Apuntes de
COMPLEJIDAD COMPUTACIONAL
(Traducción con modificaciones y adiciones de las Lecture Notes Complexity of Algorithms, de P. Gács y L. Lovász, 1999)

Claudio Gutiérrez
Departamento de Ciencias de la Computación, Universidad de Chile
cgutierr@dcc.uchile.cl

Ilustraciones de Pedro Ortega

Estos apuntes son algunos capítulos del proyecto de traducción al castellano de las Lecture Notes on Complexity of Algorithms de P. Gács y L. Lovász, que gracias al permiso de los autores estará accesible (pronto) públicamente en `http://www.dcc.uchile.cl/~cgutierr`. En ella tomaron parte los alumnos del curso de Complejidad Computacional (PUC, 2do. Semestre 2000, Chile), C. Acuña, P. Barceló, R. Cabello, P. Castañeda, A. Cortés, P. Moraga, E. Onetto, F. Orchard, L. Ramírez, R. Rojas y C. Tondreau. Contribuyó también M. Cecilia Bastarrica.

La presente versión se benefició de las sugerencias, comentarios y correcciones de Pedro Ortega, quien es el autor de todas las ilustraciones.

La edición, modificaciones, adiciones y por supuesto los errores corresponden a C. Gutiérrez. Cualquier comentario es muy bienvenido. El texto original en inglés se encuentra disponible en `http://www.cs.yale.edu/homes/lovasz/notes.html`.

Índice General

1 Modelos de Computación **5**
1.1 Introducción 5
1.2 La máquina de Turing 7
1.3 Máquina de Turing Universal 9
1.4 Varias Cintas Versus Una Cinta 11
1.5 La máquina de Acceso Aleatorio (RAM) 14
1.6 Otros modelos 18
1.7 Ejercicios 20

2 Computación con recursos limitados **23**
2.1 Problemas e instancias 23
2.2 Solubilidad algorítmica versus solubilidad práctica 24
2.3 Tiempo y espacio 25
2.3.1 Tiempo versus espacio 26
2.3.2 Medidas para RAM 27
2.4 Consideraciones técnicas sobre las medidas 28
2.5 Complejidad Intrínseca 30
2.6 Ejercicios 30

3 Tiempo polinomial **31**
3.1 Algoritmos en Aritmética 32
3.1.1 Operaciones aritméticas 32
3.1.2 Eliminación Gaussiana 34
3.2 Algoritmos en grafos 37
3.2.1 ¿Como representar un grafo? 37
3.2.2 Búsqueda en grafos 38
3.2.3 Búsqueda en anchura y caminos más cortos 38
3.2.4 Correspondencia máximo en grafos bipartitos y caminos alternativos. 39
3.3 Ejercicios 41

4 Computación No-determinística 43
4.1 Máquinas de Turing no-determinísticas . . . 43
4.2 Testigos . . . 45
4.3 Resultados generales sobre clases no determinísticas . . . 47
4.4 Ejemplos de Lenguajes en NP . . . 50
4.4.1 Grafos . . . 50
4.4.2 Aritmética y álgebra . . . 53
4.4.3 Sistemas de Inecuaciones Lineales . . . 55

5 Complejidad y Aleatoriedad 59
5.1 Introducción . . . 59
5.2 Verificación de una identidad polinomial . . . 59
5.3 Correspondencia perfecta en grafos . . . 61
5.4 Verificación de primalidad . . . 62
5.5 Clases de complejidad aleatorias . . . 65
5.6 Ejercicios . . . 69

6 Complejidad y Paralelismo 71
6.1 Máquinas de Acceso Aleatorio Paralelas . . . 71
6.2 La clase NC . . . 75
6.3 Problemas en NC . . . 75
6.4 Ejercicios . . . 77

7 Aproximación 79
7.1 Problemas de Optimización . . . 79
7.2 Algoritmos de aproximación . . . 80
7.3 Clases de Complejidad Aproximadas . . . 82
7.3.1 Relaciones entre clases y reducibilidad . . . 82
7.4 Ejercicios . . . 82

8 Apéndices 85
8.1 La notación asintótica $\mathcal{O}$, Ω y Θ . . . 85
8.2 Computabilidad . . . 85

Capítulo 1

Modelos de Computación

1.1 Introducción

En esta sección trataremos el concepto de "computación" o "algoritmo". Este concepto es fundamental para nuestro trabajo, pero no lo definiremos formalmente. Lo que haremos es considerarlo como una noción intuitiva la cual es favorable para varios tipos de formalización que nos permitirán analizarlo rigurosamente.

Un *algoritmo* significa un procedimiento que sirve para una computación o una construcción (la computación de alguna función), la cual puede ser llevada a cabo "mecánicamente", sin pensar. Otro punto de vista es decir que es "una prescripción exacta del orden determinado en que ha de ejecutarse un sistema de operaciones para resolver todos los problemas de un cierto tipo."[?]. Esta no es realmente una definición rigurosa, pero uno de los propósitos de este curso es demostrar que posible hacer una discusión general en estos términos. (Esta discusión es a menudo formulada como la *tesis de Church.*) Un programa en un lenguaje de programación (C, JAVA o cualquier otro) es un buen ejemplo de una especificación de un algoritmo. Debido a que la naturaleza "mecánica" de un algoritmo es su característica más importante, para precisar la noción de algoritmo introduciremos varios conceptos de máquina abstracta que cumplen estas características.

Las *máquinas abstractas* o matemáticas computan alguna salida a partir de alguna entrada. La entrada y la salida puede ser una *palabra* de un alfabeto finito, es decir, una secuencia finita de símbolos del alfabeto, muchas veces también llamada *cadena*. Las máquinas abstractas son muy parecidas a los computadores que el lector conoce pero algo idealizadas: omitimos algunas características innecesarias para nuestra discusión (por ejemplo fallas del hardware) y agregamos una memoria que es infinitamente expansible.

El siguiente es un típico problema que a menudo resolvemos en los computadores:

dada una lista de nombres, ordenarlos en orden alfabético. La entrada es una cadena que consiste de nombres como Roberto, Carlos, Alicia. La salida es también una cadena. El problema es computar una función que asigne a cada cadena de nombres su copia alfabéticamente ordenada.

En general, un problema algorítmico típico es realmente un conjunto de problemas del mismo tipo. Por ejemplo, "encontrar la raíz cuadrada de un número" es un problema que realmente consiste de infinitos problemas particulares: uno

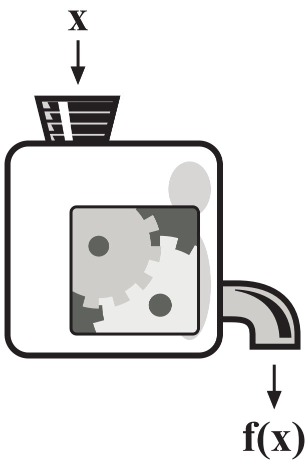

Figura 1.1: Una máquina abstracta

para cada número. Estos problemas particulares se llaman *instancias* del problema general, y su número puede ser infinito, y por lo tanto pueden tener un tamaño arbitrariamente grande. Por eso debemos considerar una familia infinita de computadores finitos de tamaño creciente, o algún computador infinito idealizado. La última aproximación tiene la ventaja de que evita las preguntas sobre qué familias infinitas son las permitidas.

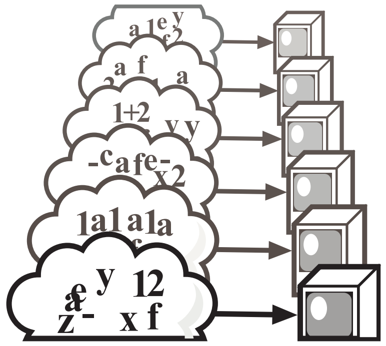

Figura 1.2: Un computador infinito

Históricamente, el primer modelo infinito puro de computación fue la *máquina de Turing*, introducida por el matemático inglés Alan Turing en 1936, antes de la creación de los computadores programables. Esencialmente el modelo consta de una unidad central que es finita (con una estructura independiente de la entrada) y una capacidad de almacenamiento (memoria) infinita. Más exactamente, la memoria es un arreglo uni-dimensional infinito de celdas. La unidad de control es un autómata finito capaz de hacer cambios locales arbitrarios a la celda de memoria explorada y de cambiar gradualmente la posición explorada. En las máquinas de Turing, es posible efectuar todos los cálculos que pueden ejecutarse en *cualquier otro* modelo de máquina abstracta. Esta noción de máquina es usada principalmente en investigaciones teóricas. Este modelo es menos apropiado para la definición de algoritmos concretos ya que su descripción es extraña, y principalmente dado que difiere de los computadores actuales en varios aspectos importantes.

La debilidad más importante de la máquina de Turing en comparación con los computadores reales es que su memoria no es accesible inmediatamente: para leer una celda de memoria distante, hay que recorrer todas las celdas intermedias. Esto es remediado por la *máquina de acceso aleatorio* (en inglés Random Access Machine, RAM).[1] La RAM puede alcanzar cualquier celda de memoria en un solo paso. Puede ser considerada un modelo simplificado de un computador del mundo real con la abstracción adicional de que tiene memoria ilimitada y la capacidad de almacenar números enteros de gran tamaño en cada una de sus celdas de memoria. La RAM puede

[1] Shepherdson y Sturgis en 1963 introdujeron este modelo que es el más similar al moderno computador digital, o más bien una abstracción de él. A diferencia de otros modelos, éste fué obtenido como una abstracción de los computadores digitales existentes ya en esa época.

ser programada en un lenguaje de programación arbitrario. Para la descripción de algoritmos, es práctico usar la RAM ya que es lo más cercano a la estructura de un programa real. Pero veremos que la máquina de Turing y la RAM son equivalentes desde varios puntos de vista, siendo el más importante entre ellos que las funciones que es posible computar en las máquinas de Turing y en la RAM son las mismas.

A pesar de sus aparentes limitaciones teóricas, también consideraremos los *circuitos lógicos* como un modelo de computación. Un circuito lógico dado permite sólo entradas de un tamaño fijo, y por lo tanto, puede resolver sólo un número finito de problemas. Sin embargo, será evidente que para entradas de un tamaño fijado, cualquier función es computable por algún circuito lógico. Más aún, si restringimos el tiempo de cálculo, entonces la diferencia entre los problemas resolubles por circuitos lógicos o por máquinas de Turing o por RAM no será esencial. Dado que la estructura y el trabajo de los circuitos lógicos son muy transparentes y manejables, ellos juegan un rol muy importante en las investigaciones teóricas (especialmente en las demostraciones de cotas inferiores en complejidad).[2]

El concepto clave usado en la discusión de modelos de máquinas es el de *simulación*. Esta noción no será definida en su completa generalidad, ya que también se refiere a máquinas o lenguajes que todavía no han sido inventados. Pero sus significados estarán claros. Decimos que la máquina M simula a la máquina N si a la máquina M se le pueden asociar los estados internos y las transiciones de N s de manera tal que para una misma entrada, M compute la misma salida que N.

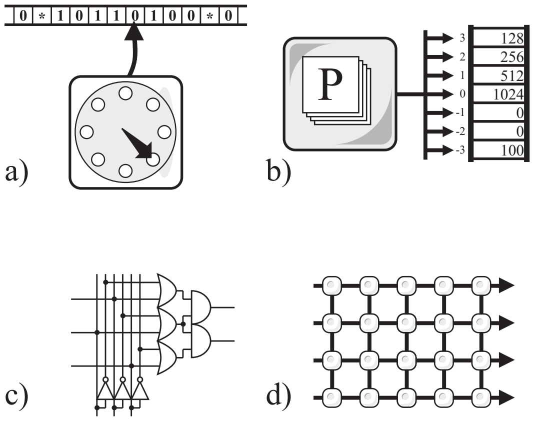

Figura 1.3: Modelos de computación

1.2 La máquina de Turing

Informalmente, una *máquina de Turing* es un autómata finito equipado con memoria ilimitada. Esta memoria está dada en la forma de una o más cintas, las cuales son infinitas en ambas direcciones. Cada cinta tiene una celda de partida especial que también podemos llamar la celda 0-ésima. Cada celda de cada cinta, excepto un número finito de celdas, contiene un símbolo especial $*$ del alfabeto, que denota la "celda vacía".

Para acceder a la información en las cintas, entregamos a cada cinta un cabezal de lectura-escritura que en cada paso se sitúa en una celda de la cinta. En esa posición, es posible leer el contenido y/o (sobre-)escribir un nuevo símbo-

[2]Si se agrega un reloj y un registro de memoria a un circuito lógico llegaremos a un *autómata finito* interconectado que conforma los componentes de hardware típicos de los computadores de hoy. Dejemos claro que un autómata finito dado, cuando es usado en entradas de tamaño arbitrario, puede computar sólo funciones muy primitivas, y no es un modelo de computación adecuado. Uno de los modelos más simples para una máquina infinita es conectar un número infinito de autómatas similares dentro de un arreglo. De esta manera obtenemos un *autómata celular*.

lo en dicha celda. Los cabezales de lectura-escritura están conectados a una unidad de control que cual es un autómata finito. Sus posibles estados forman un conjunto finito Γ. Hay un estado de partida "START" y un estado de parada "STOP". Inicialmente, la unidad de control está en el estado "START", y los cabezales se ubican en las celdas de partida de las cintas.

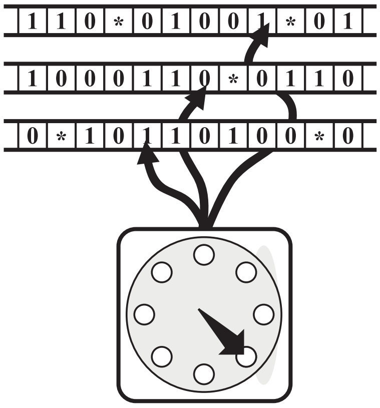

Figura 1.4: Máquina de Turing con tres cintas

En cada paso, cada cabezal lee el símbolo que contiene la celda de la cinta correspondiente, y lo envía a la unidad de control. Dependiendo de este símbolo y de su propio estado, la unidad de control hace tres operaciones:

1. Envía un símbolo a cada cabezal para sobreescribir el símbolo en la cinta;

2. Envía uno de los comandos "MOVER A DERECHA", "MOVER A IZQUIERDA", "PERMANECER" a cada cabezal;

3. Hace una transición a un nuevo estado (que puede ser el mismo que el anterior).

Por supuesto, los cabezales se encargan de estos comandos, los cuales completan un paso de la computación. La máquina se detiene cuando la unidad de control alcanza el estado "STOP".

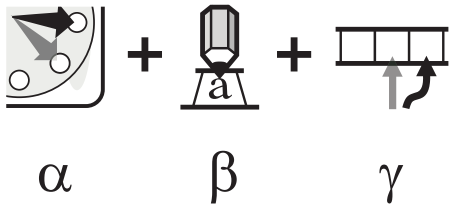

Figura 1.5: Tres funciones de una máquina de Turing

Aunque la descripción informal de arriba usa alguna jerga de ingeniería, no es difícil traducirla a términos puramente formales (matemáticos). Para nuestros propósitos, una máquina de Turing está completamente especificada por los siguientes datos: $T = (k, \Sigma, \Gamma, \alpha, \beta, \gamma)$ donde $k \geq 1$ es un número natural, Σ y Γ son conjuntos finitos, $* \in \Sigma$, START, STOP $\in \Gamma$, y α, β y γ son funciones arbitrarias,

$$\alpha : \Gamma \times \Sigma^k \longrightarrow \Gamma,$$
$$\beta : \Gamma \times \Sigma^k \longrightarrow \Sigma^k,$$
$$\gamma : \Gamma \times \Sigma^k \longrightarrow \{-1, 0, 1\}^k,$$

Aquí α especifica el nuevo estado, β entrega los símbolos a ser escritos en la cinta y γ especifica cómo se mueven los cabezales (ver Figura 1.5). En lo que sigue fijaremos el alfabeto Σ y asumiremos que contiene, además del símbolo blanco $*$, por lo menos otros dos símbolos, digamos 0 y 1 (en la mayoría de los casos estos dos símbolos serán suficientes).

Por la *entrada* de la máquina de Turing, entendemos las k palabras inicialmente escritas en las cintas. Siempre asumimos que estas son escritas en las cintas partiendo por la posición 0. Así, la entrada de la máquina de Turing de k cintas es una k-tupla ordenada, cada elemento de los cuales es una palabra en Σ^*. A menudo la entrada consiste de una sola palabra x. Entonces cuando hablamos de la entrada x entenderemos la entrada consistente en la k-tupla $(x, 0, \ldots, 0)$. La *salida* de la máquina es una k-tupla ordenada que consiste en las palabras que quedan en las cintas cuando la máquina se detiene. Frecuentemente, estamos realmente interesados sólo en una palabra, el resto es "basura". Cuando decimos que la salida es una palabra única y no especificamos cuál, entonces entenderemos que es la palabra escrita en la última cinta.

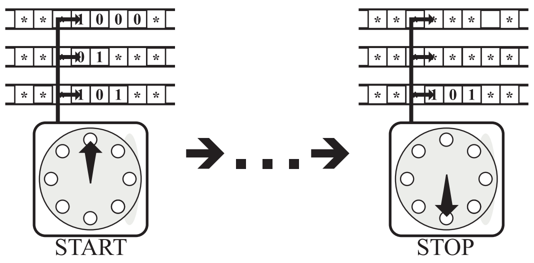

Figura 1.6: Partida y término de la computación en una máquina de Turing

Es útil asumir que las palabras de entrada no contienen el símbolo $*$. De otro modo, no sería posible saber dónde está el fin de la palabra de entrada: un simple problema tal como "encontrar el largo de la entrada" no sería soluble: como no hay manera de saber cuán lejos se ha movido el cabezal, no sabríamos si la entrada ya ha terminado. Denotamos el alfabeto $\Sigma \setminus \{*\}$ por Σ_0. (Una solución alternativa podría ser reservar un símbolo para señalar el "fin de la entrada".) También asumimos que durante su trabajo, la máquina de Turing lee su entrada completa: con esto, sólo estamos dejando de lado casos triviales.

Es posible definir las máquinas de Turing de muchas maneras, pero afortunadamente desde todos los puntos de vista importantes estas definiciones son equivalentes. Generalmente, las cintas son infinitas sólo en una dirección; su número puede ser virtualmente siempre restringido a dos y en muchos casos incluso a uno; podemos asumir que aparte del símbolo $*$ (el cual en este caso lo identificamos con el 0) el alfabeto contiene sólo el símbolo 1; sobre algunas cintas, podemos estipular que la máquina sólo puede leer desde ellas o sólo puede escribir en ellas. La equivalencia de estas variantes (desde el punto de vista de las computaciones realizables en ellas) puede ser verificada con mayor o menor trabajo, pero sin grandes dificultades. En este sentido, demostraremos en estas notas sólo lo que vayamos necesitando, pero creemos que esto da una familiaridad suficiente con las herramientas de tales simulaciones para que el lector haga algunos ejercicios por su cuenta.

1.3 Máquina de Turing Universal

Basado en lo precedente, podemos notar una diferencia significativa entre las máquinas de Turing y los computadores reales: para el cómputo de cada función, hay que construir una máquina de Turing separada, mientras que en los computadores reales controlados por programas, es suficiente escribir un programa apropiado. Mostraremos ahora que las máquinas de Turing también pueden ser operadas de esta manera: es posible construir una máquina de Turing en la cual, usando programas adecuados, es posible computar todo lo que es computable en cualquier máquina de Turing. Tales máquinas de Turing son interesantes no sólo porque semejan más a los computadores programables, sino porque también juegan un rol muy importante en muchas demostraciones.

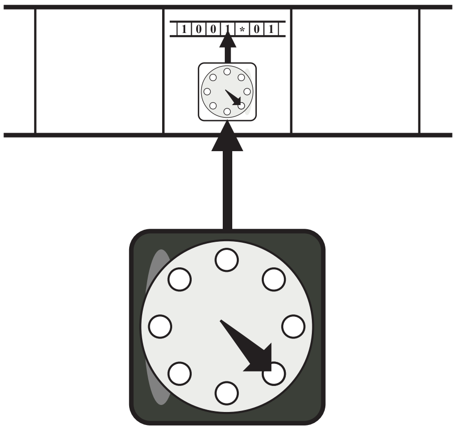

Figura 1.7: Una máquina de Turing Universal

Sean $T = (k+1, \Sigma, \Gamma_T, \alpha_T, \beta_T, \gamma_T)$ y $S = (k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S)$ dos máquinas de Turing ($k \geq 1$). Sea $p \in \Sigma_0^*$. Decimos que T *simula a* S *con el programa* p si para palabras arbitrarias $x_1, \ldots, x_k \in \Sigma_0^*$ la máquina T para en un número finito de pasos con la entrada $(x_1, \ldots, x_k, p)$ si y sólo si S para con la entrada $(x_1, \ldots, x_k)$ y si al momento de parar, las primeras k cintas de T tienen el mismo contenido que las cintas de S. Decimos que una máquina de Turing de $(k+1)$ cintas es *universal* (con respecto a máquinas de Turing de k cintas) si para cada máquina de Turing de k cintas S sobre Σ hay una palabra (programa) p con el cual T simula a S.

Teorema 1 *Para cada número $k \geq 1$ y cada alfabeto Σ existe una máquina de Turing universal de $(k+1)$ cintas.*

DEMOSTRACIÓN. La idea básica de la construcción de una máquina de Turing universal consiste en escribir en la cinta $k+1$ una tabla que describa el trabajo de la máquina de Turing S a ser simulada (esencialmente las funciones α, β, γ). Aparte de esto, la máquina de Turing universal T debe llevar la cuenta del estado en que está en cada momento la máquina simulada S. (Esto no es sencillo, pues aunque cada máquina tiene un número finito de estados, como la máquina fija T debe simular *todas* las máquinas, no podría retener los estados de cada una). En cada paso, basándose en el estado de la máquina simulada S y en los símbolos leídos en las otras cintas, busca en la tabla: (1) el estado al cual S hace la transición, (2) lo que tiene que escribir en las cintas y (3) hacia dónde debe mover los cabezales. Daremos la construcción usando $k+2$ cintas.[3]

Para simplificar, asumiremos que S contiene los símbolos "0", "1" y "-1". Sea $S = (k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S)$ una máquina de Turing de k cintas arbitraria. Identificaremos cada elemento g del conjunto de estados $\Gamma_S \setminus \{\text{STOP}\}$ con una palabra $w[g]$ de largo r sobre el alfabeto Σ_0^*. El "código" de una posición dada de la máquina S será la siguiente palabra:

$$w[g]h_1 \cdots h_k w[\alpha_S(g, h_1, \ldots, h_k)] \\ \beta_S(g, h_1, \ldots, h_k)\gamma_S(g, h_1, \ldots, h_k) \quad (1.1)$$

donde $g \in \Gamma_S$ es el estado dado de la unidad de control y $h_1, \ldots, h_k \in \Sigma$ son los símbolos leídos

[3] Podemos deshacernos fácilmente de la cinta $(k+2)$: su contenido (el cual siempre son sólo r celdas) será puesto en las celdas $-1, -2, \ldots, -r$. Sin embargo, parece que todavía necesitamos dos cabezales en esta cinta: una se mueve en la mitad positiva mientras que la otra en la negativo (no necesitan cruzarse). Resolveremos esto duplicando cada celda: el símbolo original se queda en la mitad izquierda, y en la mitad derecha hay un 1 si el cabezal correspondiente estuviera ahí (la otra mitad derecha de celdas quedan vacías). Es fácil describir como un cabezal debe moverse en esta cinta para que puedan simular los movimientos de ambos cabezales originales.

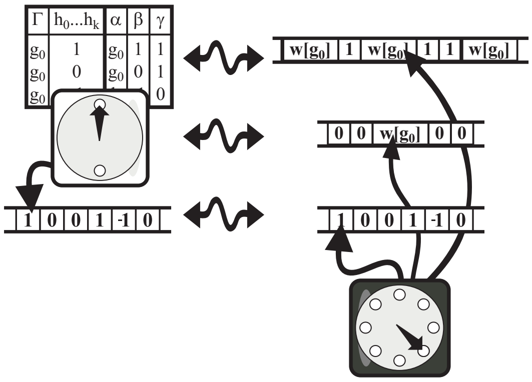

Figura 1.8: Codificación de una máquina de Turing

por cada cabezal. Concatenando todas esas palabras obtenemos una palabra a_S. Esto es lo que escribimos en la cinta $k+1$; mientras en la cinta $k+2$ escribimos un estado de la máquina S, inicialmente el nombre del estado START.

A continuación, construimos la máquina de Turing T' que simula un paso de S como sigue. Estando en la cinta $k+1$, busca la entrada correspondiente al estado recordado en la cinta $k+2$ y los símbolos leídos por los primeros k cabezales. Entonces lee desde ahí lo que tiene que hacer: escribe el nuevo estado en la cinta $k+2$, deja que los primeros k cabezales escriban el símbolo apropiado y se mueve en la dirección apropiada. $\odot$

1.4 Varias Cintas Versus Una Cinta

El siguiente resultado muestra que, en cierto sentido, no es esencial cuántas cintas tenga una máquina de Turing.

Teorema 2 *Para cada máquina de Turing S de k cintas existe una máquina de Turing T de una cinta que reemplaza a S de la siguiente manera: para cada palabra $x \in \Sigma_0^*$, la máquina S se detiene en un número finito de pasos en la entrada x si y sólo si T se detiene en la entrada x, y al momento de parar, la ultima cinta de S tiene escrito lo mismo que la cinta de T. Además si S demora N pasos entonces T demora $\mathcal{O}(N^2)$ pasos.*

Demostración. Debemos almacenar el contenido de las cintas de S en la única cinta de T. Para esto, primero debemos estirar la entrada escrita en la cinta de T: copiamos el símbolo encontrado en la i-ésima celda a la celda $(2ki)$-ésima. Esto puede ser hecho de la siguiente manera: primero empezando desde el último símbolo y saltando hacia la derecha copiamos cada símbolo $2k$ posiciones a la derecha. Mientras tanto, escribimos "*" en las posiciones $1, 2, \ldots, 2k-1$. Después, empezando desde el último símbolo, movemos cada símbolo del último bloque de no-blancos $2k$ posiciones a la derecha, etc.

Ahora, la posición $2ki+2j-2$ $(1 \leq j \leq k)$ corresponderá a la i-ésima celda de la cinta j, y la posición $2k+2j-1$ tendrá un 1 o una * dependiendo de que si el cabezal correspondiente de S, en el paso correspondiente a la computación de S, esté explorando esa celda o no. También, marcamos con un 0 la primera celda par de los finales vacíos de las cintas. De esta forma asignamos una configuración de T a cada configuración de las computaciones de S.

Ahora mostraremos cómo T puede simular los pasos de S. Primero que nada T retiene el estado en que se encuentra S. También sabe cuál es el resto del número de la celda módulo $2k$ que fue explorado por su propio cabezal. Empezando desde la derecha, dejamos que el cabe-

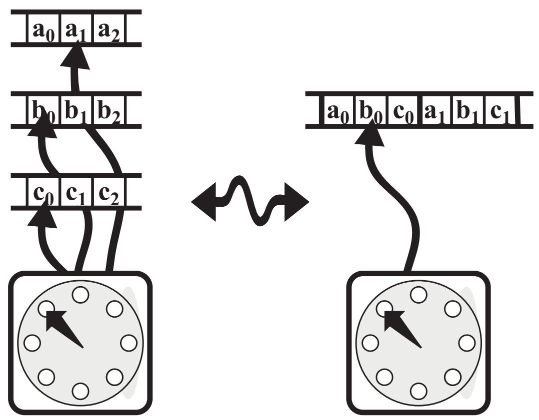

Figura 1.9: Simulación de varias cintas con una

zal ahora haga una pasada a través de la cinta completa. En el momento en que alcance el fin sabrá cuáles son los símbolos leídos por los cabezales de S en este paso. Desde aquí, computa cuál será el nuevo estado de S, lo que deben escribir los cabezales y en qué dirección se deben mover. Empezando desde atrás, por cada 1 encontrado en una celda impar, puede sobreescribir correspondientemente la celda que esta antes de esta y también puede mover el 1 en $2k$ posiciones hacia la izquierda o hacia la derecha si esto fuese necesario. (Si en el intertanto, pudiera pasar más allá del 0 del principio o del final, entonces también podría moverla en 2k posiciones en la dirección apropiada.)

Cuando la simulación de la computación de S haya terminado, el resultado todavía debe ser comprimido: el contenido de la celda $2ki$ debe ser copiado a la celda i. Esto puede ser hecho en forma similar al "estiramiento" inicial. Obviamente, la máquina T descrita arriba computará las mismas cosas que la máquina S. El número de pasos está compuesto de tres partes: los tiempos tomados por el "estiramientos", la simulación y la "compresión". Sea M el número de celdas en la máquina T qu siempre serán exploradas por la máquina; obviamente $M = \mathcal{O}(N)$. El "estiramiento" y la "compresión" necesitan tiempo $\mathcal{O}(M^2)$. La simulación de un paso de S necesita $\mathcal{O}(M)$ pasos, por lo tanto, la simulación necesita $\mathcal{O}(MN)$ pasos. Todo esto junto suma $\mathcal{O}(N^2)$ pasos. $\odot$

Como hemos visto, la simulación de una máquina de Turing de k cintas por una máquina de Turing de 1 cinta no es completamente satisfactoria: el número de pasos aumenta cuadráticamente. Esto no es sólo una debilidad de la construcción específica que hemos descrito; de hecho existen tareas computacionales que pueden ser resueltas en una máquina de Turing de 2 cintas en sólo N pasos, pero para las cuales cualquier máquina de Turing de 1 cinta necesita N^2 pasos. Describiremos un ejemplo sencillo de tales problemas. Un *palíndrome* es una palabra (digamos, sobre el alfabeto $\{0, 1\}$) que no cambia cuando se lee en orden reverso. Por ejemplo, $x_1 \cdots x_n$ es un palíndrome si y sólo si $x_i = x_{n-i+1}$ para todo i. Analicemos el problema de reconocer un palíndrome.

Teorema 3 1. *Existe una máquina de Turing de 2 cintas que decide si una palabra de entrada $x \in \{0, 1\}^n$ es un palíndrome en $\mathcal{O}(n)$ pasos.*

2. *Cada máquina de Turing de 1 cinta que decida si una palabra de entrada $x \in \{0, 1\}^n$ es un palíndrome tiene que hacer $\Omega(n^2)$ pasos en el peor de los casos.*

Demostración. La parte (1) es fácil: por ejemplo, podemos copiar la entrada en la segunda cinta en n pasos, entonces movemos el primer cabezal hasta el principio de la entrada en n pasos más allá (dejamos el segundo cabezal en el final de la palabra), y comparamos x_1 con x_n, x_2

con x_{n-1} etc., en otros n pasos. Todo junto, esto tomaría sólo $3n$ pasos.

La parte (2) es más difícil de demostrar. Considere cualquier máquina de Turing de 1 cinta que reconozca palíndromes.

Para ser más específicos, diremos que termina escribiendo un "1" en el campo de partida en la cinta si la palabra de entrada es un palíndrome, y un "0" si no lo es.

Vamos a demostrar que para cualquier entrada de largo n, la máquina tendrá que hacer $\Omega(n^2)$ movimientos.

Será conveniente asumir que n es divisible por 3 (el argumento es muy similar en el caso general). Sea $k = n/3$. Restringimos las entradas a palabras en las cuales el tercio del medio son sólo ceros, es decir, a palabras de la forma: $x_1 \cdots x_k 0 \cdots 0 x_{2k+1} \cdots x_n$. (Si podemos mostrar que entre medio ya existen tales palabras, existe una para la cual la máquina debe trabajar $\Omega(n^2)$, estamos listos.) Fijemos cualquier j tal que $k \leq j \leq 2k$. Llamemos a la línea divisora entre los campos j y $j+1$ de la cinta el *corte* después de j.

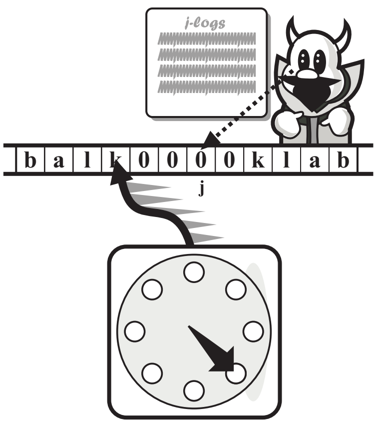

Figura 1.10: Punto de partida de la demostración

Imaginemos que tenemos un pequeño demonio sentado en esto y que está grabando el estado de la unidad central todo el tiempo que el cabezal pase entre estos campos.

Al final de la computación tendremos la secuencia $g_1 g_2 \cdots g_t$ de elementos de Γ (el largo t de la secuencia puede ser diferente para distintas entradas), el j-log de la entrada dada.

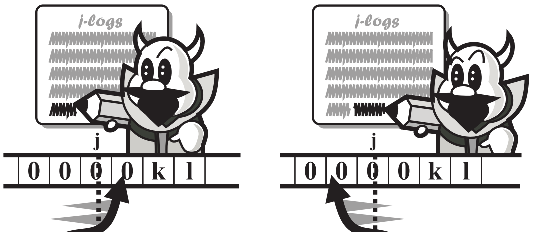

Figura 1.11: Demonio marcando el estado para formar el j-log

La clave para la demostración es el Lema 1 que demostramos más adelante. Usando este lema, es fácil probar que para una entrada de largo m dado, el número de diferentes j-logs de largo menor que m es a lo más

$$1 + |\Gamma| + |\Gamma|^2 + \cdots + |\Gamma|^{m-1} = \frac{|\Gamma|^m - 1}{|\Gamma| - 1} < 2|\Gamma|^{m-1}.$$

Esto es cierto para cualquier elección de j; por lo tanto, el número de palíndromes en que el j-log, para algún j, tiene largo menor que m es a lo más $2(k+1)|\Gamma|^{m-1}$. Existen 2^k palíndromes del tipo considerado, y también el número de palíndromes para los cuales j-log tiene largo al menos m para todo j es a lo menos

$$2^k - 2(k+1)|\Gamma|^{m-1}. \tag{1.2}$$

Así pues si eligiéramos m número positivo, entonces habría un palíndrome para el cual el j-log tendría largo de a lo menos m para todo j. Esto implica que los demonios registrarían por lo menos $(k+1)m$ movidas, para que la computación tome a lo menos $(k+1)(m+1)$ pasos. Es fácil verificar que la elección de $m = n/(6\log|\Gamma|)$ hace la expresión (1.2) positiva, y entonces hemos encontrado una entrada para la cual, la computación toma al menos $(k+1)m > n^2/(6\log|\Gamma|)$ pasos.

