilidad como en la complejidad computacional. En cualquier caso, al utilizar oráculos se establecen relaciones *relativizadas*, las cuales parten del supuesto de que se puede decidir (eficientemente en el marco de la complejidad computacional) un determinado lenguaje. Así y todo, probar relaciones relativizadas podría arrojar luz sobre las relaciones no relativizadas correspondientes.

En particular, concentrándonos en el problema P vs NP, si las clases P^L y NP^L son las clases que reúnen a los lenguajes decidibles con MT determinísticas y no determinísticas de tiempo polinomial con un oráculo de un lenguaje L, respectivamente, probar las relaciones relativizadas $P^L = NP^L$ o $P^L \neq NP^L$ podría contribuir a la resolución del problema no relativizado. Sólo por dar un ejemplo: partiendo de un lenguaje L muy difícil, reducir gradualmente su dificultad podría ir aportando información valiosa acerca de la verdadera relación entre P y NP.

En el teorema siguiente ejemplificamos una prueba de una relación relativizada $P^L = NP^L$, haciendo uso de un oráculo de un lenguaje muy difícil, PSPACE-completo:

Teorema 9.5. $P^{QSAT} = NP^{QSAT}$.

Prueba. El lenguaje QSAT es tan o más difícil que cualquier lenguaje de P y NP, por lo que resulta razonable que las máquinas de Turing correspondientes a los lenguajes de P^{QSAT} y NP^{QSAT} se equiparen en poder computacional. Formalmente, $P^{QSAT} \subseteq NP^{QSAT}$ se cumple por definición, y con respecto a la prueba $NP^{QSAT} \subseteq P^{QSAT}$, primero probaremos $NP^{QSAT} \subseteq PSPACE$ y después $PSPACE \subseteq P^{QSAT}$:

- *$NP^{QSAT} \subseteq PSPACE$*. Sean $L \in NP^{QSAT}$ y $M_N{}^{QSAT}$ una MTN con un oráculo de QSAT que decide L en tiempo polinomial. La siguiente MTD M decide L en espacio polinomial: simula $M_N{}^{QSAT}$

computación por computación reutilizando espacio, toda invocación a QSAT la reemplaza ejecutando una MTD M_{QSAT} que decide QSAT en espacio polinomial, y acepta sii alguna computación acepta.

Ejercicio. Comprobar que M decide L en espacio polinomial.

- $PSPACE \subseteq P^{QSAT}$. Sean L ∈ PSPACE y M_f una MTD que computa una reducción poly-time f de L a QSAT (posible porque QSAT es PSPACE-completo). La siguiente MTD M_D^{QSAT} decide L en tiempo polinomial: dada una entrada w, ejecuta M_f para generar f(w), invoca al oráculo de QSAT con f(w), y acepta sii el oráculo acepta.

Ejercicio. Comprobar que M_D^{QSAT} decide L en tiempo polinomial.

Fin del teorema

Por medio de los oráculos también se puede dilucidar qué tipo de métodos de prueba sirven para resolver el problema P vs NP: el Teorema de Baker, Gill y Solovay, de 1975, establece que existen oráculos de lenguajes A y B tales que $P^A = NP^A$ y $P^B \neq NP^B$ (A puede ser QSAT, de acuerdo al teorerma 9.5). La implicancia de este teorema es muy significativa. De él se deriva que se deben descartar los métodos que *relativizan*, es decir, los métodos que si permiten probar una relación también permiten probar la relación relativizada correspondiente. Dos ejemplos de métodos que relativizan son la simulación y la diagonalización: si por simulación de MT no determinísticas mediante MT determinísticas se probara P = NP, también se cumpliría $P^B = NP^B$, contradiciendo el teorema, y si por diagonalización se encontrara un lenguaje separador de los conjuntos P y NP – P y por lo tanto se demostrara P ≠ NP, también se cumpliría $P^A \neq NP^A$, volviendo a contradecirse el teorema. Por el contrario, un método que no relativiza, en el marco del problema P vs NP, es la *aritmetización* (la hemos mencionado en la sección sobre los sistemas de pruebas interactivas); lamentablemente, al día de hoy se desconoce cómo aplicarla para avanzar en la relación entre las dos clases.

La máquina PRAM y la jerarquía AC

La máquina paralela de acceso aleatorio o PRAM es un modelo computacional alternativo al circuito booleano que analizamos antes, también muy conocido y simple. Básicamente, consta de un conjunto de procesadores que ejecutan un mismo programa, todos efectuando un paso de la computación correspondiente al mismo tiempo, con la guía de un reloj centralizado. Los procesadores comparten una única memoria global, y todas las instrucciones requieren el mismo tiempo de ejecución.

Por la misma razón que en el modelo de los circuitos booleanos, sobre las máquinas PRAM se impone una condición de *uniformidad*, la existencia de una máquina de Turing que genere en

espacio logarítmico sus descripciones para todo tamaño de cadena, y una condición de *factibilidad*, la limitación a un número polinomial de procesadores. De este modo, para decidir lenguajes se consideran *familias uniformes de máquinas PRAM polinomiales*. Dado que no lo vimos con los circuitos booleanos, en el ejemplo siguiente mostramos cómo una máquina PRAM con una cantidad exponencial de procesadores podría obtener resultados irrazonables.

Ejemplo 9.15. Una máquina PRAM M con 2^n procesadores puede decidir eficientemente el lenguaje NP-completo CV correspondiente al problema del cubrimiento de vértices de un grafo. Dado un par (G, K), M hace:

1. Cada procesador P_i genera su índice i en binario.
2. Si la cantidad de dígitos 1 del índice generado por P_i es a lo sumo K, y los vértices de G correspondientes a las posiciones de dichos dígitos forman (no forman) un cubrimiento de vértices de G, entonces P_i acepta (rechaza).
3. M acepta sii algún P_i acepta.

Ejercicio. Probar que efectivamente M decide el lenguaje CV en tiempo polinomial.
Fin del ejemplo

Los lenguajes decidibles por familias uniformes de máquinas PRAM polinomiales en $O((\log_2 n)^k)$ pasos, para algún $k \geq 1$, forman la clase AC^K, y la unión infinita de las clases AC^k constituye la jerarquía AC. Las clases AC^k se pueden definir también como las clases NC^k, con circuitos booleanos, pero con vértices sin la limitación de tener a lo sumo dos arcos de entrada (la A de AC es por *alternancia* o *alternation*, en referencia a las puertas lógicas *and* y *or*). Se prueba, para todo $k \geq 1$, que $NC^k \subseteq AC^k \subseteq NC^{k+1}$, y así AC = NC, por lo que AC también caracteriza a los lenguajes decidibles por algoritmos paralelos eficientes.

Ejemplo 9.16. Describimos una máquina PRAM M que en una cantidad logarítmica de pasos, y con una cantidad polinomial de procesadores, suma N números en paralelo.

Para facilitar la descripción, asumimos que N es una potencia de 2, y usamos una estructura con la forma *for i := 1 to k do in parallel* …, cuyo efecto es activar los primeros k procesadores para que ejecuten en paralelo una misma instrucción.

Dada una entrada $(N, w_1, \ldots, w_N)$, M hace:

1. for i := 1 to $\log_2 N$ do
2. for k := 1 to $N/2^i$ do in parallel
3. $w_k := w_{2k-1} + w_{2k}$

Por ejemplo, si N = 8, M primero suma en paralelo $w_1 + w_2$, $w_3 + w_4$, $w_5 + w_6$ y $w_7 + w_8$, después hace lo mismo con los dos pares de números obtenidos, y finalmente repite con el último par.

M utiliza N/2 procesadores (en el bloque 2, el índice k alcanza a lo sumo el valor N/2), y ejecuta una cantidad logarítmica de pasos (el bloque 1 itera $\log_2 N$ veces).

Fin del ejemplo

Al igual que el modelo de las familias uniformes de circuitos polinomiales, el modelo de las familias uniformes de máquinas PRAM polinomiales satisface la Tesis de la Computación Paralela.

Las pruebas chequeables probabilísticamente

A diferencia de los sistemas de pruebas interactivas, en las *pruebas chequeables probabilísticamente* el verificador V, de tiempo probabilístico polinomial, no interactúa con ningún probador P, sino que cuenta desde el principio con una prueba completa asociada a la cadena que debe reconocer. Pero la procesa con la siguiente particularidad: puede utilizar una determinada cantidad de dígitos aleatorios, y efectuar una determinada cantidad de consultas sobre la prueba, que son lecturas independientes de un solo dígito, 1 o 0, de acceso directo (por lo que se pueden interpretar como consultas a un oráculo). En base a dicho sistema de pruebas se define una familia de clases de lenguajes denominadas PCP (r(n), q(n)). Un lenguaje L pertenece a la clase PCP (r(n), q(n)), o *admite una prueba chequeable probabilísticamente* de tipo (r(n), q(n)), siendo r y q dos funciones de $\mathcal{N}$ a $\mathcal{N}$, sii existe un verificador V de tiempo probabilístico polinomial tal que, para toda cadena w de tamaño n, cumple lo siguiente:

- *Propiedad de eficiencia*: Sobre una cadena de dígitos 1 y 0 identificada como la *prueba* de w, utilizando a lo sumo O(r(n)) dígitos aleatorios efectúa a lo sumo O(q(n)) lecturas independientes de un solo dígito, luego de lo cual acepta o rechaza w.
- *Propiedad de completitud:* Si $w \in L$, entonces existe una prueba tal que la probabilidad de que acepte w es 1.
- *Propiedad de sensatez:* Si $w \notin L$, entonces para toda prueba, la probabilidad de que acepte w es a lo sumo 1/2 (esta constante se puede reemplazar por cualquier otra menor que 1).

