#82

SERIE
COMPUTACIÓN

Jorge L. Ortega Arjona

# Breves notas sobre: complejidad

VÍNCULOS MATEMÁTICOS

AÑO
2004

# FACULTAD DE CIENCIAS

# VÍNCULOS MATEMÁTICOS

## BREVES NOTAS SOBRE COMPLEJIDAD

Jorge L. Ortega Arjona*

VÍNCULOS MATEMÁTICOS NO. 82, 2009

* Profesor del Departamento de Matemáticas de la Facultad de Ciencias de la UNAM.
Impreso en la Coordinación de Servicios Editoriales de la Facultad de Ciencias UNAM

# Breves Notas sobre
## *Complejidad*

**Jorge L. Ortega Arjona**
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Noviembre 2007

# Índice general

# Prefacio

Las *Breves Notas sobre Complejidad* introducen en forma simple y sencilla a algunos de los temas relevantes en el área de la Teoría de la Complejidad. No tiene la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que por su relevancia se consideran esenciales para el conocimiento básico de esta área, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: Números Aleatorios, Complejidad en Tiempo y Espacio, Satisfactibilidad, Funciones No Computables, NP-Completitud, el Teorema de Cook, Problemas NP-Completos y la Tesis de Church. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe aprender en las asignaturas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Noviembre 2007

# Capítulo 1

# Números Aleatorios
## *La Teoría Chaitin-Kolmogoroff*

Mencionar números aleatorios y computadoras en el mismo texto parece casi una contradicción. La esencia de lo aleatorio es la ausencia de un procedimiento o mecanismo. Considérese por ejemplo, las máquinas que se utilizan en algunos juegos de lotería. Diez bolas de colores con números que van del 0 al 9 circulan en una jaula. En algunos casos, un flujo de aire mantiene a las bolas flotando. Cuando se apaga el flujo de aire, una de las bolas entra en un canal, saliendo de la jaula. El valor de la bola se considera un número aleatorio, ¿o no?

Al punto en que las leyes de la física pueden determinar la posición de las bolas en todo momento, este procedimiento para obtener números aleatorios no resulta del todo perfecto. Los ganadores de las loterías, sin embargo, nunca han objetado este dispositivo. Y en pocas ocasiones, los usuarios de los lenguajes de programación más modernos objetan los generadores de números aleatorios provistos por tales lenguajes. Los números "aleatorios" así obtenidos son tan útiles en tantas aplicaciones que pocas personas se preocupan por ellos dado que tales números *parecen* aleatorios.

Muchos de los generadores de números aleatorios que se emplean en las computadoras modernas usan un método lineal congruencial. Lo que quiere decir es que una simple fórmula lineal opera en el número aleatorio presente para obtener el siguiente:

$$x_{n+1} \leftarrow k \times x_n + c \, mod \, m$$

El número aleatorio actual $x_n$ se multiplica por una constante $k$, y un

corrimiento $c$ se añade al producto. Finalmente, se obtiene el *módulo* de la división del número resultante entre $m$. Para inciar todo el proceso, se escoge un valor $x_0$, llamado *semilla.*

No toda combinación de parámetros $k$, $c$ y $m$ son igualmente efectivos para producir números que parecen aleatorios. Por ejemplo, los valores $k = 19$, $c = 51$, $m = 100$, $x_0 = 25$, producen la siguiente secuencia, que tiene una gran desventaja:

$$25, 26, 45, 6, 47, 44, 87, 4, 27, 64, 67, 24, 7, 84, 47, \ldots$$

El último número listado, 47, es el mismo que el quinto número en la secuencia. Dado que la fórmula utilizada es enteramente determinística, la secuencia entre los dos números 47 se repetirá sin final. La longitud de tal secuencia se conoce como su *periodo.* Obviamente, cada secuencia generada por este método se repite tarde o temprano. Entonces surge la pregunta: ¿qué es lo mejor que se puede lograr con este método?

La siguiente selección de parámetros mejora en algo la situación. También ilustra qué tan sensible es el proceso de generación a un pequeño cambio en los parámetros de los valores $k = 19$, $c = 51$, $m = 101$, $x_0 = 25$, que generan la secuencia:

$$25, 21, 46, 16, 52, 29, 97, 76, 81, 75, 62, 17, 71,$$
$$87, 88, 6, 64, 55, 86, 69, 49, 73, 24, 2, 89, 76, \ldots$$

Esta secuencia no se repite sino hasta mucho después. El periodo se mejora de 10 a 18.

Al parecer, muchas secuencias aleatorias de números pueden producirse mediante la bien conocida fórmula logística utilizada como un modelo de caos en sistemas dinámicos:

$$x_{n+1} \leftarrow r \times x_n \times (1 - x_n)$$

Comenzando con un valor semilla inicial $x_0$ entre 0 y 1, la fórmula iterativa puede producir una secuencia más convincente de números aleatorios. Aquí, el valor de la semilla no es crítico. Después de algunas iteraciones, la secuencia de valores tiende a saltar dentro del mismo subintervalo $[a, b]$ dentro de $[0, 1]$. Mediante aplicar la transformación

$$y_n \leftarrow \frac{x_n - a}{b - a}$$

a $x_n$, se obtiene un nuevo número "aleatorio" $y_n$ entre 0 y 1.

La fórmula logística sólo se comporta de forma caótica para ciertos valores del parámetro $r$. Resulta particularmente interesante considerar los valores de $r$ entre 3.57 y 4.

Mucha teoría se ha elaborado para mejorar la aleatoriedad de varios programas generadores de números, pero éstos siempre serán números *pseudoaleatorios*. De hecho, tales números no son últiles para respoder la pregunta ¿qué es aleatorio? Sin embargo, permiten ilustrar la idea de que los programas de computadora pueden generar números con varios grados de aleatoriedad aparente.

La idea de un programa de computadora que genere una secuencia de números aleatorios tiene como base la teoría Chaitin-Kolmogorov. Descubierta independientemente a mediados de los años 1960's por Gregory J. Chaitin del T.J. Watson Research Center de IBM en Yorktown, Nueva York, y por A.N. Kolmogorov, un matemático soviético, la teoría define la aleatoriedad de una secuencia finita de números en términos de la longitud del programa más corto que la genera; mientras más largo sea el programa, más aleatoria la secuencia. Obviamente, un programa que produce una secuencia dada requiere ser más largo que la secuencia misma. Esto sugiere que aquellas secuencias que requieren programas aproximadamente tan largos como ellas mismas son las más aleatorias; es razonable etiquetarlas entonces con el adjetivo "aleatorias".

Considérese el equivalente binario de la secuencia generada al principio de este capítulo. ¿Cuál es el programa más corto que genera la secuencia consistente de $m$ repeticiones de 01001? Suponiendo por el momento que el lenguaje algorítmico usado aquí es un lenguaje de programación, uno puede al menos fijar la longitud mínima de un programa generador de esta secuencia:

```
for i:=1 to m
     output 0,1,0,0,1
```

Este programa contiene 23 caracteres ASCII (sin contar los espacios en blanco) y una variable `m` que cambia de una versión del programa a otra. De hecho, para un valor particular de `m`, la longitud del programa en términos de caracteres es de $23 + log\, m$. Genera una secuencia de longitud $n = 5m$. ¿Qué tan aleatoria, entonces, es la secuencia que produce? Una forma de medir la aleatoriedad es formar la razón

entre la longitud del programa y la longitud de la secuencia, que consiste en $m$ repeticiones de 0, 1, 0, 0, 1... Por tanto se tiene una aleatoriedad de:

$$r \leq \frac{23 + logm}{5m}$$

Ya que la razón tiende a cero conforme $m$ crece, es razonable concluir que mientras más larga sea la secuencia, menos aleatoria es. En el límite, la razón tiene valor cero; en otras palabras, no es aleatoria en absoluto. La misma conclusión se mantiene para cualquier programa que tenga un número fijo de parámetros relacionados con la longitud de la secuencia; los números que produce tienden a no tener aleatoriedad.

Sobre todas las secuencias de longitud $n$, sin embargo, puede mostrarse que la gran mayoría son aleatorias. Mediante utilizar un umbral arbitrario de $n - 10$, puede preguntarse ¿cuántas secuencias de $n$ dígitos pueden generarse por programas mínimos de longitud menor que $n-10$? Para propósitos de argumentación, se supone que todos los programas se escriben en términos de dígitos binarios. Esto no produce daño alguno a la argumentación, ya que símbolos alfabéticos y de otros tipos pueden ser considerados como grupos de dígitos de 8 bits. Por lo tanto, hay ciertamente no más que:

$$2^1 + 2^2 + \cdots + 2^{n-11}$$

programas de longitud menor que $n - 10$. La suma no excede $2^{n-10}$. Por lo tanto, menos que $2^{n-10}$ programas tienen longitud menor que $n - 10$. Estos programas generan a lo mas $2^{n-10}$ secuencias, y esto último contabiliza para cerca de una secuencia de $n$ bits en cada mil.

Se pensaría que con tantas secuencias aleatorias debería ser fácil obtener una. Nada puede alejarse más de la verdad. Para comprobar que una secuencia particular $S$ es aleatoria, se debe demostrar que no hay un programa significativamente más corto que $S$ que pueda generarla.

Supóngase que tal procedimiento de demostración en sí mismo ha sido mecanizado en forma de un programa $P$. El programa opera con enunciados de Cálculo de Predicados y, por cada uno, decide si es una demostración de que una secuencia particular de $n$ bits puede solamente ser generada por un programa tan larga como la secuencia misma. En realidad, $P$ no requiere ser tan general; solo necesita verificar que la secuencia puede ser generada por un programa más largo que $P$.

Existe un procedimiento mecánico para generar fórmulas predicativas válidas, una tras otra, de tal modo que su longitud contínuamente aumenta – todas las fórmulas de longitud 1, luego todas las fórmulas de longitud 2, y así en adelante. Algunas de las fórmulas resultan ser demostraciones de que secuencias específicas no pueden ser generadas por programas tan cortos como $P$. Pero en tal caso, $P$ puede modificarse tal que reporte tales secuencias, en efecto, generándolas. Por tanto, $P$ ha generado una secuencia que no es muy corta para generarse.

La contradicción aparente forza a concluir que $P$ no puede existir. Tampoco, entonces, ningún programa (o procedimiento de demostración) puede ser más general que $P$. De tal modo, es imposible probar que una secuencia es aleatoria a pesar del hecho de que la mayoría de las secuencias son aleatorias.

La similitud entre el argumento apenas descrito y el famoso teorema de incompletitud de Gödel no es accidental. En el tipo de sistemas formales definidos por Gödel, se ha demostrado que cualquier sistema axiomático es incompleto – hay teoremas que no pueden demostrarse en el sistema. De entre tales teoremas están aquéllos aseverando que una secuencia dada de números es aleatoria. El trasfondo de la teoría Chaitin-Kolmogoroff que se ha presentado hasta aquí es que mientras no se conoce o se sabe si una secuencia dada es aleatoria, es posible al menos medir el grado de aleatoriedad que producen ciertos programas.

Para aquéllos que realmente desean producir una secuencia realmente aleatoria de números, existe un dispositivo que hace el trabajo. Un diodo Zener es un componente electrónico que permite el flujo de corriente eléctrica en una sola dirección. Sin embargo, cuando se opera bajo voltaje inverso, algunos de los electrones se filtran a trvés del dispositivo en la dirección equivocada. Su frecuencia depende de los movimientos aleatorios térmicos de los electrones dentro del diodo. Cuando se le mide con un osciloscopio sensitivo, la corriente filtrada ciertamente parece aleatoria.

Es posible incorporar un diodo Zener a un circuito que muestree la corriente en intervalos de tiempo regulares de algunos microsegundos. Si el valor excede un cierto umbral, un convertidor analógico digital genera un 1 lógico; si no es así, genera un 0 lógico. De esta forma, se produce una secuencia de números. Si la secuencia no es aleatoria, entonces la mecánica cuántica moderna está en serios problemas.

# Capítulo 2

# Complejidad en Tiempo y Espacio

## *La Notación de la O Grande*

Cuando un programa se ejecuta en una computadora, dos de las más importantes consideraciones que deben tenerse son cuánto tiempo le tomará y cuánta memoria ocupará. Hay otras cuestiones como si el programa funciona, pero las dos consideraciones de tiempo de cómputo y espacio de memoria son dominantes. Por ejemplo, en grandes computadoras, la atención a procesos de los usuarios cambia dependiendo del tiempo en que un proceso debe ejecutarse y cuánta memoria utiliza. Aun en computadoras pequeñas, se desea que un programa ejecute rápidamente y no exceda la cantidad de memoria disponible.