Lema 1 *Sea* $x = x_1 \cdots x_k 0 \cdots 0 x_k \cdots x_1$ *y* $y = y_1 \cdots y_k 0 \cdots 0 y_k \cdots y_1$ *dos palíndromes distintos y* $k \leq j \leq 2k$. *Entonces sus* j*-logs son distintos.*

Demostración. Supongamos que los j-logs de x e y son los mismos, digamos $g_1 \cdots g_t$. Considere la entrada $z = x_1 \cdots x_k 0 \cdots 0 y_k \cdots y_1$. Notar que en estas entradas, todos los x_i están ubicados a la izquierda del corte y todos los y_i están ubicados hacia la derecha. Mostraremos que la máquina concluirá que z es un palíndrome, lo cual es una contradicción.

¿Qué pasa cuando iniciamos la máquina con la entrada z? Por un momento, el cabezal se moverá en los campos a la izquierda del corte, y por lo tanto, la computación procederá exactamente como si estuviera con la entrada x. Cuando el cabezal alcance primero el campo $j+1$, entonces estará en el estado g_1 por el j-log de x. Después, el cabezal utilizará su tiempo mirando hacia la derecha del corte. Esta parte de la computación será idéntica a la parte correspondiente de la computación con la entrada y: comienza en el mismo estado que la parte correspondiente de la computación de y lo hizo, y lee los mismos caracteres de la cinta, hasta que el cabezal regresa al campo j de nuevo. Podemos seguir la computación en la entrada z similarmente, y ver que la porción de la computación durante sus m pasos que queda a la izquierda del corte es idéntica a la porción correspondiente de la computación con la entrada x, y que la porción de la computación durante sus m pasos que queda a la derecha del corte es idéntica a la porción correspondiente de la computación con la entrada y. Dado que la computación con la entrada x termina escribiendo un "1" en el campo inicial, la computación con la entrada z termina de la misma manera. Esto es una contradicción. ⊙ ⊙

1.5 La máquina de Acceso Aleatorio (RAM)

Cuando se diseñan máquinas de Turing para diversas tareas, uno nota que una máquina de Turing gasta una gran cantidad de tiempo en mover sus cabezales de lectura y escritura de un extremo de la cinta al otro. Es posible diseñar artificios para evitar algunos de estos movimientos, pero siguiendo esta línea nos alejaríamos cada vez más de los computadores reales, los cuales tienen una memoria de "acceso aleatorio", es decir, que pueden acceder a *cualquier* campo de su memoria en un solo paso. Por lo tanto, uno puede querer modificar la manera en que ha diseñado las máquinas de Turing con memoria de manera que puedan alcanzar una celda de memoria arbitraria en un solo paso.

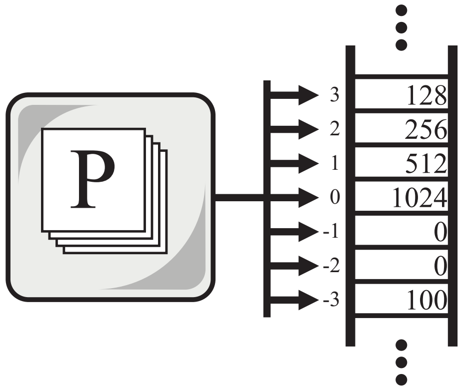

Figura 1.12: Máquina de acceso aleatorio, RAM

Por supuesto, la máquina debe saber cuáles celdas acceder y por lo tanto, tenemos que asignar direcciones a las celdas. Deseamos seguir considerando la memoria como una memoria no acotada; por lo tanto, debemos permitir cualquier entero como dirección válida. La dirección de la celda a acceder debe ser ella misma almacenada en alguna parte, por lo tanto, debemos permitir que cualquier entero sea almacenado en una celda (en vez de un solo símbolo de un alfabeto finito, como es el caso de una máquina de Turing). Finalmente, haremos el modelo más parecido a las máquinas de hoy en día al hacerlo programable (nótese que esto le da el carácter análogo a la máquina de Turing universal). De esta manera obtenemos la noción de una *Máquina de Acceso Aleatorio* o máquina RAM (de las siglas en inglés *Random Access Machine*). Ahora, seamos más precisos. La *memoria* de una Máquina de Acceso Aleatorio es una secuencia doblemente infinita $\ldots, x[-1], x[0], x[1], \ldots$ de registros de memoria. Cada registro puede almacenar un entero arbitrario. En cualquier momento dado, sólo una cantidad finita de los números almacenados en la memoria son distintos de 0. La memoria de programa es una secuencia infinita (en un solo sentido) de registros llamados *líneas*. Aquí escribimos un programa de algún largo finito en cierto lenguaje de programación similar al lenguaje *assembler* de las máquinas reales. Es suficiente, por ejemplo, permitir las siguientes declaraciones:

$$x[i] \leftarrow 0;$$
$$x[i] \leftarrow x[i] \pm 1;$$
$$x[i] \leftarrow x[i] \pm x[j];$$
$$x[i] \leftarrow x[x[j]];$$
$$x[x[i]] \leftarrow x[j];$$
$$\texttt{IF } x[i] \leq 0 \texttt{ THEN GOTO } p.$$

Aquí, i y j son direcciones de los registros de memoria (enteros arbitrarios), p es la dirección de alguna línea de programa (un número natural arbitrario). La instrucción penúltima garantiza la posibilidad de acceso inmediato. Con esto, la memoria se comporta como un arreglo en un lenguaje de programación convencional como C. El conjunto exacto de instrucciones básicas es importante sólo en la medida de que sea suficientemente simple para implementar, suficientemente expresivo para hacer posibles las computaciones deseadas, y de cardinalidad finita. Por ejemplo, es suficiente permitir los valores $-1, -2, -3$ para i, j. También podemos omitir las operaciones de suma y resta entre los elementos unitarios ya que puede ser escrito un programa para ellos. Por otro lado, también podemos incluir la multiplicación, etc. La *entrada* de una Máquina de Acceso Aleatorio es una secuencia finita de números naturales escritos en registros de memoria $x[0], x[1], \ldots$ La Máquina de Acceso Aleatorio lleva un programa finito arbitrario. Ella se detiene cuando llega a una línea del programa sin ninguna instrucción. La *salida* está definida como el contenido de los registros $x[i]$ después que el programa se detiene.

Es sencillo escribir subrutinas RAM para tareas simples que ocurren repetidamente en programas que resuelven cosas más complejas. Varias de éstas las dejamos para los ejercicios. Aquí discutiremos tres tareas que necesitaremos posteriormente en este capítulo.

Ejemplo (asignación de valores). Sean i y j dos enteros. Entonces, la asignación

$$x[i] \leftarrow j$$

puede ser realizada por el siguiente programa RAM si j es positivo:

$$x[i] \leftarrow 0;$$
$$x[i] \leftarrow x[i] + 1;$$
$$\vdots$$
$$x[i] \leftarrow x[i] + 1;$$

(donde la instrucción $x[i] \leftarrow x[i] + 1$ se ejecuta j veces).

Si j es negativo, sencillamente reemplazamos la instrucción $x[i] \leftarrow x[i] + 1$ por $x[i] \leftarrow x[i] - 1$ y la ejecutamos $|j|$ veces.

Ejemplo (Suma de una constante). Sean i y j dos enteros. Entonces, la declaración

$$x[i] \leftarrow x[i] + j$$

puede ser realizada de la misma manera que en el ejemplo anterior, sólo que omitiendo la primera línea.

Ejemplo (Ramificaciones múltiples). Sean $p_0, p_1, \ldots, p_r$ índices de líneas de programa y supóngase que conocemos que los contenidos de los registros i satisfacen la condición $0 \leq x[i] \leq r$. Entonces, la declaración

$$\texttt{GOTO}\ p_{x[i]}$$

puede ser realizada por el programa RAM

$$\begin{aligned}
&\texttt{IF}\ x[i] \leq 0\ \texttt{THEN GOTO}\ p_0;\\
&x[i] \leftarrow x[i] - 1;\\
&\texttt{IF}\ x[i] \leq 0\ \texttt{THEN GOTO}\ p_1;\\
&x[i] \leftarrow x[i] - 1;\\
&\vdots\\
&\texttt{IF}\ x[i] \leq 0\ \texttt{THEN GOTO}\ p_r;
\end{aligned}$$

Se debe tener cuidado cuando se incluye este último segmento de programa en otro programa, ya que modifica el contenido de $x[i]$. Si deseamos preservar el contenido de $x[i]$, pero tener un registro "borrador", digamos $x[-1]$, entonces podemos hacer

$$x[-1] \leftarrow x[i] - 1; \tag{1.3}$$
$$\texttt{IF}\ x[-1] \leq 0\ \text{THEN GOTO}\ p_0; \tag{1.4}$$
$$x[1] \leftarrow x[-1] - 1; \tag{1.5}$$
$$\texttt{IF}\ x[-1] \leq 0\ \texttt{THEN GOTO}\ p_1; \tag{1.6}$$
$$x[-1] \leftarrow x[-1] - 1; \tag{1.7}$$
$$\vdots \tag{1.8}$$
$$\texttt{IF}\ x[-1] \leq 0\ \texttt{THEN GOTO}\ p_r; \tag{1.9}$$

Si no tenemos un registro "borrador" tenemos que hacer espacio para uno (debido a que no necesitaremos esos detalles, los dejaremos para los ejercicios).

Ahora mostraremos que las RAM y las máquinas de Turing pueden computar esencialmente las mismas funciones y sus tiempos de ejecución no difieren notablemente. Consideremos (por simplicidad) una máquina de Turing de una cinta, con alfabeto $\{0, 1, 2\}$ donde 0 será el símbolo de espacio en blanco (nos desviamos de nuestra convención anterior, pero esto resulta más práctico aquí).

Cada entrada $x_1, \ldots, x_n$ de la máquina de Turing (que es una secuencia de símbolos 1 y 2) puede ser interpretado como una entrada de la RAM de dos maneras distintas: podemos escribir los números n, $x_1, \ldots, x_n$ en los registros $x[0], \ldots, x[n]$ o podemos asignar a la secuencia $x_1, \ldots, x_n$ un solo número natural reemplazando los 2 por 0 y anteponiendo un 1. La salida de la máquina de Turing puede ser interpretada similarmente a la salida de la RAM.

Estudiaremos la correspondencia precisa para la primera interpretación.

Teorema 4 *Para cada máquina de Turing (con múltiples cintas) sobre el alfabeto $\{0, 1, 2\}$, se puede construir un programa en la Máquina de Acceso Aleatorio con las siguientes propiedades. Esta computa para todas las entradas las mismas salidas que la máquina de Turing y si la máquina de Turing hace N pasos entonces la Máquina de Acceso Aleatorio hace $\mathcal{O}(N)$ pasos con números binarios de $\mathcal{O}(\log N)$ dígitos.*

DEMOSTRACIÓN. Sea $T = \langle 1, \{0, 1, 2\}, \Gamma, \alpha, \beta, \gamma \rangle$. Sea $\Gamma = \{1, \ldots, r\}$, donde 1 = START y r = STOP. Durante la simulación de la computación de la máquina de Turing, en el registro $2i$ de la RAM encontraremos el mismo número (0, 1 ó 2) que en la i-ésima celda de la máquina de Turing. El registro $x[1]$ almacenará dónde está el cabezal en la cinta, y el estado de la unidad de control será determinado según dónde estemos en el programa. Nuestro programa estará compuesto de trozos $Q_{i,j}$ los cuales simulan la acción de la máquina de Turing cuando se encuentra en el estado i y está leyendo el símbolo $j (1 \leq i \leq r-1, 0 \leq j \leq 2)$ y las líneas P_i saltan a $Q_{i,j}$ si la máquina de Turing se encuentra en el estado i y leyendo el símbolo j. Ambas son sencillas de construir. P_i es simplemente

$$\texttt{GOTO } Q_{i,x[1]}$$

para $1 \leq i \leq r$; la parte del programa P_r consiste de una sola línea de programa vacía. Las partes $Q_{i,j}$ del programa no son mucho más complicadas:

$$\begin{aligned}
&x[x[1]] \leftarrow \beta(i,j);\\
&x[1] \leftarrow x[1] + 2\gamma(i,j);\\
&\texttt{GOTO } P_{\alpha(i,j)};
\end{aligned}$$

El programa mismo se vería como sigue:

$$\begin{aligned}
&x[1] \leftarrow 0;\\
&P_1\\
&P_2\\
&\vdots\\
&P_r\\
&Q_{1,0}\\
&\vdots\\
&Q_{r-1,2}
\end{aligned}$$

Con esto, hemos descrito la simulación de la máquina de Turing por la RAM. Para analizar el número de pasos y el tamaño de los números utilizados, es suficiente notar que en N pasos, la máquina de Turing puede escribir en a lo más $\mathcal{O}(N)$ registros. Luego en cada paso de la máquina de Turing se trabaja con números de largo $\mathcal{O}(\log N)$. $\odot$

Otra interpretación de la entrada de la máquina de Turing es, como se mencionó más arriba, mirar la entrada como un solo número natural e ingresarlo a la RAM como tal. Entonces, este número a estará en el registro $x[0]$. En este caso, lo que podemos hacer es computar los dígitos de a con la ayuda de un programa sencillo. Escribir (borrando el 1 de la primera posición) en los registros $x[0], \ldots, x[n-1]$ y aplicar la construcción descrita en el Teorema 4.

Nota 1 *En la demostración del Teorema 4, no usamos la instrucción $x[i] \leftarrow x[i] + x[j]$; esta instrucción es necesaria cuando se computan los dígitos de la entrada. Incluso esto puede ser obtenido sin la operación de suma si descartamos la restricción en el número de pasos. Pero si permitimos números arbitrarios como entradas para la RAM, entonces, sin esta instrucción el tiempo de ejecución y el número de pasos obtenidos pueden ser exponenciales incluso para problemas muy sencillos. Por ejemplo consideremos el problema de sumar el contenido a del registro $x[1]$ con el contenido b del registro $x[0]$. Esto es fácil de llevar a cabo en la RAM en un número acotado de pasos. Pero si excluimos la instrucción $x[i] \leftarrow x[i] + x[j]$ entonces el tiempo requerido es al menos* $\min\{|a|, |b|\}$.

Tomando un programa dado para la RAM, podemos interpretar su entrada y su salida cada una como una palabra en $\{0, 1, -, \#\}^*$ (denotando todas las ocurrencias de enteros en binario, si se necesita, con signo y separados entre ellos por $\#$). En este sentido, el siguiente teorema se aplica.

Teorema 5 *Para cada programa de Máquina de Acceso Aleatorio existe una máquina de Turing que computa para cada entrada la misma salida.*

Si la Máquina de Acceso Aleatorio usa tiempo N entonces la máquina de Turing corre en $\mathcal{O}(N^2)$ pasos.

DEMOSTRACIÓN. Simularemos la computación de la RAM por una máquina de Turing de cuatro cintas. Escribimos en la primera cinta el contenido de los registros $x[i]$ (en binario, y con signo si es negativo). Podemos representar el contenido de todos los registros (representando, digamos, el contenido 0 por el símbolo "$*$"). Sin embargo, esto puede causar un problema debido a la característica de acceso inmediato ("aleatorio") de la RAM. Más exactamente, la RAM puede escribir incluso en el registro con número 2^N usando sólo un paso con un entero de N bits. Por supuesto, el contenido de la mayoría de los registros con índices menores seguirían siendo 0 durante toda la computación. No es útil mantener el contenido de éstos en la cinta dado que la cinta sería muy larga y tomaría tiempo exponencial al cabezal para dirigirse al lugar donde debe escribir. Por lo tanto, almacenaremos en la cinta de la máquina de Turing sólo el contenido de aquellos registros en los cuales la máquina generalmente escribe. Por supuesto, también debemos registrar el número del registro en cuestión.

Lo que haremos entonces es que cuando la RAM escriba un número en el registro $x[z]$, la máquina de Turing simule esto escribiendo la cadena $\#\#y\#z$ al final de su primera cinta (esto nunca reescribe esta cinta). Si la RAM lee el contenido de algún registro $x[z]$, entonces, en la primera cinta de la máquina de Turing, partiendo por atrás, el cabezal busca la primera cadena de la forma $\#\#u\#z$; este valor u muestra lo que fué escrito en el z-ésimo registro la última vez. Si no encuentra tal cadena entonces trata a $x[z]$ como 0.

Cada instrucción del "lenguaje de programación" de la RAM es fácil de simular por una máquina de Turing apropiada usando sólo las otras tres cintas. Nuestra máquina de Turing será una "supermáquina" en la cual un conjunto de estados corresponden a cada línea de programa. Estos estados generan una máquina de Turing que lleva la instrucción en cuestión y entonces trae los cabezales al fin de la primera cinta (a su última celda no vacía) y a la celda 0 de las otras cintas. El estado STOP de cada una de tales máquinas de Turing es identificado con el estado START de la máquina de Turing correspondiente a la próxima línea. (En caso de salto condicional, si $x[i] \leq 0$, la supermáquina se va al estado inicial de la máquina de Turing correspondiente a la línea p). El START de la máquina de Turing correspondiente a la línea 0 también es el START de la supermáquina. En vez de esto tenemos aún otro estado, STOP, que corresponde a la línea de programa vacía.

Es fácil de ver que la máquina de Turing así construida simula el trabajo de la RAM paso por paso. Esto conduce a que la mayoría de las líneas de programa se ejecuta en un número de pasos proporcional al número de dígitos de los números que ocurren en éste, es decir, al tiempo de ejecución que la RAM gasta en éste. La excepción es la lectura del siguiente, para la cual, posiblemente debe ser buscado en la cinta completa. Dado que el largo de la cinta es N, el número total de pasos es $\mathcal{O}(N^2)$. $\odot$

1.6 Otros modelos

Los modelos de computación están estrechamente ligados a la "definición" de algoritmo. Los lógicos formalizaron este concepto con diversas definiciones que capturan o aproximan el concepto de "procedimiento mecánico" en términos matemáticos. En este marco aparecieron casi simultáneamente, a partir de los años treinta del siglo XX, varias definiciones que intentaban capturar el concepto de algoritmo. A continuación mencionaremos algunas de las más importantes.

Funciones recursivas. Este es el intento más antiguo y el que contiene las ideas seminales sobre este tema. Es obra de los lógicos K. Gödel y J. Herbrand. Para simplificar la formulación, en vez de hablar de un problema como un conjunto infinito de instancias, se codifica cada instancia por un número natural,[4] y entonces un problema será un conjunto de números naturales (cada número corresponderá a una instancia del problema). Entonces, un procedimiento de decisión es una función "calculable" $f : \mathbb{N} \longrightarrow \{\text{Sí, No}\}$, que a cada número (problema) le asigna Sí o No.

A partir de esto, la intuición es la siguiente: cualquier función que sea "calculable" debe serlo debido a

[4] Por ejemplo una codificación como la de Gödel. Las oraciones son sucesiones de palabras, las palabras son sucesiones de símbolos (letras, espacio en blanco, puntuación). Si le asignamos un número a cada símbolo, no es difícil asignarle números únicos a sucesiones de símbolos, etc.

un proceso parecido a las definiciones por inducción matemática. Por ejemplo las ecuaciones $m(x,1) = x$ y $m(x,y+1) = m(x,y)+x$ definen inductivamente la multiplicación a partir de la suma. No es difícil ver que todas las funciones definibles por ecuaciones de este tipo son calculables mecánicamente (por ejemplo usando las ecuaciones del ejemplo; calculamos $m(2,3) = m(2,2)+2$; luego $m(2,2) = m(2,1)+2$, y finalmente $m(2,1) = 2$).

Las *funciones recursivas* se obtienen generalizando estas ideas al permitir usar parámetros y ecuaciones funcionales implícitas.

Cálculo λ. Este es el enfoque desde el punto de vista de los lenguajes funcionales y de la teoría de tipos, y fué inventado por A. Church siguiendo ideas de Shönfinkel y Curry.

Toda función calculable se puede expresar como una composición sucesiva de funciones más simples. Para formalizar esto, necesitamos desarrollar un sistema simbólico para expresar funciones y su composición. Esto es precisamente el *cálculo lambda* (cálculo λ). Las ideas básicas son: (1) Ua función consta de una entrada y una salida, y necesitamos una notación que indique esto explícitamente. Church eligió $\lambda x.f(x)$, donde la primera x indica el parámetro de entrada, lo que viene después del punto es la expresión a ser evaluada (la salida). (2) Estudiar la mecánica de la composición y evaluacion de funciones; Por ejemplo, ¿cómo se evalúa $(f \circ g)(5)$, donde $f(n) = n^3$ y $g(x) = 2n$? Si queremos analizar procesos mecánicos de cálculo, hay muchas soluciones. Por ejemplo, ¿evaluamos $2n = 10$ y luego multiplicamos $10\times10\times10$?, o ¿evaluamos $g(n)^3 = 2n \times 2n \times 2n = 8n^3$ y luego 5^3, etc.?

Es posible acercarse a este enfoque desde un punto de vista algo más abstracto (pero clave en lenguajes de computación): nótese la odiosa diferencia que hacemos en general entre una función y los elementos a los que se aplica. Si les damos a las funciones y a sus argumentos los mismos "derechos de ciudadanía" tenemos una teoría general de funciones. Así podemos hablar de funciones aplicándose sobre funciones, etc. Si se llevan estas ideas a su límite, se obtiene lo que se conoce como *cálculo λ*.

Algoritmos de Markov. Es posible ver un cómputo como la aplicación sucesiva de reglas. Este es el enfoque desde el punto de vista de los lenguajes y las gramáticas formales. Fué introducido por Markov en los años cuarenta.

Un proceso de cómputo mecánico puede verse como un proceso de transformaciones simbólicas $A_1 \cdots A_n \longrightarrow B_1 \cdots B_m$, donde hay muchas reglas como ésta. Un número finito de ellas determina un algoritmo; si uno agrega ciertas condiciones de normalización se obtienen los *algoritmos de Markov*. Este ha sido un enfoque muy productivo, especialmente en el estudio de gramáticas formales y reescritura.

Sistemas de Post. Tempranamente ciertas personas se dieron cuenta que la computación no tenía que ver sólo con números. (Hagamos notar que incluso Turing presentó su primera máquina como calculando números.) El lógico E. Post introdujo ciertos sistemas en la década del cuarenta, que contienen un fuerte componente simbólico, es decir, calculan con símbolos. En ciertos aspectos, es bastante similar en espíritu al anterior. Es muy útil para codificar problemas insolubles.

Los sistemas anteriores representan diversos puntos de vista, tanto desde el punto de vista de los datos que manejan, como de los procesos en que están basado. ¿Cómo compararlos? Es posible hacerlo haciendo una correspondencia de números con palabras y viceversa, y notando que en el fondo todos pueden considerarse como funciones con ciertas entradas y salidas. Usando estas técnicas es posible demostrar:

Teorema 6 (Gödel, Turing, Church, Post, Markov, etc.) *Si codificamos adecuadamente las entradas y salidas, cada uno de los sistemas anteriores define exactamente la misma clase de funciones.*

Es decir, las definiciones son equivalentes. Esta es una gran sorpresa, considerando la diversidad de puntos de vista que representan. Las funciones de números que calculan estos sistemas se llaman *recursivas* y todas son calculables por algoritmos. ¿Hay algoritmos fuera de los representados por estas funciones? Obsérvese que de nuevo tenemos el mismo problema: ¿que se entiende por algoritmo? Respondiendo a esta pregunta, A. Church y A. Turing presentaron en forma independiente y casi paralelamente la siguiente conjetura:

Tesis de Church. *Las funciones de números susceptibles de ser calculadas por algoritmos son* precisamente *aquellas definidas por el teorema anterior.*

Nótese el carácter de la *tesis de Church*: Una afirmación que no es posible demostrar ni refutar (pues no tenemos una definición formal de algoritmo). Aún a pesar de esto, la tesis rápidamente obtuvo aprobación general debido a la abrumadora cantidad de evidencia en su favor (en particular el teorema antes mencionado).[5]

1.7 Ejercicios

1. Construya una máquina de Turing que calcule las siguientes funciones:

   (a) $x_1 \cdots x_m \mapsto x_m \cdots x_1$.

   (b) $x_1 \cdots x_m \mapsto x_1 \cdots x_m x_1 \cdots x_m$.

   (c) $x_1 \cdots x_m \mapsto x_1 x_1 \cdots x_m x_m$.

   (d) Para cada entrada de largo m que consiste sólo en 1's, la forma binaria de m; para todas las otras entradas, nunca se detiene.

2. Asuma que tenemos dos máquinas de Turing, calculando las funciones $f : \Sigma_0^* \longrightarrow \Sigma_0^*$ y $g : \Sigma_0^* \longrightarrow \Sigma_0^*$. Construya una máquina de Turing que calcule la función $f \circ g$.

3. Construya una máquina de Turing que realice $2^{|x|}$ pasos para cada entrada x.

4. Construya una máquina de Turing que con la entrada x, se detenga en un número finitos pasos si y sólo si el símbolo 0 aparece en x.

5. Escriba una simulación de una máquina de Turing con cinta infinita para ambos lados por una maquina Turing con una cinta que es infinita en una sola dirección.

6. Muestre que si simulamos una máquina con k cintas en la máquina de Turing universal de $k+1$ cintas, entonces para una entrada arbitraria, el número de pasos aumenta sólo por un factor multiplicativo proporcional al largo del programa simulado.

7. Sean T y S dos máquinas de Turing con una cinta. Decimos que T simula el trabajo de S por un programa p (aquí $p \in \Sigma_0^*$) si para todas las palabras $x \in \Sigma_0^*$, la máquina T se detiene en las entradas $p * x$ en un número finito de pasos si y sólo si S se detiene en la entrada x, y al momento de parar, la cinta de T y la cinta de S tienen el mismo contenido.

   Demuestre que existe una máquina de Turing T de una cinta que puede simular el trabajo de cualquier otra máquina de Turing de una cinta en este sentido.

8. En la simulación de máquinas de k cintas por máquinas de una cinta presentada en el texto, el control finito de la máquina simuladora T es un poco más grande que la de la máquina simulada S: más aún, el número de estados de la máquina simuladora depende de k. Demuestre que esto no es necesario: existe una máquina de una cinta que puede simular cualquier máquina de k cintas.

9. (*) Muestre que toda máquina de Turing de k cintas puede ser simulada por una de dos cintas de manera que si en alguna entrada, la máquina de k cintas hace N pasos, entonces la de dos cintas hace a lo más $\mathcal{O}(N \log N)$ pasos. (Ayuda: en vez de mo-

[5] En la época que se formuló la tesis, el lógico húngaro Kalmar fué unos de los pocos que se opuso a aceptarla debido a consideraciones lógicas.

Es bueno hacer notar que detrás de la tesis de Church hay una serie de consideraciones de carácter *físico*: determinismo, finitud, comunicación de información, etc. No es sorprendente por ello que cuando la validez de la universalidad de estos conceptos se pone en duda, aparezcan nuevos modelos de computación. Nótese que la computación digital moderna es la prueba más fehaciente de que la tesis de Church domina el ámbito de la computación.

ver los cabezales simulados, mueva las cintas simuladas (Hennie-Stearns)).

10. Cinta bidimensional.

    (a) Defina la noción de máquina de Turing con cinta bidimensional.
    (b) Muestre que una máquina de Turing de dos cintas puede simular una máquina de Turing con cinta bidimensional.
    (c) Estime la eficiencia de la simulación anterior.

11. Sea $f : \Sigma_0^* \longrightarrow \Sigma_0^*$ una función. Una máquina de Turing *online* contiene, aparte de las cintas usuales, dos cintas extras. La cinta de entrada es legible sólo en una dirección, la cinta de salida puede ser escrita sólo en una dirección. Una máquina de Turing online T computa la función f si en una sola corrida, para cada n, después de haber recibido n símbolos $x_1, \ldots, x_n$, escribe $f(x_1, \ldots, x_n)$ en la cinta de salida, terminada con un blanco.

    Encuentre un problema que puede ser resuelto más eficientemente en una máquina de Turing online con cinta de trabajo bidimensional que con una cinta de trabajo unidimensional. (Ayuda: en una cinta bidimensional, cualquiera de los n bits puede ser accesados en $\sqrt{n}$ pasos. Para aprovechar esto, deje a la entrada representar una secuencia de operaciones en una "base de datos": inserciones y consultas, y defina f como la interpretación de estas operaciones.)

12. Cinta de árbol.

    (a) Defina la noción de una máquina de Turing con una cinta de árbol.
    (b) Muestre que una máquina de Turing con dos cintas puede simular una máquina de Turing con cintas de árbol. (Ayuda: Almacene en la cinta 1 cada símbolo de la cinta bidimensional, un número arbitrario que identifique su posición original y los números que identifican estos padres e hijos.)
    (c) Estime la eficiencia de la simulación anterior.
    (d) Encuentre un problema que pueda ser resulto más eficientemente con una cinta de árbol que con cualquier cinta de dimensión finita.

13. Escriba un programa para la RAM que para un número positivo dado a

    (a) determine el número m más grande con $2^m \leq a$;
    (b) compute su representación en base 2.

14. Sea $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ un polinomio con coeficientes enteros $a_0, \ldots, a_n$. Escriba un programa RAM que compute los coeficientes del polinomio $(p(x))^2$ a partir de aquellos de $p(x)$. Estime el tiempo de ejecución de su programa en términos de n y $K = \max\{|a_0|, \ldots, |a_n|\}$.

15. Demuestre que si a una RAM no se le permite utilizar la instrucción $x[i] \leftarrow x[i] + x[j]$, entonces sumar el contenido a de $x[1]$ al contenido de b de $x[2]$ toma al menos $\min\{|a|, |b|\}$ pasos.

16. Dado que la RAM es una sóla máquina, el problema de la universalidad no puede ser tomado exactamente de la misma manera que para las máquinas de Turing: en

cierto sentido esta máquina RAM es universal. Por lo tanto, la siguiente propiedad de "auto-simulación" de la RAM se acerca más. Para un programa RAM p y una entrada x, sea $R(p, x)$ la salida de la RAM. Sea $\langle p, x\rangle$ la entrada de la RAM que obtenemos escribiendo los símbolos de p uno a uno en los registros $1, 2, \ldots$ seguido por un símbolo # y entonces por registros conteniendo la secuencia original x. Demuestre que existe un programa RAM u tal que para todos los programas RAM p y entradas x se cumple que $R(u, \langle p, x\rangle) = R(p, x)$.

17. (Máquina puntero). Después de haber visto cintas dimensionalmente finitas y una cinta de árbol podríamos buscar considerar una máquina con un grafo dirigido más general como su medio de almacenamiento. Cada celda c tiene un número fijo de aristas numeradas $1, \ldots, r$. Cuando el cabezal explora una cierta celda se puede mover a cualquiera de las celdas $\lambda(c, i)(i = 1, \ldots, r)$ alcanzables desde ella a través de las aristas salientes. Dado que parece imposible concordar en el mejor grafo, introducimos un nuevo tipo de operación elemental: cambiar la estructura de almacenamiento del grafo localmente, alrededor de el cabezal explorada. Transformaciones arbitrarias pueden ser logradas aplicando repetidamente las siguientes tres operaciones (e ignorando los nodos que están inaccesibles): $\lambda(c, i) \leftarrow$ New, donde New es un nuevo nodo; $\lambda(c, i) \leftarrow \lambda(\lambda(c, j))$ y $\lambda(\lambda(c, i)) \leftarrow \lambda(c, j)$. Una máquina con esta estructura de almacenamiento y estas tres operaciones además de las usuales operaciones de las máquinas de Turing será llamada una *Máquina Puntero*. Llamemos RAMá la RAM en que las operaciones de suma y resta son omitidas; sólo la operación $x[i] \leftarrow x[i] + 1$ es permitida. Demuestre que la Máquina Puntero es equivalente a la RAM', en el siguiente sentido. Para cada Máquina Puntero existe un programa RAM'que computa para cada entrada la misma salida. Si la Máquina Puntero corre en tiempo N entonces la RAM'corre en $\mathcal{O}(N)$ pasos. Para cada programa RAMéxiste una Máquina Puntero que computa para cada entrada la misma salida. Si el programa RAM'corre en tiempo N entonces la Máquina Puntero corre en $\mathcal{O}(N)$ pasos.

Fíjese en lo que la Nota 1 dice para esta simulación.

Capítulo 2

Computación con recursos limitados

2.1 Problemas e instancias

Para formalizar los conceptos de complejidad, es necesario antes formalizar el concepto de "problema".

> **Problema.** Cuestión que se trata de aclarar; Proposición dudosa. || *Mat.* Proposición dirigida a averiguar el modo de obtener un resultado cuando ciertos datos son conocidos.

Pero hay problemas y problemas, particulares y generales. Los que aquí nos conciernen son problemas genéricos, aquellos que dependen de parámetros, y para los cuales queremos soluciones uniformes, es decir, que sirvan para todos los parámetros de entrada. Así, no nos interesarán problemas del tipo: "encontrar la factorización del número 125.675.441", o "encontrar el grafo dual del grafo de Petersen". Nuestro objeto de estudio serán problemas genéricos del tipo: "encontrar la factorización de un número", "dado un sistema de ecuaciones lineales, determinar si tiene solución". Obsérvese que estos dos últimos problemas no son *un* problema, sino un conjunto infinito de problemas, uno por cada número natural en el primer caso, y uno por cada conjunto finito de números reales en el segundo (los coeficientes de las ecuaciones).

Luego para nosotros un *problema* será una *proposición lógica* junto a ciertos *datos conocidos.* Nos interesará sobre todo ser muy precisos en el *planteamiento* del problema (el cual generalmente identificaremos con el problema mismo). Este planteamiento constará de:

1. *Entrada* (o instancia). El tipo de dato que necesitamos ingresar.
2. *Respuesta* (o salida). El tipo de dato que queremos obtener como respuesta.