En la figura 9.8 mostramos los componentes y características del sistema descripto.

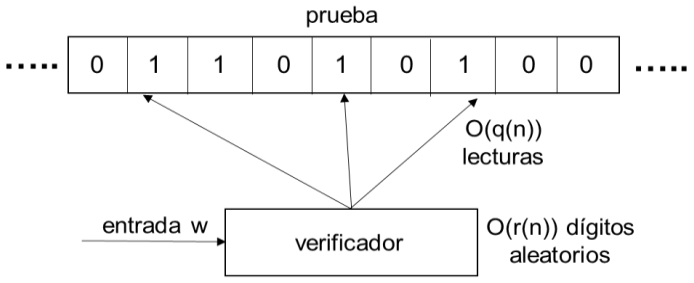

Figura 9.8. Componentes y características de un sistema de pruebas chequeables probabilísticamente.

Ejemplo 9.17. Describimos un verificador V para probar que el lenguaje de los pares de grafos no isomorfos pertenece a la clase PCP(poly(n), 1).

Dados dos grafos G_1 y G_2 de m vértices, V asume como prueba un arreglo de dígitos 0 y 1, con un dígito i por cada grafo G_i de m vértices, tal que i = 0 o 1 según G_i sea isomorfo a G_1 o G_2, respectivamente (en otro caso, i tiene un valor cualquiera). Con estas consideraciones, V hace lo siguiente:

1. Elige aleatoriamente un número j entre 1 y 2 y una permutación π de (1, …, m), y obtiene el grafo $H = \pi(G_j)$.
2. Consulta el dígito i correspondiente al grafo H en la prueba y acepta sii i = j.

V tarda tiempo probabilístico polinomial, utiliza un número polinomial de dígitos aleatorios y hace una sola consulta a la prueba. Además, si los grafos G_1 y G_2 no son isomorfos, claramente la probabilidad de que V acepte es 1, y si lo son, se puede comprobar que para toda prueba la probabilidad de que V acepte es a lo sumo 1/2.

Fin del ejemplo

Se prueba (Teorema PCP) que todo lenguaje de la clase NP admite una prueba chequeable probabilísticamente de tipo ($\log_2 n$, 1), caracterización de NP indudablemente contraintuitiva. Por ejemplo, considerando un típico sistema axiomático S de las matemáticas, con pruebas verificables en tiempo polinomial, el lenguaje $L = \{(\Phi, 1^K) \mid$ la fórmula Φ tiene una prueba en S de longitud a lo sumo K$\}$ pertenece a NP, y entonces, por el Teorema PCP, cuenta con certificados en los que es suficiente examinar unos pocos bits, así y todo tan válidos para asegurar la correctitud de las fórmulas de S como las pruebas matemáticas estándar que deben revisarse línea por línea.

Más allá de observaciones como ésta, producto de un modelo de computación ciertamente particular, el Teorema PCP es especialmente relevante en la complejidad computacional por permitir derivar resultados negativos en el área de las aproximaciones polinomiales de varios problemas de optimización. Ya nos referimos antes a la dificultad para obtener resultados de este tipo en dicho ámbito, y que la prueba típica de no aproximabilidad polinomial de un problema de optimización consiste en generar, mediante una reducción, instancias tales que la brecha entre la medida de una solución óptima cuando la instancia original pertenece al lenguaje de referencia, y la medida de una solución óptima cuando la instancia original no pertenece al lenguaje de referencia, es muy grande (recordar el ejemplo 9.3 relacionado con el problema de optimización del viajante de comercio). En este sentido, las pruebas chequeables probabilísticamente de los lenguajes de la clase NP facilitan la tarea, por su vinculación con los problemas de optimización. La idea general consiste en relacionar la brecha que una prueba chequeable probabilísticamente requiere entre la probabilidad de aceptar correcta e incorrectamente, con la brecha entre las medidas de las soluciones óptimas de las instancias que la reducción correspondiente debe generar. De esta forma, por ejemplo, se puede probar que el problema de optimización del

máximo clique no es aproximable. También se pueden obtener resultados sobre límites de aproximabilidad polinomial, como por ejemplo que el problema de optimización del cubrimiento de vértices no es aproximable polinomialmente con cualquier Ɛ (recordar el ejemplo 9.2 en el que construimos una 1/2-aproximación polinomial), lo mismo que el problema de optimización del conjunto independiente de vértices.

Ejemplo de algoritmo cuántico

Mostramos un algoritmo cuántico que refleja una característica distintiva del paradigma, la posibilidad de evaluar una función simultáneamente sobre diferentes valores. Si bien el algoritmo no es de mucha utilidad, sirve para ilustrar la ventaja de emplear amplitudes en lugar de probabilidades (seguimos considerando sólo amplitudes reales).

Ejemplo 9.18. Se cuenta con una función booleana f, de la que sólo se sabe que a partir de un argumento 0 o 1 devuelve un valor 0 o 1, y se quiere calcular el *or exclusivo* de f(0) y f(1) utilizando f una única vez. Obviamente, de f(0) no se puede inferir nada acerca de f(1), y lo mismo sucede al revés. Así que la idea es evaluar la función f sobre 0 y 1 simultáneamente, explotando la posibilidad de superponer diferentes estados. Describimos a continuación la solución propuesta:

Utilizamos un registro de dos cubits, inicializado en el estado |01>.

Para llevar el registro a un estado de superposición, primero le aplicamos una operación cuántica F_1, representada por la matriz:

$$\begin{pmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 & -1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/2 & -1/2 & -1/2 & 1/2 \end{pmatrix}$$

Luego le aplicamos una operación cuántica F_2, que es la que implementa la función f. La idea es que F_2 se comporte del siguiente modo:

- |00> ⟶ |0 f(0>
- |01> ⟶ |0 ¬f(0)>
- |10> ⟶ |1 f(1)>
- |11> ⟶ |1 ¬f(1)>

es decir, alterando sólo el estado del segundo cúbit de acuerdo al estado del primero, empleando una sola vez la función f. F_2 se representa por la siguiente matriz:

$$\begin{pmatrix} 1-f(0) & f(0) & 0 & 0 \\ f(0) & 1-f(0) & 0 & 0 \\ 0 & 0 & 1-f(1) & f(1) \\ 0 & 0 & f(1) & 1-f(1) \end{pmatrix}$$

Finalmente, volvemos a aplicar F_1 sobre el registro y lo medimos, considerando como resultado el valor 0 o 1 según el estado final del primer cúbit sea |0> o |1>, respectivamente.

En definitiva, el algoritmo propuesto es:

1. Inicializar el registro en el estado |01>.
2. Aplicar sobre el registro las operaciones cuánticas F_1, F_2 y F_1.
3. Medir el registro.

Veamos que el algoritmo efectivamente resuelve el problema. Vamos a verificar:

- Si el *or exclusivo* de f(0) y f(1) es 0, es decir si f(0) = f(1), entonces el estado final del registro queda en |01>, y así obtenemos el valor 0.
- Si el *or exclusivo* de f(0) y f(1) es 1, es decir si f(0) ≠ f(1), entonces el estado final del registro queda en |11>, y así obtenemos el valor 1.

Dado el registro en el estado inicial |01>, representado por el vector (0, 1, 0, 0), luego de aplicarle F_1 pasa al estado representado por el vector:

$$(1/2, -1/2, 1/2, -1/2)$$

Aplicándole luego F_2, el registro pasa al estado representado por el vector:

$$(1/2 - f(0), -1/2 + f(0), 1/2 - f(1), -1/2 + f(1))$$

Notar la dependencia tanto de f(0) como de f(1).

Finalmente, aplicando sobre el registro otra vez F_1, se alcanza el estado representado por el vector:

$$(0, 1 - f(0) - f(1), 0, f(1) - f(0))$$

por lo que luego de la medición se obtiene:

- Si f(0) = f(1), la probabilidad de que el registro esté en el estado |01> es $(1 - f(0) - f(1))^2 = 1$.
- Si f(0) ≠ f(1), la probabilidad de que el registro esté en el estado |11> es $(f(1) - f(0))^2 = 1$.

Fin del ejemplo

El ejemplo anterior se basa en el *algoritmo cuántico de Deutsch*. Refleja claramente la diferencia entre el paradigma clásico probabilístico y el paradigma cuántico. Con un algoritmo probabilístico podemos evaluar tanto f(0) como f(1), con probabilidad 1/2 cada alternativa, pero una *excluyendo* a la otra. Con un algoritmo cuántico, en cambio, como acabamos de mostrar, ambas alternativas *pueden interferirse*, y así producir *conjuntamente* el resultado buscado.