Un problema sencillo que puede utilizarse para ilustrar estos dos aspectos de la eficiencia algorítmica es el siguiente: supóngase $n$ enteros positivos almacenados en un arreglo $A$. ¿Son todos los enteros distintos, o al menos dos de ellos son el mismo? No es siempre fácil notar enteros duplicados:

$$A = [86, 63, 39, 98, 96, 38, 68, 88, 36, 83, 17, 33, 69, 66, 89, 96, 93]$$

Una forma algorítmica directa de detectar duplicados considera uno a uno los enteros, revisando el arreglo $A$ y buscando una coincidencia:

*REVISA*

1. **for** $i \leftarrow 1$ **to** $n - 1$ **do**

   *a*) **for** $j \leftarrow i + 1$ **to** $n$ **do**

      1) **if** $A[i] = A[j]$ **then output** $(i, j)$
         **exit**
         **else continue**

El algoritmo *REVISA* utiliza muy poco almacenamiento, tan solo un arreglo de $n$ localidades y dos variables $i$ y $j$. Consecuentemente, se puede decir que *REVISA* require $n+2$ palabras de almacenamiento, suponiendo que cada palabra es lo suficientemente larga para contener cualquier entero que se desee.

Para determinar cuánto tiempo require *REVISA*, se utiliza una técnica simple de diagramación que supone que cada línea del algoritmo tiene un costo de 1 unidad de tiempo para ejecutarse. Para detectar duplicados entre cuatro enteros positivos, el patrón de ejecución de *REVISA* puede dibujarse como sigue:

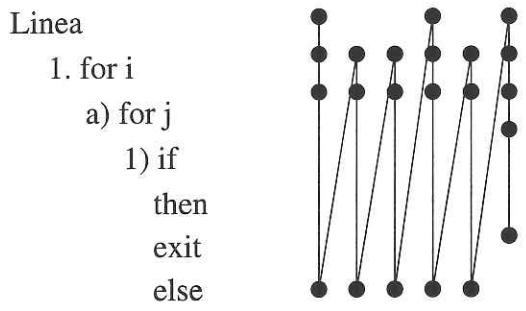


Figura 2.1: Ejecución de *REVISA*

Cada punto representa la ejecución de la instrucción ocupando el mismo nivel que el punto. Las líneas conectan a los puntos en orden de arriba hacia abajo, e izquierda a derecha, como una representación de cómo el algoritmo se ejecuta para una particular instancia del problema. En el ejemplo anterior, la última secuencia de puntos representa el punto donde $i = 3$ y $j = 4$. En este caso, la prueba $if A[3] = A[4]$ pasa, y el algoritmo se detiene.

Ya que el diagrama anterior tiene 22 puntos, se puede decir que *REVISA* requiere de 22 unidades de tiempo para procesar una secuencia como 7, 8, 4, 4. Si la secuencia hubiera sido 7, 8, 4, 5, entonces la última línea vertical se vería como las anteriores. Al menos, es necesario adoptar esta convención: alcanzar el enunciado **continue** después de pasar por el enunciado **then**, significa que el algoritmo ha recorrido todos los índices, y termina.

Este ejemplo da pie a algunas preguntas interesantes. Primero, dados $n$ enteros, ¿cuál es el mayor número de unidades de tiempo que *REVISA* requiere para determinar si la secuencia contiene enteros duplicados?, ¿cuánto tiempo, en promedio, requiere *REVISA* para determinar esto? Aquí, solamente se analiza la primera pregunta.

Evidentemente, *REVISA* siempre toma el mayor tiempo cuando en una secuencia de $n$ enteros, los dos últimos (y solo esos dos últimos) son iguales. Primero, cada ejecución del ciclo interior cuando el enunciado **if** falla requiere tres pasos.

Cuando $REVISA$ comienza con $i = 1$, $j$ va de 2 a $n$ y el ciclo interior se ejecuta $n - 1$ veces para un total de $3(n - 2)$ pasos. La expresión del tiempo total que require $REVISA$ es:

$$1 + 3(n - 1) + 1 + 3(n - 2) + \cdots + 1 + 3(2) + 1 + 4$$

Agrupando la expresión, se tiene que:

$$n + 1 + 3\sum_{k=1}^{n-1} = n + 1 + \frac{3n(n-1)}{2} = \frac{3n^2 - n + 2}{2}$$

La complejidad de tiempo del peor caso de un algoritmo operando en una entrada de tamaño $n$ es simplemente el máximo tiempo que el algoritmo requiere para operar cualquier entrada de tamaño $n$. La complejidad en tiempo del peor caso de $REVISA$, por la medida adoptada, es $(3n^2 - n + 2)/2$. Por supuesto, esta fórmula se basa en la suposición básica de que todas las instrucciones en un programa requieren el mismo tiempo. Ciertamente, este no es el caso para programas reales que se ejecutan en computadoras reales. Pero es frecuentemente posible asignar tiempos de ejecución (microsegundos o menos) a instrucciones individuales y repetir un análisis esencialmente similar al anterior. En tal caso, se puede obtener una fórmula con coeficientes diferentes, por ejemplo $(7{,}25n^2 - 1{,}14n + 2{,}83)/2$. Cualquier implementación concebible de $REVISA$ tiene una complejidad de tiempo en el peor caso de tipo cuadrático.

Por tal razón, se ha convenido la utiulización de una notación en orden-de-magnitud cuando se expresan las complejidades de tiempo y espacio para algoritmos y programas. Una función $f(n)$ se conoce como "la O grande de una función $g(n)$" si existe un entero $N$ y una constante $c$ tal que:

$$f(n) \leq c \cdot g(n) \qquad \text{para todo } n \geq N$$

Esto se expresa como:

$$f(n) = O(g(n))$$

Y si sucede que $g(n) = O(f(n))$, se dice entonces que ambas funciones tienen al mismo orden de magnitud.

Regresando a $REVISA$ y denotando su complejidad en tiempo del peor caso como $T_w(n)$, se tiene que:

$$T_w(n) = O(n^2)$$

Resulta que todas las funciones cuadráticas de $n$ tienen la misma orden de magnitud. Por tal razón, tiene sentido escribir la función cuadrática más sencilla de $n$ disponible para indicar el orden de magnitud de la complejidad en tiempo de $REVISA$. Claramente, $REVISA$ requiere solo de $O(n)$ de espacio de almacenamiento, ya que $n + 2 = O(n)$.

Constrastando a $REVISA$ se examina a continuación una aproximación algorítmica diferente del problema. El algoritmo $ALMACENA$ adopta la táctica simple de almacenar cada entero del arreglo $A$ en otro arreglo $B$ con un índice igual al entero mismo.

$ALMACENA$

1. **for** $i \leftarrow 1$ **to** $n$ **do**

   *a*) **if** $B[A[i]] \neq 0$
   **then output** $A[i]$
   **exit**
   **else** $B[A[i]] \leftarrow 1$

Se supone que el arreglo $B$ incialmente contiene ceros. Cada vez que un entero $a$ se encuentra, $B[a]$ toma el valor 1. De esta forma, enteros previamente encontrados se detectan por el enunciado **if**.

El uso de un número como base del cómputo de su dirección de almacenamiento es la base de la técnica conocida como *hashing*. En la versión primitiva de hashing que se utiliza en el algoritmo, se debe suponer que el arreglo $B$ es lo suficientemente largo para contener todos los enteros almacenados en $A$.

Es instructivo comparar las complejidades en tiempo del peor caso de ambos algoritmos. Considérese, por ejemplo, el desempeño de $ALMACENA$ sobre la misma secuencia de números 7, 8, 4, 4:

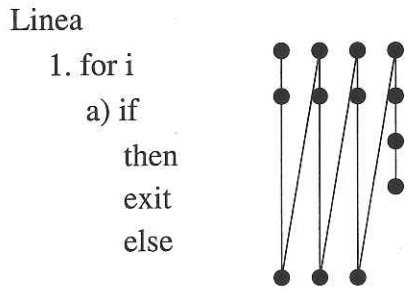


Figura 2.2: Ejecución de $ALMACENA$

La complejidad en el tiempo de $ALMACENA$ en este secuencia es de 13 unidades de tiempo, mientras que $REVISA$ requiere 22 unidades de tiempo. Este resulta ser el peor caso para $ALMACENA$ cuando $n = 4$, y en general, la complejidad en tiempo del peor caso para almacena es:

$$T_w(n) = 3n + 1 = O(n)$$

Por otro lado, $ALMACENA$ requiere mucho más almacenamiento que $REVISA$: si se involucran números de hasta $m$ bits, entonces $ALMACENA$ requiere hasta $2^m$ localidades de memoria para su operación.

Mediante comparar la complejidad en tiempo del peor caso de $REVISA$ y $ALMACENA$, es obvio qué algoritmo es superior cuando se grafican ambas complejidades (figura 2.3).

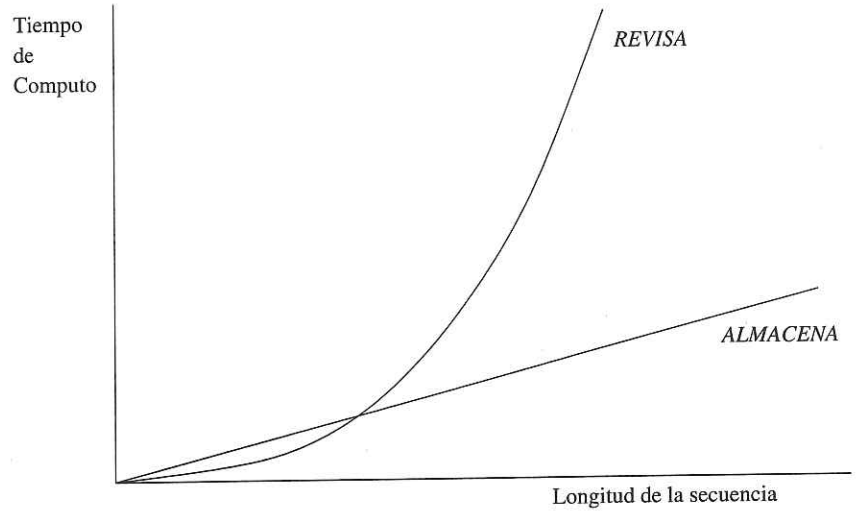


Figura 2.3: Las complejidades en tiempo del peor caso para $REVISA$ y $ALMACENA$

Aun cuando $REVISA$ requiere menos tiempo que $ALMACENA$ en secuencias con longitudes de 1 y 2, para cuando $n = 3$, $ALMACENA$ muestra ya superioridad. Tal conclusión puede observarse, sin importar qué complejidad en tiempo tengan ambos algoritmos – mientras $REVISA$ sea cuadrática y $ALMACENA$ sea lineal. Tarde o temprano un algoritmo con complejidad en tiempo lineal se desempeña mejor que un algoritmo con complejidad cuadrática.

Lo mismo sucede con la complejidad en espacio de los algoritmos: un algoritmo con requerimientos de memoria cuadráticos excede mucho más rápido la capacidad de memoria de una computadora que un algoritmo con complejidad lineal.

Los estudios de las complejidades en tiempo y espacio de algoritmos tienden a concentrarse en la pregunta ¿qué tan rápido un problema dado puede resolverse algorítmicamente? A menos que los requerimientos de almacenamiento sean exorbitantes, la pregunta principal sobre la eficiencia de un algoritmo es, en general, ¿podemos encontrar un algoritmo más rápido para resolver el mismo problema?

Supóngase que hay un problema $P$ para el cual el algoritmo conocido más rápido tiene una complejidad en el peor caso de $O(f(n))$. Si puede demostrarse que no hay un algoritmo más rápido, uno que tenga una complejidad $g(n)$ donde $g(n) = O(f(n))$ pero $f(n) \neq O(g(n))$, entonces el problema mismo puede decirse que tiene complejidad $O(f(n))$. Por ejemplo, bajo las suposiciones razonables acerca de cómo un algoritmo de ordenamiento puede realizar comparaciones, puede demostrarse que ordenar $n$ enteros es un problema $O(n \log n)$. Desafortunadamente, muy pocos problemas conocidos tienen complejidad conocida; probar que un algoritmo es el más rápido posible para un problema dado es notoriamente difícil.

Ciertamente, algunos problemas parecen tener complejidades exponenciales. Nadie ha hallado un algoritmo en tiempo polinomial para ninguno de ellos. Aun cuando un algoritmo $O(2^n)$ puede existir, nadie ha descubierto un algoritmo que se ejecute en tiempo $O(n^{1000})$. No es difícil notar que cuando $n$ es suficientemente grande, $n^{1000} < 2^n$.