Veamos algunos ejemplos, cuyo plantemiento nos servirá de modelo para el resto de estos apuntes. Partamos por el conocido problema que consiste en ordenar una lista de números:

Problema 1 *(Ordenar una lista)*
ENTRADA. *Una lista finita de números enteros.*
RESPUESTA. *La lista ordenada de menor a mayor.*

El segundo es un conocido problema de aritmética:

Problema 2 *(Adición)*
ENTRADA. *Un par a, b de números naturales.*
RESPUESTA. *Un número natural c tal que $c = a + b$.*

Problema 3 *(Primalidad)*
ENTRADA. *Un número natural.*
RESPUESTA. *Sí o No, dependiendo si el número tiene factores no triviales.*

Nótese que para todos los efectos prácticos, vía una adecuada codificación cada entrada y cada salida puede codificarse por palabras de un alfabeto Σ_0. Consideremos entonces un alfabeto finito (que incluya el símbolo en blanco $*$) y definamos $\Sigma_0 = \Sigma \setminus \{*\}$. En este sentido, definiremos un *problema* como un conjunto P de pares ordenados (I, A) de palabras en Σ_0^* donde I se llama la *instancia*, A se llama una *respuesta* para esa instancia, y cada palabra de Σ_0^* aparece como primera componente en *al menos* un par. (Esta última condición es necesaria por razones técnicas que veremos más adelante.)

dib. que muestre instancias

Esta definición es suficientemente general para abarcar la mayoría de las necesidades. Para enfatizar esta generalidad, muchas veces estos problemas son llamados *problemas de búsqueda.* Sin embargo, para estudiar la complejidad, es conveniente restringirse a problemas cuya respuesta (segunda componente del par) es SI o NO (o "1" o "0", según la codificación). Estos problemas son llamados problemas de *decisión.* Un ejemplo es *Primalidad* que vimos más arriba. Así, un *problema de decisión* puede considerarse *un* lenguaje: el conjunto de palabras x para las cuales (x, SI) están en el problema. En este sentido hablaremos indistintamente de *problema* (de decisión) o de *lenguaje.*

Nota 2 *La asimetría en la definición de los* problema de decisión *(el lenguaje está constituído por las palabras x cuya segunda componente es SI; ¿porque no elegimos NO?) jugará un importante rol en el futuro.*

2.2 Solubilidad algorítmica versus solubilidad práctica

En estas notas asumimos que el lector conoce la diferencia que existe para un problema entre ser *soluble* y *no ser soluble* algorítmicamente, es decir, por medio de computadores. Informalmente, un problema (de decisión) es *soluble algorítmicamente*[1] si existe un método, procedimiento, algoritmo, por el cual pudiéramos responder afirmativa o negativamente a cada una de las instancias del problema. En este sentido, la primera gran sorpresa de la computación fué descubrir que existían problemas que *no* se pueden resolver por computadores. (Para aquellos que quieran refrescar estos conceptos, sugerimos leer el Apéndice). Por supuesto que nosotros nos restringimos a aquellos problemas *solubles* algorítmicamente.

Pero la *solubilidad algorítmica* de un problema puede estar muy lejos de su solubilidad *práctica.* Hay problemas solubles algorítmicamente que no pueden ser resueltos, para una entrada de tamaño dado, en menos de un número exponencial o doble exponencial de pasos. La *teoría complejidad computacional*, una rama mayor de la teoría de algoritmos, investiga la solubilidad de problemas particulares bajo ciertas restricciones de recursos. Los recursos considerados más importantes son tiempo y espacio (almacenamiento).

Definiremos estas nociones en términos del modelo de computación de la máquina de Turing. Esta definición es adecuada para el estudio teórico; para describir algoritmos el modelo RAM es más conveniente y es una mejor aproximación a la realidad. Veremos, sin embargo, que desde el punto de vista de las restricciones de

dib. que muestre recursos mas import. que son tiempo y espacio

[1] Cuando el problema es de decisión, en vez de "soluble algorítmicamente" generalmente se dice *decidible.*

recursos más importantes (por ejemplo tiempo y espacio limitados por una función polinomial) no importa que modelo de máquina es utilizado en la definición.

Esto nos lleva a la definición de varias *clases de complejidad*: clases de problemas solubles en determinado tiempo fijo, parametrizado por el largo de la entrada. Toda función positiva que dependa de la entrada define una clase, pero algunas de ellas son particularmente importantes. La clase de complejidad más importante es la de *tiempo polinomial.* Muchos algoritmos importantes en la práctica corren en un tiempo limitado por un una expresión polinomial en el tamaño de la entrada (en breve, "son polinomiales").

Los algoritmos polinomiales son a menudo muy interesantes matemáticamente, debido a que son construidos con una profunda visión de de la estructura matemática de los problemas, y a menudo usan potentes herramientas matemáticas.

Restringiremos las computaciones a problemas de decisión (es decir, con respuesta sí o no). Esta no es una restricción muy limitante y simplifica bastante la representación. Notemos que la tarea de computar cualquier salida puede ser reducida a computar sus bits en alguna representación binaria razonable.

2.3 Tiempo y espacio

En este capítulo, cuando una máquina de Turing es utilizada en alguna computación, asumiremos que tiene una cinta de entrada, una de salida y $k \geq 1$ cintas disponibles. Al inicio, hay una palabra en Σ_0^* escrita en la cinta de entrada.

La *demanda de tiempo* de una máquina de Turing T es una función $\mathrm{time}_T(n)$ definida como el máximo número de pasos que toma T para todas las posibles entradas de largo n. Asumiremos que $\mathrm{time}_T(n) \geq n$, es decir que la máquina debe leer la entrada; esto no tiene por que ser así necesariamente, pero con este supuesto exclυímos sólo casos triviales. Podría suceder que $\mathrm{time}_T(n) = \infty$.

Similarmente, la función $\mathrm{space}_T(n)$ está definida como el número máximo, sobre todas las entradas de largo n, de celdas sobre las que la máquina escribe. De esta manera, las celdas ocupadas por la entrada no se cuentan dentro de los requerimientos de tiempo. Asumiremos que $\mathrm{space}_T(n) \geq 1$, es decir, la máquina debe entregar alguna salida.

Una máquina de Turing será llamada *de tiempo polinomial*, si hay una función polinomial $f(n)$ para la cual $\mathrm{time}_T(n) \in \mathcal{O}(f(n))$. Esto es equivalente a decir que hay una constante c para la cual la demanda de tiempo de T está en $\mathcal{O}(n^c)$. En otras palabras, una máquina de Turing T es polinomial si para toda entrada de tamaño n, entrega la respuesta en un número de pasos no superior a $f(n)$, donde f es algún polinomio. Similarmente, podemos definir una máquina de Turing *exponencial* (en este caso la demanda de tiempo es $\mathcal{O}(2^{n^c})$ para algún $c \geq 0$). Análogamente, también podemos definir máquinas de Turing que trabajan en *espacio* polinomial y en *espacio* exponencial.

Recordemos que una máquina de Turing T *decide* un lenguaje $\mathcal{L} \subseteq \Sigma^*$ si para cada entrada $x \in \Sigma^*$ se detiene y responde 1 si $x \in \mathcal{L}$ y 0 si $x \notin \mathcal{L}$. Ahora estamos en condiciones de definir la complejidad de un problema. Diremos que un lenguaje $\mathcal{L} \subseteq \Sigma_0^*$ tiene *complejidad en tiempo a lo más* $f(n)$, si existe una máquina de Turing con demanda de tiempo a lo más $f(n)$ que lo decide. Esto es, si existe una máquina de Turing que para cada palabra x, responde en tiempo no mayor a $f(|x|)$ con 1 si $x \in \mathcal{L}$ o con 0 si $x \notin \mathcal{L}$.

L	Espacio Logarítmico
P	Tiempo polinomial
PSPACE	Espacio polinomial
EXPTIME	Tiempo exponencial

Tabla 2.1: Algunas clases determinísticas

Gran parte de la teoría de complejidad computacional consiste en "clasificar" los lenguajes (i.e. los problemas) de acuerdo a su complejidad (ya sea en tiempo, espacio, etc.). En este sentido, es conveniente agrupar todos los lenguajes "similares" desde este punto de vista en clases. Denotaremos por DTIME($f(n)$) la clase de los lenguajes que tienen complejidad en tiempo a lo más $f(n)$. (La letra "D" indica que sólo consideramos algoritmos determinísticos; más adelante consideraremos algoritmos que son "no-determinísticos" o que usan aleatoriedad). Denotaremos por PTIME, o simplemente por P, la clase de lenguajes decidibles por una máquina de Turing de tiempo polinomial. Similarmente cuando un lenguaje tiene complejidad en espacio a lo más $f(n)$, diremos que pertenece a la clase DSPACE($f(n)$). Así por ejemplo, PSPACE es la clase de los lenguajes de espacio polinomial.

dib. de clases

Nota 3 *Podría ser tentador definir la complejidad en tiempo de un lenguaje $\mathcal{L}$ como el tiempo óptimo que toma una máquina de Turing T que decide el lenguaje. Note que fuimos más cuidadosos anteriormente y solamente definimos cuando la complejidad en tiempo era a lo más $f(n)$. La razón es que podría no haber un mejor algoritmo (máquina de Turing) que resuelva un problema dado: algunos algoritmos funcionarán mejor para instancias pequeñas, otros para largas, u otros para las todavía más largas etc. La sección ?? contiene un teorema que nos presenta tales ejemplos.*

Nota 4 *Cuando decimos que la multiplicación entre dos números de largo n puede obtenerse en tiempo n^2, entonces hemos encontrado una cota superior a la complejidad de una* función *(multiplicación de dos números representados por las cadenas de entrada) en lugar de un* lenguaje. *Las clases* DTIME($f(n)$), DSPACE($f(n)$), *etc. están definidas como clases de* lenguajes; *las clases correspondientes para* funciones *también pueden ser definidas.*

A veces es trivial dar cotas inferiores a la complejidad de una función. Consideremos por ejemplo la función $x \times y$ donde x e y son números en notación binaria. Su computación requiere a lo menos $|x| + |y|$ pasos, debido a que este es el largo de la salida. Cotas inferiores sobre la complejidad de lenguajes nunca son así de triviales, puesto que la salida de un problema de decisión es un solo bit.

2.3.1 Tiempo versus espacio

En los capítulos que siguen demostraremos teoremas generales respecto a las medidas de complejidad y relaciones entre las clases. Mostraremos que hay lenguajes que requieren una gran cantidad de tiempo para decidirlos. Teoremas análogos pueden probarse para complejidad de espacio. Es natural preguntarse acerca de la relación de estas dos medidas de complejidad.

Hay una variedad de preguntas naturales e interesantes acerca del compromiso entre espacio y tiempo. Mencionemos primero que todos los problemas prácticos bien conocidos que corren en la mayoría de los computadores pueden ser acelerados significativamente agregando memoria. La relación aquí no es realmente entre complejidad de almacenamiento y tiempo de computación, sino sólo entre memoria más lenta y más rápida. Posiblemente, entre "memoria RAM" versus memoria en disco, la que es cercana al acceso en serie del modelo de las máquinas de Turing.

Pero existen algunos ejemplos reales de compromiso espacio-tiempo en la práctica. Supongamos que durante una computación los valores de una función boolenana pequeña pero compleja sean usados repetidamente. Entonces, en una máquina de acceso aleatorio vale la pena calcular estos valores una vez al comienzo para todas los entradas y utilizar posteriormente una tabla de accesos. Similarmente, si cierto campo de nuestros registros en una base de datos es utilizado frecuentemente, entonces vale la pena computar una tabla facilitando este tipo de búsqueda. Todos estos ejemplos caen en la siguiente categoría. Conocemos algún problema P y un algoritmo A que lo resuelve. Otro algoritmo $A\prime$ se sabe que también resuelve P en menos tiempo y mayor espacio que A. Pero, generalmente, no tenemos ninguna demostración de que con menor cantidad de tiempo efectivamente mayor espacio es necesario para resolver P. Además, cuando se conoce una cota inferior para complejidad tiempo de alguna función generalmente no tenemos una estimación de la comlejidad de espacio más que la trivial mencionada arriba (y vice versa).

dib. sobre compromiso memoria-tiempo

Pero hay resultados generales que se pueden demostrar, como los siguientes. Es trivial que el almacenamiento demandado por una máquina de Turing de k cintas es a lo más k veces su demanda de tiempo (ya que en un paso no se pueden escribir más de k). Por lo tanto, si tenemos $\mathcal{L} \in \mathrm{DTIME}(f)$ para un lenguaje, entonces existe una constante c (dependiendo del lenguaje) tal que $\mathcal{L} \in \mathrm{DSPACE}(cf)$, es decir, si se permite extender el alfabeto y $f(n) < n$ entonces $\mathrm{DSPACE}(cf) = \mathrm{DSPACE}(f)$ y por lo tanto

$$\mathrm{DTIME}(f) \subseteq \mathrm{DSPACE}(f).$$

Por otro lado, el tiempo requerido no es mayor que una función exponencial del espacio requerido (ya que exactamente la misma configuración, tomando en cuenta también las posiciones de los cabezales y el estado de la unidad de control, no puede ocurrir más de una vez sin entrar en un ciclo). Calculando más precisamente, el número de configuraciones diferentes de memoria es a lo más $c \cdot f(n)^d m^{f(n)}$, donde m es el tamaño del alfabeto y c, d son constantes.

Puesto que de acuerdo a lo anterior, la complejidad en tiempo de un lenguaje no depende de un factor constante, y en esta cota superior los números c, d, m son constantes, se sigue que si $f(n) > \log n$ y $\mathcal{L} \in \mathrm{DSPACE}(f(n))$ entonces $\mathcal{L} \in \mathrm{DTIME}((m+1)^{f(n)})$, esto es para alguna constante c,

$$\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DTIME}(c^{f(n)}).$$

2.3.2 Medidas para RAM

¿Como definir la demanda de tiempo en una máquina RAM? El número de pasos de la Máquina de Acceso Aleatorio no es la mejor medida de "el tiempo que toma trabajando". Uno podría (mal) usar el hecho de que las instrucciones operan sobre números naturales de tamaño arbitrario, y utilizar trucos computacionales que funcionen en ese modelo computacional, pero que utilicen números tan grandes, que se transformen en computaciones prácticamente imposibles. Por ejemplo, podemos simular suma de vectores sumando dos números naturales muy grandes.

Por lo tanto, preferimos caracterizar el tiempo de trabajo de algoritmos en RAM como dos números, y decimos que "la máquina toma a lo más n pasos con números de a lo más k bits". Similarmente, los requerimientos de espacio se caracterizan diciendo "La máquina almacena a lo más n números de a lo más k bits".

Si quisiéramos un solo número para caracterizar el tiempo de trabajo de un algoritmo RAM, podemos considerar como unidad de tiempo, no cada paso sino que el número de bits que ocurren en cada paso. (tanto registros de direcciones como datos contenidos en ellos). Como el número de bits de un número es esencialmente el logaritmo en base dos de su valor absoluto, usualmente se llama a este modelo *costo RAM logarítmico.*

En algoritmos aritméticos y algebraicos, frecuentemente conviene contar las operaciones aritméticas; en una Máquina de Acceso Aleatorio, esto corresponde a extender el set de operaciones básicas del lenguaje de programación, incluyendo la resta, multiplicación, división (con resto) y comparación de enteros. Y contar el número de operaciones (en términos del largo de la entrada) sobre números con a lo más un polinomial número de dígitos, por lo tanto, nuestro algoritmo sería polinomial en el modelo de costo logarítmico.

2.4 Consideraciones técnicas sobre las medidas

Por que la notación asintótica. El lector habrá notado que desde los comienzos, hemos preferido hablar de cotas *asintóticas* (notación $\mathcal{O}$, etc.) en vez de cotas precisas. Esto tiene sus razones. De hecho, no parece posible hacer una teoría elegante si consideramos las constantes multiplicativas y aditivas cuando hablamos de los recursos de tiempo y espacio. Veremos también que *no cualquier función* sirve como cota.

El siguiente teorema muestra porque las constantes no jugarán rol en nuestra teoría.

Teorema 7 (Aceleramiento lineal) *Para toda máquina de Turing T y $c > 0$ existe una máquina de Turing S sobre el mismo alfabeto que decide el mismo lenguaje y para la cual* $\mathrm{time}_S(n) \leqslant c \cdot \mathrm{time}_T(n) + n$.

Demostración. Por simplicidad, asumamos que T tiene sólo una cinta de trabajo (la demostración sería similar para k cintas) y que $c = 1/p$, donde p es un entero.

Sea S la máquina de Turing con una cinta de entrada, $2p-1$ cintas de "partida" y $2p-1$ cintas de trabajo. Numeremos estas cintas desde $1-p$ hasta $p-1$. Definamos el *índice* de la celda j de la cinta (de partida o de trabajo) i como el número $j(2p-1)+i$. La celda de partida (resp. de trabajo) con índice t va a corresponder a la celda t en la cinta de entrada (resp. de trabajo) de la máquina T. La máquina S tendrá además una cinta de salida.

La máquina S comienza su trabajo copiando cada letra de la entrada x desde su cinta de entrada a la celda con el índice correspondiente en sus cintas de partida, luego mueve cada cabezal de vuelta a la celda 0. De ahí en adelante ignora la cinta de entrada "real".

Cada paso sucesivo de la máquina S va a corresponder a p pasos consecutivos en la máquina T. Luego de pk pasos de la máquina T, los cabezales de la cinta de entrada y la cinta de trabajo estarán en las celdas t y s respectivamente. Vamos a planificar la máquina S de manera tal que en este caso, cada celda de cada cinta de partida (resp. de trabajo) de S mantenga el mismo símbolo que la correspondiente celda de la correspondiente cinta de T, y los cabezales descansan en las celdas de la cinta de partida con índices $t-p+1, \ldots, t+p-1$ y las celdas de la cinta de trabajo con índices $s-p+1, \ldots, s+p-1$. Asumimos que la unidad de control de la máquina S "sabe" también qué cabezal explora la celda corresponde a t (resp s). Conoce además cual es el estado de la unidad de control de T.

Debido a que la unidad de control de S ve no sólo lo que es leído por la unidad de control de T en el momento actual en su cintas de entrada y de trabajo, sino que además las celdas a una distancia a lo más $p-1$ de éstas, puede computar hacia dónde se moverán y qué escribirán los cabezales de T en los siguientes p pasos. Luego de p pasos los cabezales de T estarán en las posiciones, digamos $t+i$ y $s+j$ (donde $i, j > 0$). Obviamente, $i, j < p$. Notemos que mientras tanto el cabezal de "trabajo" puede cambiar los símbolos escritos en la cinta de trabajo sólo en el intervalo $[s-p+1, s+p-1]$.

Hagamos ahora que la unidad de control de S realice lo siguiente: que compute y recuerde cuál será el estado de la unidad de control de T p pasos después. Que recuerde qué cabezales descansan en las celdas correspondientes a las posiciones $(t+i)$ y $(s+j)$. Que reescriba los símbolos en la cinta de trabajo de acuerdo a la configuración p pasos después (esto es posible debido a que existe un cabezal en cada celda de trabajo con índices en el intervalo $[s-p+1, s+p-1]$). Finalmente, que mueva el cabezal de partida con índices en el intervalo $[t-p+1, t-p+i]$ y los cabezales de trabajo con índices en el intervalo $[s-p+1, s-p+j]$ un paso a la derecha; de esta manera, los índices utilizados por ellos llenarán el intervalo $[t+p, t+p+i-1]$ (resp. $[s+p, s+p+i-1]$), los cuales, junto con los cabezales que permanecieron en su lugar, entregan el intervalo $[t+i-p, t+i+p-1]$ (resp. $[s+j-p+1, s+j+p-1]$).

Si durante los p pasos considerados T escribe en la cinta de salida (ya sea 0 ó 1) y se detiene, entonces hagamos que S también lo haga. Asi entonces construimos una máquina de Turing S que (aparte del copiado inicial) realiza sólo p veces el número de pasos de T y decide el mismo lenguaje. $\odot$

Es posible probar un teorema similar para medidas de espacio.

No cualquier función sirve de cota. La teoría que hemos venido delineando se comporta bien para las funciones usuales: logaritmos, polinomios, exponenciales, combinaciones de ellas. Desgraciadamente hay funciones anómalas que pueden destruir todo el edificio. La buena noticia es que son funciones muy raras.

Llamamos a una función $f : Z_+ \to Z_+$ *constructible completamente en tiempo* si existe una máquina de Turing de múltiples cintas tal que para cada entrada de largo n se detiene usando exactamente $f(n)$ pasos. El significado de esta extraña definición es que con funciones constructibles completamente en tiempo es fácil acotar el tiempo de ejecución de máquina de Turing: si existe una máquina de Turing que realiza exactamente $f(n)$ pasos en cada entrada de largo n, entonces podemos incluir ésta en cualquier otra máquina de Turing como un reloj: sus cintas, excepto las cintas de trabajo, son distintas, y la máquina de Turing combinada entrega en cada paso el trabajo de ambas cintas.

Obviamente, toda función constructible completamente en tiempo es recursiva. Por otro lado, es fácil ver que $n^2, 2^n, n!$ y toda función "razonable" es constructible completamente en tiempo. El lema que sigue garantiza la existencia de muchas funciones completamente constructibles en tiempo.

Llamemos una función $f : Z_+ \to Z_+$ *buena* si existe una máquina de Turing que compute $f(n)$ en tiempo $\mathcal{O}(f(n))$. (Aquí escribimos n y $f(n)$ en notacion unaria: el número n es dado por una secuencia $1 \cdots 1$ de largo n y queremos como salida una secuencia $1 \cdots 1$ de largo $f(n)$. El resultado no variaría, sin embargo, si n y $f(n)$ fuesen representadas en notación binaria).

Ahora el siguiente lema es fácil de demostrar:

Lema 2
1. *Para toda función buena $f(n)$ existe una función completamente constructible en tiempo $g(n)$ tal que $f(n) \leq g(n) \leq cf(n)$ para alguna constante c.*
2. *Para toda función completamente constructible en tiempo $g(n)$ existe una función buena $f(n)$ con $g(n) \leq f(n) \leq cg(n)$ para alguna constante c.*
3. *Para toda función recursiva $f(n)$ existe una función completamente constructible en tiempo $g(n)$ con $f(n) \leq g(n)$.*

Este lema nos permite utilizar indistintamente, en la mayoría de los casos, funciones completamente constructibles en tiempo y funciones buenas.

Teorema 8 *Si $f(n)$ es completamente constructible en tiempo y $g(n)\log f(n) = o(f(n))$, entonces existe un lenguaje en $\mathrm{DTIME}(f)$ que no pertenece a $\mathrm{DTIME}(g)$.*

Este resultado dice que las complejidades en tiempo de los lenguajes recursivos son "suficientemente densas". Análogamente, pero más simple, el resultado se mantiene para complejidades de espacio. La constructibilidad completamente en tiempo de la función f juega un rol muy importante en el último teorema. Si no la consideramos, es posible encontrar funciones tales que $\mathrm{DTIME}(2^{2^{f(n)}}) = \mathrm{DTIME}(f(n))$. Su construcción está dada por el llamado teorema de brecha cuyo enunciado es el siguiente:

Teorema 9 (de la Brecha) *Para toda función recursiva $\varphi(n) \geq n$ existe una función recursiva $f(n)$ tal que:*

$$\mathrm{DTIME}(\varphi \circ f) = \mathrm{DTIME}(f).$$

Luego, existe una función recursiva f con $\mathrm{DTIME}(f^2) = \mathrm{DTIME}(f)$ e incluso existe una tal que $\mathrm{DTIME}(2^{2^{f(n)}}) = \mathrm{DTIME}(f(n))$.

Otro resultado interesante, el llamado *teorema de aceleramiento*, muestra que existen problemas (lenguajes) arbitrariamente "acelerables", o de otra forma, que para un problema dado, generalmente no hay un "mejor" algoritmo:

Teorema 10 (de aceleramiento) *Para toda función recursiva $g(n)$ existe un lenguaje recursivo $\mathcal{L}$ con la siguiente propiedad: para toda máquina de Turing T que decide $\mathcal{L}$ existe una máquina de Turing S que decide $\mathcal{L}$ con $g(\mathrm{time}_S(n)) < \mathrm{time}_T(n)$.*

El teorema de aceleramiento lineal se aplica a todo lenguaje; este teorema expresa sólo la existencia de un lenguaje arbitrariamente "acelerable". En general, para un lenguaje arbitrario, no puede esperarse algo mejor que aceleramiento lineal. La esencia de la demostración del teorema es que permitimos máquinas más complicadas para almacenar más información en la unidad de control. Luego, la máquina necesita trabajar sólo con entradas mas largas que "su propio mérito"

La conclusión más importante que se desprende del teorema de aceleramiento es que que aun cuando es conveniente hablar de complejidad computacional de un cierto lenguaje $\mathcal{L}$, las afirmaciones rigurosas referentes a la complejidad generalmente no se refieren a una única función $t(n)$, sino sólo dan *cotas superiores* $t'(n)$ (a través de la construcción de una máquina de Turing que decida el lenguage en tiempo $t'(n)$)) o *cotas inferiores* $t''(n)$ (mostrando que ninguna máquina de Turing puede resolver el problema en tiempo $t''(n)$ para todo n).

2.5 Complejidad Intrínseca

Resumen de complejidad descriptiva.

2.6 Ejercicios

1. Ordenar en una lista $\mathcal{O}$-creciente la siguiente lista de funciones (para $n \geq 1$): n^2, n^3, $n^2 \lg n$, $5n^2 + 100$, 2^n, $n^{\lg n}$, $n!$, 2^{2^n}, n^n, n^2 si n es par, 2^n de lo contrario.

2. Para toda máquina de Turing T y $c < 0$, se puede encontrar una máquina de Turing S con el mismo número de cintas que decide el mismo lenguaje y para la cual $\mathrm{time}_S(n) \leqslant c \cdot \mathrm{time}_T(n) + n$ (aquí permitimos la extensión del alfabeto).

3. Formule y demuestre el análogo al problema anterior para espacio en vez de tiempo.

4. Probar el Teorema 8 y la siguiente sentencia relacionada. Sea $t\prime(n) \log t\prime(n) = o(t(n))$. Entonces el problema de la parada con cota t no puede decidirse en una máquina de Turing de dos cintas en tiempo $t\prime(n)$.

5. Muestre que si $s(n)$ es cualquier función y $s\prime(n) = o(s(n))$ entonces el problema de la parada para espacio con cota $s(n)$ no puede ser resuelto en tiempo $s\prime(n)$.

Capítulo 3

Tiempo polinomial

Hay consenso en la comunidad computacional que *tiempo polinomial* (es decir, tiempo acotado por una función polinomial en el tamaño del dato de entrada) es una propiedad que se acerca a la *resolubilidad práctica* o *eficiente*.[1] Formalmente,

$$\mathrm{PTIME} = \bigcup_{k \geq 0} \mathrm{DTIME}(n^k).$$

Como en el caso de la tesis de Church, no hay (ni puede haber) demostración formal de que esta clase realmente *es* la clase de los algoritmos eficientes. Pero la experiencia de quienes trabajan en computación, ya sea en materias aplicadas, en aspectos teóricos, en lógica, en optimización, ha llevado al convencimiento que esta es la clase que mejor captura los problemas tratables. Un fuerte argumento reside en el hecho de que todos los modelos conocidos de computación (máquinas de Turing simples, con varias cintas, máquinas RAM, etc.) son reducibles (simulables) unas a otras por algoritmos polinomiales.

[1]No todo es tan perfecto desgraciadamente. No todos los problemas en esta clase son necesariamente tratables en la práctica. Esto se debe a varios factores, entre ellos, la posibilidad que las constantes del polinomio sean muy grandes, que el grado del polinomio sea muy grande, que incluso pudiera haber problemas solubles por algoritmos polinomiales pero su descripción muy difícil o imposible.

La siguiente tabla muestra uno de los argumentos de más peso para asociar tiempo polinomial con resolubilidad práctica. La tabla representa el comportamiento en tiempo de un algoritmo de complejidad T en función del tamaño del dato de entrada D suponiendo que cada operación elemental toma 10^{-9} segundos. (En la tabla, ms abrevia milisegundo, es decir 10^{-3} segundos).

$T \backslash D$	**10**	**30**	**50**
$\mathbf{n^2}$	0,0001 ms	0,0009 ms	0,0025 ms
$\mathbf{n^3}$	0,001 ms	0,027 ms	0,125 ms
$\mathbf{n^5}$	0,01 ms	24,3 ms	310 ms
$\mathbf{2^n}$	0,001 ms	1 s	13 días
$\mathbf{3^n}$	0,06 s	2,4 días	$\approx 10^7$ años

Tabla 3.1: Tiempo que toma un algoritmo de complejidad en tiempo T en una instancia de tamaño D.

Obsérvese especialmente como crece el tiempo para las dos últimas filas, que son algoritmos exponenciales, versus el comportamiento de los algoritmos polinomiales.

La mayor parte de esta sección se dedica a ilustrar como ciertos problemas pueden ser resueltos con restricciones de recursos. Mostraremos que la mayoría de las tareas computacionales básicas pueden ser resueltas en tiempo polinomial. Esto incluye problemas de la teoría de números (operaciones aritméticas, máximo común divisor, aritmética modular), álgebra lineal (eliminación Gaussiana) y teoría de grafos.

3.1 Algoritmos en Aritmética

3.1.1 Operaciones aritméticas

Todas las operaciones aritméticas básicas son polinomiales: suma, resta, multiplicación y división de enteros con resto. (Recordemos que el largo de un entero n de entrada es el número de bits que se necesitan para representarle, es decir, $\log_2 n + \mathcal{O}(1)$). Aprendimos algoritmos polinomiales para todas estas operaciones en la escuela elemental (algoritmos lineales en el caso de la suma y la resta, cuadráticos en el caso de la multiplicación y división). Consideramos también la comparación de dos números como una operación aritmética trivial, aunque esencial, y esto también puede hacerse en tiempo polinomial (lineal).

Un algoritmo aritmético polinomial menos trivial es algoritmo de Euclides, que resuelve el siguiente problema

Problema 4 *(Máximo común divisor)*

ENTRADA. *Dos números enteros a y b.*
RESPUESTA. *El máximo común divisor de a y b, denotado* $\mathrm{mcd}(a, b)$.

El algoritmo de Euclides funciona como sigue. Podemos suponer que a es más pequeño que b. Si $a = 0$ entonces el máximo común divisor de a y b es $\mathrm{mcd}(a, b) = b$. Si $a > 0$ entonces, dividimos b por a con resto, que denotamos por r. Entonces $\mathrm{mcd}(a, b) = \mathrm{mcd}(a, r)$ y es suficiente entonces calcular el máximo común divisor entre a y r. Como $r < a$ la recurrencia terminará en un número finito de iteraciones y tenemos el máximo común divisor entre a y b.

	a	b	r
1	84	48	36
2	48	36	12
3	36	**12**	0

mcd(84, 48) = 12

Figura 3.1: Algoritmo de Euclides

Notemos que estrictamente hablando, el algoritmo dado anteriormente no es un programa para una máquina de acceso aleatorio. Es un programa recursivo y tal como está es bastante informal. Pero sabemos que un programa informal puede ser llevado a uno formal, y que un programa recursivo puede ser traducido a un programa en lenguaje de máquina (la mayoría de los compiladores pueden hacerlo).

Lema 3 *El algoritmo de Euclides toma tiempo polinomial. Más exactamente realiza $\mathcal{O}(\log a + \log b)$ operaciones aritméticas para la entrada (a, b).*

DEMOSTRACIÓN. Dado que $0 \leq r < a \leq b$, el algoritmo de Euclides terminará tarde o temprano. Veamos que lo hace en tiempo polinomial. Notemos que $b \geq a + r > 2r$ y además $r < b/2$. Luego $ar < ab/2$. Por lo tanto después de $\lceil \log ab \rceil$ iteraciones el producto de los dos números será menor que 1, es decir, uno de ellos es cero, lo que dice que el algoritmo termina. Cada iteración consiste en operaciones aritméticas básicas y se hace por lo tanto en tiempo polinomial. ⊙

Otra importante cualidad del algoritmo de Euclides es que no sólo entrega el valor del máximo común divisor, sino que además es posible obtener dos enteros p, q tales que $\mathrm{mcd}(a, b) = pa + qb$. Para esto simplemente tenemos que llevar el siguiente cálculo durante cada iteración. Si $a' = p_1 a + q_1 b$ y $b' = p_2 a + q_2 b$ y si dividimos b' por a' con resto r', es decir $b' = ha' + r'$, entonces

$$r' = (p_2 - hp_1)a + (q_2 - hp_2)b,$$

y así obtenemos la representación del nuevo número r' de la forma $p'a + q'b$.

Nota 5 *El algoritmo de Euclides es presentado a veces como la siguiente iteración: si $a = 0$ entonces fin. Si $a > b$ entonces intercambiemos los números. Si $0 < a \leq b$ entonces sea $b := b - a$. Esencialmente lo mismo desde el punto de vista matemático (de hecho el algoritmo original de Euclides era más cercano a esto). Este algoritmo no es polinomial, debido a que la computación de* $\mathrm{mcd}(1, b)$ *necesita b iteraciones, lo que es exponencial en el número de bits* $\log b + \mathcal{O}(1)$.

Las operaciones de suma, resta y multiplicación pueden también ser calculadas en tiempo polinomial en el escenario de la *aritmética modular* (anillo de las clases de restos módulo m). Representamos una clase de restos por el menor resto no-negativo que contiene. Realizamos la operación entre ellos como enteros y posteriormente realizamos división con resto por m. Así por ejemplo, 47 módulo 5 es 2. Es conveniente introducir la siguiente notación: $a \equiv b \mod m$ significa que m divide $(a - b)$; luego $47 \equiv 2 \mod 5$.

Si m es número primo, entonces también podemos realizar la *división modular* en tiempo polinomial. ¡Esto no es lo mismo que división con resto! Dados números a, b y m tales que $0 \leq a, b \leq m - 1$ y $b \neq 0$, podemos calcular un entero x con $0 \leq x < m$ tal que

$$bx \equiv a \mod m.$$

(x es denotado a veces como $x \equiv a/b \mod m$).

La solución es aplicar el algoritmo de Euclides para calcular el máximo común divisor entre b y m. Obviamente ya sabemos que el resultado es 1, pero lo que nos interesa son los números p, q tales que $bp + mq = 1$. En otras palabras $bp \equiv 1 \mod m$, y por lo tanto $b(ap) \equiv a \mod m$. Así, el cuociente x que buscamos se obtiene del resto ap después de dividirlo por m.