Observaciones finales

- Del estudio de la complejidad computacional de los problemas de decisión, o lo que es lo mismo, de los lenguajes que los representan, se derivan muchos resultados de la complejidad computacional de los problemas de búsqueda y de conteo correspondientes. Un resultado obvio es que si un lenguaje A es NP-completo, entonces los problemas de búsqueda FA y de conteo #A no pueden resolverse en tiempo polinomial a menos que se cumpla P = NP. Otros resultados relevantes, menos intuitivos, establecen que todo lenguaje NP-completo es polinomialmente equivalente al problema de búsqueda asociado (es decir que también se cumple la implicación recíproca de la anterior en el caso de los problemas de búsqueda), y que P = NP implica FP = FNP.
- La manera habitual de resolver los problemas de optimización, si sus lenguajes asociados son NP-completos, es recurriendo a aproximaciones polinomiales. Asumiendo P $\neq$ NP, se prueba que no todos los problemas de optimización se pueden aproximar polinomialmente. Como de costumbre, para clasificar a los problemas se usan reducciones y el concepto de completitud, en este caso reducciones que preservan la propiedad de ser aproximable polinomialmente y la completitud en términos de dichas reducciones. Se conocen pocos problemas completos en este marco en comparación con los lenguajes NP-completos, lo que se justifica por la mayor dificultad para construir las reducciones que se requieren, y porque muchos problemas de optimización asociados a lenguajes NP-completos son aproximables polinomialmente. Un camino alternativo para obtener resultados de no aproximabilidad polinomial, o de aproximabilidad polinomial acotada por un determinado error relativo, es aplicando el Teorema PCP.
- La jerarquía polinomial o PH se extiende desde P hasta PSPACE, y permite clasificar a muchos lenguajes de interés computacional que no pertenecerían a NP ni CO-NP. Los lenguajes de PH se pueden especificar con predicados con una cantidad constante de alternancias de cuantificadores existenciales y universales, computables en tiempo polinomial, y también utilizando MT alternantes y MT con oráculo de determinadas características. PH se estructura por niveles, y la conjetura más aceptada es que todos los niveles son distintos, es decir que la jerarquía no colapsa, conjetura más fuerte que P ≠ NP y que sirve para establecer distintos resultados de la complejidad computacional.
- Los lenguajes de la clase P que se considera que cuentan con algoritmos paralelos eficientes son aquéllos que pueden decidirse por familias uniformes de circuitos booleanos polinomiales

de profundidad polilogarítmica (existe una definición alternativa en términos de las máquinas PRAM). La jerarquía de lenguajes correspondiente se denomina NC, y la conjetura más aceptada es que está incluida estrictamente en P. Los circuitos booleanos también son útiles para tratar el problema P vs NP, considerando el aspecto de las cotas temporales mínimas.

- Los algoritmos probabilísticos resuelven muchos problemas de interés computacional de una manera más eficiente, o al menos más simple, que los algoritmos determinísticos existentes. Los lenguajes decidibles por algoritmos probabilísticos de tiempo polinomial forman la clase BPP, considerada así la clase P probabilística. Ejecutando de manera independiente varias veces los algoritmos asociados a la clase, hasta un límite polinomial, se puede reducir de manera significativa la probabilidad de que respondan erróneamente. Se cumple $P \subseteq BPP$, y se desconoce la relación entre BPP y NP. La conjetura más aceptada es que P = BPP, significando que todos los algoritmos probabilísticos eficientes pueden desaleatorizarse con un retardo sólo polinomial.
- Las pruebas interactivas determinísticas, con un probador de poder ilimitado y un verificador de tiempo polinomial, constituyen otra manera de demostrar la pertenencia a la clase NP. Cuando el verificador es probabilístico, el sistema de pruebas parece ganar mucho poder computacional, dado que su alcance se extiende a toda la clase PSPACE. Existen distintas variantes de sistemas de verificaciones probabilísticas, con aplicaciones en la criptografía y las aproximaciones polinomiales, entre otras áreas.
- Las máquinas cuánticas, basadas en la mecánica cuántica, desafían la Tesis Fuerte de Church-Turing, porque permiten resolver en tiempo polinomial algunos problemas que actualmente cuentan con algoritmos clásicos sólo de tiempo mínimamente exponencial. Constituyen un modelo físicamente realizable, asumiendo que pueden aislarse de manera apropiada y complementarse con adecuados mecanismos de corrección de errores. Los lenguajes decidibles eficientemente por máquinas cuánticas conforman la clase BQP, es decir que BQP es la clase P cuántica. BQP incluiría estrictamente a las clases P y BPP, y sería incomparable con la clase NP. Se conocen pocos algoritmos cuánticos mejores que sus contrapartes clásicas, por la dificultad que implica construirlos, y también por nuestra intuición más próxima a la algorítmica clásica.

Referencias

Para presentar los temas de este capítulo nos hemos basado fundamentalmente en los contenidos de los capítulos respectivos de (Bovet y Crescenzi, 1994), (Papadimitriou, 1994), (Goldreich, 2008), (Arora y Barak, 2009) y (Moore y Mertens, 2011). Dada la cantidad de tópicos tratados, en lo que sigue nos limitamos a mostrar esencialmente publicaciones fundacionales:

S. Cook, R. Karp y L. Levin desarrollaron sus trabajos sobre la NP-completitud utilizando, cada uno, un tipo diferente de reducción polinomial (ver en las referencias del capítulo 7). Mientras que las reducciones polinomiales utilizadas por Karp, o Karp-reducciones, relacionan

sólo problemas de decisión, y las reducciones polinomiales de Levin, o Levin-reducciones, relacionan sólo problemas de búsqueda, las reducciones polinomiales de Cook o Cook-reducciones, enfocadas inicialmente en la dificultad de las pruebas del cálculo proposicional, son las más generales, permitiendo relacionar por medio de oráculos problemas de decisión con problemas de búsqueda.

La clase #P se presentó en (Valiant, 1979a), incluyendo la descripción de varios miembros de la clase. La #P-completitud del cálculo de la permanente, a la que nos referimos en el capítulo, la demostró el mismo autor en (Valiant, 1979b). La clase #P y la jerarquía polinomial o PH son generalizaciones de la clase NP, y no parecen tener características comparables (habilidad para contar certificados y alternancia, respectivamente), pero sin embargo se prueba que todo lenguaje de PH se puede decidir en tiempo polinomial con un oráculo de #SAT (Toda, 1991).

La idea de utilizar aproximaciones polinomiales para resolver los problemas de optimización es anterior al descubrimiento de la NP-completitud, pero recién en (Johnson, 1974) se formalizaron distintos aspectos sobre las mismas y se presentaron algunos ejemplos, al tiempo que se planteaba si existían mejores algoritmos. El trabajo que se considera más relevante sobre las reducciones entre los problemas de optimización es (Papadimitriou y Yannakakis, 1991), en el que se define la clase MAXSNP, con varios problemas completos de interés computacional. (Feige, Goldwasser, Lovasz, Safra y Szegedy, 1991) fue el primer artículo que relacionó la dificultad para encontrar resultados sobre la aproximabilidad polinomial con las pruebas chequeables probabilísticamente; varios de estos resultados figuran en (Arora y Lund, 1995), donde además se sistematiza el esquema de prueba con un marco de estudio sencillo.

La jerarquía polinomial o PH se presentó en (Stockmeyer, 1977), como la versión acotada temporalmente de la jerarquía aritmética de Kleene (mencionada en las referencias del capítulo 3), la cual se prueba que no colapsa, a diferencia de PH, de la que sólo se conjetura que no lo hace. La equivalencia entre la definición original de PH con MT con oráculo y la definición con predicados cuantificados se probó en (Wrathall, 1977). Que PH no colapse es condición suficiente para que se cumplan importantes resultados de la complejidad computacional, como la no inclusión de NP en $P_{/poly}$ y la no NP-completitud del lenguaje correspondiente al problema del isomorfismo de grafos. Otro resultado importante, derivado directamente del concepto de alternancia, es que el problema de satisfactibilidad no puede resolverse simultáneamente en tiempo lineal y espacio logarítmico. Como en otras jerarquías, el mayor interés en PH se centra en las clases de lenguajes de sus primeros niveles, en este caso particularmente útiles para analizar varios problemas de la inteligencia artificial.

N. Pippenger, en (Pippenger, 1979), fue quien inició el estudio de la jerarquía NC, considerando los lenguajes decidibles por máquinas de Turing de tiempo polinomial con una cantidad logarítmica de cambios de dirección del cabezal. La definición de la jerarquía NC con circuitos booleanos se formuló en (Cook, 1979), y la de la jerarquía AC con máquinas PRAM, en (Stockmeyer y Vishkin, 1984). Otras caracterizaciones de NC, por ejemplo con máquinas de Turing alternantes de acceso aleatorio, refuerzan la idea de que dicha jerarquía captura el concepto de lenguaje que admite un algoritmo paralelo eficiente. Con respecto a la utilidad de

los circuitos booleanos para resolver el problema P vs NP considerando cotas temporales mínimas, una publicación muy completa sobre el estado del arte es (Boppana y Sipser, 1990)

Los algoritmos probabilísticos comenzaron a estudiarse sistemáticamente en (Gill, 1977), en donde se definieron las clases de lenguajes más relevantes, y recibieron especial atención cuando se encontraron algoritmos eficientes para el problema de primalidad, como los que se publicaron en (Solovay y Strassen, 1977) y (Rabin, 1980). (Karp, 1990) incluye muchos ejemplos de algoritmos probabilísticos en diversas áreas (teoría de números, álgebra, teoría de grafos, combinatoria, búsquedas, ordenamientos, computación concurrente, etc.), y los principios generales que caracterizan sus diseños. Los generadores pseudoaleatorios se estudiaron inicialmente en el contexto de la criptografía, a comienzos de los años 1980; en particular, la publicación (Yao, 1982) fue la primera que consideró su uso potencial para desaleatorizar los algoritmos correspondientes a la clase BPP.