Todos los problemas que se pueden desear resolver, y se cuenta con una medida $n$ de tamaño en ejemplos, puede presumiblemente ser clasificada en orden de su complejidad. En la parte más alta de la lista vienen aquellos problemas que requieren $O(2^n)$ pasos (o más) para resolverse. Estos están para siempre fuera del alcance de las computadoras secuenciales. En seguida, aparecen los problemas con soluciones que tienen tiempos polinomiales, es decir, en orden de polinomios. Algunos problemas pueden tener complejidad $O(n^2)$; otros pueden tener complejidad $O(n)$, o menos. De hecho, hay problemas con complejidad intermedia, por ejemplo:

$$O(n) < O(n \log n) < O(n^{1/2}) < O(n^2)$$

En 1965, el matemático Jack Edmonds fue el primero en llamar la atención en la distinción entre problemas con tiempos exponenciales y polinomiales y sus algoritmos. Parecía muy extraño que algunos problemas, como encontrar el camino más corto entre dos puntos de una gráfica, tuvieran una solución algorítmica de $O(n^2)$, mientras que otros problemas que parecían relacionados cercanamente, como encontrar el camino más largo, tuvieran solo una solución algortítmica de $O(2^n)$. Esta distinción fue explorada y, hasta cierto punto, explicada por Stephen Cook en 1971.

# Capítulo 3

# Satisfactibilidad
## *Un Problema Central*

El problema de "satisfactibilidad" (*satisfiability*) en el algebra boolena parece, a primer vista, bastante simple. Dada una expresión como:

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

encontrar una asignación de valores de verdad (0 ó 1) para las variables $x_1$, $x_2$, $x_3$ y $x_4$, de modo que la expresión misma sea verdadera. Esto significa que cada subexpresión como $x_1 + \bar{x_3} + x_4$ debe ser verdadera (tener valor 1). Si se busca una asignación satisfactoria para la expresióin que se muestra anteriormente, se podría intentar algo como $x_1 = 1$, $x_2 = 1$, $x_3 = 1$ y $x_4 = 1$, sólo para descubrir que la subexpresión $\bar{x_1} + \bar{x_2} + \bar{x_4}$ tiene valor $0 + 0 + 0 = 0$ bajo tal asignación, violando la condición de que cada una de las subexpresiones debe ser verdadera. Así, se sigue que tal asignación no satisface la expresión. Sin embargo, se puede intentar ahora con $x_1 = 0$, $x_2 = 1$, $x_3 = 1$ y $x_4 = 1$. Se puede demostrar que todas las subexpresiones son verdaderas para esta asignación, y por lo tanto, ésta "satisface" a la expresión. Las expresiones que tienen la forma anterior se dice que están como "suma de productos".

Otra forma de ver el problema de satisfactibilidad es examinar el circuito correspondiente a la expresión en producto de sumas, y preguntarse cuál combinación de variables de entrada causa que el circuito como un todo genere en su salida un valor 1 (figura 3.1).

Si el problema anterior da la impresión que los problemas de satisfactibilidad son siempre fáciles de resolver, considérese el siguiente ejemplo:

$$(x_1+\bar{x_2}+x_3)(\bar{x_1}+\bar{x_2}+\bar{x_3})(\bar{x_1}+\bar{x_2}+x_3)(\bar{x_1}+x_2+\bar{x_3})(x_1+x_2+x_3)(\bar{x_1}+x_2+x_3)(x_1+x_2+\bar{x_3})$$

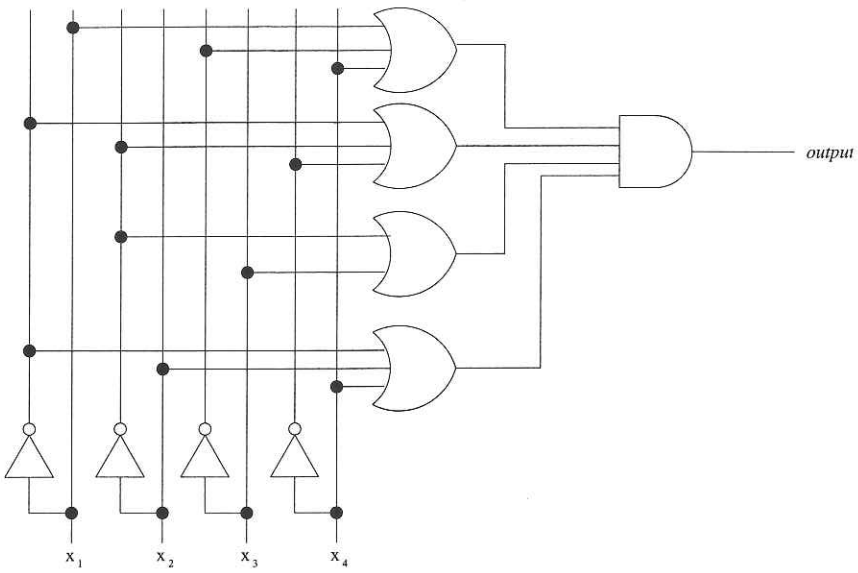


Figura 3.1: ¿Qué entradas producen una salida 1?

Este producto de sumas tiene más subexpresiones que el anterior, pero menos variables. Se podría intentar un número de asignaciones antes de llegar a:

$$x_1 = 0$$
$$x_2 = 1$$
$$x_3 = 1$$

De hecho, ésta es la única asignación de variables que satisface la expresión. Es también posible que para una expresión en producto de sumas no haya una asignación que la satisfaga, de modo que se le llama "insatisfactible".

La manera obvia de resolver algorítmicamente un problema de satisfactibilidad con $n$ variables lógicas es generar todas las posibles combinaciones sistemáticamente y probar cada una con la expresión dada. Si satisface la expresión, se ha terminado; de otra forma, se continúa generando y probando. Si ninguna de las $2^n$ posibles asignaciones de variables satisface la expresión, entonces ésta es insatisfactible. Desafortunadamente, el tiempo que toma este algoritmo tiende a crecer exponencialmente con $n$; con instancias satisfactibles, el problema puede generar cualquiera entre 1 y $2^n$ asignaciones antes de obtener una solución. Con instancias insatisfactibles, debe probar con todas las $2^n$ asignaciones, lo que representa una larga espera excepto para los problemas más pequeños.

Un algoritmo algo mejor fue descubierto por M. Davis y H. Putnam a principios de los años 1960s. En su forma más simple, el algoritmo Davis-Putnam puede presentarse como sigue:

*DAVIS – PUTNAM*

1. **procedure** *split(E)*
   - *a*) **if** $E$ tiene una subexpresión vacía **then return**
   - *b*) **if** $E$ no tiene subexpresiones **then exit**
   - *c*) Seleccione la siguiente variable no asignada $x_i$ en $E$
   - *d*) **split**$(E(x_i = 0))$
   - *e*) **split**$(E(x_i = 1))$

Inicialmente, $E$ es la expresión dada en producto de sumas para la cual se busca una asignación de variables satisfactoria. A cada paso de la recursión, $E$ representa la expresión en ese paso. Si la asignación actual parcial falla en satisfacer una de las subexpresiones o las ha satisfecho todas, el algoritmo regresa o sale, respectivamente. De otra forma, la subexpresión actual se revisa de izquierda a derecha; y si la primera variable que aparece, por ejemplo $x_i$, se vuelve la base de dos llamadas a **split**. Para la primera, se forma la expresión $E(x_i = 0)$. Cada subexpresión que contenga a $\bar{x}_i$ se satisface por $x_i = 0$ y por lo tanto se borra de la expresión y cada aparición de $x_i$ se elimina de toda subexpresión en la que aparezca. Para la segunda llamada, se forma la expresión $E(x_i = 0)$. Las subexpresiones que contienen $x_i$ y apariciones de $\bar{x}_i$ se eliminan de $E$.

Si en cualquier momento se genera una expresión conteniendo una subexpresión vacía, entonces esa subexpresión falla en ser satisfecha por la asignación actual parcial de variables, y no hay caso en continuar. Si, sin embargo, todas las subexpresiones han sido eliminadas, entonces todas han sido satisfechas por la asignacion actual parcial de variables, y pueden asignarse valores arbitrarios a cualquier variable remanente.

El algoritmo Davis-Putnam, siendo un procedimiento recursivo que se llama a sí mismo dos veces en cada paso, explora un árbol implícito de búsqueda para cada expresión que se le de. El árbol correspondiente al primer ejemplo se muestra en la figura 3.2. Cada $\varnothing$ representa una expresión de la cual todas sus subexpresiones han sido eliminadas, y por tanto, se tiene una asignación parcial exitosa. Cada aparición de $(\varnothing)$ indica una subexpresión vacía, pero una asignación parcial fallida. En este ejemplo en particular, el algoritmo Davis-Putnam encontraría la asignación satisfactoria casi inmediatamente con $x_1 = 0$, $x_2 = 0$ y $x_3 = 0$. La última variable, $x_4$, puede tomar el valor de 0 ó 1 arbitrariamente. El éxito de este algoritmo con esta instancia, sin embargo, tiene más que ver con la relativa abundancia de soluciones que con cualquier otro factor. En general, el algoritmo Davis-Putnam podría tomar mucho tiempo con ciertas expresiones, pero normalmente encuentra una solución en un tiempo considerablemente menor que el algoritmo de búsqueda exhaustiva

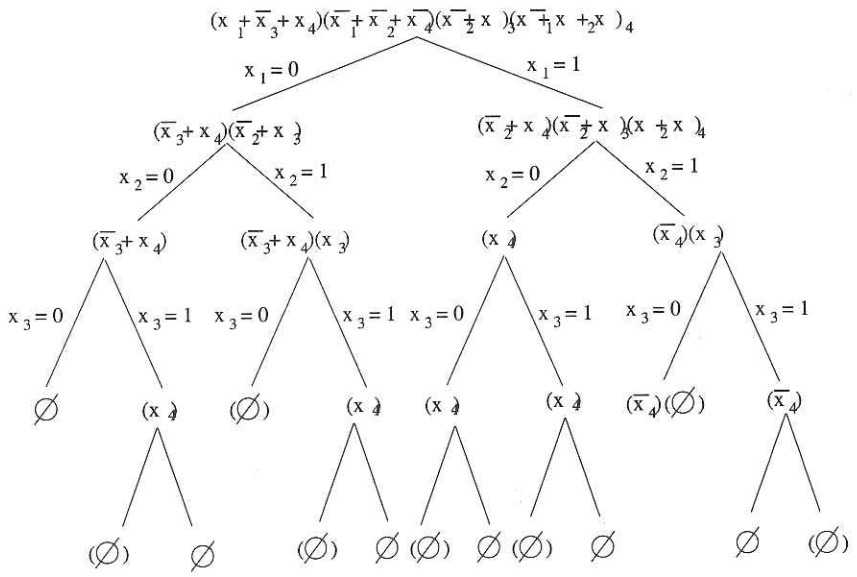

Figura 3.2: El árbol implícito en el algoritmo Davis-Putnam

descrito anteriormente. La razón de esto recae en la habilidad del algoritmo Davis-Putnam de "podar" ramas no exitosas de su árbol de búsqueda.

No hay algoritmo conocido en el presente que garantice resolver un problema de satisfactibilidad de $n$ variables en un tiempo menor que el tiempo exponencial. Específicamente, no hay algoritmo que garantice resolverlo en tiempo polinomial, es decir, en un tiempo proporcional a $n^k$ o menos para alguna potencia fija $k$. ¿Encontrar un algoritmo eficiente para el problema de satisfactibilidad se debe a la relativa incapacidad humana, o es sólo posible que tal algoritmo no exista?

Para hacer las cosas más complejas, hay una versión mucho más sencilla de este problema que parece ser igualmente difícil: encuentre un algoritmo eficiente (en tiempo polinomial) que, para cada expresión en producto de sumas, genere un 1 si la expresión es satisfactible, y un 0 si no. A esto se le conoce con el nombre de "problema de decisión de safisfactibilidad" (*safisfiability decision problem*, o simplemente problema de satisfactibilidad cuando el contexto es claro). No se requiere una asignación de variables, solamente decidir si la expresión es o no satisfactible.

Para ver porqué el problema de satisfactibilidad es central (y difícil), es necesario revisar las expresiones en producto de sumas bajo una luz diferente. En lugar de verlas sólo como expresiones lógicas que pueden ser verdaderas o falsas dependiendo de cómo se asigne sus variables, se pueden considerar sus fórmulas como un tipo de lenguaje en el cual muchas ideas matemáticas pueden expresarse. Por

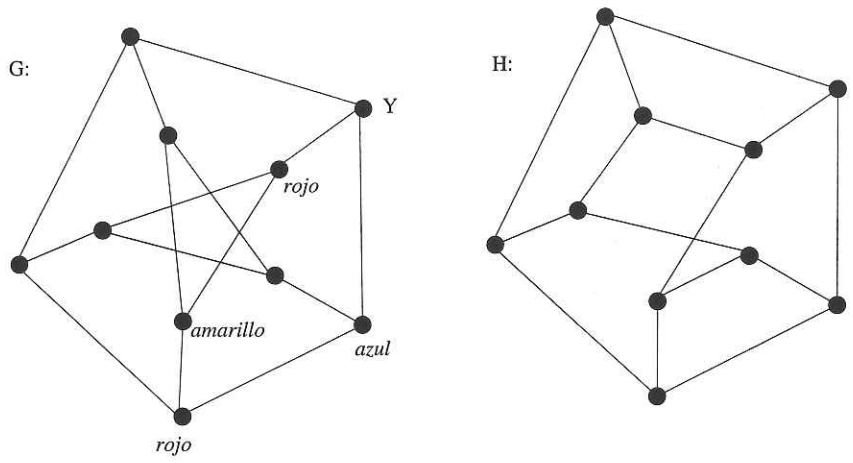


Figura 3.3: ¿Qué grafo es coloreable usando tres colores?

ejemplo, considérese el bien conocido problema en teoría de grafos: dado un grafo $G$, coloréese sus vértices en rojo, amarillo y azul de modo que si dos vértices se unen por una arista, entonces tales vértices reciben colores diferentes.