···	0	1	2	**3**	0	1	2	3	0	**1**	2	···	$\mathbb{Z}$ mod 4
···	0	1	2	**3**	4	5	6	7	8	**9**	10	···	$\mathbb{Z}$

$\mathbb{Z}$	$\mathbb{Z}$ mod 4
3·3 = 9	3·3 = 1 mod 4
3 = 9/3	3 = 1/3 mod 4

Figura 3.2: Aritmética módulo 4

Otra aplicación. Mencionamos antes otra aplicación del algoritmo de Euclides. Supongamos que tenemos un número x desconocido, pero que conocemos sus restos $x_1, \ldots, x_k$ respecto de los módulos $m_1, \ldots, m_k$ que son primos relativos entre ellos. El *teorema chino del resto* dice que los restos determinan de manera única a x módulo el producto $m = m_1 \cdots m_k$. Pero ¿cómo podríamos computar ese resto?

Es suficiente considerar el caso $k = 2$, dado que para un k general el algoritmo se obtiene de éste por inducción. Estamos buscando un entero x tal que $x \leq x_1 \mod m_1$ y $x \equiv x_2 \mod m_2$ (además queremos que $0 \leq x \leq m_1 m_2 - 1$, pero esto puede ser logrado dividiendo por el resto al final).

En otras palabras , buscamos x, q_1, q_2 tales que $x = x_1 + q_1 m_1$ y $x = x_2 + q_2 m_2$. Restando, tenemos $x_2 - x_1 = q_1 m_1 - q_2 m_2$. Esta ecuación no determina de manera única a q_1 y q_2, pero eso no es importante. Usando el algoritmo de Euclides podemos encontrar q_1 y q_2 tales que

$$x_2 - x_1 = q_1 m_1 - q_2 m_2,$$

y se computa $x = x_1 + q_1 m_1 = x_2 + q_2 m_2$. Entonces $x \leq x_1 \mod m_1$ y $x \equiv x_2 \mod m_2$, como deseábamos.

Ahora discutiremos la operación de exponenciación. Para escribir el número 2^n, necesitamos un número exponencial de dígitos (en términos del largo de la representación binaria de la entrada n). Obviamente este número no es computable en tiempo polinomial. Sin embargo, la situación cambia si queremos el resultado de la exponenciación módulo m: así a^b es también una clase módulo m, y luego puede ser representado por $\log m + \mathcal{O}(1)$ bits. Mostraremos que no solamente pude ser representado en forma polinomial, sino que también lo podemos calcular en tiempo polinomial.

Lema 4 *Sean a, b y m tres números naturales. Entonces $a^b \mod m$ puede ser computado en tiempo polinomial, o más exactamente con $\mathcal{O}(\log n)$ operaciones aritméticas, sobre números naturales con $\mathcal{O}(\log m + \log a)$ dígitos.*

Demostración. Escribamos b en notación binaria:

$$b = 2^{r_1} + \cdots + 2^{r_k},$$

donde $0 \leq r_1 < \cdots < r_k$. Es obvio que $r_k < \log b$ y de aquí se sigue que $k \leq \log b$. Ahora bien, los números $a \mod n$ para $0 \leq t \leq \log b$ son fácilmente calculables elevando al cuadrado repetidas veces, y luego multiplicamos aquellos k que juntos construyen a^b. Por supuesto realizamos todas las operaciones módulo m, es decir, luego de cada multiplicación además realizaremos una división con resto por m. $\odot$

Nota 6 *No se sabe si es posible calcular $a! \mod m$ o $\binom{a}{b} \mod m$ en tiempo polinomial.*

3.1.2 Eliminación Gaussiana

Las operaciones básicas del álgebra lineal son polinomiales: suma y producto punto de vectores, multiplicación e inversión de matrices, el cálculo de determinantes. No obstante, estos resultados no son triviales en los últimos dos casos, por lo que les veremos en detalle.

Sea $A = (a_{ij})$ una matriz arbitraria de enteros de $n \times n$.

Veamos primero que todo, que el cálculo de $\det(A)$ no es inherentemente imposible, en el sentido de que puede ser escrito con un número polinomial de bits. Sea $K = \max |a_{ij}|$; entonces para escribir la matriz A necesitamos obviamente a lo menos $\mathcal{L} = n^2 + \log K$ bits. Por otro lado, de la definición de determinante se desprende que

$$|\det(A)| \leq n!K^n,$$

y por lo tanto $\det(A)$ puede ser escrito usando

$$\log(n!K^n) + \mathcal{O}(1) \leq n(\log n + \log K) + \mathcal{O}(1)$$

bits. Y esto es polinomial en $\mathcal{L}$.

El álgebra da una fórmula para calcular cada elemento de $\det(A^{-1})$ como el cuociente de dos subdeterminantes de A. Esto muestra que también A^{-1} puede ser escrita con una cantidad polinomial de bits.

El procedimiento usual para calcular un determinante es la eliminación Gaussiana. Podemos ver esto como la transformación de la matriz en otra triangular inferior por medio de operaciones sobre las columnas. Estas transformaciones no afectan el determinante, y en la matriz resultante el cálculo del determinante es trivial. Tan sólo tenemos que multiplicar los elementos de la diagonal para obtenerlo. Es también fácil obtener la matriz inversa de esta manera y por ello no trataremos este caso separadamente.

Eliminación Gaussiana. Sea $1 \leq t \leq n$. Supongamos que para todo i tal que $1 \leq i \leq t$, hemos logrado que en la i-ésima fila solamente los primeros i elementos sean distintos de cero. Tomemos uno de estos elementos de las últimas $n-t$ columnas (paramos si no existiera tal elemento). Llamaremos a este elemento *pivote* de esa fase. Reordenamos las filas y las columnas de manera que el pivote quede en la posición $(t+1, t+1)$. Luego restamos la columna $t+1$, multiplicada por $a_{t+1,i}/a_{t+1,t+1}$, de las columna i-ésima para todo $i = t+2, \ldots, n$. De esta forma obtenemos ceros en los elementos $(t+1, t+1), \ldots, (t+1, n)$. Estas restas no cambian el valor del determinante y las operaciones de reordenamiento cambian a lo más el signo del determinante, de lo que puede llevarse la cuenta sin problemas.

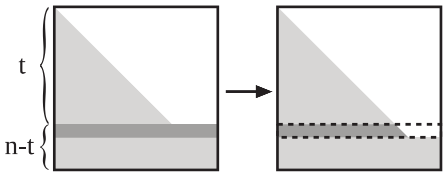

Figura 3.3: Un paso en la eliminación Gaussiana

Debido a que una operación Gaussiana ocupa $\mathcal{O}(n^2)$ operaciones aritméticas y a que estas iteraciones deben ser realizadas n veces, este procedimiento usa $\mathcal{O}(n^3)$ operaciones aritméticas. Pero el problema está en que nosotros queremos además dividir y no con resto. Esto no sería un problema sobre un cuerpo finito, pero éste no es el caso de los racionales. Nosotros asumimos que los elementos de la matriz original son enteros, pero durante el proceso pueden aparecer números racionales. ¿Cómo deberían ser almacenados estos racionales? La respuesta natural es: como pares de enteros (cuyo cuociente es el número racional) .

Necesitamos entonces que las fracciones estén expresadas en su forma simplificada, en que numerador y denominador sean primos relativos entre si. Podemos hacer lo siguiente, simplificar cada elemento de la matriz luego de cada iteración, para esto tenemos que implementar el algoritmo de Euclides. Este puede ser ejecutado en tiempo polinomial, pero es un gran trabajo extra, lo que hace deseable intentar eludirle. (Obviamente aún tenemos que demostrar que en la forma simplificada, los numeradores y denominadores de los elementos de la matriz tienen a lo más un número polinomial de dígitos. Retomaremos esta discusión después.)

Podríamos no obligar a las fracciones a estar en su forma simplificada. Entonces definimos la suma y el producto entre dos números racionales a/b y c/d por medio las siguientes fórmulas: $(ad+bc)/(bd)$ y $(ac)/(bd)$. Con esta convención, el problema es que los numeradores y denominadores que aparecen durante la ejecución del algoritmo pueden volverse muy largos (De hecho, ¡tienen un número no polinomial de dígitos!)

Afortunadamente, podemos dar un procedimiento que guarde fracciones en forma parcialmente simplificada, y elude tanto la simplificación como el crecimiento excesivo del número de dígitos. Para esto, analicemos un poco las matrices que aparecen en la eliminación Gaussiana. Podemos asumir que los elementos pivotes vienen dados por las posiciones $(1,1), \ldots, (n,n)$, es decir, que no tenemos que permutar ni filas ni columnas. Sea $(a_{ij}^{(k)})$ $(1 \leq i, j \leq n)$ la matriz obtenida luego de k iteraciones. Para simplificar, denotemos los elementos de la diagonal principal de la matriz final, como $d_1, \ldots, d_n$ (esto es $d_i = a_{ii}^{(n)}$). Sea $D^{(k)}$ la submatriz determinada por las primeras k filas y columnas de la matriz A. Y sea $D_{ij}^{(k)}$, con $k+1 \leq i, j \leq n$, la submatriz determinada por las primeras k filas más la i-ésima fila, y las primeras k columnas más la j-ésima columna. Sea $d_{ij}^{(k)} = \det(D_{ij}^{(k)})$. Obviamente, $\det(D^{(k)}) = d_{kk}^{(k-1)}$.

Lema 5

$$a_{ij}^{(k)} = \frac{d_{ij}^{(k)}}{\det(D^{(k)})}.$$

Demostración. Si calculamos $\det(D_{ij}^{(k)})$ usando eliminación Gaussiana, entonces en la diagonal obtenemos los elementos $d_1, \ldots, d_k, a_{ij}^{(k)}$. Entonces $d_{ij}^{(k)} = d_1 \cdots d_k \cdot a_{ij}^{(k)}$. Similarmente, $\det(D^{(k)}) = d_1 \cdots d_k$. Dividiendo estas dos últimas ecuaciones se obtiene el lema. $\odot$.

De este lema se sigue que todo elemento que ocurre en una eliminación Gaussiana puede ser representado como una fracción donde tanto el numerador como el denominador son determinantes de alguna submatriz de la matriz original A. De esta manera un número polinomial de dígitos es ciertamente suficiente para representar todas las fracciones obtenidas.

Por lo tanto, no es necesario calcular las simplificaciones de todas las fracciones obtenidas en el proceso. Por la definición de eliminación Gaussiana, tenemos que

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{i,k+1}^{(k)} a_{k+1,j}^{(k)}}{a_{k+1,k+1}^{(k)}}$$

y por lo tanto

$$d_{ij}^{(k+1)} = \frac{d_{ij}^{(k)} d_{k+1,k+1}^{(k)} - d_{i,k+1}^{(k)} d_{k+1,j}^{(k)}}{d_{k,k}^{(k-1)}}.$$

Esta fórmula puede considerarse como una recurrencia para calcular los números $d_{ij}^{(k)}$. Debido a que el lado izquierdo es un entero, la división puede efectuarse perfectamente. Usando la consideraciones anteriores concluimos que el número de dígitos en el cuociente es polinomial en términos del largo de la entrada.

Hay otras dos posibilidades para solucionar el problema de las fracciones en la eliminación Gaussiana.

Podemos aproximar los números por "decimales binarios" de precisión limitada (como sería natural desde el punto de vista de la implementación computacional), permitiendo, digamos, p bits después de la "coma decimal". Entonces el resultado es sólo una aproximación, pero como el determinante es un entero, es suficiente con computar con un error menor a $1/2$. Usando métodos de análisis numérico se puede determinar cuan grande debe ser p para que el error del cálculo sea menor que $1/2$. Se sabe que un número polinomial de dígitos es suficiente, y así se llega a un algoritmo polinomial.

La tercera posibilidad está basada en el hecho de que si $m > |\det(A)|$ entonces es suficiente determinar el valor de $\det(A)$ módulo m. Si m es un número primo, entonces calcular modulo m no requiere usar ningún tipo de fracciones. Dado que sabemos que $|\det(A)| < n!K^n$, es suficiente elegir m como un número primo mayor que $n!K^n$.

Sin embargo, no es fácil elegir un primo grande (ver capítulo de algoritmos aleatorios). Un método fácil es el de elegir m como el producto de distintos números primos pequeños: $m = 2 \cdot 3 \cdots p_k$ donde k puede ser elegido como el número total de bits que aparecen en la representación de A. Entonces es fácil calcular $\det(A) \bmod p_i$ para todo p_i usando eliminación Gaussiana en el escenario de la clase modular de p_i. Así podemos calcular el resto de $\det(A)$ modulo m usando el teorema chino de los restos. (Dado que k es pequeño, podemos encontrar los primeros k números primos simplemente por fuerza bruta. Pero el costo de esta computación debe ser juzgada de modo diferente debido a que los mismos números primos podrían ser utilizados para calcular un número arbitrario de determinantes.)

Nota 7 *El método modular puede ser utilizado con éxito en muchos otros casos. Una forma de ver este método es considerarlo como una codificación de enteros diferente del sistema numérico binario (o decimal): codificamos el entero n por sus restos de las divisiones con los primos 2, 3, etc. Estos son un número infinito de bits, pero si nosotros sabemos de partida que en la computación no aparecerán número más grandes que N entonces basta con considerar los primeros k primos cuyo producto es mayor que N. En esta codificación, las operaciones aritméticas pueden realizarse muy simplemente y en forma paralela para diferentes primos. La comparación por magnitud es, sin embargo, tortuosa.*

3.2 Algoritmos en grafos

Los algoritmos más importantes de la teoría de grafos son polinomiales. Sería imposible abarcar en totalidad este tema en este libro. Los algoritmos en grafos pueden llenar muchos libros. Nos limitaremos a la discusión de unos pocos ejemplos que nos darán una idea particularmente importante de algunas técnicas básicas. Agruparemos nuestros ejemplos en torno de dos métodos algorítmicos: búsqueda en grafos y extensión de caminos.

3.2.1 ¿Como representar un grafo?

Es importante la forma de representar computacionalmente un grafo. Matemáticamente, un grafo G está determinado por un conjunto de nodos V y un conjunto de arcos E (parejas no ordenadas de nodos). Las estructuras de datos más usuales para representar un grafo son dos:

1. Matriz de adyacencia: Se enumeran los nodos de 1 a $|V|$. El grafo es entonces representado por una matriz M de tamaño $|V| \times |V|$ cuyos elementos son bits. El elemento i, j es 1 o 0 de acuerdo a si existe un eje entre los nodos i y j.

2. Arreglo de listas: El grafo es representado por un arreglo de listas. Cada lista corresponde a un nodo v y sus elementos son los nodos adyacentes a v.

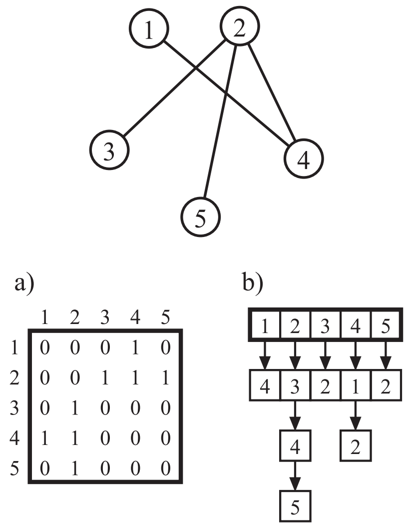

Figura 3.4: Representación de Grafos: a) matriz de incidencia; b) lista de adyacencia

Aunque la representación matricial es mucho más sencilla de manejar, cuando el grafo tiene muy pocos arcos comparados con el número de nodos, esta representación ocupa mucho espacio innecesario. Muchos de los algoritmos más usados de grafos corren en tiempo $\mathcal{O}(|V| + |E|)$, lo que para grafos con pocos arcos es prácticamente lineal en el número de nodos. Si usamos la representación matricial, siempre necesitaremos al menos $\mathcal{O}(|V|^2)$ tiempo (debemos al menos leer la entrada). Para estos casos el arreglo de listas es mucho más deseable.

3.2.2 Búsqueda en grafos

Búsqueda general y verificación de conectividad. Quizás la pregunta fundamental sobre un grafo es si es o no conexo. Si no lo es, generalmente nos interesa encontrar sus componentes conexas. Esta tarea se realiza trivialmente en tiempo polinomial, de varias maneras. La razón por la cual describiremos una solución es por que sus elementos nos serán útiles posteriormente.

Sea G un grafo. Elejimos un nodo r. Construimos un árbol T de la siguiente manera: al principio, T consiste sólo de r. En cada iteración, si aún T no contiene todos los nodos de G, vemos si hay un arco entre los nodos de T y el resto de los nodos. Si hay un eje, lo agregamos a T junto al nodo externo correspondiente. Si no hay, sabemos que G no es conexo y T es un árbol generador de una componente conexa de G. Podemos eliminar de G los nodos de T y proceder recursivamente.

Se acostumbra a llamar a los nodos de T *etiquetados*. La manera usual de buscar un arco entre un nodo etiquetado y uno que no lo es es buscar los nodos etiquetados e investigar los caminos que van fuera de ellos. Una observación simple pero importante, es que si en alguna etapa descubrimos que ninguno de los arcos de algún nodo etiquetado van hacia alguno no etiquetado, entonces no tendremos que investigar ese nodo en ninguna de las iteraciones posteriores (puesto que sólo un nodo no etiquetado puede pasar a ser etiquetado y no viceversa). Por consiguiente podemos marcar ese nodo como tomado en cuenta o *revisado*. En cualquier momento, sólo necesitamos considerar aquellos no que estén etiquetados pero no revisados. Este proceso general de etiquetar y revisar es llamado *búsqueda en el grafo*.

Particularizando el orden en que investigamos arcos que salen de nodos etiquetados, daremos algoritmos especiales que son buenos para resolver distintos tipos de problemas de grafos.

3.2.3 Búsqueda en anchura y caminos más cortos

Quizás la especialización más simple es la llamada *búsqueda en anchura*. El problema es encontrar el camino más corto entre dos nodos dados. La solución a este problema es muy simple si incluímos este problema dentro de una familia de problemas más grande: pedir el camino más corto desde un nodo de partida s hasta cualquier otro nodo del grafo. Procedemos recursivamente, marcamos los vecinos del nodo s con el número 1. Marcamos con $k+1$ aquellos nodos no marcados y que sean vecinos de vértices marcados con k. Es claro que el número anotado en cada nodo es la distancia que lo separa de el nodo de partida. (Este método es quizás el ejemplo más simple de una técnica llamada programación dinámica).

Esta idea reaparece en problemas más generales como el de encontrar el camino más corto entre dos nodos s y t en un grafo con *pesos* en los ejes. (El peso c_{ij} entre los nodos i, j se interpreta como el largo del eje.) Nuevamente introducimos este problema en la gran familia de problemas que preguntan por las maneras más rápidas de llegar desde un nodo s a cualquier otro del grafo. El siguiente *algoritmo de Dijkstra* reconoce cuando un camino entre dos vértices podría ser de peso mínimo, aun cuando otros caminos tengan menos ejes. Así, partiendo desde el nodo s, elegimos entre todos sus vecinos j al de menor costo c_{s,j_0}. Podemos asegurar con certeza que el arco de mínimo costo entre s y éste vecino tiene peso c_{s,j_0}. Pero no podemos asegurar lo que pasa entre s y sus otros vecinos. Etiquetemos este nodo j_0 con el valor c_{s,j_0}. (Podemos pensar que s está etiquetado con 0). Durante el desarrollo del algoritmo, mantenemos un conjunto T de nodos donde están todos aquellos que hemos etiquetado con certeza (es decir, con el costo mínimo de ir de s a él). Denotemos estas etiquetas por $d(s,j)$, el costo mínimo de ir de s a j. Observemos que estos caminos pasan sólo por nodos de T. En cada etapa del algoritmo definamos la frontera de T como aquellos nodos fuera de T que son vecinos a un nodo de T. Consideremos los arcos ij entre nodos i de T y j de la frontera de T. Entonces si consideramos el

$$\min_{i\in T,\, j\in \mathbf{Frontera}(T)} \{d(s,i)+c_{ij}\},$$

y el arco ij es el que logra este mínimo, podemos con certeza etiquetar el nodo j con $d(s,j)+c_{ij}$. (Cualquier camino de s a j debe dejar T en un nodo i' y luego pasar por un nodo j' de la frontera, es decir su costo será al menos $d(s,i')+c_{i'j'}$. Por la elección de i,j el camino de s a i y luego inmediatamente a j tiene costo no mayor que esto.)

3.2.4 Correspondencia máximo en grafos bipartitos y caminos alternativos.

Sea G un grafo. Queremos encontrar el correspondencia de mayor tamaño en G. Recordemos que un *correspondencia* es un subconjunto de arcos que no tienen nodos en común. Un correspondencia es *perfecto* si abarca todos los nodos.

Describiremos un algoritmo polinomial para encontrar correspondencias maximales en un grafo *bipartito*. Un grafo bipartito es aquel cuyo conjunto de nodos se puede particionar en dos clases disjuntas de tal forma que no hayan ejes del grafo entre dos nodos de una misma clase.

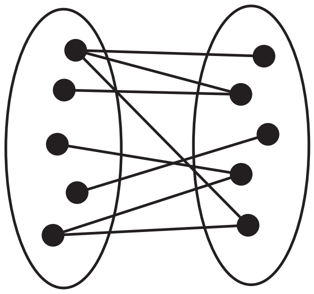

Figura 3.5: Grafo bipartito

Supongamos que tenemos un correspondencia M; ¿cómo podríamos saber que M es máximo? Una simple respuesta a esta pregunta nos la da el siguiente criterio (debido a Berge). Un *camino alternante* (con respecto a M) es aquel en el cual arco por medio pertenece a M (el camino puede o no comenzar con un arco en M). Un camino alternante se dice *aumentativo* si empieza y termina con ejes que no está en M y además, sus nodos inicial y final no son incidentes con ningún arco de$M$.

Lema 6 *Un correspondencia en G es maximal si y sólo si no existe ningún camino aumentativo con respecto a él.*

Demostración. Obviamente, si tenemos un camino aumentativo P con respecto a M, podemos mejorar M: reemplazando en M los arcos de $P\cap M$ por los de $P\setminus M$ se obtiene un correspondencia más grande.

En la otra dirección, si M no es máximo y existe un correspondencia mayor M', considere las componentes conexas de $M\cup M'$. Estos son caminos alternantes o circuitos alternantes con respecto a M. Al menos una de esas componentes debe contener más arcos de M' que de M; entonces esa componente es un camino aumentativo con respecto a M. $\odot$

Este hecho no es muy profundo, pues explica la existencia de algo (correspondencia mayor) a través de la existencia de otra cosa (caminos aumentativos). Pero sí entrega la motivación para la mayoría de los algoritmos polinomiales para correspondencia conocidos: buscar un camino aumentativo por medio de la construcción de un árbol de búsqueda. Y esto se puede lograr en tiempo polinomial.

DEMOSTRACIÓN. Dado un grafo bipartito $G = (V, E)$, queremos encontrar un correspondencia maximal M. Usamos la noción de *árboles alternantes.* Dado un correspondencia M, decimos que un vértice del grafo es *expuesto* (por M) si no está cubierto por M. Un *árbol alternante* es un subgrafo de G que (1) contiene exactamente un nodo expuesto r, (2) todo nodo a distancia impar de r tiene rango 2 en el árbol, (3) a lo largo de cualquier camino por r arco por medio pertenece al correspondencia, y (4) sus nodos estén todos a distancia par de r.

Los nodos del árbol a distancia par de r (incluyendo r mismo) son llamados nodos *externos* y los nodos a distancia impar de r son llamados nodos *internos.*

Un *bosque alternante* es un bosque para el cual todas sus componentes son árboles alternantes, y todo nodo expuesto del grafo está contenido en el bosque.

Si tenemos un correspondencia M y quisiéramos extenderlo, tendíamos que tomar un vértice expuesto r y tratar de emparejarlo con un vecino; pero el vecino pudiera estar ya en el correspondencia, así que dejamos este arco afuera; esto crea un nuevo nodo expuesto que tenemos que intentar emparejar, etc. Esto lleva a la noción de caminos alternantes. Buscar caminos alternantes desde el nodo expuesto r lleva a la construcción de árboles alternantes.

Describiremos ahora el algoritmo. Empezamos con el correspondencia vacío y el bosque alternante consistente de todos los nodos y ningún eje. En cada nueva etapa, podemos incrementar el tamaño del correspondencia (borrando algunos arcos pero agrandando más), o podemos extender el bosque ya construido (manteniendo el correspondencia intacto), o estamos copados y el correspondencia es óptimo.

Supongamos que ya tenemos un candidato a correspondencia M y un bosque F con respecto a M. Hay tres casos posibles:

1. Si hay un arco e conectando dos nodos exteriores del bosque, entonces esos nodos deben pertenecer a árboles diferentes debido a que el grafo es bipartito. Sean u, v los nodos expuestos de los dos árboles. Tenemos un camino alternante desde u a v, e intercambiando los arcos de emparejadores a través del camino obtenemos un nuevo correspondencia M' con un arco más que M. El bosque consistente de todos los nodos expuestos de M' es alternante con respecto a M'. (Ver Figura 3.6).

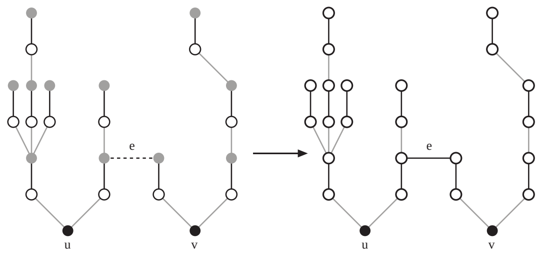

Figura 3.6: Extensión de un correspondencia de un grafo bipartito

2. Por otro lado, si hay un arco entre un nodo exterior p del bosque y un nodo u de $V(G) \setminus V(F)$, ese nodo u no puede ser expuesto debido a la definición de un bosque alternante. Luego es el nodo de un arco del correspondencia, digamos uv. El nodo v está también en $V(G) \setminus V(F)$ (de lo contrario u debería estar en F), y de esta forma podemos extender F agregando los arcos pu y ev. (Ver Figura 3.7).

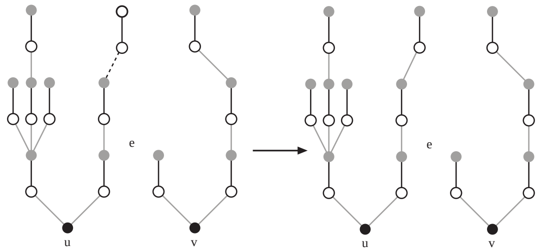

Figura 3.7: Extensión de un bosque alternante

3. Si ninguno de los casos de arriba se aplica, significa que los nodos exteriores deben estar conectados sólo con nodos internos. Un argumento de conteo simple, pero crucial, mostrará que el correspondencia debe

ser óptimo. (Usaremos este argumento en varias ocasiones durante estas notas).

Hay un nodo expuesto en cada componente de F y ninguno afuera de F, así que el total de nodos expuestos en el correspondencia es igual al número de componentes de F. Cada nodo exterior de los árboles de F es emparejado a un nodo interno, que es su antecesor en el árbol, excepto por aquellos nodos expuestos de cada árbol. Por lo tanto

$$\begin{aligned}&\#\text{ nodos exteriores} = \\ &\quad \#\text{ nodos internos} + \#\text{ nodos expuestos.} \qquad (3.1)\end{aligned}$$

Sea D el conjunto de nodos exteriores y A el de los internos de F. El conjunto de los vecinos de D en el grafo está incluido en A y el número de nodos expuestos de M es exactamente $|D| - |A|$.

Sea M' cualquier correspondencia del grafo. M' sólo puede emparejar nodos de D a nodos de A, y debe dejar al menos $|D| - |A|$ nodos expuestos. De aquí que M sea un correspondencia maximal.

$\odot$

Por lo tanto, tenemos un algoritmo válido para encontrar correspondencias maximales en un grafo bipartito. En cada paso debemos considerar a lo más n^2 ejes, y como el tamaño de M o F incrementa con cada paso, hay a lo más n^2 pasos, por lo tanto es un algoritmo que corre en $\mathcal{O}(n^4)$ pasos. Consideraciones simples permiten reducir esta cota a $\mathcal{O}(n^3)$. Es aún posible, pero bastante difícil, reducirla a $\mathcal{O}(n^{2,5})$.

3.3 Ejercicios

1. Los números de Fibonacci están definidos por la siguiente recurrencia: $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$ para $k > 1$. Sean $1 \leq a \leq b$ y sea F_k el mayor número de Fibonacci que no es mayor que b. Pruebe que el algoritmo de Euclides, cuando es aplicado sobre el par (a, b) termina en a lo más k pasos.

   ¿Cuantos pasos toma el algoritmo cuando se aplica sobre (F_k, F_{k-1})?

2. (a) Muestre que para calcular $a/b \mod m$ no necesitamos que m sea primo; es suficiente con asumir que $\text{mcd}(b, m) = 1$.

   (b) Más en general, muestre que dados enteros a, b y m, podemos encontrar en tiempo polinomial un entero x tal que $bx \equiv a \mod m$, o concluya que ese entero no existe.

3. Muestre que si A es una matriz cuadrada formada por enteros, entonces para escribir $\det(A)$ necesitamos a lo más tantos bits como para escribir A. (Ayuda: Si $a_1, \ldots, a_n$ son vectores fila de A entonces $|\det(A)| \leq |a_1| \cdots |a_n|$. Esta llamada "inecuación de Hadamard" es análoga al hecho de que el área de un paralelogramo es menor que el producto de sus lados).

4. Sea A una matriz de $m \times n$ con 2's en la diagonal principal y 1's debajo y 0's arriba. Muestre que si se aplica eliminación Gaussiana sin simplificación a esta matriz, después de k iteraciones cada uno de los denominadores será $2^{2^{k-1}}$.

5. De un algoritmo que resuelve 2-colorabilidad en tiempo polinomial.

6. Una fórmula es de Horn si cada cláusula contine a lo más una variable no negada. De un algoritmo polinomial que decida si una fórmula de Horn es satisfacible (Ayuda: parta asignando falso a todas las variables y luego proceda ...).

Capítulo 4

Computación No-determinística

4.1 Máquinas de Turing no-determinísticas

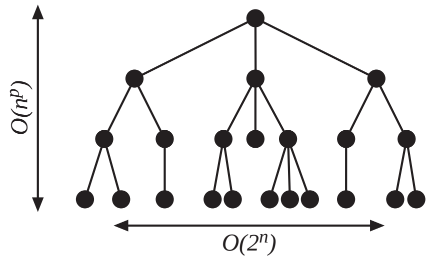

Figura 4.1: No determinismo significa sondear *simultáneamente* todas las posibilidades

Una máquina de Turing no-determinística difiere de una determinística solamente en el hecho de que en cada posición, el estado de la unidad de control y los símbolos leídos por los cabezales permiten *más de una* posible acción. Para cada estado $g \in \Gamma$ y conjunto de símbolos $h_1, \ldots, h_k$ se fija un conjunto de "acciones legales", donde una *acción legal* es una tupla de $(2k+1)$ elementos consistente de un nuevo estado $g' \in \Gamma$, nuevos símbolos $h'_1, \ldots, h'_k$, y movidas $j_1, \ldots, j_k \in \{-1, 0, 1\}$. Más exactamente, una máquina de Turing no-determinística es una tupla ordenada $T = (k, \Sigma, \Gamma, \Phi)$, donde $k \geq 1$, Σ y Γ son conjuntos finitos, $* \in \Sigma$, START, STOP $\in \Gamma$ (hasta el momento todo igual a una máquina determinística) y

$$\Phi \subseteq (\Gamma \times \Sigma^k) \times (\Gamma \times \Sigma^k \times \{-1, 0, 1\}^k)$$

es una relación arbitraria.

Una *computación legal* de la máquina es una sucesión de pasos tal que en cada paso (al igual que en una máquina determinística) la unidad de control entra a un nuevo estado, los cabezales escriben nuevas letras sobre las cintas y se mueven a lo más un paso hacia la derecha o hacia la izquierda. Los pasos deben satisfacer las siguientes condiciones: si el estado de la unidad de control era $g \in \Gamma$ antes del paso y los cabezales leían en las correspondientes cintas los símbolos $h_1, \ldots, h_k \in \Sigma$, entonces para el nuevo estado g', los nuevos símbolos escritos $h'_1, \ldots, h'_k$ y los pasos $\varepsilon_1, \ldots, \varepsilon_k \in \{-1, 0, 1\}$ se tiene

$$(g, h_1, \ldots, h_k, g', h'_1, \ldots, h'_k, \varepsilon_1, \ldots, \varepsilon_k) \in \Phi.$$

Por consiguiente, una máquina de Turing no determinística puede tener varias computaciones legales para la misma entrada. Este es un punto muy importante de recordar.

Una máquina de Turing no determinística T *acepta* una palabra $x \in \Sigma_0^*$ en tiempo t si cada vez que escribimos x en la primera cinta y la palabra vacía en las otras, la máquina tiene una computación legal que demora t pasos con esta entrada y tal que en su posición de parada tiene en la posición 0 de la primera cinta el símbolo "1". (Pueden haber otras computaciones legales que tardan mucho más, otras en que podría no detenerse nunca, otras que rechazan la palabra.)

Decimos que una máquina de Turing no determinística T *reconoce* un lenguaje $\mathcal{L}$ si $\mathcal{L}$ consiste exactamente de aquellas palabras aceptadas por T (no importa cuanto se demore siempre que sea tiempo finito). Adicionalmente a esto, si la máquina acepta cada palabra de $x \in \mathcal{L}$ en tiempo $f(|x|)$ (para cierta función $f : \mathbb{N} \longrightarrow \mathbb{N}$), decimos que T reconoce $\mathcal{L}$ en tiempo $f(n)$. El reconocimiento en términos de almacenamiento o espacio se define análogamente. La clase de lenguajes reconocidos por máquinas de Turing no determinísticas en tiempo $f(n)$ se denotará por NTIME(f) (respectivamente NSPACE(f) para espacio).