Los trabajos fundacionales sobre las pruebas interactivas fueron (Goldwasser, Micali y Rackoff, 1985) y (Babai, 1985). El sistema de pruebas interactivas para el lenguaje del problema de los grafos no isomorfos que describimos en el capítulo se presentó en (Goldreich, Micali y Wigderson, 1986). Un descubrimiento de mucho impacto en la comunidad científica, un poco después, fue la demostración de la igualdad IP = PSPACE (Shamir, 1990). Las pruebas chequeables probabilísticamente y la clase PCP fueron introducidas en (Arora y Safra, 1992), y en el mismo año se formuló el Teorema PCP (Arora, Lund, Motwani, Sudan y Szegedy, 1992), caracterizando a la clase NP como la clase PCP($\log_2 n$, 1).

Por la posible incapacidad de las computadoras clásicas para simular eficientemente los procesos cuánticos, R. Feynman planteó a comienzos de los años 1980 la necesidad de construir computadoras cuánticas. La primera definición formal de una máquina cuántica la hizo D. Deutsch (Deutsch, 1985); la que presentamos en el capítulo, en términos de circuitos cuánticos, corresponde a A. Yao (Yao, 1993). Los algoritmos cuánticos de factorización y de cálculo del logaritmo discreto de P. Shor, y de búsqueda de L. Grover, se publicaron en (Shor, 1994) y (Grover, 1996), respectivamente. La computación cuántica tiene una directa conexión con la criptografía: por el lado negativo, el algoritmo cuántico de Shor inutiliza el sistema de seguridad de clave pública RSA mencionado en el capítulo anterior (y cualquier otro basado en la factorización o el logaritmo discreto); y por el lado positivo, posibilita la construcción de mecanismos de seguridad a prueba de adversarios no limitados computacionalmente, es decir de complejidad temporal superpolinomial.

La relativización de las conjeturas sobre las relaciones entre las clases de lenguajes por medio de oráculos se inició con el artículo (Baker, Gill y Solovay, 1975), referido específicamente al problema P vs NP. Desde entonces, numerosos resultados de este tipo se publicaron con respecto a distintas clases de la complejidad computacional.

Otras lecturas recomendadas:

El capítulo 8 de (Goldreich, 2008): sobre los generadores pseudoaleatorios. Incluye una presentación del paradigma general, y la descripción de generadores pseudoaleatorios de propósito general y especial y de algoritmos canónicos de desaleatorización.

Los capítulos 11 y 22 de (Arora y Barak, 2009): sobre el Teorema PCP. Incluyen la prueba de equivalencia entre la formulación con pruebas chequeables probabilísticamente y la vinculada a las aproximaciones polinomiales, y varios resultados relacionados con la segunda formulación.

Los libros (Nielsen y Chuang, 2010) y (Aaronson, 2013): sobre la computación cuántica. El primero es el libro clásico por excelencia de la disciplina; incluye capítulos sobre los circuitos cuánticos, los algoritmos cuánticos de búsqueda y los basados en la transformada cuántica de Fourier, distintos aspectos implementativos de las máquinas cuánticas, la corrección de errores y la teoría de la información cuántica (el algoritmo cuántico presentado en una nota adicional se basa en un ejemplo de este libro). El segundo libro, ya mencionado en las referencias de otros capítulos, contiene los contenidos de un curso dictado en la Universidad de Waterloo en 2006 por su autor, uno de los principales referentes en la materia; de estilo muy didáctico, antes de tratar la computación cuántica dedica varios capítulos introductorios que arrancan con la teoría de conjuntos, siguen con la computabilidad, y llegan a la complejidad computacional de los algoritmos clásicos determinísticos y probabilísticos.

El capítulo 12 de (Papadimitriou, 1994): sobre criptografía. Varios temas que hemos desarrollado se relacionan con dicha disciplina. En este capítulo, el autor trata las funciones de un solo sentido, la criptografía de clave pública, la criptografía aleatorizada y las pruebas interactivas.

El libro (Kearns, 1990): sobre la complejidad computacional en el marco de la inteligencia artificial. Este trabajo es una muy buena introducción a la teoría del aprendizaje computacional, que estudia matemáticamente el aprendizaje eficiente llevado a cabo por un sistema computacional.

Ejercicios

9.1 Probar que si el problema de búsqueda del isomorfismo de grafos pudiera resolverse eficientemente, entonces el lenguaje asociado (lenguaje ISO) pertenecería a la clase P.

9.2 Probar que las Cook-reducciones son transitivas.

9.3 Probar que los siguientes enunciados son lógicamente equivalentes (es decir, se implican dos a dos):

a. A es Cook-reducible a B.
b. A^C es Cook-reducible a B.
c. A es Cook-reducible a B^C.
d. A^C es Cook-reducible a B^C.

9.4 Probar que el problema de búsqueda de satisfactibilidad FSAT es FNP-completo con respecto a las Cook-reducciones, es decir que todo problema de búsqueda de FNP es

Cook-reducible a FSAT. *Ayuda: combinar una MT con un oráculo de FSAT con una Levin-reducción*.

9.5 Probar que si el problema de conteo del circuito hamiltoniano pudiera resolverse eficientemente, entonces el lenguaje asociado (lenguaje CH) pertenecería a la clase P.

9.6 Probar que el problema de conteo mencionado en el ejercicio anterior se puede resolver en espacio polinomial. *Ayuda: ver el ejercicio 8.11*.

9.7 Probar que si existe una máquina de Turing de tiempo polinomial que obtiene un clique de tamaño máximo de un grafo, entonces P = NP.

9.8 Una función A es una *aproximación funcional* de una función f sii para toda cadena w cumple: $f(w)/c \leq A(w) \leq c.f(w)$, con $c \geq 1$. Sea g una función que calcula el número de asignaciones de valores de verdad que satisfacen una fórmula booleana. Probar que si existe una aproximación funcional de g computable en tiempo polinomial, entonces se cumple P = NP.

9.9 El problema de optimización OCM del *corte máximo* consiste en encontrar, dado un grafo G = (V, E), una partición o *corte* de V en dos subconjuntos V_1 y V_2 que maximice el número de arcos que conectan sus vértices (*arcos de corte*). Se prueba que el lenguaje asociado es NP-completo, y que el siguiente algoritmo es una 1/2-aproximación polinomial para el problema: dado un grafo G = (V, E), el algoritmo arranca con un corte en el que $V_1 = \emptyset$ y $V_2 = V$, y luego, iterativamente, mueve un vértice de V_1 a V_2 o de V_2 a V_1 en tanto aumente la cantidad de los arcos de corte. Verificar que el algoritmo es efectivamente una 1/2-aproximación polinomial. *Ayuda: los arcos de corte que salen de un vértice v suman al menos como los arcos que no son de corte que salen de v, porque de lo contrario v estaría en el otro subconjunto del corte*.

9.10 Clasificar a los siguientes lenguajes dentro de la jerarquía polinomial:

a. MIN-PVC = {(G, K) | G es un grafo completo ponderado y su circuito de Hamilton de menor longitud mide K}.
b. UNO-SAT = {φ | φ es una fórmula booleana satisfactible con una sola asignación de valores de verdad}.
c. SAT-NOSAT = {(φ_1, φ_2) | φ_1 y φ_2 son fórmulas booleanas tales que φ_1 es satisfactible y φ_2 no es satisfactible}.

9.11 Probar que PH $\subseteq$ PSPACE.

9.12 Describir la idea general de una MT alternante con una alternancia a partir de un estado de tipo universal, que decida el lenguaje MIN-FORM de la clase Π_2P de PH.

9.13 Describir la idea general de una máquina de Turing no determinística con un oráculo de SAT, que decida el lenguaje MAX-IND de la clase Σ_2P de PH. *Ayuda: el lenguaje IND está integrado por pares (G, K) tales que el grafo G tiene un conjunto independiente de K vértices, y pertenece a NP, por lo que existe una reducción poly-time de IND a SAT*.

9.14 Describir para cada lenguaje una familia de circuitos booleanos que lo decida:

a. El lenguaje de las cadenas 1^n, con $n \geq 1$.
b. El lenguaje de los pares (w, w), siendo w una cadena de unos y ceros.

9.15 Describir una familia de circuitos booleanos que decida el lenguaje de las cadenas 1^n, tales que la codificación binaria de n corresponde a la de los pares (<M>, w) pertenecientes al lenguaje HP que representa el problema de la detención. *Ayuda: tener en cuenta que no se requiere que la familia sea uniforme*.