La figura 3.3 muestra dos grafos $G$ y $H$. Uno es coloreable con tres colores y el otro no. Una manera de hacerlo, por ejemplo, es comenzar a colorear cada grafo en la forma indicada, pero tarde o temprano en uno de ellos fallan todos los intentos para extender el uso de tres colores para todo el grafo. De entre los grafos que pueden ser coloreados (en este caso, etiquetados) con tres colores, algunos son fáciles y otros muy difíciles. De hecho, nadie ha logrado encontrar un algoritmo eficiente para el problema de los tres colores. ¿Hay un algoritmo que, para cualquier grafo $G$ con $n$ vértices, encuentre una forma de usar los colores (si existe) para $G$ en no más de $n^k$ pasos? Se puede igualmente solicitar un algoritmo eficiente para el problema de decisión correspondiente: ¿existe un algoritmo que, para cualquier grafo $G$ con $n$ vértices, decida si $G$ es coloreable con tres colores en no más de $n^k$ pasos?

Resulta que el problema de colorear grafos con tres colores está íntimamente ligado al problema de satisfactibilidad para expresiones lógicas. Esto se debe al potencial de tales expresiones de describir ideas matemáticas.

Hay un algoritmo, que se muestra a continuación, el cual toma un grafo arbitrario $G$ y genera una expresión en producto de sumas $E(G)$. Este algoritmo tiene dos propiedades importantes:

- Si $G$ tiene $n$ vértices, entonces el algoritmo requiere no más de $n^2$ pasos para obtener $E(G)$.
- El grafo $G$ es coloreable con tres colores si y solo si $E(G)$ es satisfactible.

Antes de mostrar el algoritmo, es importante remarcar que su existencia implica que el problema de decisión de satisfactibilidad es sólo tan difícil (no es más difícil que) el problema de los tres colores. Esto porque si se encuentra un algoritmo en tiempo polinomial para el problema de satisfactibilidad, sería entonces sólo necesario acoplarlo con el algoritmo siguiente para volverlo un algoritmo eficiente para el problema de decisión de los tres colores. La composición de dos algoritmos en tiempo polinomial es aún un algoritmo en tiempo polinomial, y una respuesta de sí o no para el primer problema se traduce en una respuesta de sí o no para el segundo.

*TRANSFORM*

1. **for each** vértice $v_i$ en $G$ **output** $(r_i + y_i + b_i)$
2. **for each** arista $(v_i, v_j)$ en $G$ **output** $(\bar{r_i} + \bar{r_j})(\bar{y_i} + \bar{y_j})(\bar{b_i} + \bar{b_j})$

Suponiendo que la cadena de expresiones de salida del algoritmo se da de forma correcta, es decir, que las subexpresiones se escriben como un producto, no es difícil establecer la segunda propiedad anteriormente presentada. Sin embargo, si $G$ es coloreable con tres colores, entonces es posible asignar valores a las variables lógicas $r_i$, $y_i$ y $b_i$ como sigue: sea "color" un arreglo de tres colores de $G$ y sea:

$$r_i = \begin{cases} 1 & \text{si color } (v_i) \text{ es rojo} \\ 0 & \text{de otra forma} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{si color } (v_i) \text{ es amarillo} \\ 0 & \text{de otra forma} \end{cases}$$

$$b_i = \begin{cases} 1 & \text{si color } (v_i) \text{ es azul} \\ 0 & \text{de otra forma} \end{cases}$$

Ciertamente, cada vértice recibe un color, así que una de las variables en $r_i + y_i + b_i$ es verdadera. Al mismo tiempo, cuando $v_i$ y $v_j$ se encuentran unidas por una arista, reciben colores diferentes. No pueden ambos ser rojos, de modo que uno de $\bar{r_i}$ y $\bar{r_j}$ debe ser verdadero en $\bar{r_i} + \bar{r_j}$. Similarmente, $\bar{y_i} + \bar{y_j}$ y $\bar{b_i} + \bar{b_j}$ se satisfacen. Es tan solo un poco más difícil mostrar que si se da una asiganción satisfactoria para la expresión en producto de sumas producido por el algoritmo, entonces se obtiene un arreglo de tres colores para $G$ de ella. Ya que esta última solución puede obtenerse de la anterior muy rápido, lo que se ha dicho de la dificultad relativa de los problemas de decisión se aplica son una fuerza igual a los problemas más generales. Algorítmicamente hablando, es al menos tan difícil encontrar una solución para el problema de satisfactibilidad (cuando existe) que encontrar una solución para el problema de los tres colores.

Hay muchos problemas como el problema de los tres colores. Cada uno puede transformarse al problema de satisfactibilidad, y cada uno parece ser muy difícil por

el hecho de que no hay un algoritmo conocido que lo solucione en tiempo polinomial. En este sentido, satisfactibilidad es un problems "central", y en el mismo sentido, satisfactibiidad es ciertamente un problema muy difícil. Sería razonable sospechar fuertemente que un algoritmo rápido para solucionar el problema de satisfactibilidad no existe.

# Capítulo 4

# Funciones No Computables
## *El Problema del Castor Ocupado*

Los castores son animales bien conocidos por su capacidad de realizar una labor hasta que la han terminado. Se ocupan en la construcción de sus presas mediante cortar y apilar troncos en los ríos. Los troncos son traídos y llevados uno a uno en cada viaje.

Las máquinas de Turing que van y vienen sobre sus cintas leyendo y escribiendo símbolos, parecen castores. ¿Qué tan ocupada puede estar una máquina de Turing? Algunas máquinas de Turing se encuentran infinitamente ocupadas en el sentido de que nunca se detienen. Más aún, muchas de las que se detienen pueden estar ocupadas por largos periodos de tiempo, alterando la apariencia de su cinta en cada ejecución. Así, parece sensato proponer esta pregunta en el contexto de una cinta inicialmente vacía, para todas las máquinas que se detienen con tal cinta como entrada.

En 1962, Tibor Rado, un matemático húngaro, inventa lo que se conoce actualmente como el "problema del castor ocupado" (*busy beaver problem*): dada una máquina de Turing de $n$ estados con un alfabeto de dos símbolos $\{0, 1\}$, ¿cuál es el máximo número de unos que la máquina puede imprimir en una cinta blanca (llena de ceros) antes de detenerse? No hay duda de que este número, que se denota como $\Sigma(n)$, existe ya que el número de máquinas de Turing de $n$ estados es finito.

Lo que hace este problema interesante, entre otras cosas, es su dificultad. El problema del castor ocupado no puede resolverse en general por computadoras, ya que la función $\Sigma(n)$ crece más rápido que cualquier función computable $f(n)$. Para observar esto, supóngase que $B$ es una máquina de Turing que obtiene la función del castor ocupado $\Sigma(n)$, y que tiene $q$ estados. Esto significa que cuando $B$ se confronta con una cinta de entrada con el entero $n$ escrito en ella, $B$ produce el número $\Sigma(n)$. Si perder generalidad, se puede suponer que ambos $n$ y $\Sigma(n)$ se

escriben en notación binaria.

Ahora bien, sea $B_n$ una máquina de Turing con $n$ estados e imprime $\Sigma(n)$ unos antes de detenerse, sea $C$ una máquina que convierte un número binario a su equivalente en notación unaria, y sea $A$ una máquina que convierte una cinta en blanco a una con el número $n$ escrita en binario sobre de ella. Las tres máquinas, agrupadas, pueden representarse como $ABC$. Esta máquina comienza con una cinta en blanco y termina con $\Sigma(n)$ unos en la cinta. Más aún, tiene sólo $\lceil log\, n \rceil + q + r$ estados, ya que se requieren $\lceil log\, n \rceil$ estados para el funcionamiento de $A$ y un número constante de estados $q$ y $r$ para el funcionamiento de $B$ y $C$, respectivamente.

Para cualquier entero $n$ tal que $n \geq \lceil log\, n \rceil + q + r$, la máquina $ABC$ imprime tantos unos como $B_n$, y sin embargo, usa menos estados. De hecho, es fácil demostrar que $\Sigma(n) > \Sigma(m)$ para dos enteros $n$ y $m$ si $n > m$, obteniendo una obvia contradicción. Ya que las máquinas $A$ y $C$ claramente pueden existir, entonces $B$ no puede existir. De acuerdo con esto, $\Sigma(n)$ no es una función computable.

La siguiente tabla resume el estado presente sobre el conocimiento que se tiene de algunos valores de $\Sigma(n)$.

| $n$ | $\Sigma(n)$ |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 6 |
| 4 | 13 |
| 5 | $\geq 1915$ |

El salto de $\Sigma(4) = 13$ a $\Sigma(5) \geq 1915$ es sintomático de la naturaleza no-computable de $\Sigma(n)$. La inequidad se debe a George Uhing, un programador de Nueva York que descubre en 1984 una máquina de Turing capaz de producir 1915 unos antes de detenerse. En general, se puede seleccionar una función computable arbitraria como $2^n$ y escribir:

$$\Sigma(n) \geq 2^n$$

(para una $n$ lo suficientemente grande) con la confianza de estar en lo correcto. Pero aún hay más.

La expresión anterior fue descubierta por Rado en 1962. Más tarde, fue superada en cierto sentido por un resultado algo más dramático encontrado por M.W. Green en 1964. Para cualquier función computable $f(n)$:

$$f(\Sigma(n)) < \Sigma(n+1)$$

para infinitamente muchos valores de $n$.

Así, por ejemplo, si se considera $f$ como:

$$f(m) = m^{m^{m^{\cdots^{m}}}}$$

Para cualquier número fijo de exponenciaciones, de cualquier modo se debe concluir que $f(\Sigma(n)) < \Sigma(n+1)$. Es decir:

$$\Sigma(n+1) > \Sigma(n)^{\Sigma(n)^{\Sigma(n)^{\cdots^{\Sigma(n)}}}}$$

para infinitamente muchos valores de $n$.

Parecería que lo más indicado sería intentar resolver el problema del castor ocupado mediante mejorar los límites inferiores descritos en la tabla anterior. Sin embargo, en un concurso sobre el problema del castor ocupado realizado en los 1980s se demostró que esto no es una tarea fácil aún cuando $n = 5$.

Una razón de la enorme dificultad del problema del castor ocupado recae en la capacidad de las relativamente pequeñas máquinas de Turing para codificar conjeturas matemáticas profundas, como el último "teorema" de Fermat o la conjetura de Goldbach (todo número par es la suma de dos primos). Saber si tales máquinas se detienen es realmente importante para comprobar o no tales conjeturas. Si $\Sigma(n)$ es conocido para el valor de $n$ correspondiente a la conjetura, se sabría (en principio, al menos) cuánto tiempo habría que esperar para que la máquina se detenga.

Algunos teóricos dudan que se pueda obtener $\Sigma(6)$.

# Capítulo 5

# NP-Completitud
## *Un Muro Inextricable*

El término "NP-completo" se usa comúnmente, pero frecuentemente no se entiende correctamente. En términos prácticos, un problema NP-completo es aquél que puede resolverse en una computadora solo si se espera un tiempo extraordinariamente largo para obtener la solución. En términos teóricos, un problema NP-completo se entiende mejor como una aplicación del Teorema de Cook. El primer problema NP-completo, "satisfactibilidad" (*satisfiability*), fue descubierto por Stephen Cook mientras terminaba su doctorado en la Universidad de California en Berkeley, en 1970. Cook descubrió una transformación genérica para todo problema en una cierta clase llamada NP a un solo problema en lógica llamado satisfactibilidad (SAT) (figura 5.1).