A diferencia de las clases determinísticas, el reconocimiento no determinístico de un lenguaje $\mathcal{L}$ no significa que el lenguaje complementario $\Sigma_0^* \setminus \mathcal{L}$ sea reconocible (un ejemplo de esto son los lenguajes recursivamente enumerables pero no recursivos). Por consiguiente tiene sentido (e importancia) introducir las clases complementarias co-NTIME(f). Un lenguaje $\mathcal{L}$ pertenece a co-NTIME(f) si y sólo si el lenguaje complementario $\Sigma_0^* \setminus \mathcal{L}$ pertenece a NTIME(f). De nuevo, la noción de aceptancia en espacio y las correspondientes clases NSPACE(f) y co-NSPACE(f) se definen en forma análoga.

Nota 8 *1. Por supuesto que las máquinas de Turing pueden ser consideradas como un caso particular de máquinas de Turing no determinísticas. Por lo tanto,*

$$\mathrm{TIME}(f) \subseteq \mathrm{NTIME}(f). \tag{4.1}$$

En analogía con el hecho de que existen lenguajes recursivamente enumerables que no son recursivos (es decir, sin límites de tiempo ni espacio las máquinas de Turing no determinísticas son más poderosas que las determinísticas), uno esperaría que la inclusión (4.1) fuese estricta. Esto ha sido demostrado, sin embargo, sólo en casos muy especiales (por ejemplo para funciones f lineales; ver [?]).

Más adelante trataremos el caso más conocido, cuando f es un polinomio, es decir las clases P *y* NP.

2. Una máquina de Turing no determinística puede hacer varios tipos de pasos a partir de determinada configuración. No suponemos ninguna probabilidad de distribución sobre éstos, luego no podemos hablar sobre la probabilidad de cierta computación. Si hiciéramos esto estaríamos en el ámbito de las máquinas aleatorias, *que veremos en una sección más adelante.*

Teorema 11 *Los lenguajes reconocibles por máquinas de Turing no determinísticas son exactamente los lenguajes recursivamente enumerables.*

DEMOSTRACIÓN. Supongamos primero que $\mathcal{L}$ es recursivamente enumerable. Entonces hay una máquina de Turing T con la propiedad de que para en tiempo finito para las entradas x si y sólo si $x \in \mathcal{L}$. Modifiquemos T de tal forma que antes de detenerse, escriba el símbolo 1 en la posición 0 de la primera cinta. Obviamente, esta máquina modificada acepta x si y sólo si $x \in \mathcal{L}$.

En la otra dirección, supongamos que $\mathcal{L}$ es reconocible por una máquina de Turing no determinística T. Podemos suponer que $\mathcal{L}$ no es vacío. Sea $a \in \mathcal{L}$. Sea $\mathcal{L}^\#$ el lenguaje consistente de todas las computaciones finitas de T. Cada elemento de $\mathcal{L}^{\#}$ produce, con alguna codificación adecuada, una sucesión con las configuraciones, o descripciones instantáneas, a medida que va transcurriendo la computación.

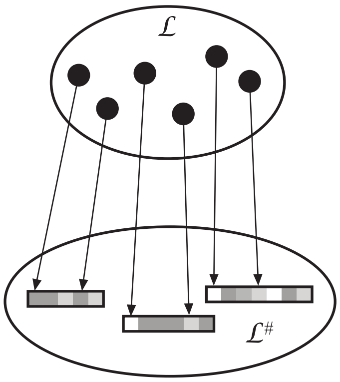

Figura 4.2: Cada elemento de $\mathcal{L}^{\#}$ codifica sucesiones de configuraciones

Cada configuración muestra el estado interno, el símbolo a ser leído en cada cinta y las posiciones de los cursores. Claramente el conjunto $\mathcal{L}^{\#}$ es recursivo puesto que dadas dos configuraciones, es posible decidir si la segunda se obtiene a partir de la primera en una computación de T. Sea S una máquina de Turing que para cada entrada y decide si ella está en $\mathcal{L}^{\#}$, y en caso positivo, describe la computación legal de alguna palabra x que ella representa. El rango de valores de la función recursiva definida por S es exactamente $\mathcal{L}$. $\odot$

4.2 Testigos

Fijemos un alfabeto finito Σ_0 y consideremos un lenguaje $\mathcal{L}$ sobre él. Nos proponemos presentar en esta sección una forma muy útil e intuitiva del hecho que $\mathcal{L}$ sea reconocible por una máquina de Turing no determinística en un tiempo dado.

Un *testigo* es un sujeto que tiene información adicional que permite dilucidar un hecho generalmente difícil de desentrañar por sí mismo.

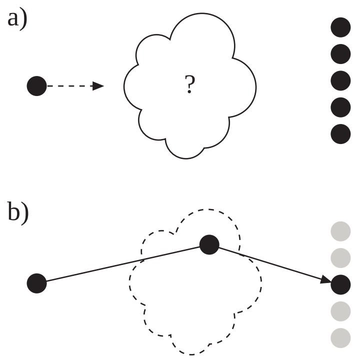

Figura 4.3: Un testigo da información adicional para dilucidar un hecho

Nuestro hecho es la colección de problemas que representa el lenguaje $\mathcal{L}$. Necesitamos para cada instancia, información adicional que permita dilucidar el problema en forma más sencilla. Por ejemplo, para determinar que un número no es primo, un testigo podría ser un divisor de él; para determinar si hay un circuito hamiltoniano en un grafo, basta con presentar un tal circuito como testigo. Hay muchos problemas que tiene esta característica: una vez presentado el testigo, la solución es sencilla de verificar. Lo difícil en general es encontrar el testigo. Por supuesto nos interesarán testigos de tamaño razonable.

Sean f, g dos funciones con $g(n) \geq n$. Decimos que el lenguaje $\mathcal{L}_T$ es un *testigo* (también llamado *certificado*) de largo $f(n)$ y tiempo $g(n)$ para el lenguaje $\mathcal{L}$ si se tiene que para cada $x \in \mathcal{L}$ hay una palabra $y \in \mathcal{L}_T$ de tamaño $|y| \leq f(|x|)$ y una relación entre las palabras x e y, $R(x, y)$, verificable en tiempo $g(|x|)$.

Teorema 12 *1. Cada lenguaje* $\mathcal{L} \in$ NTIME(f) *tiene un testigo de largo* $\mathcal{O}(f)$ *y tiempo* $\mathcal{O}(n)$.

2. *Si un lenguaje $\mathcal{L}$ tiene un testigo de largo $f(n)$ y tiempo $g(n)$, entonces $\mathcal{L} \in$ NTIME$(g(f(n)+n+1))$.*

DEMOSTRACIÓN. (1). Sea T máquina de Turing no determinística que reconoce el lenguaje $\mathcal{L}$ en tiempo $f(n)$ con, digamos, dos cintas. Siguiendo el patrón de la demostración del Teorema 11, asociemos a cada palabra $x \in \mathcal{L}$ una descripción de una computación legal de T que acepte x en tiempo $f(|x|)$. No es difícil construir una máquina de Turing que decida, para una palabra de largo N, en $\mathcal{O}(N)$ pasos si ésta es una descripción de una computación legal, y en caso positivo, acepte esta palabra x. Luego el testigo será el lenguaje constituido por las descripciones antes mencionadas. Es trivial verificar que cumple con las condiciones requeridas.

(2). Sea $\mathcal{L}_T$ el testigo de $\mathcal{L}$, y considérese una máquina de Turing determinística S que decida $\mathcal{L}_T$ en tiempo $g(n)$. Construyamos una máquina de Turing no determinística T que haga lo siguiente: cuando x es ingresado en su primera cinta, entonces computa (determinísticamente) el valor de $f(|x|)$ y escribe este número de unos en la segunda cinta. Entonces escribe el símbolo & al final de x y hace una transición a su único estado en el cual su comportamiento es no determinístico. Estando en este estado, escribe una palabra y de largo a lo más $f(|x|)$ después de $x\&$. Esto lo hace así: mientras lee un 1 en la segunda cinta, tiene $|\Sigma_0|+1$ movidas legales: o escribe un símbolo del alfabeto en la primera cinta, se mueve a la derecha en la primera cinta, a la izquierda en la segunda, o no escribe nada y hace una transición al estado START2.

Desde el estado START2, en la primera cinta, la máquina mueve su cabeza a la celda de partida, borra la segunda cinta, y procede a trabajar como la máquina S.

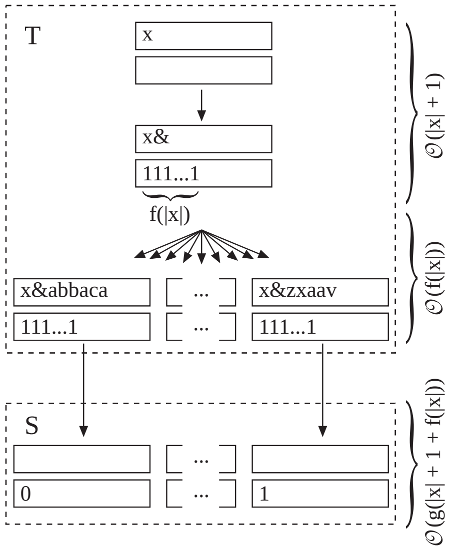

Figura 4.4: Las máquinas T y S de la demostración

Esta máquina T acepta una palabra si y sólo si hay una palabra $y \in \Sigma_0^*$ de largo a lo más $f(|x|)$ para la cual S acepta la palabra $x\&y$, es decir, $y \in \mathcal{L}_T$. El tiempo ocupado por esta computación es claramente $\mathcal{O}(f(|x|)+g(|x|+1+f(|x|)))$. $\odot$

El caso más importante que estudiaremos es el caso cuando los testigos son de tamaño polinomial y la relación R es computable en tiempo polinomial.

Corolario 1 *Para un lenguaje arbitrario $\mathcal{L}$, las siguientes propiedades son equivalentes:*

1. *$\mathcal{L}$ es reconocible por una máquina de Turing no determinística en tiempo polinomial,*

2. *$\mathcal{L}$ tiene un testigo de tamaño polinomial (cuya relación es) reconocible en tiempo polinomial.*

Nota 9 *Mencionaremos sin probarlo (ni siquiera sin formularlo formalmente) que estas propiedades son equivalentes a lo siguiente: es posible dar una definición de $\mathcal{L}$ en el sistema formal de axiomas de la teoría de conjuntos tal que, para una palabra $x \in \mathcal{L}$, la afirmación "$x \in \mathcal{L}$" puede ser demostrada de los axiomas de la teoría de conjuntos en un número de pasos polinomial en $|x|$.*

Denotaremos la clase de lenguajes que tienen la propiedad señalada en el Corolario 1 por NP. Su clase complementaria, es decir la clase de aquellos lenguajes $\mathcal{L}$ para los cuales $\Sigma_0^* \setminus \mathcal{L}$ está en NP forman la clase co-NP. Como mencionamos antes, con estas clases de lenguajes, la verificación de testigos es sencilla, aunque no necesariamente el reconocimiento del lenguaje como tal. Veremos que esta clase es fundamental: contiene una gran parte de los problemas algorítmicos importantes desde el punto de vista de la práctica.

Muchos lenguajes importantes están dados por sus testigos. En lo sucesivo, abusando del lenguaje, llamaremos indistintamente *testigo* al lenguaje $\mathcal{L}_T$ o a alguna de sus palabras que funciona como testigo para alguna palabra $x \in \mathcal{L}$. A menudo, y como es de esperar, nos interesará no sólo la existencia de testigos, sino cómo encontrarlos. Esto se llama el *problema de búsqueda* asociado al lenguaje $\mathcal{L}$. Por supuesto, pueden haber muchos tales problemas para un lenguaje dado. Un problema de búsqueda puede tener sentido aún si su correspondiente problema de decisión es trivial. Por ejemplo, todo número natural tiene una descomposición prima, pero ésta no es fácil de encontrar.

4.3 Resultados generales sobre clases no determinísticas

El siguiente teorema nos presenta la conexión entre las clases determinísticas con sus correspondientes no determinísticas para el caso de tiempo y espacio.

Teorema 13 *Sea f una función buena. Entonces:*

1. $\mathrm{NTIME}(f) \subseteq \mathrm{DSPACE}(f)$,
2. $\mathrm{NSPACE}(f) \subseteq \bigcup_{c>0} \mathrm{DTIME}(2^{cf})$.

Demostración. (1) La esencia de la construcción es que todas las computaciones de una máquina no determinística pueden hacerse sucesivamente una tras otra usando no más espacio que el que necesario para una de ellas. Sólo necesitaremos algún espacio extra para llevar la cuenta de dónde estamos en esta simulación.

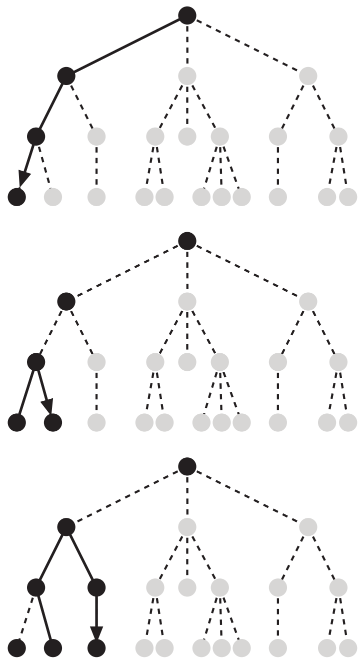

Figura 4.5: Simulación de una computación no determinística por una determinística

Más formalmente: sea T una máquina de Turing no determinística que reconoce $\mathcal{L}$ en tiempo

$f(n)$. Como se mencionó, podemos suponer que todas las computaciones legales de T toman a lo más $f(n)$ pasos, donde n es el largo de la entrada. Modifiquemos T de forma de que para una entrada x elija primero la primera acción en orden lexicográfico (fijamos algún orden en Σ y Γ). Damos a la nueva máquina determinística, llamémosla S, una cinta extra para llevar cuentas en la cual escribe las acciones legales que ha elegido. Si la computación correspondiente (en T) no termina con la aceptación de x, entonces S no se detiene, sino busca en su cinta de cuentas la última acción realizada, y la cambia por la siguiente en el orden lexicográfico.

Es claro que la máquina S intenta todas las computaciones legales de T y usa sólo el espacio necesario para una computación de la máquina T (no más de $f(n)$) más el usado en la cinta de cuentas que no es más que $f(n)$. (Obsérvese que S no es más que simulación de búsqueda en profundidad limitada a profundidad $f(n)$).

(b). Sea $T = \langle k, \Sigma, \Gamma, \Phi \rangle$ una máquina de Turing no determinística que reconoce $\mathcal{L}$ con espacio de almacenamiento $f(n)$. Podemos suponer que T tien una sola cinta. Queremos simular todas las computaciones legales de T. Hay que tener algo de cuidado, pues una computación legal de éstas puede tomar $2^{f(n)}$ pasos, luego pudiesen haber $2^{2^{f(n)}}$ computaciones legales y no tenemos tiempo para verificar todas estas computaciones.

Para organizar mejor la verificación, organizaremos las computaciones en un grafo de la siguiente manera. Fijemos el largo n de las entradas. Por una configuración de la máquina entendemos una tripleta (g, p, h), donde $g \in \Gamma$, $-f(n) \leq p \leq f(n)$ y $h \in \Sigma^{2f(n)+1}$. El estado g es el que tiene la unidad de control en ese momento, el número p dice donde está el cursor y h especifica los símbolos en la cinta (puesto que estamos interesados en computaciones cuya necesidad de espacio es a lo más $f(n)$ es suficiente considerar $2f(n)+1$ celdas).

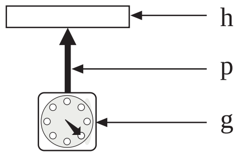

Figura 4.6: Estado g, cursor p, cinta h.

No es difícil deducir entonces que el número de configuraciones es a lo más $|\Gamma|(2f(n)+1)m^{2f(n)+1} = 2^{\mathcal{O}(f(n))}$. Cada configuración puede ser codificada por una palabra de largo $\mathcal{O}(f(n))$ sobre Σ.

Consideremos un grafo dirigido cuyos nodos son las configuraciones. Dibujamos un eje del nodo u al v si la máquina tiene una acción legal que conduzca de la configuración u a la v. Agreguemos un nodo v_{STOP} y dibujemos un eje a v_{STOP} desde cada configuración en la cual la máquina está en estado STOP y tiene un 1 en la celda 0 de su cinta. Denotemos por u_x la configuración correspondiente a la entrada x. La palabra x está en $\mathcal{L}$ si y sólo si en este grafo dirigido un camino dirigido lleva de u_x a v_{STOP}.

En el modelo RAM podemos contruir el grafo G en tiempo $2^{\mathcal{O}(f(n))}$ y (por ejemplo usando búsqueda en anchura) podemos decidir en tiempo $\mathcal{O}(|V(G)|) = 2^{\mathcal{O}(f(n))}$ si el contiene un camino dirigido de u_x a v_{STOP}. Puesto que la RAM puede ser simulada por una máquina de Turing en tiempo cuadrático, la cota de tiempo permanece $2^{\mathcal{O}(f(n))}$ en la máquina de Turing. $\odot$

El siguiente teorema muestra que los requerimientos de espacio no decrecen esencialmente si permitimos no determinismo.

Teorema 14 (Savitch) *Si f es buena y $f(n) \geq \log n$, entonces*

$$\mathrm{NSPACE}(f) \subseteq \mathrm{DSPACE}(f^2).$$

DEMOSTRACIÓN. Consideremos el grafo G de la demostración anterior. Por supuesto, no podemos construir el grafo completo, pues es demasiado grande. Pero lo que necesitamos es esencialmente decidir si este grafo contiene un camino que lleve de u_x a v_{STOP}.

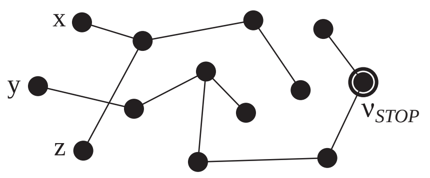

Figura 4.7: Un grafo y posibles caminos

Los miraremos, por consiguiente, como un "oráculo". Esto significa que, dado cualquier par de nodos, podemos decidir en un solo paso si ellos están conectados por un eje. ¿Cuańto cuesta esto? El Lema 7 da la respuesta. De allí podemos deducimos que el problema de decidir si desde un nodo dado u_x existe un camino hacia v_{STOP} puede ser resuelto en espacio que toma codificar una configuración más el llevar las cuentas ($\mathcal{O}(f(n))$, ver demostración anterior) más el costo del oráculo. $\mathcal{O}(f(n)^2)$, ver Lema siguiente (recordemos que el grafo nuestro tiene $2^{f(n)}$ nodos. $\odot$

Lema 7 *Sea G un grafo dirigido con n nodos. Entonces el problema de decidir si entre dos nodos dados hay un camino se puede resolver en espacio $\mathcal{O}((\log n)^2)$.*

DEMOSTRACIÓN. Construiremos una máquina de Turing con dos cintas. Al principio, la primera cinta contiene el par (u, q) y la segunda el par (v, q). El trabajo de la máquina procede por etapas. Al comienzo de algun etapa intermedia, ambas cintas contendrán algunos pocos pares (x, r), donde x es el nombre del nodo y $t \leq q$ es un número natural. (Obsérvese que como tenemos n nodos, podemos enumerarlos y usar la codificación en binario de su número como nombre; luego necesitamos nombres de tamaño no mayor que $\log n$).

Sea (x, r) y (y, s) el primer par en las dos cintas. En la etapa actual, la máquina pregunta si hay un camino de largo a lo más $\min\{2^r, 2^s\}$ de x a y. Si $\min\{r, s\} \geq 1$, entonces sea $m = \min\{r, s\} - 1$. Escribimos un par (w, m) al final de la primera cinta y determinamos recursivamente si hay un par de largo a lo más 2^m de w a y. Si hay uno, entonces escribimos el par (v, m) al final de la segunda cinta, lo borramos del final de la primera cinta, y determinamos si hay un par de largo a lo más 2^m de x a w. Si hay uno, entonces borramos (v, m) del final de la segunda cinta: ya sabemos que hay un camino de largo a lo más $\min\{2^r, 2^s\}$ de x a y. Si no hay camino de largo a lo más 2^m ya sea entre x y w o entre w e y, entonces intentamos con el siguiente w en orden lexicográfico. Si hemos intentado con todos los w's, entonces sabemos que no existe camino de largo $\min\{2^r, 2^s\}$ de x a y.

Es fácil de ver que los segundos elementos de los pares decrecen de izquierda a derecha en ambas cintas, de tal forma que en cada cinta habrán en todo momento a lo más q pares. Un par requiere $\mathcal{O}(\log n + \log q)$ símbolos. El espacio usado es entonces $\mathcal{O}((\log n)^2 + \log n \log q)$. Como $q \leq n$ se obtiene el enunciado del Lema. $\odot$

Como observamos, la clase PSPACE de lenguajes decidibles en máquinas de Turing determinísticas en tiempo polinomial es muy importante. Debido al teorema de Savitch, esta clase es también igual a su complemento, es decir tenemos:

Corolario 2

$$\mathrm{PSPACE} = \mathrm{NPSPACE}.$$

4.4 Ejemplos de Lenguajes en NP

4.4.1 Grafos

En esta sección, llamamos grafo a un *grafo simple*: un grafo no dirigido sin arcos múltiples ni ciclos. El número n denotará siempre el número de nodos. A menos que lo digamos explícitamente, suponemos que el grafo se describe con una matriz de adyacencia, la cual consideraremos como una cadena en $\{0,1\}^{n^2}$. De esta forma, una propiedad de un grafo puede considerarse como un lenguaje sobre $\{0,1\}$. Podemos entonces preguntar si una cierta propiedad de un grafo está en NP.[1] Las siguientes propiedades de los grafos están en NP.

CONECTIVIDAD DE GRAFOS. Un grafo se dice *conexo* si todo par de nodos del grafo está unido por un camino.

El certificado es un conjunto de $n/2$ caminos, uno para cada par de nodos.

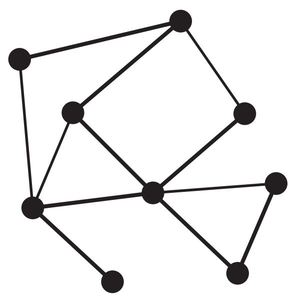

Figura 4.8: Certificado de conectividad

[1]Nótese que describir un grafo de alguna de las otras formas habituales, e.g. dando la lista de vecinos de cada nodo, no afectaría el hecho que las propiedades del grafo estuviesen en la clase NP. Es fácil calcular estas representaciones, una a partir de la otra, en tiempo polinomial.

NO-CONECTIVIDAD DE GRAFOS. La negación de la propiedad anterior. El certificado es un subconjunto propio del conjunto de nodos que no está conectado por ningún arco al resto de los nodos.

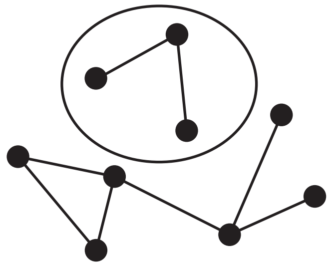

Figura 4.9: Certificado de No-conectividad

Planaridad. Un grafo es *planar* si es posible dibujarlo en un plano sin que se corte ningún par de arcos.

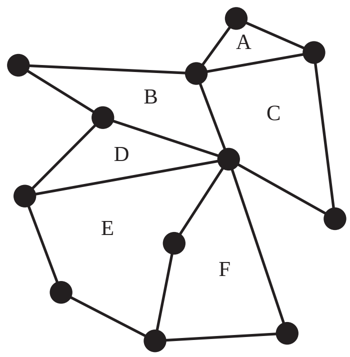

Figura 4.10: Certificado de planaridad

El testigo natural, entonces, es un diagrama (dibujo) concreto. Pero es necesario cierto análisis para ver que en caso que dicho diagrama exista, entonces existe también un diagrama en que las coordenadas de todos los nodos son números enteros cuyo número de bits es polinomial en n.

Pero falta la representación de los arcos. Es interesante destacar el hecho conocido en teoría de grafos que el diagrama puede realizarse usando solamente segmentos de línea recta para los arcos y por lo tanto es suficiente especificar las coordenadas de los nodos. Debemos ser cuidadosos, sin embargo, dado que las coordenadas de los nodos que se usan en el dibujo pueden necesitar demasiados bits en su descripción, violando entonces el requisito sobre la longitud del testigo.[2]

Es posible también dar una forma puramente combinatorial de dibujar el grafo. Sea G un grafo con n nodos y m arcos, que suponemos por simplicidad que es conexo. Una vez dibujado en el plano, los arcos particionan el plano en dominios que pueden llamarse "países" (el dominio ilimitado es también un país). Necesitamos el siguiente hecho, conocido como la *fórmula de Euler*:

Teorema 15 *Un grafo planar conexo con n nodos y m arcos tiene $n + m - 2$ países.*

Usando este resultado, para especificar el dibujo damos un conjunto de $m - n + 2$ nombres de países y para cada país especificamos una secuencia de arcos que forman sus límites.

El hecho de que exista este conjunto de secuencias de arcos es una condición necesaria de planaridad que se deduce de la fórmula de Euler. Probar la suficiencia de esta condición requiere herramientas más sofisticadas de topología; no entraremos en estos detalles.[3]

NO-PLANARIDAD. Esta es la negación de la propiedad de planaridad. ¿Que testigo tenemos aquí? Revisemos los siguientes hechos.

dibujo de K5

1. Sea K_5 el grafo obtenido mediante la conexión de cinco nodos en todas las formas posibles. Este grafo también se llama un "pentágono completo". Sea K_3^3 el grafo bipartito de 6 nodos que contiene dos conjuntos A y B de tres nodos cada uno con todos los arcos posibles entre A y B. Este grafo también se llama "tres casas, tres paredes" por cierto puzzle con ese nombre. Es fácil ver que K_5 y K_3^3 no son planares.

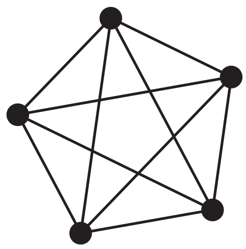

Figura 4.11: El grafo K_5

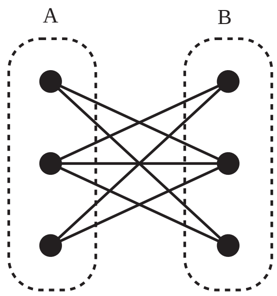

Figura 4.12: El grafo $K_{3,3}$

[2]Afortunadamente puede probarse que todo grafo planar puede dibujarse en el plano de modo que cada arco es un segmento de línea recta y las coordenadas de cada nodo son números enteros cuyo número de bits es polinomial en n.

[3]La especificación de un conjunto de secuencias de arcos como los límites de un país corresponde a definir una superficie de dos dimensiones con el grafo dibujado sobre ella. Un teorema de topología dice que si un grafo conexo dibujado sobre esta superficie satisface la fórmula de Euler, entonces la superficie es topológicamente equivalente (homomórfica) al plano.

2. Dado un grafo G, decimos que un grafo G' es una *subdivisión* de G si se obtiene remplazando en G los arcos por caminos arbitrariamente largos que no se intersectan. Entonces es fácil ver que si G no es planar, entonces cualquier subdivisión de él es también no planar.

3. Si un grafo no es planar, entonces obviamente, todo grafo que lo contenga es también no planar.

El siguiente teorema fundamental de la teoría de grafos dice que los grafos no planares son solamente aquellos obtenidos con el procedimiento anterior:

Teorema 16 (Kuratowski) *Un grafo es no planar si y sólo si contiene un subgrafo que es una subdivisión de K_5 o K_3^3.*

Si un grafo es no planar, entonces el subgrafo cuya existencia establece el teorema de Kuratowski puede servir para certificar esto.

EXISTENCIA DE LA CORRESPONDENCIA PERFECTA. Un conjunto M de arcos independientes (i.e. que no tienen nodos en común) se llama una *correspondencia (matching)*. M es una correspondencia del subconjunto de nodos U si todo nodos de U es incidente con un arco de M. Una correspondencia M es *perfecta* si todo nodo del grafo es incidente con algún arco de M.

Entonces un certificado de esta propiedad es la correspondencia perfecta en sí.

INEXISTENCIA DE LA CORRESPONDENCIA PERFECTA. Estudiemos primero posibles testigos de esta inexistencia en caso de grafos bipartitos. Sea G un grafo bipartito que consiste en la bipartición en las clases U y W. Necesitaremos el siguiente resultado:

Teorema 17 (Frobenius) *Un grafo bipartito G tiene una correspondencia perfecta si y sólo si $|U| = |W|$ y para todo k, todo conjunto de k nodos en W siempre tiene al menos k vecinos en U.*

Por lo tanto, si en algún grafo bipartito no hay una correspondencia perfecta, entonces esto puede certificarse o bien constatando que U y W tienen diferente cardinalidad, o bien encontrando un subconjunto de W que viole las condiciones del teorema.

Consideremos ahora el caso cuando G es un grafo arbitrario. Si hay una correspondencia perfecta, entonces es fácil ver que para cualquier k, si quitamos k nodos cualesquiera, quedarán a lo sumo k componentes conexas de tamaño impar. El siguiente teorema fundamental (y más profundo que el anterior) dice que esta condición no solamente es necesaria para la existencia de una correspondencia perfecta, sino que además es suficiente.

Teorema 18 (Tutte) *Un grafo G tiene una correspondencia perfecta si y sólo si para cualquier k, si quitamos k nodos cualesquiera, quedarán a lo sumo k componentes conectadas de tamaño impar.*

De esta forma, si no hay una correspondencia perfecta en el grafo, esto puede certificarse mediante un conjunto de nodos tales que al quitarlos se crean demasiadas componentes de tamaño impar.

EXISTENCIA DE UN CICLO HAMILTONIANO. Recordemos que un *ciclo Hamiltoniano* en un grafo es un ciclo que recorre cada nodo exactamente una vez.

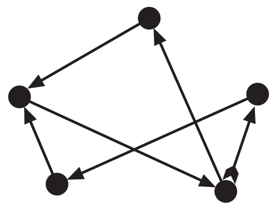

Figura 4.13: Un ciclo hamiltoniano

Entonces un certificado de esta propiedad es el ciclo Hamiltoniano en sí.

COLOREAMIENTO CON TRES COLORES. Un *coloreamiento* de un grafo es la asignación de algún símbolo llamado "color" a cada nodo de tal modo que todos los nodos vecinos tengan un color diferente.

Si un grafo puede ser coloreado con tres colores, entonces el coloreamiento en sí es un certificado. Por cierto, ésto es válido para cualquier número de colores.

Nota. La mayor parte de las propiedades antes mencionadas, hasta la no existencia de una coincidencia perfecta inclusive, pueden ser verificadas en tiempo polinomial (i.e. están en la clase P). Para los problemas del ciclo Hamiltoniano y el coloreamiento con tres colores, sin embargo, no se conoce ningún algoritmo polinomial (volveremos sobre ésto más adelante).

4.4.2 Aritmética y álgebra

Para mostrar que muchos problemas fundamentales de la aritmética y el álgebra también pertenecen a la clase NP, recordemos que todo número natural puede considerarse como una palabra en $\{0,1\}^*$ (que representa el número en binario). Comenzamos con el problema de decidir si un número natural es primo.

PRIMALIDAD. ¿Es primo o no? Es importante hacer notar la diferencia de este problema con el de factorización. El problema de *primalidad* es un problema de decisión, es decir, requiere respuesta Si o No. El problema de *factorización* pide como respuesta los factores (en caso de no ser primo).

Para la propiedad de no primalidad, es decir, para determinar si un número es compuesto, un certificado es un divisor propio.

Para primalidad es sustancialmente más difícil encontrar un testigo. Para ello usaremos el siguiente teorema fundamental de teoría de números:

Teorema 19 *Un entero $n \geq 2$ es primo si y sólo si existe un número natural a tal que $a^{n-1} \equiv 1 \pmod{n}$ pero $a^m \not\equiv 1 \pmod{n}$ para ningún m tal que $1 \leq m < n-1$.*

Este teorema dice que existe una llamada *raíz primitiva* a para n, cuyas potencias recorren todos los residuos de *mod* n distintos de 0.

Con este teorema en mente, nos gustaría usar el número a como testigo de la primalidad de n. Dado que obviamente solamente el resto del número a después de la división por n tiene relevancia aquí, habrá otro testigo a con $1 \leq a \leq n$. De esta forma, la restricción sobre la longitud del testigo se satisface: a no tiene más bits que n. Sea k el número de bits de n.

Como hemos visto en la sección sobre tiempo polinomial, también podemos comprobar en tiempo polinomial la condición:

$$a^{n-1} \equiv 1 \pmod{n}. \tag{4.2}$$

Sin embargo, verificar las otras condiciones es una tarea mucho más difícil:

$$a^m \not\equiv 1 \pmod{n} \quad (1 \leq m < n-1). \tag{4.3}$$

Hemos visto que podemos hacer esto para un m específico, pero parece que debemos hacerlo $n-2$ veces, i.e. un número exponencial de veces en términos de k (recordemos que k es el tamaño de la entrada del problema). Sin embargo, podemos usar el (sencillo) hecho de teoría de números que dice que si se satisface la ecuación (4.2), entonces el menor $m = m_0$ que viola la ecuación (4.3) (si existe alguno) es un divisor de $n-1$. Es también fácil observar que entonces la ecuación (4.3) la violan todos los múltiplos de m_0 que sean menores que $n-1$. Por lo tanto, si la descomposición en factores primos de $n-1$ es $n-1 = p_1^{r_1} \ldots p_t^{r_t}$, entonces la ecuación (4.3) es violada por algunos $m = (n-1)/p_i$. Es suficiente entonces verificar que para todo i con $1 \leq i \leq t$,

$$a^{(n-1)/p_1} \not\equiv 1 \quad (mod\, n).$$

Ahora resulta obvio que $t \leq k$ y por lo tanto tenemos que comprobar si la ecuación (4.3) se satisface para a lo sumo k valores, lo cual puede hacerse de la forma antes descrita, en un tiempo polinomial.

Pero existe aún otra dificultad: ¿cómo haremos para calcular la descomposición en números primos de $n-1$? Este es un problema más difícil en sí mismo que determinar si n es primo. Podemos agregar la descomposición en números primos de $n-1$ a los "testigos"; esta consta, además del número a, de los números $p_1, r_1, \ldots, p_t, r_t$ (es fácil ver que esto tiene a lo sumo $3k$ bits). Ahora sólo resta comprobar que ésta es una descomposición en factores primos, i.e. que $n-1 = p_1^{r_1} \ldots p_t^{r_t}$ (esto es fácil) y que $p_1, \ldots p_t$ son efectivamente primos. Para esto último, ¡usamos la misma idea recursivamente!