9.16 Probar que la clase NC es cerrada con respecto a las reducciones log-space.

9.17 Probar que si L es un lenguaje P-completo con respecto a las reducciones log-space y pertenece a NC, entonces NC = P.

9.18 Probar: $P \subseteq ZPP \subseteq RP \subseteq BPP \subseteq PP$.

9.19 Probar: $RP \subseteq NP \subseteq PP$.

9.20 Probar que las clases ZPP, RP, BPP y PP son cerradas con respecto a las reducciones poly-time.

9.21 Probar que las clases ZPP, BPP y PP son cerradas con respecto al complemento.

9.22 Probar que las definiciones de la clase NP con MT determinísticas con certificados y con sistemas interactivos de pruebas determinísticas son equivalentes.

9.23 Una persona A tiene que convencer a una persona B, daltónica, de que tiene un pañuelo rojo y un pañuelo amarillo. Para ello, A le da a B un pañuelo en cada mano, le dice de qué color es cada uno, y luego ambos acuerdan repetir el siguiente proceso 20 veces: A se da vuelta; luego B tira una moneda, si sale cara cambia los pañuelos de mano y si sale ceca los deja como están; finalmente A vuelve a darse vuelta y le dice a B de qué color es el pañuelo que tiene en cada mano. Explicar por qué es muy poco probable que con el proceso acordado, A engañe a B.

9.24 Probar: IP $\subseteq$ PSPACE.

9.25 La operación cuántica *not* se aplica sobre un solo cúbit, y su efecto es cambiar |0> por |1> y |1> por |0>. Representar la operación con una matriz.

9.26 La operación cuántica *cnot* o *not controlado* se aplica sobre dos cubits, y su efecto es el siguiente: a partir de $|d_1d_2>$, con $d_i = 0$ o 1, cambia d_2 de 0 a 1 y de 1 a 0 sólo si se cumple $d_1 = 1$. Es decir: |00> ⟶ |00>, |01> ⟶ |01>, |10> ⟶ |11>, |11> ⟶ |10>. La matriz que representa la operación es:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Probar que la matriz es unitaria y que efectivamente representa la operación descripta.

9.27 Dado un registro de dos cubits en un estado |ψ0>, con |ψ> arbitrario, se quiere copiar el primer cúbit en el segundo por medio de la puerta cuántica *cnot*. ¿Se logra el objetivo cualquiera sea |ψ>? *Ayuda: |ψ> puede ser algún estado de superposición* $\alpha_0|0> + \alpha_1|1>$, *en cuyo caso* $|\psi\psi> = \alpha_0^2|00> + \alpha_0\alpha_1|01> + \alpha_1\alpha_0|10> + \alpha_1^2|11>$.

Apéndice

En este apéndice repasamos símbolos, abreviaturas, jerarquías, clases, problemas y lenguajes que aparecen en el libro, y enumeramos los teoremas, ejemplos y figuras que presentamos. Algunos teoremas y ejemplos se reformulan para mayor claridad.

Símbolos y abreviaturas

$\leq_C$: Cook-reducción.

$\leq_{log}$: reducción log-space.

$\leq_m$: m-reducción.

$\leq_p$: reducción poly-time.

$\leq_T$: Turing-reducción.

AF: autómata finito.

AP: autómata con pila.

BFS: búsqueda a lo ancho.

DFS: búsqueda en profundidad.

FNC: forma normal conjuntiva.

FND: forma normal disyuntiva.

FNP: forma normal prenexa.

M_f: máquina de Turing que computa una m-reducción f.

M^L: máquina de Turing con un oráculo de un lenguaje L.

MT: máquina de Turing.

MTD: máquina de Turing determinística.

MTN: máquina de Turing no determinística.

MTP: máquina de Turing probabilística.

Jerarquías y clases

#P: problemas de conteo asociados a los lenguajes de NP.

APX: problemas de optimización aproximables.

BPP: lenguajes decidibles en tiempo polinomial probabilístico acotado.

BQP: lenguajes decidibles en tiempo polinomial cuántico.

CO-NP: complementos de los lenguajes de NP.

CO-NPC: complementos de los lenguajes de NPC.

CO-RE: complementos de los lenguajes de RE.

EXP: lenguajes decidibles en tiempo exponencial.

FNP: problemas de búsqueda asociados a los lenguajes de NP.

FP: problemas de búsqueda asociados a los lenguajes de P.

IP: lenguajes que admiten una prueba interactiva.

$\mathcal{L}$: todos los lenguajes.

LOGSPACE: lenguajes decidibles en espacio logarítmico.

NC: jerarquía de los lenguajes decidibles por algoritmos paralelos eficientes.

NC^1: clase de la jerarquía NC con circuitos booleanos de profundidad $O(\log_2 n)$.

NC^2: clase de la jerarquía NC con circuitos booleanos de profundidad $O((\log_2 n)^2)$.

NLOGSPACE: lenguajes verificables en espacio logarítmico.

NP: lenguajes verificables en tiempo polinomial.

NPC: lenguajes NP-completos.

NPI: lenguajes de NP de dificultad intermedia.

ONP: problemas de optimización asociados a los lenguajes de NP.

OP: problemas de optimización asociados a los lenguajes de P.

P: lenguajes decidibles en tiempo polinomial.

PAS: problemas de optimización aproximables con cualquier ε.

PCP(r(n), q(n)): lenguajes que admiten una prueba chequeable probabilísticamente cuyo tipo es (r(n), q(n)).

PH: jerarquía polinomial.

PP: lenguajes decidibles en tiempo polinomial probabilístico.

PSPACE: lenguajes decidibles en espacio polinomial.

R: lenguajes recursivos.

RE: lenguajes recursivamente enumerables.

RP: lenguajes decidibles en tiempo polinomial aleatorio.

ZPP: lenguajes decidibles en tiempo polinomial probabilístico de error cero.

Problemas y lenguajes

#SAT: problema de conteo asociado al lenguaje SAT.

ACC = {(G, i, j) | G es un grafo no dirigido que tiene un camino del vértice i al vértice j}. Representa el problema de accesibilidad en grafos no dirigidos.

CH = {G | G es un grafo no dirigido que tiene un circuito de Hamilton}.

CLIQUE = {(G, K) | G es un grafo que tiene un clique de tamaño K}.

$CLIQUE_K$ = {G | G es un grafo que tiene un clique de tamaño K}.

COMP: lenguaje de los números compuestos.

CSAT = {φ | φ es una fórmula booleana sin cuantificadores, en FNC, satisfactible}.

CV = {(G, K) | G es un grafo que tiene un cubrimiento de vértices de tamaño K}.

D = {w_i | la MT M_i acepta la cadena w_i, según el orden canónico}.

D-ACC = {(G, i, j) | G es un grafo dirigido que tiene un camino del vértice i al vértice j}. Representa el problema de accesibilidad en grafos dirigidos.

DIV-3 = {N | N es un número natural que tiene un divisor que termina en 3}.

DOM = {(G, K) | G es un grafo que tiene un conjunto dominante de vértices de tamaño K}.

FACT = {(N, M_1, M_2) | N, M_1 y M_2 son números naturales y N tiene un factor primo en el intervalo [M_1, M_2]}. Representa el problema de factorización.

FSAT: problema de búsqueda asociado al lenguaje SAT.

HP: {(<M>, w) | la MT M se detiene a partir de w}. Representa el problema de la detención.

IND = {(G, K) | G es un grafo que tiene un conjunto independiente de vértices de tamaño K}.

ISO = {(G_1, G_2) | G_1 y G_2 son grafos isomorfos}. Representa el problema del isomorfismo de grafos.

K-COLOR = {G | G es un grafo y sus vértices pueden colorearse con K colores, de manera tal que ningún par de vértices adyacentes reciban el mismo color}.

K-SAT = {φ | φ es una fórmula booleana sin cuantificadores, en FNC, con K literales por cláusula, satisfactible}.

L_{20} = {<M> | la MT M tiene una cinta, y ejecutada a partir de la entrada vacía su cabezal nunca sale del fragmento de la cinta delimitado por las primeras 20 celdas en el sentido de izquierda a derecha}.

$L_{\varnothing}$ = {<M> | L(M) = ∅}.

L_{Σ^*} = {<M> | L(M) = Σ*}.

L_{EQ} = {(<M_1>, <M_2>) | L(M_1) = L(M_2)}. Representa el problema de equivalencia de máquinas de Turing.

L_{LP} = {ψ | ψ es un teorema de la lógica de predicados}. Representa el problema de decisión en la lógica de predicados.

$L_{\mathcal{N}}$ = {ω | ω es un enunciado verdadero de la aritmética}. Representa el problema de decisión en la aritmética.

L_U = {(<M>, w) | la MT M acepta w}. Representa el problema de aceptación.

$L_{U\text{-}SK}$ = {(<M>, w, 1^K) | M es una MT con una cinta de trabajo, y acepta w utilizando a lo sumo K celdas}. Representa el problema de aceptación en espacio acotado.

$L_{U\text{-}TK}$ = {(<M>, w, 1^K) | M es una MTN con una cinta, y acepta w en a lo sumo K pasos}. Representa el problema de aceptación en tiempo acotado (con MT no determinísticas).

MAX-IND = {(G, K) | G es un grafo que tiene un conjunto independiente de vértices de tamaño K, y no tiene conjuntos independientes de vértices más grandes}.