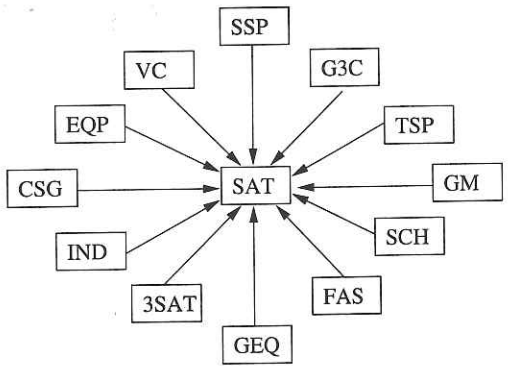


Figura 5.1: Algunos problemas que se transforman a satisfactibilidad

La transformación descubierta por Cook es *genérica* debido a que puede es-

pecializarse para actuar sobre un solo problema en NP. Por ejemplo, transforma cualquier instancia del problema del agente viajero (*Traveling Salesperson Problem* ó TSP) a una instancia del problema de satisfactibilidad de tal manera que:

- La transformación puede computarse en tiempo polinomial.
- La instancia de TSP tiene una solución si y solo si la correspondiente instancia SAT tiene solución.

Estas dos propiedades se mantienen cuando la transformación genérica se especializa a cualquier problema del número infinito de problemas en NP: solo es necesario substituir en el segundo punto anterior "TSP" por el nombre del problema actual.

La implicación principal del Teorema de Cook es que la satisfactibilidad es al menos difícil desde un punto de vista de tiempo polinomial, como cualquier otro problema en NP. Debe ser al menos tan difícil encontrar un algoritmo de tiempo polinomial para la satisfactibilidad como para cualquier otro problema en NP. En el momento en que se tenga un buen algoritmo para la satisfactibilidad a la mano, se puede seleccionar cualquier problema en NP y aplicar la transformación genérica de Cook para obtener, en tiempo polinomial, una instancia de satisfactibilidad. Aplicando el algoritmo de satisfactibilidad determinaría la existencia de una solución a la instancia de satisfactibilidad, y por lo tanto, una instancia del problema en NP, que por la segunda condición, tendría solución en tiempo polinomial.

La mayoría de los problemas que se intenta resolver mediante una computadora tienen soluciones que exponen una estructura. La solución al TSP, por ejemplo, sería encontrar una ruta de costo mínimo que cubra todas las ciudades de un territorio dado. Los problemas en la clase NP tienen respuestas sencillas: sí o no. Muchos problemas que tienen respuestas en forma de estructuras, como TSP, pueden fácilmente convertirse en problemas de decisión simplemente mediante preguntar si una solución de cierto tamaño existe. Por ejemplo, un instancia de decisión de TSP podría consistir en una red de ciudades y un límite en kilómetros. ¿Hay una ruta que cubra todas las ciudades con una distancia total menor que el límite? ¿sí o no?

NP es un acrónimo: N por *No-deterministico*, y P por *tiempo Polinomial*. La clase NP, entonces, se define como el conjunto de todos los problemas de decisión que pueden resolverse por una computadora no-determinística en tiempo polinomial. Varios modelos para las computadoras actuales son o pueden hacerse no-determinísticos. De hecho, en general, no es difícil considerar una máquina de Turing no-determinística. Tan solo con equiparla con un dispositivo que escriba una suposición en alguna porción reservada de la cinta, y considerar otra porción donde se coloque una instancia de algún problema de decisión. Se carga la máquina de Turing con un programa que averigüe, basado en la suposición, si la respuesta es

sí o no. Si por todas las instancias positivas (que responden sí) al problema, y solo para estas, hay un conjunto de valores de la suposición que causan que la máquina de Turing responda "sí", entonces se dice que el programa resuelve el problema de decisión. Y lo hace en un tiempo polinomial fijo si hay algún polinomio que limite el número de pasos en al menos un cómputo positivo, por cada posible instancia positiva que la máquina de Turing pueda encontrar. En tal caso, se dice que el problema pertenece a NP.

Como es usual en análisis de complejidad en tiempo, el tamaño del problema sirve como variable independiente para el polinomio. Normalmente en la teoría de NP-completitud el tamaño de una instancia es la longitud de la cadena de símbolos que la codifica en la cinta de una máquina de Turing. Se supone que la representación es razonablemente económica, ya que muchos símbolos innecesarios podrían distorsionar la verdadera complejidad del cómputo.

El problema de la partición (PRT) se presenta enseguida en forma de decisión: dado un conjunto de enteros positivos, ¿hay una partición del conjunto en dos partes de modo que ambas sumen el mismo número? ¿sí o no? La figura 5.2 muestra cómo una instancia particular del problema puede resolverse con una máquina no-determinística de Turing.

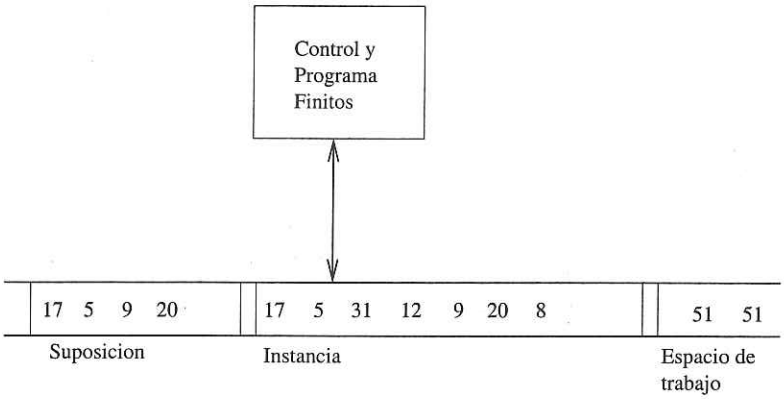


Figura 5.2: Una máquina no-determinística de Turing resolviendo el problema de la partición.

En este ejemplo, la cinta de la máquina se divide en tres regiones: una para la suposición, una para la instancia, y otra como espacio de trabajo para manejar los resultados intermedios. ¿Hay una partición del conjunto de enteros 17, 5, 31, 12, 9, 20, 8? Sí, sí la hay, y mediante cierto procedimiento la máquina no-determinística de Turing especifica una de las partes. La suma de 17, 5, 9 y 20 es 51, y la suma del complemento 31, 12 y 8 es también 51. La máquina de Turing, después de suponer

aleatoriamente, funciona de manera determinística. Moviéndose hacia adelante y hacia atrás sobre la cinta, la máquina (dirigida por su programa, por supuesto), primero se asegura que los enteros de su suposición también existan en la instancia. Suma los números y escribe el resultado en el espacio de trabajo. Finalmente, suma todos los enteros en la instancia y lo divide entre 2, comparando el resultado con la suma almacenada en el espacio de trabajo. Si los números son iguales, la salida es "sí"; de otra forma, la salida es "no". Algunas suposiciones hacen que la máquina de Turing nunca se detenga, pero estas no son considerados en la complejidad no-determinística en tiempo.

Para cada instancia positiva de PRT, existe un cómputo positivo de longitud mínima. De entre todas las instancias positivas de tamaño $n$, uno de estos cómputos de mínima longitud resulta ser un máximo. Este se toma como la complejidad del procedimiento para una entrada de tamaño $n$. Claramente, limita la longitud de los cómputos positivos por cada instancia positiva de tal tamaño. Si, de entre todos los valores de $n$, hay un polinomio que limite este máximo, entonces PRT está en NP.

Sería una pérdida de tiempo escribir el programa para la máquina no-determinística de Turing para PRT y entonces analizarlo para determinar el valor preciso de la complejidad del programa para cada tamaño de la entrada. Aquí, sólo concierne si la complejidad está acotada por un polinomio. Es suficiente decir que en el caso de PRT, el procedimiento descrito anteriormente puede llevarse a cabo por un programa que nunca requiere más de $O(n^3)$ pasos, donde $n$ es el tamaño de la instancia.

De hecho, PRT, como SAT, son NP-completos. Pero no todos los problemas en NP son necesariamente NP-completos. Por ejemplo, si se altera levemente PRT mediante requerir además que cada número en una parte sea al menos tan grande como un número en la otra parte, se obtiene un nuevo problema. El programa no-determinístico para este nuevo problema hace todo lo que el programa para PRT hace. También verifica que cada número en la parte supuesta predominen en la parte remanente de la instancia. Esto, también, puede realizarse en un número de pasos, es decir, $O(n^3)$. Por lo tanto, en nuevo problema está también en NP.

Sin embargo, es innecesario apelar a una computadora no-determinística para resolver este último problema. Puede resolverse determinísticamente en tiempo polinomial. Primero, ordene los números dados en orden decreciente, y entonces obtenga su suma. Enseguida, revise los números de principio a fin, acumulando la suma parcial durante el proceso. Si la suma parcial llega a ser igual que la mitad del total, la salida es "sí". Este algoritmo puede ciertamente hacerse que opere en un número polinomial de pasos.

La clase P consiste de todos aquellos problemas de decisión que, como el problema anterior, pueden resolverse en tiempo polinomial por una computadora determinística.

Ahora bien, la famosa pregunta ¿P = NP? pregunta si las dos clases de problemas son iguales. No es difícil mostrar que P es un subconjunto de NP, de modo que la verdadera pregunta es si hay problemas de decisión en NP que no tengan una solución determinística en tiempo polinimial. A pesar de la enorme potencia de cómputo de un dispositivo que siempre hace las suposiciones correctas, no es posible comprobar que esto haga ninguna diferencia.

Por otro lado, la inhabilidad para resolver determinística y eficientemente algunos problemas en NP hace sospechar que la contención es propia. Los principales candidatos para ejemplos de problemas en NP-P son los NP-completos. Como se ha mencionado anteriormente, SAT es NP-completo. Antes de describir otros problemas insolubles similares, vale la pena ser más específicos acerca del Teorema de Cook y lo que significa ser NP-completo.

El punto central del Teorema de Cook es la transformación genérica mencionada anteriormente. Como entrada, toma una instancia del problema $ABC$ (por decir un nombre) y un algoritmo no-determinístico que resuelve $ABC$ en tiempo polinomial. La transformación genérica genera un número de enunciados lógicos que describen el comportamiento de máquinas de Turing en general. También se generan enunciados que describen la acción de una máquina de Turing bajo la dirección de un programa arbitrario con entrada arbitraria. En este caso, se usa el programa para la máquina de Turing que resuelve el problema $ABC$ y una instancia específica de $ABC$. El programa genera los enunciados en un tiempo $O(p^3(n))$, donde $p(n)$ es la complejidad en tiempo del programa (no-determinístico) que resuelve $ABC$. Así, en un tiempo $O(p^3(n))$, la transformación genérica genera un conjunto de enunciados lógicos que describen al programa que resuelve $ABC$ completamente en términos de sus acciones sobre una instancia dada. El Teorema de Cook se propone de tal modo que el conjunto resultante de enunciados tiene una asignación satisfactoria si y solo si la instancia original de $ABC$ es una instancia positiva.

Supóngase ahora que se obtiene un algoritmo determinístico que resuelve el problema de la satisfactibilidad en tiempo polinomial. ¿Cómo usarlo para resolver el problema $ABC$?

Primero, se debe transformar la instancia a resolver a un sistema de enunciados equivalentes que se resuelve en $O(p^3(n))$, usando la transformación genérica de Cook. Enseguida, se aplica el algoritmo anterior, que es capaz de detectar las instancias positivas (que obtienen "sí" como respuesta) de SAT en un tiempo $O(q(m))$ (polinomial, por supuesto), donde $m$ es el tamaño de la instancia de SAT que se le proporciona como entrada. Resulta sencillo reconocer que la composición de dos procedimientos da como resultado un "sí" si y solo si la instancia original de $ABC$ es una instancia positiva. Más aún, el procedimiento compuesto se ejecutaría en:

$$O(q(p^3(n))) \text{ pasos}$$

Este podría ser un polinomio muy grande, pero de cualquier modo es tan solo un polinomio. Obviamente, cualquier problema de decisión en NP puede ahora

resolverse en tiempo polinomial. Esto significaría que P = NP.

Sin embargo, es poco probable obtener un algoritmo con tales características, no sólo por las previas y poco exitosas experiencias en la búsqueda de algoritmos veloces, sino también por que una gran cantidad de problemas en NP (aquéllos con los que se realmente se tienen problemas para resolverse) resultan ser NP-completos.

Recuérdese: lo que hace que SAT sea un problema NP-completo es el hecho de que al resolver cualquier instancia de cualquier problema en NP es equivalente a resolver alguna instancia de SAT. Lo mismo resulta cierto para otros problemas particulares de decisión.

# Capítulo 6

# El Teorema de Cook
## *Lo Básico*

El Teorema de Cook establece que el problema de "satistactibilidad" (*satisfiability*) es NP-completo. Lo hace mediante presentar una transformación genérica que en tiempo polinomial mapea cada y todo problema en NP al problema de satisfactibilidad. Los detalles de tal transformación, lo básico, consiste meramente en varios sistemas de enunciados que expresan en forma lógica la operación de una máquina de Turing. Para cada problema en NP, la máquina de Turing involucrada es una no-determinística, y resuelve el problema en tiempo polinomial. En cualquier caso, la transformación genérica puede verse como la expresión en lenguaje lógico de todo lo que una máquina de Turing puede y no puede hacer.