Aún resta comprobar que esta recursión da testigos de longitud polinomial y que puede decidirse en tiempo polinomial que éstos son testigos. Sea $L(k)$ la máxima longitud de los testigos en caso de números n de k bits. Entonces, de acuerdo con la recursión anterior:

$$L(k) \leq 3k + \sum_{i=1}^{t} L(k_i),$$

donde k_i es el número de bits del número primo p_i. Dado que $p_1 \ldots p_t \leq n-1 < n$, podemos concluir fácilmente que

$$k_1 + \cdots + k_t < k + t.$$

También obviamente $k_i < k$. Usando esto, se deduce de la recursión anterior que $L(k) \leq 3k^2$. De hecho, es más fácil de probar la desigualdad $L(k) \leq 3(k-1)^2$. Esto es obvio para $k = 1, 2$, y si sabemos que es válida para todos los números menores que k, entonces

$$\begin{aligned} L(k) &\leq 3k + \sum_{i=1}^{t} L(k_i) \\ &\leq 3k + \sum_{i=1}^{t} 3(k_i - 1)^2 \\ &\leq 3k + 3(k-2) \sum_{i=1}^{t} k_i - 1 \\ &\leq 3k + 3(k-2).(k-1) \leq 3(k-1)^2. \end{aligned}$$

Del mismo modo podemos probar que el problema de determinar si una cadena es un certificado definido del modo anterior es decidible en tiempo polinomial.

Dijimos al comenzar que el problema de Primalidad era diferente al de Factorización. En general no es suficiente saber si un número n es primo o no (Primalidad), sino que si no es primo queremos además obtener uno de sus divisores propios. Si podemos resolver este último problema, podremos hallar toda la descomposición (Factorización) con sólo repetir el procedimiento. Este no es un problema de decisión, pero puede ser reformulado como un problema de decisión:

Problema 5 Existencia de un divisor acotado
ENTRADA. *Dos números naturales n y k.*
RESPUESTA. *Si o No, de acuerdo a si n tiene un divisor propio menor o igual que k.*

Es claro que este problema está en la clase NP: el certificado es el divisor.

El problema complementario está también en NP:

NO EXISTENCIA DE UN DIVISOR ACOTADO El lenguaje asociado a este problema es el conjunto de todos los pares (n, k) tal que todo divisor propio de n es mayor que k. Un certificado de esto es la descomposición prima de n, conjuntamente con un certificado de primalidad de cada factor primo.

Nota 10 *No se sabe si el problema de composición (aún menos, el de la existencia de un divisor acotado) está en* P. *Extendiendo la noción de algoritmo y usando números aleatorios, es posible decidir en tiempo polinomial si un número es primo (ver la sección de algoritmos aleatorios). Al mismo tiempo, el problema de buúsqueda asociado (búsqueda de un divisor propio), o en forma equivalente, decidir acerca de la existencia de divisores acotados, es significativamente más difícil; por ello, aún no se ha encontrado un algoritmo polinomial ni siquiera usando números aleatorios.*

Volvamos ahora a ciertos asuntos básicos del álgebra. Una noción análoga a la de primalidad de un entero positivo es la irreductibilidad de un polinomio (con una sola variable y con coeficientes racionales por simplicidad). Un polinomio es *reducible* si puede escribirse como el producto de dos polinomios no constantes con coeficientes racionales.

REDUCIBILIDAD DE UN POLINOMIO SOBRE UN CUERPO RACIONAL Para este problema, un certificado es un divisor propio; pero algunos comentarios son necesarios.

Sea f un polinomio. Para probar que este problema está en NP debemos estar seguros de que el número de bits necesarios para escribir un divisor propio puede ser acotado por un número de bits de tamaño polinomial en la representación de f. (Omitimos la demostración.)

Más aún, se puede demostrar que este lenguaje está en P.

4.4.3 Sistemas de Inecuaciones Lineales

Un sistema de inecuaciones lineales $Ax \leq b$ (donde A es una matriz de enteros con m filas y n columnas y b es un vector con m elementos) puede considerarse una palabra sobre el alfabeto de símbolos "0", "1", "," y ";" cuando representamos sus elementos en el sistema de números binarios, escribimos la matriz fila tras fila con una coma después de cada número y un punto y coma después de cada fila. Las siguientes propiedades de los sistemas de inecuaciones lineales están en NP.

Existencia de Solución. La sóla existencia de una solución se ofrece como un testigo de la solubilidad, pero debemos ser cuidadosos: debemos estar convencidos de que si el sistema de ecuaciones lineales con coeficientes enteros tiene una solución, entonces tiene una solución en los números racionales, y más aún, una solución donde los numeradores y los denominadores de cada número tienen solamente un número polinomial de bits. Estos hechos se deducen de los elementos de la teoría de la programación lineal.

NO EXISTENCIA DE SOLUCIÓN. Podemos encontrar un testigo de la no existencia de solución usando el siguiente conocido teorema fundamental de la programación lineal:

Lema 8 (Farkas) *El sistema de inecuaciones $Ax \leq b$ no es resoluble si y sólo si el siguiente sistema de inecuaciones es soluble: $y^T A = 0$, $y^T b = -1$, $y \geq 0$.*

En palabras, este lema dice que un sistema de inecuaciones lineales no es resoluble si y sólo si podemos obtener una contradicción mediante una combinación lineal de las inecuaciones con coeficientes no negativos.

Usando esto, la existencia de una solución a este otro sistema de inecuaciones del lema (con coeficientes no negativos) es un testigo de la inexistencia de una solución.

EXISTENCIA DE UNA SOLUCIÓN ENTERA. (Es decir, una solución en números enteros.) La solución en sí es un testigo, pero necesitamos algún razonamiento una vez más para limitar el tamaño de los testigos, que aquí es más difícil.

No se sabe si la inexistencia de una solución entera está en NP, i.e., si este hecho puede comprobarse mediante un certificado de largo polinomial y en tiempo polinomial.

Es importante notar que el problema fundamental de la programación lineal, i.e., buscar el óptimo de una función lineal objetivo sujeta a una serie de condiciones lineales, puede reducirse fácilmente al problema de resolubilidad de sistemas de inecuaciones lineales. En forma similar, la búsqueda de soluciones óptimas puede reducirse a decidir la existencia de una solución entera.

Por largo tiempo no se sabía si el problema de la resolubilidad de un sistema de inecuaciones lineales estaba en P (el conocido método simplex no es polinomial). El primer algoritmo polinomial para este problema fue el método del elipsoide de L. G. Khachian (basado en trabajos de Yudin y Nemirovskii).

El tiempo de ejecución de este método nos ha llevado sin embargo a un polinomio de grado muy alto; no puede entonces competir en la práctica con el método simplex, el cual, a pesar de ser exponencial en el peor de los casos, en promedio (como lo muestra la experiencia) es mucho más rápido que el método del elipsoide.

Se han encontrado muchos algoritmos para programación lineal de tiempo polinomial desde entonces; entre ellos, el método de Karmarkar puede competir con el simplex en la práctica.

No existe ningún algoritmo polinomial conocido para resolver sistemas de ecuaciones lineales enteras; no se puede siquiera tener la esperanza de encontrar dicho algoritmo (ver la noción de NP-completitud más adelante).

Revisando la lista de ejemplos anteriores podemos hacer las siguientes observaciones:

- Para muchas propiedades que están en NP, su negación (i.e. el complemento del correspondiente lenguaje) está también en NP.

Sin embargo, este hecho es generalmente no trivial; en varias ramas de las matemáticas, frecuentemente los teoremas más fundamentales afirman esto para ciertos lenguajes.

- Frecuentemente sucede que si cierta propiedad (lenguaje) resulta estar en NP $\cap$ co-NP entonces tarde o temprano resulta estar en P. Esto sucedió, por ejemplo, con la existencia de la correspondencia perfercta, la planaridad, la solución de los sistemas lineales de inecuaciones. La investigación es intensa en el caso de Primalidad. Si se considera NP como un análogo de "recursivamente enumerable" y P como un análogo de "recursivo" entonces podemos esperar que esto siempre sea así. Sin embargo, no existe una demostración de esto; más aún, esto no puede esperarse que sea realmente una verdad general.

- Con otros problemas en NP, su solución en tiempo polinomial parece no tener esperanzas porque son muy difíciles de manejar (ciclos Hamiltonianos, coloreamiento de grafos, y soluciones enteras a sistemas de inecuaciones lineales). No podemos probar que estos problemas no están en P (no sabemos si realmente P = NP); pero aún pueden probarse ciertas propiedades exactas de estos problemas que muestran que son difíciles. Volveremos sobre esto más adelante.

- Hay muchos problemas en NP con un problema de búsqueda naturalmente asociado y con la propiedad de que si podemos resolver el problema de decisión entonces tambien podemos resolver (de una forma natural) el problema de búsqueda. Por ejemplo, si podemos decidir si hay una correspondencia perfecta en cierto grafo, entonces podemos buscar la correspondencia perfecta en tiempo polinomial de la siguiente forma: eliminamos los arcos del grafo de modo que aún conserve la correspondencia perfecta. En cuanto nos quedamos atascados, el grafo restante será la correspondencia perfecta. Usando trucos simples similares, el problema de búsqueda asociado a la existencia de ciclos Hamiltonianos, colorabilidad con 3 colores, etc. puede reducirse a un problema de decisión. Sin embargo, esto no siempre es así. Por ejemplo, nuestra capacidad de decidir en tiempo polinomial (al menos en algún sentido) si un número es primo no era aplicable al problema de encontra un divisor propio.

- Una serie de problemas en NP tiene un *problema de optimización* relacionado el cual es más fácil de formular, aún si no es un problema NP por su forma. Por ejemplo, en lugar del problema general de correspondencia, es más fácil decir que el problema consiste en encontrar el máximo tamaño de una correspondencia del grafo. En el caso del problema del coloreamiento, podemos buscar el número cromático, el menor número de colores con los cuales se puede colorear el grafo. La resolubilidad de un conjunto de inecuaciones lineales está íntimamente relacionado con el problema de encontrar una solución que maximice cierta forma lineal: este es el problema de la programación lineal. Habrá muchos otros ejemplos más adelante. Si hay un algoritmo polinomial que resuelve el problema de optimización, entonces automáticamente también resuelve el problema NP asociado. Si existe un algoritmo polinomial que resuelva el problema NP, entonces usando búsqueda binaria, proporcionaría un algoritmo polinomial para resolver el problema de optimización asociado.

Por cierto que hay problemas (lenguajes) interesantes también en otras clases de complejidad no determinísticas. La clase *tiempo exponencial no determinístico* (NEXPTIME) puede definirse como la unión de las clases NTIME(2^{n^c}) para todo $c > 0$. Podemos formular un ejemplo en conexión con el teorema de Ramsey. Sea G un grafo; el *número de Ramsey* $R(G)$ del grafo G es el menor $N > 0$ para el cual no importando cómo coloreemos los arcos del grafo completo de N nodos con dos colores, algún color contiene una copia de G. (El teorema de Ramsey dice que este número finito existe, lo cual es no trivial). Sea $\mathcal{L}$ formado por el par (G, N) para los cuales $R(G) > N$. El tamaño de la entrada (G, N) (si G se describe por su matriz de adyacencia) es $O(|V(G)|^2 + \log N)$.

Ahora bien, $\mathcal{L}$ está en NEXPTIME dado que $(G, N) \in \mathcal{L}$ tiene un testigo en el coloreamiento del grafo completo con N nodos en los cuales no hay una copia homogéneamente coloreada de G; esta propiedad puede comprobarse en tiempo $O(N^{|V(G)|})$ el cual es exponencial en el tamaño de la entrada (pero no peor). Por otra parte, no conocemos ningún otro algoritmo mejor para decidir determinísticamente que $(G, N) \in \mathcal{L}$ que uno doblemente exponencial. El algoritmo trivial, el cual desafortunadamente es el más conocido, recorre todos los coloreamientos de los arcos del grafo completo de N nodos, y este número es $2^{N(N-1)/2}$.

Falta agregar problemas

Capítulo 5

Complejidad y Aleatoriedad

5.1 Introducción

La Tesis de Church afirma que cada "algoritmo" (en el sentido informal de la palabra) es realizable en una máquina de Turing. Vimos también varios otros modelos de computación propuestos, y como resultó que todos ellos resolvían exactamente la misma clase de problemas.

Sin embargo, hay una extensión de la noción de algoritmo (luego de máquina) que es más poderosa que una máquina de Turing, y aún así realizable en el mundo real: la noción de *algoritmo aleatorio.* Aquí permitimos "tirar una moneda al aire", i.e., tener acceso a un generador de números aleatorios. Dichos algoritmos son capaces de resolver problemas que las máquinas de Turing no pueden resolver (vamos a formular y probar esto de forma exacta más adelante), y en muchos casos, resolver más eficientemente aquellos realizables por máquinas de Turing. Comenzaremos con una discusión de esos ejemplos. El más simple es el de la aplicación de la aleatoriedad a la verificación de identidades algebraicas; el más importante es sin duda el determinar si un entero dado es primo o no.

Puesto que de esta forma obtenemos un concepto matemático nuevo, más fuerte, de máquina, tiene sentido introducir los conceptos correspondientes de clases de complejidad aleatoria. Trataremos algunas de las más importantes al final del capítulo.

5.2 Verificación de una identidad polinomial

Sea $f(x_1, x_2, ..., x_n)$ un polinomio de n variables con coeficientes racionales y de grado no mayor que k. Nos gustaría decidir en qué casos f es idénticamente 0 (como una función de n variables). Sabemos del álgebra clásica que un polinomio es idénticamente 0 si y sólo si, después de "desarrollar los paréntesis", todos los términos se cancelan. Este criterio, sin embargo, no siempre es útil. Es posible por ejemplo que se nos presente un polinomio como $(x_1 + y_1)(x_2 + y_2) \cdots (x_n + y_n) + 1$. En este caso, el "desarrollar los paréntesis" produciría un número exponencial de términos. Otro ejemplo es cuando un polinomio se nos presenta como el cálculo de un determinante (ver siguiente subsección), que como hemos visto se calcula usualmente de alguna manera especial.

Un método alternativo es reemplazar las variables $x_1, \ldots, x_n$ por números aleatorios $\xi_1, \ldots, \xi_n$ y calcular el valor de $f(\xi_1, \ldots, \xi_n)$. Si se tiene que $f(\xi_1, \ldots, \xi_n) \neq 0$, entonces naturalmente el polinomio f no puede ser idénticamente 0. Al contario, si el valor calculado es 0, entonces aún podría ocurrir que el polinomio no sea idénticamente 0. Ahora bien, si sabemos que encontrar una de sus raíces tiene una probabilidad muy baja, entonces podríamos concluir que el polinomio es idénticamente 0 con una probabilidad baja de haber cometido un error. Este es el principio que expresa el siguiente teorema. Obsérvese que debido a que las variables son discretas (valores enteros elegidas en $[0, N]$) no podemos obtener un resultado acerca de probabilidad de error 0, sino solamente acerca de probabilidad pequeña cuando N es suficientemente grande.

Teorema 20 (Lema de Schwartz) *Sea f un polinomio no idénticamente nulo, en n variables, con coeficientes en $\mathbb{Q}$ y de grado*[1] *a lo más k. Si $\xi_1, \ldots, \xi_n$ son números enteros elegidos en el intervalo $[0, N-1]$ aleatoria e independientemente de acuerdo a la distribución uniforme, entonces*

$$\Pr\{f(\xi_1, \ldots, \xi_n) = 0\} \leq \frac{k}{N}.$$

DEMOSTRACIÓN. Inducción sobre n. El enunciado es verdadero para $n = 1$ dado que un polinomio en una variable de grado k puede tener a lo más k raíces. Sea $n > 1$. Ordenemos f de acuerdo a las potencias de x_1: $f = f_0 + f_1x_1 + f_2x_1^2 + \cdots + f_tx_1^t$, donde $f_0, \ldots, f_t$ son polinomios en las variables $x_2, \ldots, x_n$, el término f_t no es idénticamente 0, y $t \leq k$. Entonces, denotando $\xi_j, \ldots, \xi_n$ por $\bar{\xi}_{jn}$, se tiene:

$$\begin{aligned}
&\Pr\{f(\bar{\xi}_{1n}) = 0\}\\
&\quad\leq \Pr\{f(\bar{\xi}_{1n}) = 0 | f_t(\bar{\xi}_{2n}) = 0\} \cdot \Pr\{f_t(\bar{\xi}_{2n}) = 0\}\\
&\qquad + \Pr\{f(\bar{\xi}_{1n}) = 0 | f_t(\bar{\xi}_{2n}) \neq 0\} \cdot \Pr\{f_t(\bar{\xi}_{2n}) \neq 0\}\\
&\quad\leq Pr\{f_t(\bar{\xi}_{2n}) = 0\} + \Pr\{f(\bar{\xi}_{1n}) = 0 | f_t(\bar{\xi}_{2n}) \neq 0\}.
\end{aligned}$$

[1] El grado de un polinomio en varias variables está definido como el grado mayor de sus términos (monomios); el grado de un monomio es la suma de los exponentes de las variables en él.

Usando la hipótesis de inducción y el hecho de que el grado de f_t es a lo más $k - t$, podemos estimar cada uno de los términos. El primero es a lo más $(k-t)/N$. El segundo es a lo más t/N, puesto que ξ_1 es independiente de las variables $\xi_2, \ldots, \xi_n$, y luego, no importando cómo estén fijados de tal forma que $f_t \neq 0$ (por lo tanto f, como polinomio de x_1, no es idénticamente 0), la probabilidad de que ξ_1 es su raíz es a lo más t/N.

La conclusión del teorema se sigue inmediatamente. ⊙

Algoritmo 1 *Calculamos $f(\xi_1, \ldots, \xi_n)$ con valores enteros ξ_i escogidos aleatoria e independientemente unos de otros de acuerdo a la distribución uniforme en el intervalo $[0, 2kn]$. Si no obtenemos el valor 0, paramos: f no es idénticamente 0. Si obtenemos el valor 0, repetimos el cómputo. Si obtenemos el valor 0 100 veces, paramos y declaramos que f es idénticamente 0.*

Si f es idénticamente 0 entonces este algoritmo lo determinará. Si f no es idénticamente 0, entonces en cada iteración separada–de acuerdo al lemma de Schwartz–la probabilidad de que el resultado sea 0 es menor a $1/2$. Con 100 experimentos repetidos independientemente unos de otros, la probabilidad de que esto ocurra todas las veces, es decir, de que el algoritmo asegure erróneamente que f es idénticamente 0, es menor a 2^{-100}.

Para ser capaces de llevar a cabo este algoritmo en tiempo razonable necesitamos dos cosas: (1) ser capaces de generar números aleatorios eficientemente (aquí asumimos que esto puede ser implementado en tiempo polinomial en el número de bits de los enteros a generar), y (2) saber evaluar f en tiempo polinomial en el largo de la entrada, que es el de la definición de f; esta definición puede ser una expresión conteniendo multiplicaciones y sumas con paréntesis, pero también algo completamente diferente, por ejemplo un determinante.

5.3 Correspondencia perfecta en grafos

Como un ejemplo de la aplicación del método presentamos un algoritmo de correspondencia (matching). (Ya hemos tratado el problema de correspondencia en los capítulos ?? y ??). Sea G un grafo bipartito y $E(G)$ su conjunto de aristas, donde las aristas están dentro de los conjuntos $A = \{a_1, a_2, \ldots, a_n\}$ y $B = \{b_1, b_2, \ldots, b_n\}$. Asignamos a cada arista a_ib_j una variable x_{ij} y construimos la matriz M de $n \times n$ cuyos elementos son:

$$m_{ij} = \begin{cases} x_{ij} & \text{si } a_ib_j \in E(G), \\ 0 & \text{en caso contrario.} \end{cases}$$

El determinante de esta matriz está estrechamente relacionado con las correspondencias del grafo G:

Teorema 21 (Frobenius-König) *Un grafo bipartito tiene una correspondencia perfecta si y sólo si* $\det(M)$ *no es idénticamente* 0.

DEMOSTRACIÓN. Consideremos un término en el desarrollo del determinante de la matriz $M = (m_{ij})$

$$\pm m_{1\pi(1)} m_{2\pi(2)} \cdots m_{n\pi(n)},$$

donde π es una permutación de los números $1, \ldots, n$. Para que este término no sea 0, necesitamos que los vértices a_i y $b_{\pi(i)}$ estén conectados para todo i; en otras palabras, que $\{a_1b_{\pi(1)}, \ldots, a_nb_{\pi(n)}\}$ sea una correspondencia perfecta en G. De esta forma, si no hay correspondencias perfectas en G entonces el determinante es idénticamente 0. Si hay correspondencias perfectas en G, entonces a cada uno de ellos le corresponde un término diferente de cero del desarrollo . Dado que estos términos no se cancelan unos con otros (cualquier pareja tiene al menos dos variables distintas), el determinante no es idénticamente 0. ⊙

Dado que $\det(M)$ es un polinomio de los elementos de la matriz M que es computable en tiempo polinomial (por ejemplo usando eliminación Gausiana), este teorema ofrece un algoritmo aleatorio de tiempo polinomial para el problema de correspondencia en grafos bipartitos. Mencionamos anteriormente que también hay un algoritmo determinístico de tiempo polinomial para este problema (el método húngaro). Una ventaja del algoritmo aquí tratado es que es muy fácil de programar (el cáluclo de determinantes se encuentra usualmente en bibliotecas de programas). Si usamos métodos rápidos de multiplicación de matrices, entonces este algoritmo aleatorio es algo más rápido que el determinístico más rápido: puede ser completado en tiempo $\mathcal{O}(n^{2,4})$ en vez de $\mathcal{O}(n^{2,5})$. Su principal ventaja es, sin embargo, que es adecuado para su paralelización, como veremos en una sección posterior.

El método anterio se puede generalizar a grafos no-bipartitos. Sea $V = \{v_1, \ldots, v_n\}$ el conjunto de vértices del grafo G. Asignamos nuevamente a cada arco v_iv_j (donde $i < j$) una variable x_{ij} y construimos una matriz asimétrica $T = (t_{ij})$ de $n \times n$ como sigue:

$$t_{ij} = \begin{cases} x_{ij} & \text{si } v_iv_j \in E(G) \text{ y } i < j, \\ -x_{ij} & \text{si } v_iv_j \in E(G) \text{ y } i > j, \\ 0 & \text{en los otros casos.} \end{cases}$$

El siguiente teorema de Tutte que generaliza el de Frobenius-König hace el resto del trabajo. El algoritmo es el mismo mencionado anteriormente.

Teorema 22 *Un grafo tiene una correspondencia perfecta si y sólo si* $\det(T)$ *no es idénticamente* 0.

5.4 Verificación de primalidad

Sea m un número natural impar; queremos decidir si m es primo. Vimos en el capítulo anterior que este problema está en NP $\cap$ co-NP. Los testigos descritos entonces no nos llevaron, sin embargo, (al menos en ese momento) a un algoritmo de verificación de primos en tiempo polinomial.

Por lo tanto daremos primero una nueva descripción de la composición, más complicada, que forma la esencia de los métodos probabilísticos para verificar primalidad. Recordemos que a, b son relativamente primos si no tienen divisores en común.

Teorema 23 (Pequeño Teorema de Fermat) *Si m es un primo entonces $a^{m-1} - 1$ es divisible por m para todos los números naturales $a < m$ relativamente primos con m.*

Decimos que la pareja (m, a) satisface la *condición de Fermat* si y sólo si $a^{m-1} - 1$ es divisible por m. Con esta notación, el Pequeño Teorema de Fermat dice que si m es primo entonces todos los números $a = 1, \ldots, m-1$ forman parejas (m, a) que satisfacen la condición de Fermat.

Más aún, (1) si m no es primo entonces la *mayoría* de los a violan la condición de Fermat.[2] Además, hagamos notar que (2) verificar que una pareja (m, a) satisface o no la condición de Fermat es posible hacerlo en tiempo polinomial (esto fue discutido en el capítulo ??). Los hechos (1) y (2) proveen las ideas esenciales para un algoritmo aleatorio polinomial para verificar si m es primo: *elegir a aleatoriamente y determinar si (m, a) satisface la condición de Fermat.*

Desgraciadamente, los números "raros" a que hicimos referencia complican una idea tan sencilla. Estos son números m que no siendo primos, se comportan exactamente igual que los números primos respecto del pequeño teorema de Fermat, es decir, tienen la propiedad de que para todos los números a que son relativamente primos con m la pareja (m, a) satisface la condición de Fermat. (Un tal ejemplo es $561 = 3 \cdot 11 \cdot 17$.) Por esta razón, a esto números se les denomina *pseudo-primos.*

Lema 9 *Si m no es primo ni pseudo-primo, entonces a lo más la mitad de los enteros a $(1 \leq a < m)$ relativamente primos con m satisfacen la condición de Fermat. (Nótese que ninguno de los a relativamente primo con m satisface la condición de Fermat.)*

DEMOSTRACIÓN. Dado que m no es un pseudo-primo, existe al menos un b relativamente primo a m tal que $b^{m-1} - 1$ no es divisible por m. Ahora si a es "malo", i.e., $a^{m-1} - 1$ es divisible por m, entonces hay otro a (por ejemplo $ab \mod m$) que es "bueno":

$$(ab)^{m-1} - 1 = (a^{m-1} - 1)b^{m-1} + b^{m-1} - 1,$$

y aquí el primer término es divisible por m pero el segundo no.

Dado que para cada a "malo" encontramos otro a (a saber $ab \mod m$) que es "bueno". Es fácil ver que diferentes a's dan diferentes números "buenos". Luego al menos la mitad de los números a deben ser buenos. ⊙

Luego, si m no es un pseudoprimo, obtenemos el siguiente algoritmo aleatorio para la verificación de primos: revisar si un entero a $(1 \leq a < m)$ elegido aleatoriamente satisface la condición de Fermat. Si no, entonces sabemos que m no es primo. Si la satisface, repetimos el procedimiento. Entonces, si repetimos un número grande de veces este procedimiento, independiente uno de otro, y todas las veces se satisface la condición de Fermat, podemos afirmar que m es primo con bastante seguridad. De hecho, con k repeticio-

[2] Para ser rigurosos, como veremos en el Lema 9, hay algunos pocos números "raros" que no cumplen esto.

nes, la probabilidad de error es 2^{-k} (1/2 en cada repetición).

La mala noticia es que este método falla cuando m es pseudo-primo, pues todas las iteraciones pasarían la condición de Fermat, y nosotros creeríamos que es primo. La buena noticia es que es posible modificar ligeramente la condición de Fermat para salvar esta dificultad. Escribamos el número $m-1$ de la forma $2^k M$ donde M es impar. Diremos que a satisface la *condición de Miller* si al menos uno de los números

$$a^M - 1, a^M + 1, a^{2M} + 1, a^{4M} + 1, a^{2^{k-1}M} + 1$$

es divisible por m.

Nótese que el producto de estos números es $a^{m-1} - 1$, luego todos los números que satisfacen la condición de Miller también satisfacen la condición de Fermat. Más aún, si m es primo, entonces m divide este producto, y por lo tanto divide a uno de estos factores, i.e., cada a $(1 \leq a < m)$ satisface la condición de Miller. Pero si m es compuesto, podría suceder que algunos a satisfagan la condición de Fermat pero no la de Miller (m puede ser un divisor de un producto sin ser divisor de ninguno de sus factores).

Tenemos que la condición de Miller provee una prueba de primalidad potencialmente más fuerte que la condición de Fermat, y de hecho, suficiente para nuestros propósitos.

Teorema 24 *Si m es un número compuesto, entonces al menos la mitad de los números $1, \ldots, m-1$ viola la condición de Miller.*

Demostración. Si m no es pseudo-primo, el resultado se sigue del Lemma 9. Luego podemos suponer que m es pseudo-primo. Necesitamos una caracterización de los números pseudo-primos, que no demostraremos aquí.

Lema 10 *m es un pseudo-primo si y sólo si $m = p_1 \cdots p_t$, donde p_i son primos distintos, $t \geq 2$, y $(p_i - 1)|(m-1)$ para todo $i = 1, \ldots, t$.*

Sea $p_1 \cdots p_t$ $(t \geq 2)$ la descomposición prima de m. Luego los primos p_i son todos impares y distintos, y tenemos $(pi-1)|(m-1) = 2^k M$ para todo $i = 1, \ldots, t$.

Sea l el exponente más grande con la propiedad de que ninguno de los números $p_i - 1$ divide a $2^t M$. Dado que los números $p^i - 1$ son pares cuando M es impar, dicho exponente existe (e.g. 0) y claramente $0 \leq l \leq k$. Más aún, por la definición de l, existe un j para el cual $p_j - 1$ divide a $2^{t+1}M$. Por lo tanto $p_j - 1$ divide a $2^s M$ para todo s con $l \leq s \leq k$, y además p_j divide a $a^{2^s M} - 1$ para todas las clases de residuo primitivas a. Consecuentemente, p_j no puede dividir a $a^{2^s M} + 1$, que es $(a^{2^s M} - 1) + 2$, y luego m no divide a $a^{2^s M} + 1$. Si a es una clase residual que cumple la condición de Miller, entonces m debe ser un divisor de una de las clases restantes $a^M - 1, a^M + 1, a^{2M} + 1, \ldots, a^{2^{l-1}M} + 1, a^{2^l M} + 1$. Entonces, para cada tal a, el número m divide, ya sea al producto de los primeros $l+1$ términos, el cual es $(a^{2^l M} - 1)$, o al último, $(a^{2^l M} + 1)$. Llamemos a las clases residuales primitivas $a \mod m$ "del primer tipo" si $a^{2^l M} = 1(\mod m)$ y como "del segundo tipo" si $a^{2^l M} = -1(\mod m)$.

Estimemos primero el número de números del primer tipo. Consideremos un índice i con $1 \leq i \leq t$. Dado que $p_i - 1$ no divide al exponente $2^l M$, entonces hay a no divisible por pi tal que $a^{2^l M} - 1$ no es divisible por p_i (Usar: si m divide a $a^k - 1$ para todo $(a, m) = 1$, entonces $p_i - 1$ divide k para todo $i + 1, \ldots, t$. Esto se sigue del teorema chino de los restos). El razonamiento del Lema 9 muestra que a lo más la mitad de las clases residuales $\mod p_i$ satisfacerán la condición de Fermat con ese exponente , i.e. tal que $a^{2^l M} - 1$ es divisible por pi. De acuerdo al teorema chino de los restos, existe una correspondencia 1 a 1 entre las clases residuales primitivas módulo $p_1 \cdots p_t$ y las t-tuplas de clases residuales primitivas módulo los primos $p_1, \ldots, p_t$. Luego, módulo $p_1 \cdots p_t$, a lo más una fracción 2^t-ésima de las clases residuales primitivas es tal que cada p_i divide $(a^{2^t M} - 1)$. Entonces a lo más una 2^t-ésima fracción de las clases primitivas residuales módulo m son del primer tipo. Es fácil ver que el producto de dos del segundo tipo es uno del primer tipo. Luego multiplicando todos los elementos del segundo tipo por uno fijo del primer tipo, obtenemos uno del primer tipo, y luego el número de elementos del segundo tipo es al menos tan grande como el número del primer tipo (si no hay elementos del segundo tipo para multiplicar, la situación es mejor: cero es sin duda menor que el número de elementos del primer tipo). Por lo tanto, incluso los dos tipos juntos suman a lo más una $2^t - 1$-ésima parte de las clases residuales primitivas, y por lo tanto a lo más la mitad (dado que $t \geq 2$). ⊙

Ahora podemos probar el hecho principal que hace al tset de Miller mejor que el test de Fermat.

Lema 11 *Para m y a dados, se puede determinar en tiempo polinomial si a satisface la condición de Miller.*

Para esto, basta con recordar del capítulo **??** que el resto de $ab \mod c$ es computable en tiempo polinomial. Basado en los resultados anteriores obtenemos el siguiente algoritmo aleatorio, conocido como *test de Miller-Rabin*,

Algoritmo 2 (Test de Miller-Rabin)
Escogemos un número entre 1 y $m-1$ aleatoriamente y determinamos si satisface la condición de Miller. Si no la satisface, m es compuesto. Si la satisface, escogemos un nuevo a. Si la condición de Miller es satisfecha k veces consecutivamente, podemos asegurar que m es primo con probabilidad de error menor que 2^{-k}.

Notas.

1. Si el algoritmo descubre que m es compuesto, esto no proviene de haber encontrado un divisor, sino que del hecho que uno de los residuos viola la condición de Miller. Si al mismo tiempo, el residuo a no viola la condición de Fermat, m no puede ser primo relativo de cada uno de los números $a^M - 1, a^M + 1, a^{2M} + 1, a^{4M+1}, \ldots, a^{2^{k-1}M} + 1$, por lo que calcular su máximo común divisor con cada uno de ellos entregará un divisor apropiado de m. No se conoce hoy día ningún algoritmo polinomial (ya sea determinístico o aleatorio) para encontrar una factorización en el caso de que la condición de Fermat sea también violada. Este problema es significativamente más difícil en la práctica que la verificación de primalidad.

2. Para m dado, podemos intentar encontrar un entero a que viole la condición de Miller, no de forma aleatoria, sino que probando con cada uno de los números $1, 2, \ldots$. El problema es que no sabemos hasta dónde hay que buscar si m es compuesto. Sin embargo, usando una conjetura del siglo pasado de la teoría analítica de números, la así llamada *hipótesis generalizada de Riemann* (la cual es muy técnica como para ser formulada aquí), es posible demostrar que no es más grande que $\log m$. Luego, si la hipótesis generalizada de Riemann es cierta, esto da un algoritmo determinístico de tiempo polinomial para verificación de primalidad.

3. Podemos utilizar el algoritmo de verificación de primos recién aprendido para encontrar un número primo de n dígitos (digamos en el sistema binario). Escogemos un número k del intervalo $[2^{n-1}, 2^n - 1]$ y verificar si es primo, con una probabilidad de error de a lo más $2^{-100}/n$. Si lo es, paramos. Si no lo es, escogemos un nuevo número k. De la teoría de números primos sabemos que en este intervalo el número de primos es bastante grande: asintóticamente es $(\log e)2^{n-1}/n$, i.e., un número de n dígitos

escogido aleatoriamente será un primo con probabilidad $1.44/n$. Repitiendo entonces este experimento $\mathcal{O}(n)$ veces encontraremos un número primo con muy buena probabilidad. Análogamente es posible escoger un primo aleatorio en cualquier intervalo suficientemente grande, por ejemplo en el intervalo $[1, 2^n]$.