MAY-SAT: lenguaje de las fórmulas booleanas satisfactibles por más de la mitad de las asignaciones de valores de verdad posibles.

MIN-FORM = {(φ, K) | φ es una fórmula booleana con K símbolos, y no existe ninguna fórmula booleana equivalente más chica}.

OCV: problema de optimización del cubrimiento de vértices.

OPVC: problema de optimización del viajante de comercio.

PARIDAD = {w | w es una cadena de unos y ceros con una cantidad impar de unos}.

PARTICIÓN = {T | T es un conjunto de números naturales que se puede particionar en dos subconjuntos, de manera tal que las sumas de sus elementos coincidan}.

PRIMOS: lenguaje de los números primos. Representa el problema de primalidad.

PRI-REL = {(N_1, N_2) | N_1 y N_2 son dos números naturales primos relativos}.

PVC = {(G, B) | G es un grafo completo ponderado y tiene un circuito de Hamilton tal que la suma de los valores asociados a sus arcos es menor o igual que B}. Representa el problema del viajante de comercio.

QSAT = {θ | θ es una fórmula booleana totalmente cuantificada, en FNP, verdadera}.

SAT = {φ | φ es una fórmula booleana sin cuantificadores satisfactible}. Representa el problema de satisfactibilidad.

SH = {(G, i, j) | G es un grafo que tiene un camino de Hamilton del vértice i al vértice j}.

SUB-ISO = {(G_1, G_2) | G_1 y G_2 son dos grafos tales que G_1 es isomorfo a un subgrafo de G_2}.

SUM-SUB = {(S, K) | S es un conjunto de números naturales, K es un número natural, y existe un subconjunto de S cuyos elementos suman K}.

TAUT = {φ | φ es una tautología}. Representa el problema de validez.

Otros problemas: cubrimiento exacto, determinante de una matriz, emparejamiento perfecto en un grafo bipartito, logaritmo discreto, máximo flujo en una red, permanente de una matriz, problema de la mochila, problema del circuito euleriano, programación lineal, programación lineal entera, programación lineal entera con coeficientes 0 y 1.

Teoremas

Teorema 2.1. Toda MT M_1 con varias cintas se puede simular con una MT M_2 con una cinta.

Teorema 2.2. Toda MTN M_1 se puede simular con una MTD M_2.

Teorema 2.3. Las máquinas de Turing y las máquinas RAM tienen poder computacional equivalente.

Teorema 3.1. Si $L \in R$, entonces $L^C \in R$.

Teorema 3.2. Si $L_1 \in R$ y $L_2 \in R$, entonces $L_1 \cap L_2 \in R$ y $L_1 \cup L_2 \in R$.

Teorema 3.3. Si $L_1 \in RE$ y $L_2 \in RE$, entonces $L_1 \cap L_2 \in RE$ y $L_1 \cup L_2 \in RE$.

Teorema 3.4. R = RE ∩ CO-RE.

Teorema 3.5. Un lenguaje es recursivamente enumerable sii existe una máquina de Turing que lo genera.

Teorema 3.6. Un lenguaje es recursivo sii existe una máquina de Turing que lo genera en el orden canónico.

Teorema 3.7. Los lenguajes sensibles al contexto son recursivos.

Teorema 4.1. $R \subset RE \subset \mathcal{L}$.

Teorema 4.2. (Teorema de Indecibilidad del *Halting Problem*). El lenguaje HP no es recursivo.

Teorema 5.1

a. Si $L_1 \leq_m L_2$ y $L_2 \in R$, entonces $L_1 \in R$.
b. Si $L_1 \leq_m L_2$ y $L_2 \in RE$, entonces $L_1 \in RE$.

Teorema 5.2. Las m-reducciones son reflexivas y transitivas.

Teorema 5.3. Los lenguajes L_U y HP son RE-completos.

Teorema 5.4. Si $L_1 \leq_T L_2$ y $L_2 \in R$, entonces $L_1 \in R$.

Teorema 5.5 (Teorema de Rice). Si Π es un conjunto de lenguajes recursivamente enumerables tal que $\emptyset \subset \Pi \subset RE$, entonces el lenguaje L_Π = {<M> | L(M) ∈ Π} no es recursivo.

Teorema 5.6. El problema de decisión en la aritmética es indecidible.

Teorema 5.7 (Teorema de Indecibilidad del Entscheidungsproblem). El problema de decisión en la lógica de predicados es indecidible.

Teorema 6.1. Un lenguaje pertenece a la clase NP sii existe una máquina de Turing no determinística que lo decide en tiempo polinomial.

Teorema 6.2 (Teorema de la Jerarquía Temporal). Dadas dos funciones temporales $T_1(n)$ y $T_2(n)$ tiempo-construibles, si $T_2(n) > \log_2 T_1(n) \cdot T_1(n)$ cuando n tiende a infinito, entonces se cumple que la clase temporal TIME(T(n)), siendo T(n) el máximo entre $T_1(n)$ y $T_2(n)$ para todo n, incluye estrictamente a TIME($T_1(n)$).

Teorema 6.3. El lenguaje L = {<M> | M acepta <M> en $2^{|<M>|}$ pasos} pertenece a R – P.

Teorema 7.1

a. Si $L_1 \leq_p L_2$ y $L_2 \in P$, entonces $L_1 \in P$.
b. Si $L_1 \leq_p L_2$ y $L_2 \in NP$, entonces $L_1 \in NP$.

Teorema 7.2. Sea L un lenguaje NP-completo. Si $L \in P$, entonces P = NP.

Teorema 7.3. Sea L_1 un lenguaje de NP. Si existe un lenguaje NP-completo L tal que $L \leq_p L_1$, entonces L_1 también es NP-completo.

Teorema 7.4 (Teorema de Cook-Levin). El lenguaje SAT es NP-completo.

Teorema 8.1 (Teorema de Ladner). Si $P \neq NP$, entonces dado un lenguaje recursivo $L_1 \notin P$, existe un lenguaje $L_2 \in P$ tal que:

a. $L_1 \cap L_2 \notin P$.
b. $L_1 \cap L_2$ se reduce polinomialmente a L_1.
c. L_1 no se reduce polinomialmente a $L_1 \cap L_2$.

Teorema 8.2. Sea L un lenguaje NP-completo. Si L pertenece a CO-NP, entonces NP = CO-NP.

Teorema 8.3. NLOGSPACE $\subseteq$ P.

Teorema 8.4. El lenguaje D-ACC es NLOGSPACE-completo con respecto a las reducciones log-space.

Teorema 8.5. El lenguaje QSAT es PSPACE-completo con respecto a las reducciones poly-time.

Teorema 8.6 (Teorema de la Jerarquía Espacial). Si $S_1(n)$ y $S_2(n)$ son dos funciones espacio-construibles mayores o iguales que $\log_2 n$, y $S_2(n)$ es mayor que $S_1(n)$ cuando n tiende a infinito, entonces se cumple que la clase espacial SPACE(S(n)), siendo S(n) el máximo entre $S_1(n)$ y $S_2(n)$ para todo n, incluye estrictamente a SPACE($S_1(n)$).

Teorema 8.7 (Teorema de Savitch). Toda MTN M_1 de espacio S(n), tal que S(n) es una función espacio-construible mayor o igual que $\log_2 n$, se puede simular con una MTD M_2 de espacio $O(S^2(n))$.

Teorema 8.8 (Teorema de Immerman). Si M es una MTN que decide un lenguaje L en espacio S(n), tal que S(n) es una función espacio-construible mayor o igual que $\log_2 n$, entonces existe una MTN M^C que decide el lenguaje L^C en el mismo espacio.

Teorema 9.1. Asumiendo P = NP, todo problema de búsqueda correspondiente a un lenguaje de NP se puede resolver en tiempo polinomial.

Teorema 9.2. Si existe un lenguaje PH-completo, entonces PH colapsa.

Teorema 9.3. Un lenguaje L pertenece a la clase P sii existe una familia uniforme de circuitos booleanos polinomiales $\{C_n\}_{n\geq 1}$ que lo decide.

Teorema 9.4. $NC^1 \subseteq$ LOGSPACE y NLOGSPACE $\subseteq NC^2$.

Teorema 9.5. $P^{QSAT} = NP^{QSAT}$.

Ejemplos

Ejemplo 2.1. MT M con una cinta que decide el lenguaje de las cadenas a^nb^n, con $n \geq 1$.

Ejemplo 2.2. MT M con varias cintas que decide el lenguaje de las cadenas a^nb^n, con $n \geq 1$.

Ejemplo 2.3. MTD M que decide el lenguaje ACC.

Ejemplo 2.4. MTN M que decide el lenguaje ACC.

Ejemplo 2.5. AF M que decide el lenguaje de las cadenas de *a* y b con una cantidad par de b.

Ejemplo 2.6. AP M que decide el lenguaje de las cadenas a^nb^n, con $n \geq 1$.

Ejemplo 3.1. Gramática que genera el lenguaje de las cadenas a^nb^n, con $n \geq 1$.

Ejemplo 4.1. MT M que genera el código de la máquina de Turing i-ésima según el orden canónico.