El énfasis es sobre la palabra *todo*. Aquéllo que apenas se puede considerar merecedor de mención debe ser expresado en el sistema de enunciados a construirse. Por ejemplo, una máquina de Turing del tipo usado en el Teorema de Cook puede hallarse en un estado a la vez, pero no en dos estados simultáneos.

Aquí, se utiliza notación de predicados para expresar el estado actual de la actividad de una máquina de Turing. El predicado $S(t, q)$ tiene valor verdadero (ó 1) si al tiempo $t$ la máquina de Turing está en el estado $q$. Si no, su valor es falso (ó 0). Debido a que se intenta capturar la operación de una máquina (no-determinística) de Turnig que requiere sólo $p(n)$ pasos para un polinomio $p$, el tiempo $t$ corre de 0 a $p(n)$. Los valores posibles para $q$, por otro lado, varían entre el conjunto $Q$ de estados disponibles para la máquina de Turing bajo consideración.

Para expresar la idea de que en cada tiempo la máquina está en sólo un estado, se escribe el siguiente enunciado para todos los valores posibles de las variables indicadas:

$$[S(t,q) \rightarrow\sim S(t,q')]$$

con las condiciones:

$$\begin{aligned} q, q' &\in Q \\ q &\neq q' \\ t &= 0, 1, \ldots, p(n) \end{aligned}$$

Este enunciado puede interpretarse como sigue: si al tiempo $t$ el estado es $q$, entonces en el tiempo $t$ el estado no puede ser ningún $q'$ que no sea igual a $q$. Se puede objetar que los $(p(n)+1) \times (\mid Q \mid (\mid Q \mid -1))$ enunciados resultantes no están en forma apropiada; los enunciados en una instancia de satisfactibilidad deben todos estar en forma disyuntiva. Tal situación se corrige fácilmente, sin embargo, mediante el uso de la conocida equivalencia de las siguientes dos expresiones:

$$A \rightarrow B \text{ y } \sim A \vee B$$

Así, el enunciado general puede escribirse como $[\sim S(t,q) \vee \sim S(t,q')]$.

La expresión lógica final que se construye debe también de alguna manera codificar el hecho de que cada celda en la cinta de la máquina de Turing puede contener sólo un símbolo en cada momento. Para ello, se utiliza el predicado $T(t,c,s)$. Este predicado es verdadero si en el tiempo $t$ la celda $c$ contiene el símbolo $s$, y falso de cualquier otro modo. Para expresar la unicidad del contenido de la celda en cualquier momento, parece requerirse un sistema un poco más grande de enunciados que para expresar el caso de unicidad de estados.

$$[T(t,c,s) \rightarrow\sim T(t,c,s')]$$

con las condiciones:

$$\begin{aligned} t &= 0, 1, \ldots, p(n) \\ c &= -p(n), -p(n)+1, \ldots, -1, 0, 1, \ldots, p(n) \\ s, s' &\in \Sigma \\ s &\neq s' \end{aligned}$$

Aquí, $\Sigma$ denota el alfabeto de la cinta de la máquina de Turing.

Cuando todos los posibles valores de $t$, $c$ y $s$ se substituyen en este enunciado genérico, resultan $p(n) \times (2p(n)+1) \times \mid \Sigma \mid (\mid \Sigma \mid -1)$ enunciados. De nuevo, se requiere tan solo convertir la implicación a una disyunción. En general, la transformación genérica se entiende más fácilmente si se usan los operadores lógicos más comunes. Los enunciados que aparecen aquí siempre pueden convertirse eficientemente (en tiempo polinomial) a su forma disyuntiva.

Es también un hecho obvio de la operación de la máquina de Turing que en cualquier momento la cabeza lectora/escritora opera sólo una celda. Para expresar

esta idea, se usa un predicado como $H(t, c)$. Este es verdadero en el tiempo $t$ si la máquina de Turing revisa la celda número $c$. De nuevo, el mismo estilo de enunciado puede utilizarse con precisamente el mismo efecto:

$$[H(t, c) \rightarrow\sim H(t, c')]$$

con las condiciones:

$$\begin{aligned} t &= 0, 1, \ldots, p(n) \\ c, c' &= -p(n), -p(n)+1, \ldots, -1, 0, 1, \ldots, p(n) \\ c &\neq c' \end{aligned}$$

Substituyendo los valores de $t$, $c$ y $c'$ indicados, es notorio que el número total de enunciados generados es $(p(n)+1) \times (2p(n)+1)$. En este sistema, como en el anterior, nótese que las celdas van de $-p(n)$ a $p(n)$. Esto simplemente significa que para el tiempo previsto para la máquina, ésta no puede revisar más allá a la izquierda que $-p(n)$ y más allá a la derecha de $p(n)$.

El siguiente enunciado garantiza que, en efecto, en ningún momento, más de una operación puede realizarse en la máquina Turing. Esta es la esencia de la máquina secuencial, y es también la esencia de la garantía general de que la máquina hace lo que se supone debe hacer. Al mismo tiempo, se debe asegurar que la máquina no hace cosas que no debe hacer, como por ejemplo, cambiar símbolos de la cinta que no están siendo revisados. Dos de los predicados anteriores sirven para esto, en los siguientes enunciados:

$$[(T(t, c, s) \vee \sim H(t, c)) \rightarrow T(t+1, c, s)]$$

con las condiciones:

$$\begin{aligned} t &= 0, 1, \ldots, p(n)-1 \\ c &= -p(n), -p(n)+1, \ldots, -1, 0, 1, \ldots, p(n) \\ s &\in \Sigma \end{aligned}$$

En el tiempo $t$ la celda $c$ puede contener el símbolo $s$, y sin embargo, la cabeza no está revisando la celda $c$. En tal caso, el símbolo permanece sin cambio en $c$ en el tiempo $t+1$. Usando la implicación y las leyes de De Morgan, es fácil convertir este enunciado en su forma disyuntiva.

Por convención, en el tiempo $t = 0$ la máquina de Turing revisa la celda 0. Esto se puede aseverar en un enunciado como:

$$[H(0, 0)]$$

Otras condiciones iniciales incluyen al estado inicial, por convención 0:

$$[S(0, 0)]$$

En el tiempo $t = 0$, sin embargo, la máquina de Turing tiene dos cosas importantes en su cinta. Primero, un número de celdas a la izquierda de la celda 0, conteniendo un conjunto de símbolos supuestos, colocados ahí, si se desea, por un dispositivo aleatorio. Esta idea se expresa por omisión: no hay condiciones que gobiernen los contenidos de las celdas a la izquierda de la celda 0. Se puede en realidad suponer que un número específico de estas celdas contiene símbolos supuestos y que todos los símbolos más allá de estos hasta la celda $-p(n)$ contienen ceros.

Segundo, las celdas 1 a $n$ se supone contienen una instancia del problema actual:

$$[T(0, i, s_i)]\, i = 1, 2, \ldots, n$$

Esto significa simplemente que en el tiempo 0 la $i$-ésima celda contiene el símbolo $s_i$, es decir, el $i$-ésimo símbolo en la cadena de la instancia que se alimenta a la transformación genérica.

Si la máquina de Turing tiene $m$ estados, no hay problema en suponer que su estado de detención es el $m$-ésimo. Tal y como se han especificado las condiciones iniciales de operación, es necesario especificar las condiciones cuando el tiempo provisto para el cómputo ha terminado.

$$[S(p(n), m)]$$

La máquina de Turing debe detenerse para el tiempo $p(n)$. Debe también haber contestado positivamente: "sí". Esto se simboliza mediante la aparición de un 1 en la celda 0 para ese tiempo (y "no" puede simbolizarse de la misma manera mediante un 0):

$$[T(p(n), 0, 1)]$$

El sistema final de enunciados hace referencias directas o indirectas al programa de la máquina de Turing. El programa consiste de una colección de quíntuplas de la forma:

$$(q, s \rightarrow q', s', d)$$

Esto significa que en cualquier momento en que la máquina de Turing se encuentre en estado $q$ y lee el símbolo $s$ en la cinta, entra en estado $q'$, escribe un símbolo $s'$ en lugar de la $s$ y se mueve una celda en dirección $d$. Esta dirección se indica simplemente mediante una R para la derecha y una L para la izquierda.

Tres conjuntos separados de enunciados especifican los tres posibles efectos de estar en un estado dado y leyendo un símbolo particular. Debe entrarse a unnuevo estado, escribirse un nuevo símbolo y tomarse una dirección de la cabeza.

$$[(S(t, q) \wedge T(t, c, s) \wedge H(t, c)) \rightarrow S(t + 1, q')]$$

$$[(S(t, q) \wedge T(t, c, s) \wedge H(t, c)) \rightarrow T(t + 1, c, s')]$$

$$[(S(t,q) \wedge T(t,c,s) \wedge H(t,c)) \rightarrow H(t+1,c')]$$

con las condiciones:

$$\begin{aligned} t &= 0,1,\ldots,p(n) \\ c &= -p(n),\ldots,p(n) \\ q &\in Q \\ s &\in \Sigma \end{aligned}$$

Para cada combinación de las cuatro variables $t$, $c$, $q$ y $s$ hay un valor único de $q'$, $s'$ y $c'$ especificado en el programa. Se entiende que en cada caso estas son las variables para generar el enunciado. Ciertamente, solo se requieren $q$ y $s$ para determinar estos valores. La transformación obtiene los valores meramente mediante buscarlos en una tabla que contiene el programa no-determinístico.

Esto finaliza la especificación de los enunciados generados por la transformación genérica en respuesta a un programa específico y una instancia del problema. Los enunciados pueden generarse en tiempo polinomial en cualquier modelo de cómputo que se adopte. Mediante utilizar una máquina de acceso aleatorio que pueda ejecutar un lenguaje de alto nivel, una serie de ciclos anidados con índices limitados de la forma arriba especificada es perfectamente capaz de producir los enunciados adecuados tan rápidamente como es posible.

La corrección de la transformación no se prueba aquí. Tal demostración depende grandemente de observar que los enunciados limitan la máquina virtual de Turing a comportarse como debe. Si este es el caso, los enunciados pueden satisfacerse sólo si la máquina se detiene antes del límite de tiempo polinomial con un valor 1 ocupando la celda 0. Esto, a la vez, es posible sólo si la instancia del problema original tiene como respuesta "sí".

El conjunto algo grande (pero aún de tamaño polinomial) de enunciados generados por la transformación de Cook no solo muestra que la satisfactibilidad es NP-completa, sino también demuestra el uso de ambos cálculos de predicados y proposicional como lenguajes de codificación. La gran dificultad en resolver el problema de satisfactibilidad en tiempo polinomial es seguramente debido a este poder de codificación. Quizá la satisfactibilidad codifica otros procesos que no podrían posiblemente resolverse en tiempo polinomial.

# Capítulo 7

# Problemas NP-Completos
## *El Arbol Insoluble*

Hasta ahora, se sabe de más de 2000 problemas que son NP-completos. Desde 1972, cuando Stephen Cook encuentra que el problema de satisfactibilidad (*satisfiability*) es NP-completo, varios autores han publicado una serie de resultados sobre la NP-completitud de otros problemas. La lista de problemas NP-completos crece contínuamente.

Los problemas de esta lista provienen de una variedad de fuentes: computación, ingeniería, investigación de operaciones, matemáticas, etc. Algunos de estos problemas surgen de aplicaciones prácticas como optimización de compiladores, análisis de estructuras de acero y planificaciones en tiendas. Otros surgen en un contexto puramente teórico tal como la teoría de ecuaciones, cálculo y lógica-matemática.

Un problema se considera como NP-completo mediante encontrar una transformación especial a partir de un problema que se sabe NP-completo. Un árbol de tal transformación conecta al problema de satisfactibilidad SAT con todos los problemas NP-completos (figura 7.1). Tales problemas incluyen el problema de cobertura de vértices (*vertex-cover problem* ó VC), el problema del agente viajero (*travel salesman problem* ó TSP) y muchos otros. La transformación que demuestra que un problema es NP-completo comparte dos importantes propiedades con la transformación genérica originalmente propuesta por Cook:

- Puede completarse en tiempo polinomial.
- Preserva las soluciones.

Supóngase, por ejemplo, que $A$ y $B$ son dos problemas en NP y que $f$ es una transformación de $A$ a $B$ con estas propiedades. Si $A$ es NP-completo, hay una transformación genérica $g$ para todo problema en NP al problema $A$. No está mal considerar esta transformación como aquélla especificada por Cook en su teorema.