5.5 Clases de complejidad aleatorias

En las secciones previas tratamos algoritmos que utilizan números aleatorios. Ahora definiremos clases de problemas solubles por dichos algoritmos. Análogamente a como lo hicimos con algoritmos determinísticos y no determinísticos, definiremos un modelo de computación correspondiente a los algoritmos aleatorios.

Sea $T = (k, \Sigma, \Gamma, \Phi)$ una máquina de Turing no-determinística, es decir Φ es una relación en $(\Gamma \times \Sigma^k) \times (\Gamma \times \Sigma^k \times \{-1, 0, 1\}^k))$ y consideremos dada una distribución probabilística para cada $g \in \Gamma$ y $h_1, \ldots, h_k \in \Sigma$ en el conjunto

$$\{(g', h_1', \ldots, h_k', \varepsilon_1, \ldots, \varepsilon_k) : \\ (g, h_1, \ldots, h_k, g', h'1, \ldots, h'k, \varepsilon_1, \ldots, \varepsilon_k) \in \Phi\}.$$

(Es útil suponer que las probabilidades de los eventos son numeros racionales, dado que los eventos con dichas probabilidades son fáciles de generar suponiendo que podemos generar bits mutuamente independientes). Una máquina de Turing no-determinística junto con estas distribuciones se llama *máquina de Turing aleatoria.* La máquina T computa de la siguiente forma: en un estado g y leyendo $h_1, \ldots, h_k$ se mueve a la siguiente configuración de acuerdo a la relación Φ y la distribución de probabilidades correspondiente. Luego, hay ciertas probabilidades de obtener ciertas respuestas.

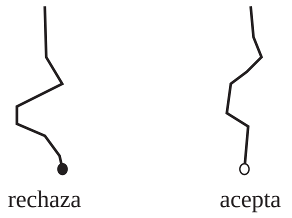

Figura 5.1: Un árbol determinístico

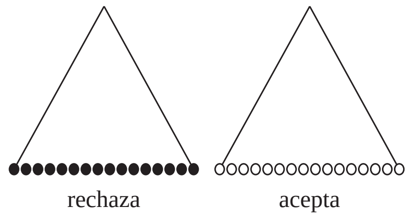

Figura 5.2: Un árbol no determinístico

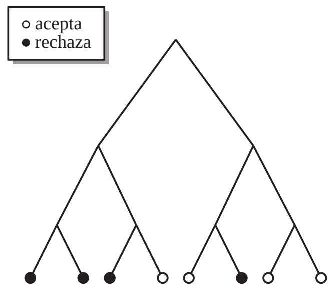

Figura 5.3: Un árbol con las posibles computaciones

Un punto importante es definir cómo decidirán un lenguaje estas nuevas máquinas. La novedad es que las respuestas vienen asociadas con probabilidades de ser correctas o no.

Una máquina aleatoria T que entrega las respuestas con alguna probabilidad de error se dice que procede en *sentido Monte-Carlo*. Si el error se produce en la aceptación y en el rechazo, se dice que *decide* según Monte-Carlo. Cuando el error se produce sólo en la aceptación, *acepta* según Monte-Carlo. Nótese que todos nuestros ejemplos caen en esta última categoría.

Por otro lado, una máquina aleatoria T que entrega las respuestas con total certeza (no erra nunca) se dice que decide en el *sentido Las Vegas*. Nótese que dependiendo de la distribución de probabilidades que tenga, puede demorar diferentes tiempos para diferentes ejecuciones de la misma entrada.

Ahora bien, ¿cómo medir el tiempo? Nótese que en el caso de una máquina aleatoria hay una manera natural de hacerlo: el número esperado[3] de pasos. Con estas consideraciones, surgen tres importantes clases aleatorias:

1. Clase BPP (Bounded-error Probabilistic Polynomial time). Un lenguaje $\mathcal{L} \in$ BPP si hay máquina aleatoria T que corre en tiempo polinomial esperado tal que:
   - Si $x \in \mathcal{L}$ entonces $\Pr\{T \text{ acepta } x\} > 3/4$,
   - Si $x \notin \mathcal{L}$ entonces $\Pr\{T \text{ rechaza } x\} > 3/4$.

   Es decir, acepta o rechaza con "razonable" certeza tanto en aceptación como rechazo.

2. Clase RP (Randomized Polynomial time). Un lenguaje $\mathcal{L} \in$ RP si hay máquina aleatoria T que corre en tiempo polinomial esperado tal que:
   - Si $x \in \mathcal{L}$ entonces $\Pr\{T \text{ acepta } x\} \geq 1/2$,
   - Si $x \notin \mathcal{L}$ entonces $\Pr\{T \text{ acepta } x\} = 0$.

   Es decir, acepta con razonable certeza y rechaza con toda seguridad.

3. Clase ZPP (Zero-error Probabilistic Polynomial time). Un lenguaje $\mathcal{L} \in$ ZPP si hay máquina aleatoria que corre en tiempo polinomial esperado tal que $x \in \mathcal{L}$ si y sólo si T acepta x (i.e. no se equivoca nunca).

[3] Recordemos que la *esperanza* de una variable aleatoria X es $E(X) = \sum_{x \in X} \Pr\{X = x\} x$.

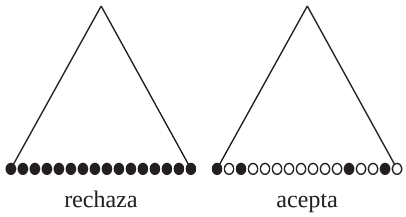

Figura 5.4: Una máquina RP

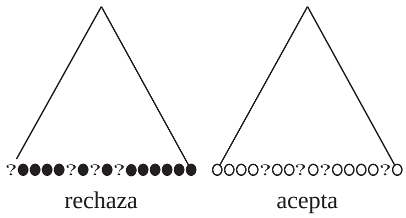

Figura 5.5: Una máquina ZPP

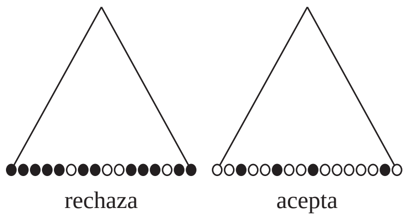

Figura 5.6: Una máquina BPP

Algunas observaciones sobre las definiciones anteriores.

Las constantes en las definiciones Monte-Carlo. La constante $3/4$ en la definición es arbitraria: cualquier número c con $1/2 < c < 1$ daría la misma definición de la clase BPP. De hecho, si la máquina dá una respuesta correcta con probabilidad c, repetimos la computación t veces para la entrada x y aceptamos como respuesta aquella entregada más frecuentemente. Es fácil ver, usando la ley de grandes números grandes, que la probabilidad de que esta respuesta sea errónea es menor a c_1^t donde $c_1 < 1$ y depende sólo de c. Para t suficientemente grande esto puede ser hecho arbitrariamente pequeño y esto cambia el número de pasos esperado sólo en un factor constante. (Nótese que c no puede ser $1/2$, pues entonces podemos dar una respuesta correcta con un "lanzamiento de moneda").

Análogamente, se puede demostrar que la constante $1/2$ en la definición de aceptación puede ser reemplazada por un número positivo arbitrario menor que 1.

Tiempo esperado y peor caso. En vez del número esperado de pasos en la definición de las clases BPP y RP, pudiéramos también haber considerado el peor caso y esto no cambiaría las clases. Veamos: si el número de pasos del peor caso es polinomial, obviamente también lo será el número de pasos esperado. Inversamente, supongamos que el número esperado de pasos es polinomial, digamos a lo más $|x|^d$. Entonces de acuerdo a la desigualdad de Markov[4], la probabilidad de que una computación dure más que, por ejemplo, $8|x|^d$ pasos, es a lo más $1/8$. Podemos entonces construir un contador que pare la máquina después de $8|x|^d$ pasos, y escriba 0 en la cinta de resultados. Esto aumenta la probabilidad de error a lo más en $1/8$.

Lo mismo, sin embargo, no se sabe si es cierto para la clase ZPP. Restringir a polinomial el tiempo de ejecución más grande nos llevaría a un algoritmo determinístico, y no se sabe si ZPP $= P$ (de hecho, se cree que son distintas; existen ejemplos para problemas resolubles por algoritmos de Las Vegas polinomiales para los que no se conoce ningún algoritmo determinístico polinomial).

Otros modelos de computación aleatoria. El modelo presentado no es único. Consideremos una máquina de Turing determinista que, además de las cintas usuales (entrada, trabajo, y resultado), tiene una cinta que en que cada celda tiene un bit (0 o 1) seleccionado aleatoria e independientemente con probabilidad $1/2$. La máquina en sí trabaja determinísticamente, pero su funcionamiento depende, por supuesto, del azar (los símbolos escritos en la cinta aleatoria). Es fácil ver que dicha máquina de Turing determinista con una cinta aleatoria extra y la máquina de Turing no-determinista con una distribución de probabilidad asociada pueden remplazarse una a otra en todas las definiciones.

Finalmente, también podríamos definir una máquina RAM aleatoria. Esta puede tener una celda extra w en la cual siempre hay un 0 o un 1 con probabilidad $1/2$. Agregamos la instrucción $y := w$ al lenguaje de programación, que cada vez que se ejecuta, escribe un nuevo bit aleatorio en la celda w completamente independiente de los bits anteriores. Nuevamente, no es difícil ver que esto no entrega ninguna diferencia significativa con los modelos anteriores.

[4] Si Y es variable aleatoria no-negativa, entonces para todo $k > 0$ entero se tiene que $\Pr[Y \geq kE[Y]] \leq 1/k$.

Relaciones entre clases de complejidad

Claramente

$$P \subseteq \mathrm{ZPP} \subseteq \mathrm{RP} \subseteq \mathrm{BPP}\,.$$

Tampoco es difícil ver que RP $\subseteq$ NP y que las clases BPP y ZPP son cerradas con respecto al complemento, es decir, ellas contienen junto con un lenguaje $\mathcal{L}$ al lenguaje $\Sigma \setminus \mathcal{L}$. Esto último no es claro para la clase RP, y de hecho no se sabe si esta clase es cerrada con respecto al complemento. Vale la pena, entonces, definir la clase co-RP (un lenguaje $\mathcal{L}$ está en co-RP si $\Sigma \setminus \mathcal{L}$ está en RP).

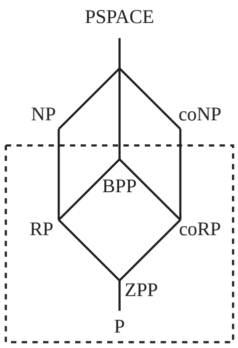

Figura 5.7: El mundo probabilístico

El siguiente resultado es una caracterización de la clase RP por medio de testigos, análogamente a lo que ocurre con la clase NP.

Teorema 25 *Un lenguaje $\mathcal{L}$ está en* RP *si y sólo si existe un lenguaje $\mathcal{L}' \in P$ y un polinomio $p(n)$ tal que:*

1. $\mathcal{L} = \{x \in \Sigma^* : y \in \Sigma^{p(|x|)} \wedge x\&y \in \mathcal{L}\}$ *y*
2. *si $x \in \mathcal{L}$ entonces para al menos la mitad de las palabras de largo $p(|x|)$ se tiene que $x\&y \in \mathcal{L}'$.*

La demostración es análoga a la del teorema correspondiente sobre NP. Siguiendo la analogía, resulta que la relación entre las clases RP y ZPP es más estrecha que la que hay entre las clases NP y P.

Teorema 26 *Las siguientes propiedades son equivalentes para un lenguaje $\mathcal{L}$:*

1. $\mathcal{L} \in \mathrm{ZPP}$*;*
2. $\mathcal{L} \in \mathrm{RP} \cap \text{co-RP}$*;*
3. *Existe una máquina de Turing aleatoria con tiempo polinomial (en el peor caso) que responde SI, NO, ó "ME RINDO" tal que: (i) las respuestas SI y NO siempre son correctas y la respuesta ME RINDO es correcta con probabilidad a lo más $1/2$, (ii) Si $x \in \mathcal{L}$ la respuesta es SI ó ME RINDO, y si $x \notin \mathcal{L}$ la respuesta es NO ó ME RINDO.*

Demostración. Es obvio que (1) implica (2). También puede verse fácilmente que (2) implica (3). Ingresemos x en paralelo a una máquina de Turing aleatoria que acepte $\mathcal{L}$ en tiempo polinomial y a otra que acepte $\Sigma \setminus \mathcal{L}$ en tiempo polinomial. Si entregan respuestas opuestas entonces la respuesta de la primera máquina es correcta. Si son idénticas respondemos ME RINDO. En este caso, una de las dos cometió un error con probabilidad a lo más $1/2$. Finalmente, para ver que (3) implica (1), sólo hay que modificar la máquina de Turing T_0 dada en (3) de tal forma que en vez de responder ME RINDO, comienze de nuevo. Si para la entrada x, el número de pasos de T_0 es t y la probabilidad de rendirse es p, entonces con esta misma entrada el número esperado de pasos de la máquina modificada es

$$\sum_1^\infty p^{t-1}(1-p)t\tau = \frac{\tau}{1-p} \leq 2\tau.$$

$\odot$

Hemos visto en la subsección anterior que el "lenguaje" de los números compuestos está en RP. Pero hay aún más: Adleman y Huang mostraron que este lenguaje está también en ZPP. Para nuestro otro ejemplo importante, los polinomios no idénticos a 0, sólo se sabe que su lenguaje está en RP. Además de los problemas algebraicos (principalmente de teoría de grupos), hay muchos otros que están en RP o ZPP pero no se conoce ningún algoritmo polinomial para su solución.

Nota. Los algoritmos que usan aleatoriedad no deben ser confundidos con los algoritmos cuyo comportamiento (por ej. el valor esperado del número de pasos) se analiza para entradas aleatorias. En este caso no se supone ninguna distribución de probabilidad sobre el conjunto de entradas, sino se considera el peor caso. La investigación del comportamiento de los algoritmos con entradas aleatorias que provienen de una distribución dada es un área importante pero difícil, aún en su infancia, que no trataremos aquí.

: borrar una vez que se agregue capitulo de analisis de promedio

5.6 Ejercicios

1. Suponga que un experimento tiene una probabilidad de éxito p. Demuestre que haciendo n^3 experimentos es posible calcular una aproximación $\hat{p}$ de p tal que la probabilidad de $|\hat{p}-p| > \sqrt{p(1-p)/n}$ es a lo más $1/n$. (Indicación: Usar la desigualdad de Chebysheff).

2. Queremos calcular una cantidad real a. Suponga que tenemos un algoritmo aleatorio que computa una aproximación A (la cual es una variable aleatoria) tal que $\Pr\{|A-a| > 1\} \leq 1/20$. Muestre que llamando algoritmo n veces se puede computar una aproximación B tal que $\Pr\{|B-a| > 1\} \leq 2-t$.

3. (*) Suponga que alguien le da tres matrices A, B, C de $n \times n$ (con enteros de tamaño máximo l) y afirma que $C = AB$. Usted está muy ocupado como para verificar esta afirmación exactamente y hace lo siguiente. Escoge un vector aleatorio x de largo n cuyas entradas son enteros escogidos uniformemente desde algún intervalo $[0, \ldots, N-1]$, y verifica $A(Bx) = Cx$. Si esto es verdad acepta la afirmación, en caso contrario la rechaza.

   (a) (*) ¿Cuán grande debe ser escogido N para que la probabilidad de aceptar erróneamente sea menor que $0,01$?

   (b) Compare la complejidad en tiempo entre el algoritmo probabilístico y el algoritmo determinista para calcular AB.

4. (*) Muestre que si m es un pseudo-primo, el test de Miller-Rabin no sólo lo descubre con una gran probabilidad, sino que también puede ser usado para encontrar una descomposición de m en 2 factores.

5. (*) Muestre que la máquina de Turing equipada con una cinta aleatoria extra y la máquina de Turing no-determinística con una distribución probabilística son equivalentes: si algún lenguaje es aceptado en tiempo polinomial por alguna, es aceptado también por la otra.

6. (*) Formule qué significa que una RAM aleatoria acepte un lenguaje determinado en tiempo polinomial y muestre que es equivalente al hecho de que alguna máquina de Turing aleatoria lo acepte.

7. (*) Llamemos *robusta* a una fórmula Booleana con n variables si, o es insatisfacible, o es satistisfacible al menos por $2^n/n^2$ asignaciones. Dé un algoritmo probabilístico polinomial para decidir la satisfabilidad de fórmulas robustas.

Capítulo 6

Complejidad y Paralelismo

Un computador paralelo es aquel que ejecuta más de una operación al mismo tiempo. En este sentido, casi todos los computadores son paralelos. Otro asunto es explotar el paralelismo en gran escala, cosa que no es sencilla de lograr. Las nuevas tecnologías hacen más urgente desarrollar los fundamentos matemáticos de la computación paralela. A pesar de la abundante investigación realizada, la búsqueda de un modelo canónico de computación paralela no ha decantado en un modelo que posea el mismo balance entre teoría y práctica como el que tiene el modelo RAM para la computación secuencial. El problema principal es la modelación de la comunicación entre los diferentes procesadores y subprogramas: esto puede ocurrir en canales cercanos, a lo largo de trayectorias determinadas de antemano, como "radio broadcast", etc. Una cuestión similar que puede ser modelada de diferentes formas es la sincronización de los relojes de los diferentes procesadores: esto puede hacerse con algunas señales comunes o sencillamente no suceder.

En esta sección, nos concentraremos en un solo modelo, el así llamado *máquina de acceso aleatorio paralelo* (Parallel Random Access Machine, PRAM), el cual ha sido elaborado fundamentalmente desde el punto de vista de la teoría de la complejidad. Aunque el modelo PRAM contiene un par de supuestos poco realistas, como son la ejecución sincronizada de los diferentes procesadores y la comunicación global a través de la memoria global, sigue siendo un buen modelo para explorar interrogantes fundamentales acerca de la paralelizabilidad de ciertos problemas. Los algoritmos presentados pueden ser considerados, por otra parte, como programas escritos en algún lenguaje de alto nivel: ellos deben ser implementados de acuerdo a las soluciones tecnológicas específicas que sean elegidas.

6.1 Máquinas de Acceso Aleatorio Paralelas

El modelo más investigado de máquinas ejecutando computación paralela es el de la máquinas de acceso aleatorio paralelo (PRAM). (Ver dibujo pag 312 Leighton)

Esta consiste de un número fijo de procesadores y una memoria común. Los procesadores son máquinas de acceso aleatorio idénticas y habrá p de ellos. El lugar de almacenamiento del programa y la memoria de ellas es común y consiste, digamos, de celdas $x[i]$ (donde los i son números enteros). Será conveniente asumir (aunque no es absolutamente necesario) que cada procesador tiene un número infinito de celdas de programa $u[i]$. Más allá de esto, cada procesador tiene una celda v de memoria separada que contiene el número de serie del procesador. El procesador puede leer su nombre v y puede leer y escribir sus propias celdas x, y, $u[i]$ asi como también las celdas de memoria común $x[i]$. En otras palabras, a las instrucciones permitidas por la máquina de acceso aleatorio (ver pág. ??) debemos agregarle las instrucciones siguientes:

$$\begin{array}{ll} u[i] \leftarrow 0; & u[i] \leftarrow u[i] \pm 1; \\ u[i] \leftarrow u[j]; & u[i] \leftarrow u[i] \pm u[j]; \\ u[i] \leftarrow u[u[j]]; & u[u[i]] \leftarrow u[j]; \\ u[i] \leftarrow x[u[j]]; & x[u[i]] \leftarrow u[j]; \end{array}$$
$$\text{if } u[i] \leq 0 \text{ then goto } p;$$

La entrada se ingresa en las celdas $x[1], x[2], \ldots$ Además de la entrada y del programa común, debemos también especificar cuántos procesadores serán usados; esto lo podemos escribir en la celda $x[-1]$. Los procesadores ejecutan el programa en paralelo pero en pasos sincronizados (lockstep). (Dado que los procesadores se pueden referir a sus propios nombres no necesariamente computarán la misma cosa) En nuestro modelo usaremos función de costo logarítmica: el costo de escribir o leer un entero k desde una celda de memoria $x[t]$ es el número total de dígitos en k y t, es decir, aproximadamente $\log 2|k| + \log 2|t|$. El siguiente paso comienza después que cada procesador ha terminado los pasos previos. La máquina se detiene cuando cada procesador llega a la línea del programa en que no hay ninguna instrucción. La salida es el contenido de la celda $x[i]$.

Una importante pregunta a decidir es cómo regular el uso de la memoria común. ¿Qué sucede si varios procesadores desean escribir o leer sobre la misma celda de memoria? Varias convenciones existen para la resolución de estos conflictos. Mencionaremos cuatro de éstas:

- Dos procesadores no deben leer o escribir sobre la misma celda al mismo tiempo. Llamamos a esto el modelo de lectura y escritura exclusiva (EREW, por exclusive-read, exclusive-write). Podríamos llamarlo también completamente libre de conflicto. Esto se debe entender de tal manera de que el programador es responsable para evitar intentos de acceso simultáneos a una misma celda. Si tales intentos ocurriesen, la máquina dará una señal de error del programa.

- Quizás el modelo más natural es el que permite a muchos procesadores leer la misma celda a la vez, pero la escritura concurrente en una misma celda es considerada un error del programa. El modelo es llamado lectura concurrente y escritura exclusiva (CREW, por concurrent-read, exclusive-write), y podría también ser llamado cuasi libre de conflicto.

- Varios procesadores pueden leer de la misma celda, pero pueden escribir concurrentemente sobre la misma celda sólo si ellos desean escribir la misma cosa. (La máquina señala un error de programa sólo si dos procesadores desean escribir diferentes números en la misma celda). Llamamos a este modelo de lectura concurrente y escritura concurrente (CRCW, por concurrent-read, concurrent-write); también podría ser llamado de conflicto limitado.

- Muchos procesadores pueden leer desde la misma celda o escribir sobre la misma celda.

Si varios desean escribir sobre la misma celda el procesador con el número de serie más pequeño lo hará: este modelo es llamado de lectura y escritura concurrente con prioridad (P-CRCW, por priority concurrent-read, concurrent-write), o más brevemente, el modelo de prioridad.

Es claro que los modelos anteriores son más poderosos en la medida de que permitan hacer más cosas. Puede ser demostrado, sin embargo, que –al menos si el número de procesadores no es demasiado grande– las computaciones que se pueden hacer en uno de los modelos más poderosos, el modelo de prioridad, no son mucho más rápidas que si fueran ejecutadas en el modelo libre de conflicto. El siguiente lema se refiere a tal afirmación.

Lema 12 *Para cada programa P, hay un programa Q tal que si P computa algúna salida desde algún entrada con p procesadores en tiempo t sobre el modelo de prioridad, entonces Q computa lo mismo sobre el modelo libre de conflicto con* $\mathcal{O}(p^2)$ *procesadores en tiempo* $\mathcal{O}(t(\log p)^2)$.

Observación. En las máquinas PRAM es necesario especificar el número de procesadores no sólo debido a que la computación depende de éste, sino también porque este es –junto al tiempo y al espacio– una importante medida de la complejidad de la computación. Si éste no está restringido, entonces podemos resolver problemas muy difíciles de manera muy rápida. Podemos decidir, por ejemplo, la 3-colorabilidad de un grafo si, para cada coloración de un conjunto de vértices y cada eje del grafo, tenemos un procesador que chequea si, en la coloración dada, el punto final de un eje dado tiene diferentes colores. Los resultados deben ser reunidos todavía por supuesto, pero en una máquina de conflicto limitado esto puede realizarse en un solo paso.

Demostración. Un procesador separado de LA máquina libre de conflicto corresponderá a cada procesador de la máquina de prioridad.

Estos son llamados procesadores supervisores. Además, cada procesador supervisor tendrá p procesador subordinados. Un paso en la computación de la máquina de prioridad será simulado por una fase (o etapa) en la computación de la máquina libre-de-conflicto. La idea básica de la construcción es que cualquier cosa que esté en la máquina de prioridad después de un paso dado en la computación dentro de una celda dada z estará contenida en la correspondiente fase de la computación en la máquina libre-de-conflicto, en cada una de las celdas con direcciones 2pz, 2pz+1, ..., 2pz+p-1. Si en un paso de la máquina de prioridad, el procesador i debe leer o escribir sobre la celda z entonces en la correspondiente fase de la máquina libre-de-conflicto, el correspondiente procesador supervisor leerá o escribirá la celda con dirección 2pz+i. Esto ciertamente evitará todos los conflictos dado que los diferentes procesadores usan distintos módulos de celdas p. Debemos estar seguros, sin embargo, que en el estado final, la máquina libre-de-conflicto escribirá sobre las celdas 2pz, 2pz+1, ..., 2pz+p-1 lo que la regla de prioridad escribiría dentro de z en el paso correspondiente de la máquina de prioridad. Para esto, insertamos una fase consistente de O(log p) pasos auxiliares ejecutados al final de cada fase. Primero, cada procesador supervisor i que en la presente fase escribió en la celda 2pz+i, escribe un 1 en la celda 2pz+p+i. Entonces, dentro del llamado primer paso de la fase, éstos mirarán si hay un 1 en la celda 2pz+p+i-1. Si hay, irán a dormir por el resto de la fase. Por otro lado, éstos escriben un 1 y despiertan a un subordinado. En general, al comienzo del paso k, el procesador i tendrá que despertar a lo más a 2k-1 subordinados (incluyendo, posiblemente, a él mismo); éstos (al menos los que serán despertados) examinarán las correspondientes celdas 2pz+p+i-2k-1, ..., 2pz+p+i-(2k-11). Los que encuentren un 1 irán a dormir. Los que escriben un 1, despiertan a un nuevo subordinado, enviándolos 2k-1 pasos a la izquierda mientras éste se ubica 2k pasos a la izquierda. El subordinado que quede bajo la ubicación 2pz+p irá a dormir; si un supervisor i termina, éste se sabe que ganó. Es fácil convencernos a nosotros mismos que si en el correspondiente paso de la máquina de prioridad, varios procesadores desean escribir en la celda z, entonces los correspondientes procesadores supervisor y subordinado no pueden entrar en conflicto mientras se muevan dentro del intervalo [2pz+p, 2pz+2p-1]. Esto puede ser visto a saber que en el k-ésimo paso, si un procesador supervisor i está activo entonces los procesadores activos j i y sus subordinados han escrito un 1 en cada una de las 2k-1 posiciones hacia abajo, comenzando en 2pz+p+i que son todavía 2pz+p. Si un procesador supervisor o su subordinado comienzan a la derecha de ellos y alcanzan una celda i en k pasos, éste necesariamente parará en alguno de los 1s e irá a dormir, antes que pueda entrar en conflicto con el i-ésimo procesador supervisor o su subordinado. Esto también muestra que siempre un solo supervisor ganará, esto es, el que tenga el número más pequeño. El ganador todavía tiene el trabajo de ver que el que escriba en la celda 2pz+i escribió en cada celda del intervalo [2pz, 2pz+p-1]. Esto es fácil de hacer por un procedimiento muy similar al hecho anteriormente: el procesador escribe el valor deseado en la celda 2pz, entonces éste despierta a un subordinado; ellos dos escriben el valor deseado en las celdas 2pz+1 y 2pz+2 entonces ellos despiertan a un subordinado cada uno, etc. Cuando todos ellos han pasado 2pz+p-1 la fase termina y la siguiente etapa de simulación puede comenzar. Dejaremos al lector planear el despertar a los subordinados. Cada uno de los pasos anteriores requieren la ejecución de varios programas de instrucciones, pero es fácil ver que sólo un número restringido es necesario, cuyo costo es, incluso en el caso del modelo de costos logarítmico, sólo O(log p + log z). De esta manera, el tiempo transcurrido entre las dos etapas de la simulación es sólo O(log p(log p + log z)). Dado que los pasos simulados de la máquina de prioridad también toman al menos log z unidades de tiempo, el tiempo de ejecución es de este modo aumentado a sólo O((log z)2). g ⊙

En lo que sigue, a menos que lo advirtamos explícitamente, usaremos el modelo libre de conflicto (EREW). De acuerdo al lema anterior, podríamos haber elegido uno de los otros modelos sin que los resultados variaran sustancialmente.

Es fácil convencerse de que la siguiente afirmación es correcta.

Proposición 1 *Si una computación puede ser ejecutada con p procesadores en t pasos con números de a lo más s bits, entonces para todo $q < p$ ésta puede ser ejecutada con q procesadores en $\mathcal{O}(tp/q)$ pasos con números de a lo más $\mathcal{O}(s + \log(p/q))$ bits. En particular, ésta puede ser ejecutada sobre una máquina RAM secuencial en $\mathcal{O}(tp)$ pasos con números de largo $\mathcal{O}(s + \log p)$.*

La cuestión fundamental en teoría de complejidad en relación a los algoritmos paralelos, es justamente lo opuesto a esto: dado un algoritmo secuencial que corre en tiempo N, se desea implementarlo con p procesadores en esencialmente $\mathcal{O}(N/p)$ pasos. Más adelante veremos algunas clases de complejidad motivadas por esta interrogante.

Veremos que la aleatorización es una herramienta más importante en el caso de la computación paralela que en el caso secuencial. La máquina de acceso paralelo aleatoria (randomized parallel Random Access Machine) difiere de la anteriormente introducida PRAM solamente en que cada procesador tiene una celda extra en la cual, con probabilidad 1/2 hay un 0 o un 1. Si el procesador lee este bit entonces un nuevo bit aleatorio se escribirá en la celda. Estos bits aleatorios son completamente independiente entre sí (tanto al interior de un procesador como entre diferentes procesadores).

6.2 La clase NC

Decimos que un programa para máquina PRAM es un programa NC si existen constantes $c, d > 0$ tal que para todas las entradas x el programa computa libre de conflicto con $\mathcal{O}(|x|^c)$ procesadores en tiempo $\mathcal{O}((\log|x|)^d)$. (De acuerdo al Lema 12, esta definición no cambia si usamos, por ejemplo, el modelo de prioridad). La clase de lenguajes NC consiste de aquellos lenguajes $\mathcal{L} \subseteq \{0,1\}^*$ cuya función característica[1] puede ser computada por un programa NC.

Nota. La finalidad buscada al introducir la clase NC no es modelar computaciones paralelas reales. En la práctica generalmente se puede usar bastante más que tiempo logarítmico, pero (al menos en un futuro previsible) sólo mucho menos que un número polinomial de procesadores polinomiales. La finalidad del concepto es describir aquellos problemas que, aparte de ser resolubles en un número polinomial de operaciones, tienen la propiedad adicional de que esas operaciones son paralelizables al máximo. Nótese que en el caso de una entrada de tamaño n, en una máquina libre de conflicto, son necesarios al menos $\log n$ pasos para permitir que todos los bits de la entrada tengan efecto sobre la salida.

Obviamente, $NC \subseteq P$. Lo que no se sabe es si la igualdad es verdadera,aunque la respuesta es que probablemente no lo sea.

Definimos la clase NC *aleatoria*, RNC (Randomized NC) siguiendo el patrón de la clase BPP. Esta consiste de los lenguajes $\mathcal{L} \subseteq \{0,1\}^*$ que son decididos con baja probabilidad de error en una máquina PRAM aleatorizada en tiempo logarítmico con un número polinomial de procesadores. Formalmente, $\mathcal{L} \in$ RNC si existe un número $c > 0$ y un programa que computa, para cada entrada x un 0 o 1, en una máquina PRAM aleatorizada con $\mathcal{O}(|x|^c)$ procesadores (dígamos, libre-de-conflicto) en tiempo $\mathcal{O}(\log|x|^c)$ Si $x \in \mathcal{L}$ entonces la probabilidad de que el resultado sea 0 es menor que $1/4$, y si $x \notin \mathcal{L}$ entonces la probabilidad de que el resultado sea 1 es menor que $1/4$.

Es posible construir una teoría de complejidad en torno a la clase NC similar a la construída en torno a la clase P. La reducción NC de un lenguaje a otro puede ser definida y, por ejemplo dentro la clase P, se puede mostrar por ejemplo, que dentro de la clase P hay lenguajes que son P-completos, es decir, que cualquier otro lenguaje en P es NC-reducible a él. No trataremos los detalles de esto aquí.

6.3 Problemas en NC

Proposición 2 *Determinar si un grafo contiene un triángulo es un problema en* NC.

DEMOSTRACIÓN. Supondremos que la entrada es la matrices de adyacencia del grafo. El algoritmo NC es esencialmente trivial. Asignamos un procesador para cada trí de vértices i, j, k, y verificamos en paralelo si el trío (i, j, n) forma un triángulo. Si no, entonces el procesador para. Si forma un triángulo entonces escribe un 1 en la posición 0 de las celdas comunes y se detiene. Da lo mismo usar el modelo de conflicto limitado o el de prioridad. Obsérgese que este algoritmo toma $\mathcal{O}(1)$ pasos. $\odot$

Nuestro siguiente ejemplo es menos trivial, y

[1] Una función característica para $\mathcal{L}$ es una función $f : \Sigma^* \to \{0,1\}$ tal que $f(w) = 1$ si y sólo si $w \in \mathcal{L}$.

a primera vista sorprendente: conectividad de grafos. Los algoritmos usuales (búsqueda en anchura o en profundidad) son fuertemente secuenciales: cada paso depende del resultado del paso anterior. Para su paralelización, usaremos un truco similar al que usamos para probar el teorema de Savitch en el Capítulo ??.

Proposición 3 *Determinar si un grafo es conexo está en* NC.

DEMOSTRACIÓN. Describiremos el algoritmo sobre el modelo de conflicto-limitado. Otra vez, instruimos el procesador con número de serie correspondiente a la tripleta (i, j, k) (por ejemplo podemos enumerarlos $i + jn + kn2$ donde n es el número de vértices) que mire (i, j, k). Si éste ve dos ejes en la tripleta entonces inserta un tercer eje (Si varios procesadores desean insertar el mismo eje, todos ellos desearán escribir la misma cosa en la misma celda y esto está permitido). Después de t repeticiones, estarán conectados exactamente los pares de puntos cuyas distancias en el grafo original es a lo más $2t$. De esta forma, repitiendo $\mathcal{O}(\log n)$ veces, obtenemos el grafo completo sí y sólo si el grafo original era conexo. ⊙

Análogamente es posible decidir si en un grafo dado hay un camino conectando dos puntos dados. Más aun, la distancia entre ambos puede ser determinada por una leve modificación al algoritmo anterior.