Ejemplo 4.2. Prueba por diagonalización de que $|\mathcal{R}| > |\mathcal{N}|$ (adaptación de la prueba de G. Cantor).

Ejemplo 4.3. Prueba de que $L_{20} \in$ R.

Ejemplo 4.4. Prueba por diagonalización de que $|2^{\mathcal{N}}| > |\mathcal{N}|$.

Ejemplo 4.5. Prueba por diagonalización de que el lenguaje $L = \{w_i \mid w_i \notin L(G_i)$ y G_i es la gramática sensible al contexto i-ésima según el orden canónico$\}$ es recursivo pero no sensible al contexto.

Ejemplo 5.1. m-reducción para caracterizar al lenguaje L_U.

Ejemplo 5.2. m-reducciones para caracterizar a los lenguajes HP y L_U.

Ejemplo 5.3. m-reducción para caracterizar al lenguaje L_{Σ^*}.

Ejemplo 5.4. m-reducción para caracterizar al lenguaje $L_\varnothing$.

Ejemplo 5.5. m-reducción para caracterizar al lenguaje L_{EQ}.

Ejemplo 5.6. Turing-reducción del lenguaje HP al lenguaje L_U.

Ejemplo 5.7. Turing-reducciones entre los lenguajes L_U y $L_\varnothing$.

Ejemplo 6.1. Tiempos de ejecución de una MT M_1 con una cinta y una MT M_2 con varias cintas que deciden el lenguaje de las cadenas a^kb^k, con $k \geq 1$.

Ejemplo 6.2. Tiempos de ejecución de una MTD M_1 y una MTN M_2 que deciden el lenguaje SAT.

Ejemplo 6.3. Tiempo de ejecución de una MT M que decide el lenguaje DIV-3, según se utilice la codificación unaria o la codificación binaria para los números.

Ejemplo 6.4. Prueba de que ACC $\in$ P.

Ejemplo 6.5. Prueba de que PRI-REL $\in$ P.

Ejemplo 6.6. Prueba de que 2-SAT $\in$ P.

Ejemplo 6.7. Prueba de que SAT $\in$ NP y DIV-3 $\in$ NP.

Ejemplo 6.8. Prueba de que SUM-SUB $\in$ NP.

Ejemplo 6.9. Prueba de que CH $\in$ NP.

Figuras

Bibliografía

Aaronson, S. (2003). *Is P versus NP formally independent?* Bulletin of the EATCS, 81, 109-136.

Aaronson, S. (2005). *NP-complete problems and physical reality*. SIGACT News, 36.

Aaronson, S. (2013). *Quantum computing since Democritus*. Cambridge University Press.

Agrawal, M., Kayal, N. y Saxena, N. (2004). *Primes is in P*. Annals of Mathematics, 160(2), 781-793.

Arora, S. y Barak, B. (2009). *Computational complexity: a modern approach*. Cambridge University Press.

Arora, S. y Lund, C. (1995). *Hardness of approximations*. Approximation algorithms for NP-hard problems, chapter 10, PWS, D. Hochbaum editor.

Arora, S., Lund, C., Motwani, R., Sudan, M. y Szegedy, M. (1992). *Proof verification and the hardness of approximation problems*. Versión preliminar en FOCS ´92. Nueva versión en JACM, 45(3), 501-555, 1998.

Arora, S. y Safra, S. (1992). *Probabilistic checking of proofs: a new characterization of NP*. Versión preliminar en FOCS ´92. Nueva versión en JACM, 45(1), 70-122, 1998.

Babai, L. (1985). *Trading group theory for randomness*. STOC, 421-429, ACM.

Babai, L. (2016). *Graph isomorphism in quasipolynomial time*. STOC '16, Proc. of the 48th annual ACM Symp. on Theory of Computing, 684-697.

Baker, T., Gill, J. y Solovay, R. (1975). *Relativizations of the P =? NP question*. SIAM Journal on Computing, 4, 431-442.

Balcázar, J., Díaz, J. y Gabarró, J. (1995). *Structural Complexity I*. Springer.

Bondy, J. y Murty, U. (2008). *Graph Theory*. Springer.

Boppana, R. y Sipser, M. (1990). *The complexity of finite functions*. Handbook of Theoretical Computer Science, Volume 1, Elsevier, MIT Press, J. van Leeuwen editor.

Bovet, D. y Crescenzi, P. (1994). *Introduction to the theory of complexity*. Prentice-Hall.

Chaitín, G. (2015). *El número omega. Límites y enigmas de las matemáticas*. Tusquets Editores.

Chomsky, N. (1956). *Three models for the description of language*. IRE Trans. on Information Theory, 2(3), 113-124.

Chomsky, N. (1959). *On certain formal properties of grammars*. Information and Control, 2(2), 137-167.

Church, A. (1936a). *An unsolvable problem of elementary number theory*. American Journal of Mathematics, 58(2), 345-363.

Church, A. (1936b). *A note on Entscheidungsproblem*. The Journal of Symbolic Logic, (1)1, 40-41.

Cobham, A. (1964). *The intrinsic computational difficulty of functions*. Proc. Congress for Logic, Mathematics, and Philosophy of Science, 24-30.

Cook, S. (1971). *The complexity of theorem-proving procedures*. Proc. of the 3rd IEEE Symp. on the Foundations of Computer Science, 151-158.

Cook, S. (1979). *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*. Proc. ACM Symp. on Theory of Computing, 338-345.

Cooper, B. y van Leeuwen, J. (2013). *Alan Turing: his work and impact*. Elsevier Science.

Deutsch, D. (1985). *Quantum theory, the Church-Turing principle and the universal quantum computer*. Proc. Roy. Soc. Lond. A, A400, 97-117.

Edmonds, J. (1965). *Paths, trees, and flowers*. Canadian Journal of Mathematics, 17(3), 449-467.

Edmonds, J. (1966-1967a). *Optimum branching*. Journal of Research of the National Bureau of Standards, Part B, 17B(4), 233-240.

Edmonds, J. (1966-1967b). *Systems of distinct representatives and linear algebra*. Journal of Research of the National Bureau of Standards, Part B, 17B(4), 241-245.

Even, S. y Tarjan, R. (1976). *A combinatorial problem which is complete in polynomial space*. JACM, 23, 710-719.

Feige, U., Goldwasser, S., Lovasz, L., Safra, S. y Szegedy, M. (1991). *Interactive proofs and the hardness of approximating cliques*. Versión preliminar en FOCS ´91. Nueva versión en JACM, 43(2), 268-292, 1996.

Fichte, J., Le Berre, D., Hecher, M. y Szeider, S. (2023). *The silent (r)evolution of SAT*. Comm. ACM, 66(6), 64-72.

Fischer, M. y Rabin, M. (1974). *Super-exponential complexity of Presburger arithmetic*. Complexity of Computation, Proc. SIAM-AMS Symp. In Applied Mathematics, R. Karp editor.

Garey, M. y Johnson, D. (1979). *Computers and intractability: a guide to the theory of NP-completeness*. Freeman.

Gasarch, W. (2002). *The P =? NP poll*. ACM SIGACT News, 33(2), 34-47.

Gasarch, W. (2012). *The second P =? NP poll*. ACM SIGACT News, 43(2), 53-77.

Gasarch, W. (2019). *The third P =? NP poll*. ACM SIGACT News, 50(1), 38-59.

Gill, J. (1977). *Computational complexity of probabilistic Turing machines*. SIAM Journal on Computing, 6, 675-695.

Gödel, K. (1929). *Die Vollständigkeit der Axiome des logischen Funktionenkalküls.* Doctoral dissertation, University of Vienna. En español: *La suficiencia de los axiomas del cálculo lógico de primer orden*. Obras Completas de Kurt Gödel, J. Mosterín editor, Alianza Editorial, 2006.

Gödel, K. (1931). *Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I*. Monatshefte fur Math, Und Physik, 38, 173-198. En español: *Sobre sentencias formalmente indecidibles de Principia Mathematica y sistemas afines*. Obras Completas de Kurt Gödel, J. Mosterín editor, Alianza Editorial, 2006.

Goldreich, O. (2008). *Computational complexity: a conceptual perspective*. Cambridge University Press.

Goldreich, O., Micali, S. y Wigderson, A. (1986). *Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems*. Versión preliminar en FOCS ´86. Nueva version en JACM, 38(3), 691-729, 1991.

Goldstein, R. (2005). *Gödel. Paradoja y vida*. Antoni Bosch editor.

Goldwasser, S., Micali, S. y Rackoff, C. (1985). *The knowledge complexity of interactive proof systems*. Versión preliminar en STOC ´85. Nueva versión en SIAM Journal on Computing, 18(1), 186-208, 1989.

Grover, L. (1996). *A fast quantum mechanical algorithm for database search*. STOC, 212-219, ACM.

Hartmanis, J., Lewis, P. y Stearns, R. (1965). *Hierarchies of memory-limited computations*. Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logic Design, 179-190.

Hartmanis, J. y Stearns, R. (1965). *On the computational complexity of algorithms*. Transactions of the AMS, 117, 285-306.

Herken, R. (1994). *The universal Turing machine. A half-century survey, second edition*. Springer-Verlag.

Hodges, A. (1992). *Alan Turing: the enigma*. Vintage.