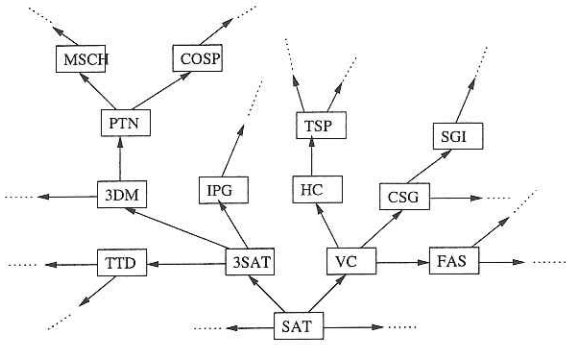


Figura 7.1: El árbol de NP-completitud.

Es decir, para cada problema $X$ en NP hay un polinomio $q$ tal que cada instancia $k$ de $X$ se transforma a $g(x)$ (una instancia de $A$) en tiempo $q(n)$, donde $n$ es el tamaño de $x$. Pero también se cuenta con una transformación $f$ que actúa sobre $g(x)$ para producir una instancia $f(g(x))$ de $B$ en tiempo $p(m)$, donde $m$ es el tamaño de de $g(k)$. Con algo más de argumentación, se establece que la transformación compuesta $f(g)$ toma una instancia $x$ a una instancia de $B$ en no más que $p(q(n)) + q(n)$ pasos. Es también verdadero que $x$ es una instancia positiva si y solo si $g(x)$ es una instancia positiva.

Estos son elementos esenciales para demostrar que el problema $B$ es NP-completo. Cuando se enfrenta un nuevo problema que se sospecha puede ser NP-completo, el primer paso es demostrar que se encuentra en NP: asegúrese que puede resolverse en tiempo polinomial por una máquina no-determinística de Turing (o computadora). El siguiente paso es encontrar un transformación específica (como $f$ anteriormente) de un problema que ya se sabe es NP-completo al nuevo problema a la mano.

Un ejemplo de una transformación de NP-completitud involucra al problema de satisfactibilidad y el problema de cubrir todas las aristas de un grafo por un número específico de vértices. Llamado el problema de cobertura de vértices (VC), la pregunta es si todas las aristas del grafo pueden cubrirse mediante un número dado $k$ de vértices; se requiere que cada arista sea incidente con al menos uno de los $k$ vértices. Una forma de establecer la pregunta es encontrar un sub-conjunto de vértices $V$ tal que:

- Cada arista en $G$ tiene al menos uno de sus vértices en $V$
- $V$ tiene $k$ (o menos) vértices.

Este problema cae en NP.

La transformación $f$ de SAT a VC puede ser descrita enteramente en términos de sus efectos en los enunciados de una instancia de SAT. Primero, sean las variables

que aparecen en una instancia específica de SAT listadas como:

$$x_1, x_2, \ldots, x_n$$

Reemplácese cada variable $x_i$ por una arista $(u_i, v_i)$ que conecta dos vértices $u_i$ y $v_i$ en un grafo que se va construyendo conforme se va avanzando. Cuando este proceso se complete, el grafo tiene $n$ aristas y $2n$ vértices.

El siguiente paso es reemplazar el $i$-ésimo enunciado:

$$(z_1, z_2, \ldots, z_m)$$

por un subgrafo completo de $m$ vértices $w_{i1}, w_{i2}, \ldots, w_{im}$. Cada $z_j$ es una literal, es decir, una variable booleana específica o su negación. Por simplicidad, supóngase que las variables son $x_1, x_2, \ldots, x_m$ $(m \leq n)$.

El siguiente paso para construir la instancia de VC que se obtiene de $f$ involucra especificar cómo los vértices de los subgrafos se unen a los vértices $u$ y $v$. Brevemente, si $z_j = x_i$, entonces se unen $w_{ij}$ con $u_l$; de otro modo, si $z_j = \bar{x}_i$, entonces se unen $w_{ij}$ con $v_l$.

El paso final es especificar el entero $k$: sea $k$ el número de variables más el número de ocurrencias de las literales menos el número de enunciados.

La construcción que lleva a cabo $f$ se ilustra en la figura 7.2 para la siguiente instancia de SAT:

$$(x_1 + x_2 + \bar{x_3})(\bar{x_1} + x_3)(x_1 + \bar{x_2} + x_3)(x_2)$$

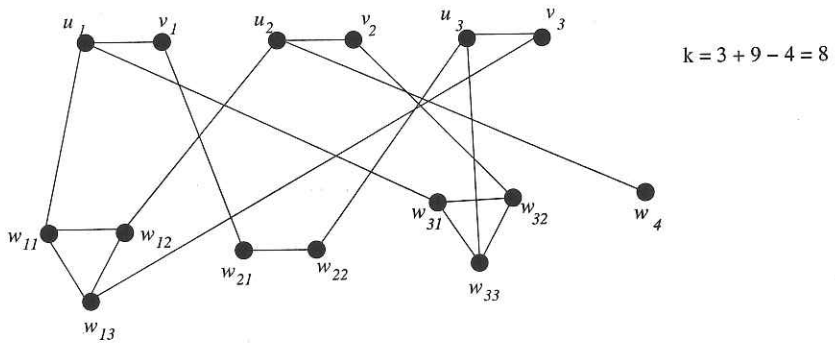


Figura 7.2: Transformación de SAT a VC.

¿Puede obtenerse $f$ en tiempo polinomial? Primero, $f$ no propiamente dibuja un grafo como el que se muestra en la figura 7.2. Meramente, necesita crear una lista de sus aristas. Las operaciones de reemplazo son directas en la mayoría de

los casos. La construcción de un subgrafo completo puede ser algo tardado, ya que su tamaño crece respecto al cuadrado del número de literales en la expresión. Sin embargo, todavía es de tamaño polinomial y puede ser manejado por un programa en tiempo polinomial, ya sea una máquina de Turing o una máquina de acceso aleatorio.

Para preservar la "positividad" de las instancias, el argumento requiere mayor explicación. Como regla general, de hecho hay dos argumentos. Primero, probar que si $S$ es una instancia positiva de satisfactibilidad (o el problema NP-completo con el que se inicia), entonces $f(S)$ es una instancia positiva de VC. Entonces, es necesario probar lo inverso: si $f(S)$ es una instancia positiva, también lo es $S$.

Supóngase que $S$ es una instancia positiva de la satisfactibilidad. Entonces, $S$ tiene una asignación de valores de verdad a sus variables que satisface cada expresión. Para cada variable $x_i$, que es verdadera por la asignación, sea $u_i$ cubierta en $V$. Si $x_i$ es falsa, entonces se coloca $v_i$ en $V$. Ya que cada expresión contiene al menos una literal verdadera, se sigue inmediatamente que cada uno de los subgrafos construidos por $f$ tienen al menos un vértice, por ejemplo $w$, adyacente a un vértice en $V$. No se coloca $w$ en $V$, sino que se añaden todos los otros vértices de tal subgrafo a $V$. No es difícil notar que la cobertura de $V$ termina con más de $k$ vértices como se especifica en términos de el número de variables, el número de literales y el número de expresiones.

El argumento inverso es ahora claro. Dado un grafo $V$ cubierto por el grafo $G$ que se ha construido por $f$, el entero $k$ acota el tamaño de $V$. Sin embargo, cada subgrafo $(u_i, v_i)$ requiere al menos un vértice en $V$ que no esté cubierto. Cada uno de los subgrafos completos tiene al menos un vértice que *no* está en $V$. El número mínimo de vértices requerido por esto es el número de variables más el número de literales menos el número de expresiones, que es precisamente $k$. Lo "compacto" de la contrucción asegura así que cada subgrafo mencionado tiene precisamente el mínimo número de vertices cubiertos permisible: cada subgrafo $(u_i, v_i)$ tiene uno, y cada subgrafo completo sobre $m$ vértices tiene $m - 1$ vértices en $V$.

Para cada $u_i$ en $V$ se asigna $x_i = 1$. Por cada $v_i$ en $V$ se asigna $x_i = 0$. Como se ha observado anteriormente, cada subgrafo contiene un vértice $w$ que no se encuentra en $V$. Pero tal vértice debe ser adyacente a un vértice, ya sea a $u_i$ ó a $v_i$. En cualquier caso, este último vértice debe pertenecer a $V$, y la literal correspondiente al vértice $w$ debe ser verdadera. Así, cada expresión contiene al menos una literal verdadera bajo la asignación. La instancia de SAT es satisfactible. La respuesta es "sí", y por lo tanto, se trata de una instancia positiva.

Esto termina el ejemplo de una transformación de NP-completitud. De las muchas demostraciones de NP-completitud en la literatura, un buen porcentaje no son más complicadas que la anterior. Otras, sin embargo, son mucho más complicadas. A veces, por sí mismo el problema se considera como un área completa por separado dentro de la computación.

La existencia de tantos problemas NP-completos meramente refuerza la idea de qué tan poca probabilidad de éxito tiene la búsqueda de algoritmos en tiempo polinomial para resolver problemas NP-completos. La transformación de Cook mapea todos los problemas en NP, y particularmente aquéllos que son NP-completos, a SAT. Pero SAT también puede transformarse a cualquier problema NP-completo. Esta observación lleva a darse cuenta que cualquier par de problemas NP-completos son mutuamente transformables entre sí en tiempo polinomial. Resolviendo uno, se han resuelto todos. Sin embargo, nadie ha encontrado un algoritmo en tiempo polinomial para ningún problema NP-completo.

Aun cuando no tiene una aplicación práctica inmediata resolver problemas de decisión como aquéllos en NP, es ciertamente importante resolver varios problemas cercanamente relacionados que tienen respuestas más complejas. Por ejemplo, dado un grafo, se desea saber el número de vértices en un conjunto que lo cubra. Se podría solicitar un algoritmo que produzca tal conjunto. Pero, si mediante preguntar si un grafo tiene un conjunto de cobertura de cierto tamaño, también se propone una pregunta que aparente no puede responderse en tiempo polinomial, entonces el problema original es al menos tan complicado. Tales problemas, aquéllos que requieren soluciones más elaboradas que responder sí o no, se les conoce con el nombre de NP-duros (*NP-hard*).

Enfrentados con la necesidad de resolución de problemas NP-duros en tiempo polinomial, se ha buscado algoritmos de aproximación que se ejecuten en tiempo polinomial.

# Capítulo 8

# La Tesis de Church
## *Todas las Computadoras son Creadas Iguales*

En 1936, Alonzo Church, un lógico estadounidense, formula una tesis que expresa precisamente qué significa computar. Church había laborado arduamente por algún tiempo con esta noción, llamándola "calculabilidad efectiva" (*effective calculability*). Tal noción describía cualquier proceso o procedimiento llevado a cabo en forma incremental mediante reglas bien definidas. Chruch creía que había podido capturar esta noción precisamente mediante un sistema formal llamado cálculo $\lambda$. Su tesis, en términos precisos, declaraba que cualquier cosa que pudiera llamarse efectivamente calculable podría ser representado dentro del cálculo $\lambda$.

Cuando el trabajo de Church aparece, ya había sido publicado otro sistema formal que declaraba representar algo similar; una clase de funciones llamadas "recursivas". Más aún, durante el primer año de aparición de la tesis de Church, aparece otra aproximación declarando lo mismo: la máquina de Turing. Un juicio *a priori* de los tres paradigmas de cómputo puede resultar en un diagrama de Venn, mostrando algunos de los traslapes entre las nociones, así como distinciones entre ellas (figura 8.1).

Resulta bastante difícil notar a primera vista que las tres nociones de cómputo son la misma. Sin embargo, ¿cómo es posible obtener una equivalencia entre las funciones computables por una máquina de Turing y las funciones recursivas, o entre éstas y el cálculo $\lambda$? En 1936, Church demuestra que las funciones recursivas son aquéllas precisamente propuestas dentro del cálculo $\lambda$. Turing, entonces, demuestra que la máquina que ha definido es también equivalente al cálculo $\lambda$ de Church.

Comenzando con algunas expresiones $\lambda$ elementales para números y operaciones sobre ellos, se puede utilizar este formalismo (de acuerdo con Church) para expresar

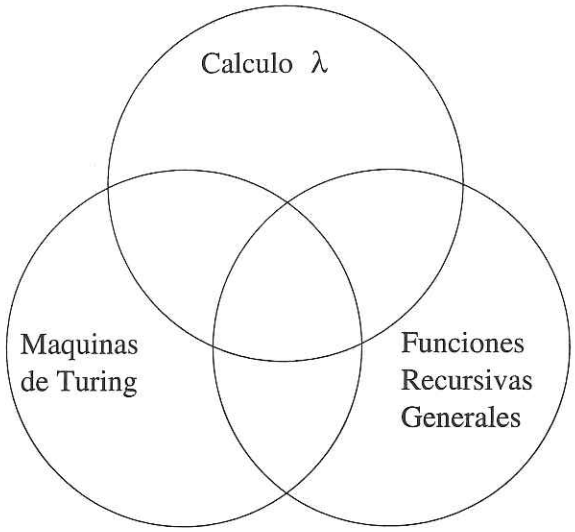


Figura 8.1: Tres nociones diferentes de cómputo.

cualquier función computable.