Proposición 4 *El producto de dos matrices (en particular, el producto escalar de dos vectores), y la k-ésima potencia de una matriz $n \times n$ $(k \leq n)$ es computable en* NC.

DEMOSTRACIÓN. Podemos computar el producto escalar de dos vectores como sigue: multiplicamos paralelamente sus elementos correspondientes. Luego agrupamos los productos obtenidos en parejas y formamos las sumas; luego agrupamos estas sumas en parejas y formamos las sumas, etc. Ahora para computar el producto de dos matrices es suficiente observar que cada elemento del producto es el producto escalar de dos vectores, y éstos pueden ser computados paralelamente. Para la k-ésima potencia de una matriz de $n \times n$ usar el mismo esquema que se usó para computar $ab(\mod c)$ (Lema ??). ⊙

El siguiente algoritmo es seguramente la herramienta más importante de la teoría de la computación paralela.

Teorema 27 (Csánky) *El determinante de cualquier matriz de enteros puede ser computado por un algoritmo* NC. *Por lo tanto, las matrices invertibles forman un lenguaje* NC.

DEMOSTRACIÓN. Presentamos un algoritmo propuesto por Chistov. La idea es tratar de representar el determinante por una conveniente serie de potencias de matrices. Sea B una matriz de $n \times n$ y sea B_k la submatriz $k \times k$ en su esquina superior izquierda. Asumamos primero que estas submatrices B_k son no-singulares, es decir, que sus determinantes no son 0. Entonces B es invertible y de acuerdo a la conocida fórmula de la inversa tenemos que $(B^{-1})_{nn} = \det B_{n-1} / \det B$, donde $(B^{-1})_{nn}$ denota los elementos que se encuentran en la esquina inferior derecha de la matriz B^{-1}. De aquí

$$\det B = \frac{\det B_{n-1}}{(B^{-1})_{nn}}.$$

Continuando esto obtenemos

$$\det B = \frac{1}{(B^{-1})_{nn}(B_{n-1}^{-1})_{n-1,n-1} \cdots (B_1^{-1})_{11}}.$$

Escribamos B en la forma $B = I - A$ donde $I = I_n$ es la matriz unitaria de $n \times n$. Asumiendo, por el momento, que los elementos de A son bastante pequeños, obtenemos el siguiente desarrollo en serie:

$$B_k^{-1} = I_k + A_k + A_k^2 + \cdots$$

que particularizando al elemento k, k da:

$$(B_k^{-1})_{kk} = 1 + (A_k)_{kk} + (A_k)_{kk}^2 + \cdots$$

Luego

$$\begin{aligned}\frac{1}{(B_k^{-1})_{kk}} =& \frac{1}{1 + (A_k)_{kk} + (A_k)_{kk}^2 + \cdots} \\ =& 1 - [(A_k)_{kk} + (A_k)_{kk}^2 + \cdots] \\ &+ [(A_k)_{kk} + (A_k)_{kk}^2 + \cdots]^2 - \cdots\end{aligned}$$

y por lo tanto

$$\begin{aligned}\det B = \prod_{k=1}^{n} (1 - [(A_k)_{kk} + (A_k)_{kk}^2 + \cdots] \\ + [(A_k)_{kk} + (A_k)_{kk}^2 + \cdots]^2 - \cdots).\end{aligned}$$

No podemos, por supuesto, computar estas series infinitas compuestas de series infinitas. Afirmamos, sin embargo, que es suficiente computar sólo n términos de cada serie. Más exactamente, sustituyamos tA en lugar de A, donde t es una variable real. Para t suficientemente pequeño, las matrices $I_k - tA_k$ son ciertamente no singulares y el desarrollo de arriba vale. Entonces substituyendo obtenemos:

$$\begin{aligned}\det(I - tA) = \prod_{k=1}^{n} (1 - [t(A_k)_{kk} + t^2(A_k)_{kk}^2 + \cdots] \\ + [t(A_k)_{kk} + t^2(A_k)_{kk}^2 + \cdots]^2 - \cdots).\end{aligned}$$

Ahora viene la idea clave: el lado izquierdo es un polinomio en t de grado a lo más n, luego en la serie de potencias del lado derecho es suficiente calcular sólo los términos de grado a lo más n. De esta forma, $\det(I - tA)$ consiste exactamente de los términos de grado a lo más n del siguiente polinomio:

$$F(t) = \prod_{k=1}^{n} [\sum_{j=0}^{n} (- \sum_{r=1}^{n} t^r (A_k^r)_{kk})^j].$$

Ahora, independientemente de lo complicado que la fórmula $F(t)$ parezca, ésta puede ser computada fácilmente en el sentido NC. Borrando de ésta los términos de grado superior a n, obtenemos un polinomio idéntico a $\det(I - tA)$. Ahora bien, como toda identidad polinomial, la nuestra vale para todos los valores de t, no sólo para los más pequeños, y no se requieren suposiciones de no singularidad. Entonces sustituyendo $t = 1$ se obtiene $\det B$. ⊙

Usando el Teorema ?? con sustituciones aleatorias se obtiene la siguiente importante aplicación:

Corolario 3 *Determinar si un grafo tiene un emparejamiento completo es un problema en* RNC.

Ninguna demostración combinatorial (i.e. que evite el uso del teorema de Csánky) es conocida para este resultado. Es importante hacer notar que el algoritmo sólo determina si el grafo tiene un emparejamiento completo, pero no da el emparejamiento si éste existe. Este último problema, significativamente más difícil, puede ser también resuelto en el sentido RNC (por un algoritmo de Karp, Upfal y Wigderson).

6.4 Ejercicios

1. (a) Probar que es posible determinar cuál de dos cadenas de ceros y unos es lexicográficamente más larga, usando n procesadores en $\mathcal{O}(1)$ pasos en el modelo de prioridad y en $\mathcal{O}(\log n)$ pasos en el modelo libre de conflicto.

   (b) (*) Mostrar que en del modelo libre de conflicto se requieren de hecho $\Omega(\log n)$ pasos.

   (c) (*) ¿Cuántos pasos se requiere en los otros dos modelos?

2. Mostrar que la suma de dos números en binario de largo no mayor que n puede ser computada con n^2 procesadores en $\mathcal{O}(1)$ pasos en el modelo de prioridad.

3. (a) Mostrar que la suma de n números en binario de largo no mayor que n, puede

ser computada, usando n^3 procesadores en $\mathcal{O}(\log n)$ pasos en el modelo de prioridad.

(b) Mostrar que n^2 procesadores son suficientes para esto.

(c) Haga lo mismo para el modelo libre de conflicto.

4. Considere el siguiente problema. Dado un circuito booleano y su entrada, computar su salida. Probar que si este problema está en NC , entonces P = NC.

5. De un algoritmo NC que en un grafo dado compute la distancia entre dos puntos.

Capítulo 7

Aproximación

7.1 Problemas de Optimización

Es bueno recordar que para los problemas de decisión no hay diferencias entre una solución y otra (nos basta saber si hay o no hay). Distinto es el caso de problemas funcionales, es decir aquellos donde la respuesta es más compleja que un simple SI o NO. Más aún, hay una gran cantidad de importantes problemas en los cuales se busca no sólo una "buena" solución, sino la "mejor" solución entre todas las posibles. Nótese que "buena", "la mejor", no se refieren aquí al tiempo o espacio usado para encontrarla, sino a alguna medida asociada a cada solución. Por ejemplo, el problema del vendedor viajero donde cada tramo entre dos ciudades tiene un costo. Una buena solución es aquella de menor costo. Estos son llamados problemas de *optimización*. Otros ejemplos se pueden encontrar en problemas de ruteo, control de flujo, scheduling, etc. Debido a su importancia, para estos problemas se ha desarrollado una infinidad de métodos particulares y una extensiva literatura. No pretendemos cubrir aquí esos aspectos algorítmicos que pueden verse en libros de *optimización combinatoria*. Lo que nos interesa a nosotros en lo que sigue son metodologías para estudiar la complejidad de este tipo de problemas y lograr alguna clasificación de ellos en función de su dificultad. Por ejemplo en que casos existen algoritmos eficientes para solucionar un problema de este tipo, y en caso negativo, cuanto es posible hacer con recursos limitados.

Un problema de optimización A está definido por un conjunto de instancias (como cualquier problema), más:

1. Un predicado $F(x)$ que determina si posibles soluciones son *soluciones factibles* para la entrada x.

2. La función de costo $c : F(x) \longrightarrow \mathbb{N}$, que a cada solución factible le asigna un costo.

3. La función objetivo: maximizar o minimizar el costo.

Resolver el problema de optimización (A, F, c, m) significa encontrar un algoritmo M tal que para cada entrada x, M compute la solución óptima para x, que denotaremos $\mathrm{opt}(x)$ y que está definida así:

$$\mathrm{opt}(x) = \max_{s \in F(x)} c(s) \quad (\text{resp. } \min_{s \in F(x)} c(s))$$

Recordemos que a todo problema de optimización se le puede asociar naturalmente un problema de decisión de la siguiente forma: Si

(A, F, c, m) es problema de optimización, entonces asociamos el problema de determinar si para una entrada x el costo de la solución $s(x)$ es menor que k.

En general nos concentraremos en problemas de optimización para los cuales las soluciones factibles tienen tamaño polinomial, verificar el predicado F puede hacerse en tiempo polinomial y la función de costo puede calcularse en tiempo polinomial. A veces se denota esta clase de problemas por NPO (Non-deterministic Polynomial time Optimization problems). Claramente esta es la versión para optimización de la clase NP. En los ejercicios proponemos demostrar esto formalmente.

7.2 Algoritmos de aproximación

Si no es factible resolver exactamente un problema de optimización, una buena alternativa es intentar resolverlo aproximadamente, es decir, encontrar soluciones que se aproximen a la solución óptima.

Sea (A, F, c, m) un problema de optimización. Para una instancia x y solución factible s_x, el *error relativo* de s_x respecto de la solución óptima $\mathrm{opt}(x)$ está dado por

$$\frac{|\,\mathrm{opt}(x) - c(s_x)|}{\max\{\mathrm{opt}(x), c(s_x)\}}.$$

El denominador de la expresión permite normalizar el error relativo para ambos casos de optimización': minimización o maximización. Así el error es siempre es un número entre 0 y 1. La solución s_x estará más cerca del óptimo $\mathrm{opt}(x)$ cuanto más cercano a cero sea el error relativo.

Un *algoritmo de aproximación de parámetro* ϵ para el problema (A, F, c, m) es un algoritmo que corre en tiempo polinomial y computa soluciones factibles cuyo error relativo es menor que ϵ.

Veremos en los ejemplos que siguen que hay problemas de optimización en NPO para los cuales hay algoritmos de aproximación con error tan pequeño como se quiera, otros para los cuales el error del mejor algoritmo está acotado inferiormente, y finalmente algunos para los cuales no existe algoritmo de aproximación, esto es el error es tan cercano a uno como se quiera.

CUBRIMIENTO DE NODOS MINIMAL
Entrada: Grafo G
Salida: Un conjunto de nodos $C \subseteq V(G)$ de tamaño minimal tal que todo eje de G es incidente con algún nodo de C.

Para este problema hay un sencillo algoritmo de aproximación: Partir con $C = \emptyset$ y con G el grafo dado. Mientras $E(G) \neq \emptyset$, elija un eje uv, agregue u, v a C y redefina G eliminando el eje uv.

Cuando $E(G) = \emptyset$, entonces claramente C será un cubrimiento de G. Observe que $|C|$ es un número par, supongamos $2t$. ¿Cuan lejos está de $\mathrm{opt}(G)$? Observe que el grafo inicial G contiene t ejes disjuntos es decir que no son incidentes a un nodo común (uno por cada ciclo del algoritmo que produjo C). Luego, cualquier cubrimiento de G debe contener al menos un nodo por cada uno de estos ejes. Es decir, $\mathrm{opt}(G) \geq t$, y de esta forma $\frac{|C|-\mathrm{opt}(G)}{|C|} \leq \frac{1}{2}$.

Demostramos entonces que hay un algoritmo de aproximación de parámetro $1/2$ para el problema de cubrimiento. Este simple procedimiento es el mejor algoritmo de aproximación que se conoce para este problema.

VENDEDOR VIAJERO
Entrada: Grafo G y función distancia $d : E(G) \longrightarrow \mathbb{N}$

Salida: Tour que pase por cada nodo exactamente una vez y cuya distancia recorrida sea minimal.

Teorema 28 *Si* P $\neq$ NP *entonces no existe algoritmo de aproximación con parámetro menor que* 1 *para* VENDEDOR VIAJERO.

DEMOSTRACIÓN. La demostración es por contradicción. Supondremos que existe un algoritmo de aproximación para el problema con parámetro $\epsilon < 1$. Probaremos que usando este hipotético algoritmo podemos encontrar un algoritmo de tiempo polinomial para el problema CIRCUITO HAMILTONIANO, que sabemos es NP-completo.

El algoritmo funciona como sigue. Dado un grafo G cuyo conjunto de nodos es V, primero construimos una instancia de VENDEDOR VIAJERO con $|V|$ nodos. Para dos nodos u, v de G, definimos su distancia $d(u,v) = 1$ si existe un eje uv en G, y $d(u,v) = |V|/(1-\epsilon)$ en caso contrario. Luego ingresamos esta instancia al hipotético ϵ-algoritmo *Vendedor Viajero*. Se nos presentan dos casos: si el algoritmo produce un tour que usa solamente ejes de largo 1, entonces existe un circuito hamiltoniano para G. Por el contrario, si el algoritmo produce un tour que usa al menos un eje de largo $|V|/(1-\epsilon)$, entonces el tour tiene largo $t > |V|/(1-\epsilon)$. Como nuestro algoritmo es aproximado de parámetro ϵ, si denotamos por o el largo del tour óptimo, se tiene que $(t-o)/t < \epsilon$, y luego $t(1-\epsilon) < o$. Concluímos entonces que $o > |V|$, y por lo tanto G no tiene circuito hamiltoniano. $\odot$

MOCHILA
Entrada: Conjunto de n pares de números enteros positivos (v_i, w_i) (que representan el valor y el peso de ciertos objetos), y peso límite W

Salida: Encontrar un subconjunto S de objetos cuyo peso total no sea mayor que W ($\sum_{i\in S} w_i \leq W$) y cuyo valor total sea el máximo posible.

(El problema se llama mochila pues modela un ladrón que intenta llevar en su mochila—que no soporta más que un peso fijo—el máximo de valores.)

Teorema 29 *Para todo* $\epsilon > 0$ *existe un algoritmo de aproximación de parámetro* ϵ *para* MOCHILA.

DEMOSTRACIÓN. Supongamos que tenemos una instancia x de MOCHILA. Sea $V = \max\{v_1, \ldots, v_n\}$ y definamos, para cada $i = 0, 1, \ldots, n$ y $0 \leq v \leq nV$, la cantidad $W(i,v)$ como el miínimo peso total obtenible si es posible seleccionar entre los i primeros objetos de tal forma que el valor sea exactamente v. Definimos $W(0,v) = \infty$, y para los restantes i, v:

$$W(i+1, v) = \min\{W(i,v), W(i, v - v_{i+1}) + w_{i+1}\}.$$

Al final, tomamos el mayor v tal que $W(n,v) \leq W$. Claramente este algoritmo resuelve el problema de la mochila en tiempo $\mathcal{O}(n^2 V)$.

Por supuesto V puede ser muy grande, y el algoritmo no es polinomial. Sin embargo, dado que estamos interesados en un algoritmo de *aproximación*, podemos trocar precisión por velocidad a través de eliminar los bits menos significativos de cada uno de los valores de v. Así, dada la instancia x definida por $\{(v_i, w_i)\}_i, W$, podemos definir la instancia *aproximada* x' definida por $\{(v'_i, w_i)\}_i$, donde los nuevos valores $v'_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$, es decir, el valor antiguo v_i con sus últimos b bits reemplazados por ceros. A su debido tiempo deberemos fijar el parámetro b.

Si resolvemos la instancia aproximada x' en vez de x, el tiempo requerido será solamente

$\mathcal{O}(n^2V/2^b)$, puesto que podemos ignorar la corrida de ceros en los v_i. La solución S' obtenida de esta forma está relacionada con la solución óptima S por la siguiente desigualdad, que más abajo fundamentamos:

$$\sum_{i\in S'} v_i' \geq \sum_{i\in S} v_i' \geq \sum_{i\in S}(v_i - 2^b) \geq (\sum_{i\in S} v_i) - n2^b$$

La primera desigualdad vale pues S' es solución óptima para x'; la segunda por la definición de v_i', y finalmente la última pues $|S| \leq n$. Comparando la primera con la última expresión, concluímos que el valor de la solución obtenida por nuestro algoritmo aproximado está a lo más $n2^b$ bajo el óptimo. Puesto que V es una cota inferior para el valor de la solución óptima (podemos suponer que $w_i \leq W$ para todo i), la desviación relativa del óptimo es a lo más $\epsilon = n2^b/V$.

Hemos probado lo siguiente: dado cualquier $\epsilon > 0$, podemos truncar los últimos $b = \lfloor \log(\epsilon V/n) \rfloor$ bits de los valores y obtener un algoritmo de aproximación con parámetro ϵ que corre en tiempo $\mathcal{O}(n^2V/2^b)$, esto es $\mathcal{O}(N^3/\epsilon)$, que es polinomial. $\odot$

7.3 Clases de Complejidad Aproximadas

1. APX: conjunto de problemas para los cuales existe un algoritmo de aproximación de parámetro $\epsilon < 1$ que corre en tiempo polinomial.

2. PAS: conjunto de problemas para los cuales existe un algoritmo $M_\epsilon(x)$ tal que para todo parámetro ϵ y entrada x computa en tiempo polinomial una solución factible con error a lo más ϵ.

Obsérvese que el algoritmo de (2) debe ser uniforme en ϵ, es decir, no se trata de un conjunto de algoritmos, uno para cada ϵ. De ahi el nombre PAS, Polynomial Approximation Scheme.

De los ejemplos que vimos, Cubrimiento de Nodos Minimal está en APX, Mochila está en PAS y Vendedor Viajero no está en ninguna de las dos.

7.3.1 Relaciones entre clases y reducibilidad

Teorema 30

$$\text{PO} \subseteq \text{PAS} \subseteq \text{APX} \subseteq \text{NPO}$$

y si P $\neq$ NP*, las inclusiones son estrictas.*

Demostración. Las inclusiones son sencillas de verificar. Respecto de la estrictez de las inclusiones si P $\neq$ NP, ya sabemos del Teorema 28 que APX $\neq$ NPO. Dejamos para los ejercicios el mostrar la otras. $\odot$

Tal como en el caso de no determinismo, aleatoriedad, etc. es posible en este ámbito definir las correspondiente noción de *reducibilidad.* Obsérvese que en este caso necesitamos un concepto que no sólo haga corresponder problemas, sino también soluciones factibles (entre otras cosas debe preservar costos).

7.4 Ejercicios

1. Demostrar que si un problema de optimización está en NPO, entonces su problema de decisión asociado está en NP.

2. Denotamos por PO los problemas de optimización para los cuales existen algoritmos que corren en tiempo polinomial. Demostrar que si P $\neq$ NP, entonces PO $\neq$ NPO.

3. Demostrar las inclusiones en el Teorema 30.

4. Demostrar que si P $\neq$ NP, entonces todas las inclusiones en el Teorema 30 son estrictas.

Capítulo 8

Apéndices

8.1 La notación asintótica $\mathcal{O}$, Ω y Θ

Para cada función $g : \mathbb{N} \longrightarrow \mathbb{R}$ definimos las clases de funciones (con dominio los números naturales) $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$. Denotemos por $f : \mathbb{N} \longrightarrow \mathbb{R}$ una función arbitraria.

1. $f \in \mathcal{O}(g)$ si y sólo si existen $n_0 \in \mathbb{N}$ y $c > 0$ tal que para todo $n \geq n_0$:
$$0 \leq f(n) \leq cg(n),$$
es decir, asintóticamente f está acotada superiormente por g salvo una constante.

2. $f \in \Omega(g)$ si y sólo si $g \in \mathcal{O}(f)$.

3. $f \in \Theta(g)$ si y sólo si $f \in \mathcal{O}(g) \cap \Omega(g)$, es decir los comportamientos asintóticos de f y g difieren sólo en una constante.

Algunos ejemplos

1. Demostrar que $5n^2 + 100 \in \Theta(n^2)$.
Demostración. $5n^2 + 100 \in \mathcal{O}(n^2)$ pues $5n^2 + 100 \leq 6(n^2)$ para $n \geq 100$ (es decir, $c = 6$ y $n_0 = 100$). También es evidente que $5n^2 + 100 \in \Omega(n^2)$, luego tenemos que n^2 y $5n^2 + 100$ son equivalentes asintóticamente salvo una constante.

2. Demostrar que $n! \in \mathcal{O}(n^n)$.
Demostración. $n! \in \mathcal{O}(n^n)$ pues $n! \leq n^n$ para todo $n \geq 1$.

3. Demostrar que $n^{\lg n} \in \mathcal{O}(2^n)$.
Demostración. Sabemos que $n = 2^{\lg n}$, luego $n^{\lg n} = (2^{\lg n})^{\lg n} = 2^{(\lg n)^2}$ y sabemos que $(\lg n)^2 \leq n$ para $n \geq 16$.

8.2 Computabilidad

Sea Σ un alfabeto finito que contiene el símbolo "$*$". Permitiremos que la entrada para una máquina de Turing sean palabras que no contengan este símbolo especial: sólo letras de $\Sigma_0 = \Sigma \setminus \{*\}$.

Llamaremos a una función $f : \Sigma_0^* \longrightarrow \Sigma_0^*$ recusiva o computable si existe una máquina de Turing que para cualquier input x que pertenezca a Σ_0^* parará despues de tiempo finito con $f(x)$ escrito en la primera cinta (nosotros hemos visto en la sección anterior que podemos asumir sin pérdida de generalidad que la máquina de Turing tiene sólo una cinta).

La noción de "recursiva" tanto como la noción de "enumerablemente recursiva" y "parcialmente recursiva" definida después puede ser fácilmente extendida de una manera única a funciones y conjuntos sobre algunos conjuntos contables diferentes de Σ_0^*, como el conjunto de los números naturales, el conjunto N^* de cadenas finitas de números naturales, etc. La extensión vendrá con la ayuda de algún tipo de codificación estándar de, por ejemplo, el conjunto de los números naturales sobre elementos de Σ_0^*. Por eso aunque definimos esas nociones sólo sobre Σ_0^*, a veces la usaremos en relación con funciones definidas sobre otros dominios. Esto es poco estricto pero no debiera causar ninguna confusión.

Llamaremos a un lenguaje L *recursivo* si su función caracteristica es

$$f_L(x) = \begin{cases} 1 & \text{si } x \in L \\ 0 & \text{en caso contrario} \end{cases}$$

es recusiva. En vez de decir que un lenguaje L es recursivo, podemos decir también que la propiedad definida por L es *decidible*. Si una máquina de Turing calcula esta función, entonces diremos que decide el lenguaje. Es obvio que todo lenguaje finito es recursivo. También si un lenguaje es recusivo su complemento también lo es.

Nota 11 *Es obvio que hay un continuo de lenguajes(michos de ellos incontables) pero solo un número contable de máquinas de Turing. Entoces deben existir lenguajes no recusivos. Nosotros veremos algunos lenguajes concretos no recursivos.*

Llamaremos a el lenguaje L recursivamente enumerable si $L = \emptyset$ o existe una función recursiva f tal que el rango de f es L. Esto significa que podemos enumerar los elementos de L: $L = \{f(w_0), f(w_1), \ldots\}$ donde $\Sigma_0^* = \{w_0, w_1, \ldots\}$. Aquí los elementos de L no necesariamente ocurren en orden creciente y la repetición es también permitida.

Daremos una definición alternativa de lenguajes recursivamente enumerablemente mediante el siguiente lema.

Lema 13 *Un lenguaje L es recursivamente enumerable ssi hay una máquina de Turing T tal que si escribimos x en la primera cinta la máquina se detiene ssi $x \in L$.*

Demostración. Sea L recursivamente enumerable: Asumiremos que es no vacío. Prepararemos una máquina de Turing que con entrada x calcule $f(y)$ en orden creciente de $y \in \Sigma_0$ y se detenga cuando encuentre un y tal que $f(y) = x$.

En la otra dirección, asumiremos que L contiene el conjunto de palabras con las cuales se detiene: asumiremos que L es no vacío y que $a \in L$. Construiremos una máquina de Turing T_0 que, cuando el número natural i es su entrada esta simule a T con input x la cual es la $(i - \lfloor\sqrt{i}\rfloor)^2$-ésima palabra de Σ_0^*, por i pasos. Si la simulada T para, entonces T_0 tine como salida x. Puesto que cada palabra de Σ_0^* ocurrirá con infinitos valores de i el rango de T_0 sera L. $\odot$

No hay ningún truco con la función $(i - \lfloor\sqrt{i}\rfloor)^2$: todo lo que necesitamos es que para $i = 0, 1, 2, \ldots$ su valor asuma cada entero no negativo infinitas veces. La técnica usada en esta demostración, esto es simular muchas computaciones por una sola, es a veces llamada "dovetailing".

Lema 14 *Todo lenguaje recursivo es enumerablemente recursivo.*

Demostración. Esto es claro si el lenguaje L es vacío. Nosotros podemos cambiar la máquina de Turing que decide F a entregar el input si este es uno un a entregar aguna arbtrario $a \in L$ si el original era 0. $\odot$

El siguiente teorema caracteriza la relación entre lenguajes recursivamente enumerables y recursivos.

Teorema 31 *Un lenguaje L es recusivo ssi ambos lenguajes L y $\Sigma_0^* \setminus L$ son recursivamente enumerables.*

DEMOSTRACIÓN. Si L es recursivo entonces su complemento también lo es y por el lema anterior es recursivamente enumerable.

Hacia el otro lado, asumamos que tanto L y su complemento son recursivamente enumerables. Construiremos 2 máquinas que los enumeren y una tercera que detecte si alguna emite como salida x. Tarde o temprano esto ocurrira y sabremos a donde x pertenece. $\odot$

Ahora mostraremos que hay lenguajes que son recusivamente enumerables (y más aun, explícitos) pero no recusivos.

Sea T una máquina de Turing con k cintas y sea L_T el conjunto de todas aquellas palabras $x \in \Sigma_0^*$ para las cuales T se detiene cuando escribimos x en todas las cintas.

Teorema 32 *Si T es una máquina de Turing universal con $k+1$ cintas entonces L_T es recursivamente enumerable pero no recursivo.*

DEMOSTRACIÓN. La primera parte se extrae del Lema 14. Probaremos el segundo, por simplicidad, para $k = 1$.

Asumiremos, a modo de contradicción, que L_T es recursivo. Entonces $\Sigma_0^* \setminus L_T$ sería recursivamente enumerable entonces existiría una máquina T_1 de una cinta que con entrada x se detendrá ssi $x \notin L_T$. La máquina T_1 puede ser simulada por T escribiendo un programa apropiado p en la segunda cinta de T. Entonces escribiendo p en ambas cintas de T, ésta se detendría si T_1 parase a causa de la simulación. La Por otro lado, la máquina T_1 fue definida para detenerse en p si y sólo si T no se detenía con entrada p en ambas cintas (es decir cuando p no pertenecia a L_T), lo cual es una contradicción. $\odot$

Esta prueba usa la llamada técnica de digonalización originaria de la teoría de conjuntos (donde fué usada por Cantor para mostrar que el conjunto de todos los números reales es no contable). La técnica forma la base para múltiples demostraciones en lógica, teoría de conjuntos, y teoría de complejidad. Veremos más de esto en lo que sigue.

Hay un número de variantes del resultado anterior que afirman la indecidibilidad de problemas similares.

Sea T una máquina de Turing. El *problema de la parada* para T es decidir para todas las entradas posibles cuando T se detiene en x. De otro punto de vista, el problema de la parada para T significa decidir el conjunto de los x para los cuales T se detiene. Podemos también hablar acerca de el problema de la parada en general, lo que significa que un par (T, x) es dado donde T es una máquina de Turing (dada por su tabla de transiciones) y x es una entrada.

Teorema 33 *Hay una máquina de Turing de 1 cinta para la cual el problema de parada es indecidible.*

DEMOSTRACIÓN. Supongamos que el problema de la parada es decidible para todas las máquinas de Turing de 1 cinta. Sea T una máquina de Turing universal con 2 cintas y construyamos una máquina de 1 cinta T_0 de manera similar a la demostración del Teorema **??** (con $k = 2$) con la diferencia de que al comienzo escribiremos la i-ésima letra de la palabra x no sólo en la celda

$4i$ sino también en la $4i - 2$. Entonces para una entrada x, la máquina T_0 simulará el trabajo de T, cuando T comienze con x en ambas cintas. Puesto que es indecidible si T se detiene para una entrada dada (x, x), es también indecidible si T_0 se detendrá en una entrada dada x. $\odot$

La demostración anterior, aunque simple, es el prototipo de un gran número de demostraciones de indecibilidad. Este procedimiento consiste en tomar cualquier problema P_1 que se sabe es indecidible (en este caso pertenencia a L_T) y mostrar que puede ser reducido au problema P_2 conocido (en este caso el problema de parada de una máquina de T_0). La reducción muestra que si P_2 es decidible, entonces P_1 también lo es: pero como ya sabemos que P_1 es indecidible , concluiremos que P_2 también lo es. La reducción de un problema a alguna que parezca relacionado es a menudo bastante escabrosa.

Una descripción de un máquina de Turing es el listado de los conjuntos Σ, Γ (donde, como antes, los elementos de Γ están copdificados por palabras sobre el conjunto Σ_0) y la tabla de funciones α, β, γ.

Corolario 4 *Es algorítmicamente indecidible cuando una máquina de Turing (dada su descripción) para en la entrada vacía.*

Demostración. Sea T una máquina de Turing cuyo problema de parada sea indecidible: mostraremos que su problema de parada puede ser reducida al problema general de la parada en entrada vacía. De hecho para cada entrada x, podemos construir una máquina de Turing T_x para la cual, cuando parte con la entrada vacía, escribe x en la cinta de entrada y entonces simula T. Si podemos decidir cuando T_x para entonces podremos decidir cuando T para en x. $\odot$

Corolario 5 *Es algoritmicamente indecidible cuando para una máquina de Turing de 1 cinta T (dada por su descripción) el conjunto L_T es vacío.*

Demostración. Para una máquina dada S, construyamos una máquina T que hace lo siguiente: primero borra todo lo que contiene la cinta y entonces se convierte en la máquina S. La descripcion de T puede ser obviamente construida sin dificultades a partir la descripción de S. Por lo tanto si S se detiene en la entrada vacía en un número de pasos finitos entonces T se detendrá en todas las entradas en un número finito de pasos, y por lo tanto $L_T = \Sigma_0^*$ es no vacío: si S funciona por tiempo infinito en la entrada vacía entonces T funcionará en todas las entradas de largo infinito y entonces L_T es vacío: por lo tanto si nosotros podemos decidir si L_T es vacío podemos decidir si S para en un input vacío, lo cual es indecidible. $\odot$

Obviamente, al igual que la propiedad de vaciedad, no es posible decidir ninguna otra propiedad P de L_T si, o el lenguaje vacío lo contiene y Σ_0^* no lo contiene, o viceversa. Es posible conseguir resultados negativos más generales aún. Llamemos *trivial* a una propiedad de un lenguaje si, o bien, todos los lenguajes la tienen, o ninguno la tiene.

Teorema 34 (Rice) *Dada cualquier propiedad de lenguajes no trivial P, es indecidible si el lenguaje L_T de una máquina de Turing T arbitraria (dada su descripción) la posee.*

Esto es, es indecidible, dada la descripción de T, cuando L_T es finito, regular, contiene una palabra dada, etc.

Demostración. Podemos asumir que el lenguaje vacío no tiene una propiedad P (si no, consideramos la negación de P). Sea T_1 una máquina de Turing para la cual L_{T_1} posee la propiedad P. Pra un máquina de Turing dada S, construyamos una máquina T como sigue: para una entrada x , primero simularemos S en la entrada vacía. Cuando la máquina simulada S para, ésta simula T_1 con entrada x. Entonces si S no se detiene en la entrada vacía, T no se detendrá en cualquier entrada y por lo tanto $L_T = \emptyset$. Si S se detiene en la entrada vacía entonces T se detendrá en las mismas entradas que T_1 y por lo tanto $L_T = L_{T_1}$. Por ende, si nosotros pudiéramos decidir cuando L_T tiene una propiedad P podríamos también decidir cuando S se detiene en la entrada vacía. $\odot$

Bibliografía

[I] **Complejidad Computacional.**

[1] J. Balcázar, J. Díaz, y J. Gabarró, *Structural Complexity I*, Springer, Segunda ed., 1995.

[2] D.P. Bovet, P. Crescenzi, *Introduction to the Theory of Complexity*, Prentice Hall International Series in Computer Science, Prentice Hall, 1994.

[3] M.R. Garey, D.S. Johnson, *Computers and Intractability*, W.H. Freeman and Company, 1979.

[4] N. Immerman, *Descriptive Complexity*, Springer, 1999.

[5] J. van Leeuwen (Ed.), *Handbook of Theoretical COmputer Science, Vol. A*, Elsevier and MIT Press 1990.

[6] B. Moret, *Theory of Computation*, Addison-Wesley, 1998.

[7] Ch.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[8] J.E. Savage, *Models of Computation, Exploring the Power of Computing*, Addison-Wesley, 1998.

[II] **Textos de Teoría de Computación.**

[9] A. Aho, J. Ullman, *Foundations of Computer Science*, (C Edition, 1995) W.H. Freeman & Co.

[10] M. Davis, R. Sigal, E. Weyuker, *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science, 2nd Ed.*, Academic Press, 1994.

[11] H. Lewis, C. Papadimitriou, *Elements of the Theory of Computation, 2nd. Ed.*, Prentice Hall, 1997.

[III] **Textos de Algoritmos.**

[12] A. Aho, J. Hopcroft, J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[13] T. Cormen, Ch. Leisserson, R. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[14] Robert Sedgewick, *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching*, Addison-Wesley Pub. Co., 1999.