Hofstadter, D. (1989). *Gödel, Escher, Bach. Un eterno y grácil bucle*. Tusquets Editores.

Hopcroft, J. y Ullman J. (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesley.

Huth, M. y Ryan, M. (2004). *Logic in computer science*. Cambridge University Press.

Immerman, N. (1988). *Nondeterministic space is closed under complementation*. SIAM Journal on Computing, 17, 935-938.

Immerman, N. (1999). *Descriptive complexity*. Springer.

Johnson, D. (1974). *Approximation algorithms for combinatorial problems*. Journal of Computer and System Sciences, 9, 256-278.

Jourdain, P. (1955). *Georg Cantor. Contributions to the founding of the theory of transfinite numbers*. Dover Publications.

Kapron, B. (2023). *Logic, automata, and computational complexity: the works of Stephen A. Cook*. ACM Books.

Karp, R. (1972). *Reducibility among combinatorial problems*. Complexity of Computer Computations, J. Thatcher y R. Miller editors., Plenum Press, New York, 85-103.

Karp, R. (1990). *An introduction to randomized algorithms*. Technical Report, University of California, Berkeley.

Kearns, M. (1990). *The computational complexity of machine learning*. The MIT Press.

Kleene, S. (1936). *General recursive functions of natural numbers*. Mathematische Annalen, 112, 727-742.

Ladner, R. (1975). *On the structure of polynomial-time reducibility*. JACM, 22, 155-171.

Levin, L. (1973). *Universal'nyie perebornyie zadachi*. Problemy Peredachi Informatsii, 9, 115,116. En inglés: *Universal search problems*. Problems of Information Transmission, 9, 265-266.

Lewis, H. y Papadimitriou, C. (1998). *Elements of the theory of computation*. Prentice-Hall.

Lewis, P., Stearns, R. y Hartmanis, J. (1965). *Memory bounds for recognition of context-free and context-sensitive languages*. Proc. 6th Annual IEEE Symp. On Switching Circuit Theory and Logical Design, 191-202.

Markov, A. (1954). *The theory of algorithms*. Trudy Mat., Inst. Steklov, Acad. Sci. USSR, 42, 3-375.

Martínez, G. y Piñeiro, G. (2009). *Gödel ∀ (para todos)*. Seix Barral.

Mendelson, E. (2010). *Introduction to mathematical logic, 5th edition*. CRC Press.

Meyer, A. y Stockmeyer, L. (1972). *The equivalence problem for regular expressions with squaring requires exponential time*. FOCS, 125-129, IEEE.

Moore. C. y Mertens, S. (2011). *The nature of computation*. Oxford University Press.

Nielsen, M. y Chuang, I. (2010). *Quantum computation and quantum information*. Cambridge University Press.

Papadimitriou, C. (1994). *Computational complexity*. Addison-Wesley.

Papadimitriou, C. y Yannakakis, M. (1991). *Optimization, approximation, and complexity classes*. Journal of Computer and System Sciences, (28)2, 244-259.

Penrose, R. (1989). *La mente nueva del emperador. En torno a la cibernética, la mente y las leyes de la física*. Consejo Nacional de Ciencia y Tecnología, Fondo de Cultura Económica, México.

Petzold, C. (2008). *The annotated Turing*. *A guided tour through Alan Turing´s historic paper on computability and the Turing machine*. Wiley.

Piñeiro, G. (2013). *Cantor. El infinito en matemáticas. Lo incontable es lo que cuenta*. Grandes Ideas de la Ciencia.

Pippenger, N. (1979). *On simultaneous resource bounds*. Proc. IEEE Symp. on Foundations of Computer Science, 307-311.

Post, E. (1936). *Finite combinatory process - formulation 1*. Journal of Symbolic Logic, 1(3), 103-105.

Post, E. (1946). *A variant of a recursively unsolvable problem*. Bulletin of the American Mathematical Society, 52(4), 264-268.

Rabin, M. (1980). *A probabilistic algorithm for testing primality*. Journal of Number Theory, 12(1), 128-138.

Rabin, M. y Scott, D. (1959). *Finite automata and their decision problems*. IBM Journal of Research and Development, 3, 114-125.

Reingold, O. (2005). *Undirected ST-connectivity in log-space*. STOC, 376-385, ACM.

Rice, H. (1953). *Classes of recursively enumerable sets and their decision problems*. Trans. Amer. Math. Soc., 89, 25-59.

Rice, H. (1956). *On completely recursively enumerable classes and their key array*s. Journal of Symbolic Logic, 21, 304-341.

Rosenfeld, R. e Irazábal, J. (2010). *Teoría de la computación y verificación de programas*. EDULP, McGraw-Hill.

Rosenfeld, R e Irazábal, J. (2013). *Computablidad, complejidad computacional y verificación de programas*. EDULP.

Savitch, W. (1970). *Relationship between nondeterministic and deterministic tape complexities*. Journal of Computer and System Sciences, 4, 177-192.

Shamir, A. (1990). *IP = PSPACE*. Versión preliminar en FOCS ´90. Nueva versión en JACM, (39)4, 869-877, 1998.

Shepherdson, J. y Sturgis, H. (1963). *Computability of recursive functions*. JACM, 10, 217-255.

Shor, P. (1994). *Algorithms for quantum computation: discrete logarithms and factoring*. Proc. 35th Annual Symp. on Foundations of Computer Science, IEEE Press.

Singh, S. (1997). *El último teorema de Fermat*. Editorial Páprika.

Sipser, M. (1992). *The history and status of the P versus NP question*. STOC, 603-618, ACM.

Sipser, M. (1997). *Introduction to the theory of computation*. PWS Publishing Company.

Solovay, R. y Strassen,V. (1977). *A fast Monte-Carlo test for primality*. SIAM Journal on Computing, 6, 84-85.

Stockmeyer, L. (1977). *The polynomial-time hierarchy*. Theoretical Computer Science, 3, 1-22.

Stockmeyer, L. y Meyer, A. (1973). *Word problems requiring exponential time*. Proc. 5th ACM Symp. on the Theory of Computing, 1-9.

Stockmeyer, L. y Vishkin, U. (1984). *Simulation of parallel random access machines by circuits*. SIAM Journal on Computing, 13, 409-422.

Toda, S. (1991). *PP is as hard as the polynomial-time hierarchy*. SIAM Journal on Computing, 20(5), 865-877.

Turing, A. (1936). *On computable numbers, with an application to the Entscheidungsproblem*. Proc. London Mathematical Society, 2(42), 230-265.

Turing, A. (1939). *Systems of logic based on ordinals*. Proc. London Mathematical Society, 2(45), 161-228.

Valiant, L. (1979a). *Completeness classes in algebra*. STOC, 249-261, ACM.

Valiant, L. (1979b). *The complexity of computing the permanent*. Theoretical Computer Science, (8)2, 189-201.

Whitehead, A. y Russell, B. (1913). *Principia Mathematica*. Cambridge University Press.

Wrathall, C. (1977). *Complete sets and polynomial time hierarchy*. Theoretical Computer Science, 3, 23-33.

Yao, A. (1982). *Theory and applications of trapdoor functions*. FOCS, 80-91, IEEE.

Yao, A. (1993). *Quantum circuit complexity*. FOCS, 352-361, IEEE.

El autor

Ricardo Rosenfeld (rosenfeldricardo@gmail.com) obtuvo el título de Calculista Científico de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata (UNLP), Argentina, en 1983, y complementó sus estudios en el Instituto de Tecnología Technión, Israel, entre 1988 y 1991 (Maestría en Ciencias de la Computación).

Desde 1991 se desempeña como Profesor Adjunto en la UNLP (primero en la Facultad de Ciencias Exactas y después en la Facultad de Informática), en las áreas de Teoría de la Computación y Verificación de Programas, y desde 2021 como Profesor e Investigador en la Universidad Abierta Interamericana, en el área de Métodos Formales en la Ingeniería de Software. Previamente, entre 1984 y 1990, fue docente en la UNLP (lenguajes y metodologías de programación), en la Universidad de Buenos Aires (derivación y verificación de programas), en la Escuela Superior Latinoamericana de Informática (algorítmica y teoría de compiladores) y en el Instituto de Tecnología Technión de Israel (programación).

Entre sus publicaciones se destacan los libros *Teoría de la Computación y Verificación de Programas* (Rosenfeld e Irazábal, 2010, EDULP y McGraw-Hill), *Computabilidad, Complejidad Computacional y Verificación de Programas* (Rosenfeld e Irazábal, 2013, EDULP), *Lógica para Informática* (Pons, Rosenfeld y Smith, 2017, EDULP) y *Verificación de Programas. Programas Secuenciales y Concurrentes* (Rosenfeld, 2024, EDULP).

En el ámbito de la industria informática, fue Líder de Proyectos y Responsable del Area de Arquitectura Informática en Telefónica Argentina, entre los años 1991 y 1997, y desde 1997 hasta 2023 formó parte del Grupo Practia, conjunto de empresas de Tecnología de la Información con oficinas en América y Europa. En dicho grupo se desenvolvió, en varios países, como Líder de Proyectos, Consultor, Responsable de Recursos Humanos y Responsable de Cuentas Comerciales, incorporándose además, a partir de 1999, como Director y Socio.