- El cálculo $\lambda$ es un procedimiento para definir funciones en términos de expresiones $\lambda$.
- Una expresión $\lambda$ es un identificador, una cadena de expresiones $\lambda$ o un número, o tiene la forma:

$$\lambda(\text{expresión de variable acotada}) \cdot \lambda\text{expresión}$$

- Una expresión de variable acotada es un identificador, los símbolos ( ) o una lista de identificadores.

El hecho de que las tres nociones de computabilidad resulten ser equivalentes favorece directamente a la tesis de Church. Para asegurarse, es posible analizar otros modelos de cómputo, como el autómata finito, que *no* son equivalentes a la máquina de Turing, pero solo por que son menos generales. La tesis de Church parecería declarar no solo que todas las nociones suficientemente generales de cómputo fueran equivalentes, sino que también hay un límite a su generalidad. Intentando hasta donde se pueda, no parece haber forma de definir un mecanismo de ningún tipo que pueda computar más de lo que una máquina de Turing es capaz de computar. No hay un esquema general que compute una función que no sea recursiva. Y no hay un procedimiento efectivo que no caiga en el entorno del cálculo $\lambda$.

De manera similar, la falla de cualquiera de estos tres esquemas generales (como por ejemplo, el problema de la detención de las máquinas de Turing) implica la falla de todos. La tesis de Church, por lo tanto, establece un límite que parece natural respecto a lo que las computadoras pueden hacer: todas las computadoras (suficientemente generales) son creadas iguales.

El apoyo a la tesis de Church proviene no sólo de la equivalencia de los tres sistemas mencionados, sino también de otras fuentes. Por ejemplo, un cuarto esquema computacional se presenta como la máquina de acceso aleatorio (*random access machine*, ó RAM). Una RAM puede computar cualquier función que una máquina de Turing pueda computar. También se puede comprobar lo contrario: las máquinas de Turing puede hacer lo mismo que las RAM. Las dos demostraciones, tomadas en conjunto, establecen la equivalencia entre los dos conceptos.

Para demostrar que una máquina de Turing puede simular una RAM, se requiere únicamente representar la memoria de la RAM sobre la cinta de una máquina de Turing y escribir un número de programas para la máquina de Turing, uno por cada instrucción posible de la RAM. De tal modo, un programa para la RAM puede traducirse intrucción por instrucción a un programa para la máquina de Turing, teniendo exactamente el mismo efecto.

Al llevar a cabo la traducción, se encuentra un problema debido al tamaño ilimitado de las palabras permitidas en la RAM. A primera vista, parece que la cinta completa de la máquina de Turing debe ocuparse en un solo registro de la memoria de la RAM. Sin embargo, en cualquier momento, la RAM que se simula requiere tan solo un tamaño de palabra finito para cualquier registro en uso actual. Si por lo tanto se dedican las suficientes celdas de la cinta de la máquina de Turing para guardar tales contenidos, entonces una operación de corrimiento se puede implementar siempre que las celdas representen un registro dado; la máquina de Turing revisa la cinta buscando una marca especial que indica el fin de las celdas de memoria que se estén utilizando. La máquina de Turing, entonces, recorre una celda a la derecha, haciendo lo mismo por cada celda de memoria toda la distancia hasta regresar a las celdas simulando el registro del problema.

Otra suposición para simplificar la labor involucra el uso de múltiples cintas para la máquina de Turing. Una máquina de Turing con múltiples cintas es equivalente a una máquina de Turing con una sola cinta. La figura 8.2 muestra cómo una máquina de Turing con tres cintas puede simular una RAM. La máquina de Turing puede tener un alfabeto infinito. Es por lo tanto posible (y conveniente) permitirle usar el alfabeto ASCII para expresar la operación de la RAM dada. La primera cinta que usa la máquina de Turing guarda los contenidos actuales del acumulador. Esta cinta también tiene espacio para copiar las instrucciones del propgrama, los nuevos contenidos computados del registro, y cosas por ese estilo. La segunda cinta se usa para almacenar el programa de la RAM, y la tercera cinta para almacenar los contenidos de la memoria de la RAM, siendo cada registro un número de celdas consecutivas.

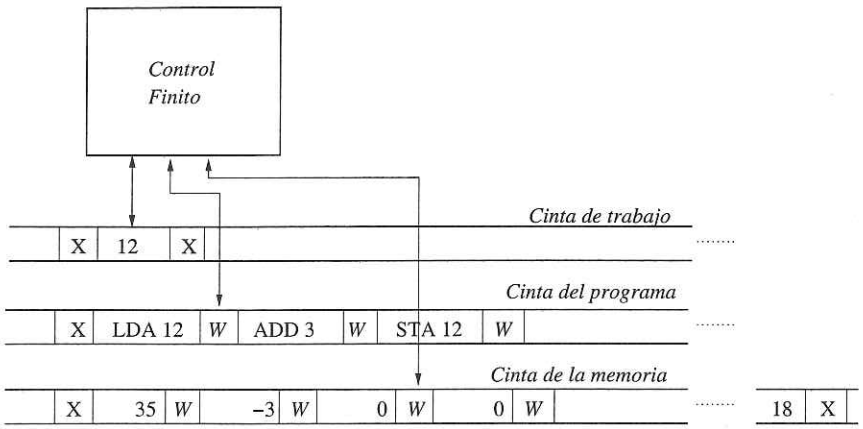


Figura 8.2: Una máquina de Turing que simula una RAM.

Considérese la instrucción:

LDA 12

Esto significa que el contenido del registro 12 de la memoria se debe transferir al registro 0 de la memoria (el acumulador).

Tal instrucción se reconoce por la máquina cuando la cabeza de la cinta 2 lee los símbolos LDA en secuencia. Entonces, entra a un sub-programa que realiza los siguientes pasos:

1. Copia el operando (12) de la cinta 2 al espacio de trabajo de la cinta 1.
2. Mueve la cabeza de la cinta 3 a lo largo de la misma hasta encontrar un marcador X.
3. Lee hacia la derecha con la cabeza de la cinta 3. Cada vez que se pasa un marcador *W*, se decrementa el número almacenado en el espacio de trabajo en una unidad.
4. Cuando el espacio de trabajo alcanza el valor 0, copia todas las celdas en la cinta 3 entre los marcadores *W* a las celdas que simulan el registro 0.

Una vez que estos pasos se han llevado a cabo, el control regresa al programa principal, el cual lee el siguiente código de instrucción para decidir qué sub-programa debe ser el siguiente.

Estas condiciones iniciales deben satisfacerse antes que el sub-programa de LDA (o de cualquier otra instrucción) pueda comenzarse: la cabeza de la cinta 1 debe

leer la celda justo a la derecha del marcador X más a la izquierda, y la cabeza de la cinta 2 debe leer la celda más a la izquierda del operando, buscando la instrucción a ser procesada. En cambio, no hay una condición particular para la cinta 3; puede encontrarse en cualquier posición.

Los diagramas de transición de estados dan la más simple indicación de cómo trabaja el sub-programa para LDA. Los cuatro diagramas que se presentan enseguida corresponden a los cuatro pasos que se listan anteriormente. No hay necesidad de etiquetar los ciclos, excepto por la palabra *inicio* que aparece en el estado que sirve como entrada al diagrama dado. Las flechas se etiquetan mediante tres conjuntos de símbolos que incluyen letras griegas. Los primeros tres símbolos sobre una flecha representan los tres símbolos que actualmente se leen por las tres cabezas de las cintas, en el orden 1,2,3. Un punto significa cualquier símbolo; una letra griega significa cualquier símbolo excepto uno que debe escribirse en otra cinta. Una letra del alfabeto significa que una transición que se dispara por la aparición en la cinta correspondiente de un caracter específico. En la segunda posición, los tres símbolos representan las mismas cosas a escribirse. En la tercera posición, se refieren los movimientos de las tres cabezas sobre las cintas: L significa izquierda, R significa derecha, y S significa ningún movimiento.

1. Copiar el operando a la cinta de trabajo (figura 8.3). Las cabezas de las cintas 1 y 2 de la máquina de Turing se mueven a la derecha, simultáneamente copiando el símbolo $\sigma$ que aparece en la cinta 2 ($\cdot\sigma\cdot$) a la cinta 1 ($\sigma\sigma\cdot$). Esto se hace para cualquier $\sigma$ que se encuentre, excepto para los marcadores *W*. Cuando este símbolo se encuentra en la cinta 2, el programa se mueve al siguiente diagrama.

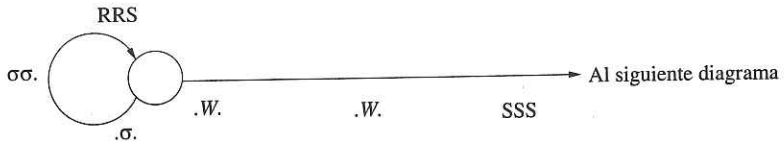


Figura 8.3: Copiar el operando a la cinta de trabajo.

2. Mover la cabeza de la cinta 3 al marcador X más a la izquierda (figura 8.4). Aquí, la máquina de Turing mueve la cabeza de la cinta 3 contínuamente a la izquierda, ignorando todos los símbolos excepto X cuando finalmente llega a él. Se entra en un nuevo estado. La cabeza de la cinta 3 se mueve contínuamente, ahora a la derecha, borrando cada símbolo que encuentra, reemplazándolo con el símbolo *B*. Esto permite a la máquina de Turing encontrar rápidamente la celda precisa del acumulador en la cual debe registrar un dígito del registro 12 de memoria. La misma idea se usa en la construcción de ciertos componentes de la máquina universal de Turing. Cuando la máqui-

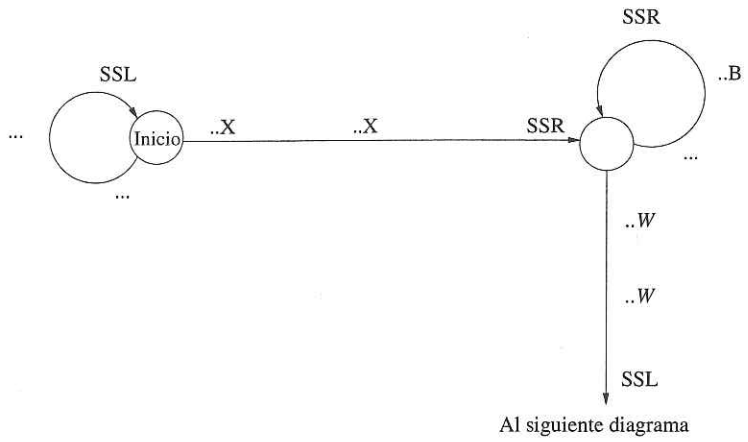


Figura 8.4: Mover la cabeza de la cinta 3.

na de Turing encuentra el marcador de límite de registro $W$, la máquina pasa al siguiente diagrama.

3. Leer hacia la derecha para "registrar" 12 (figura 8.5). En el estado inicial de este diagrama, la máquina de Turing lee hacia la derecha la cinta 3 hasta encontrar un marcador de límite de registro $W$. Si sucede que en la cinta 1 es un dígito no cero $\sigma$, se decrementa, y la máquina de Turing continua su lectura a la derecha sobre la cinta 3. Si sobre la cinta 1 se lee un 0, sin embargo, éste se reemplaza por un marcador temporal $A$, y la máquina de Turing comienza a leer ahora a la izquierda sobre la cinta 1, saltando todos los ceros hasta que encuentra un símbolo no cero. Si tal símbolo es una X, entonces el número originalmente escrito en la cinta 1 ha sido decrementado hasta 0, y la máquina sigue al siguiente diagrama. Si el símbolo es un dígito no-cero, éste se decrementa, y la cabeza sobre la cinta 1 se mueve ahora a la derecha, convirtiendo cada símbolo hasta e incluyendo a $A$ al símbolo 9. Ahora, la máquina re-entra al primer estado del diagrama, buscando el siguiente marcador $W$.

4. Copiar el registro 12 al registro 0 (figura 8.6). Cuando la máquina de Turing entra a este diagrama, la cabeza de la cinta 3 lee la celda más a la izquierda de la sección del registro que se especifica por el operando de LDA. En este caso, es la celda más a la izquierda del registro 12. Se lee uno de los dígitos 0 a 9 en tal caso. Y en realidad hay diez posibles salidas para el primer estado, pero en este diagrama solo se muestran dos: una para el 0 y otra para el 9. En cualquier caso, el símbolo se reemplaza por una $A$, y el siguiente estado al que se accede es único para el símbolo que se acaba de leer. De esta forma, la máquina de Turing "recuerda" el símbolo sin tener que escribirlo. En tal estado único, la cabeza de la cinta 3 se mueve contínuamente hacia

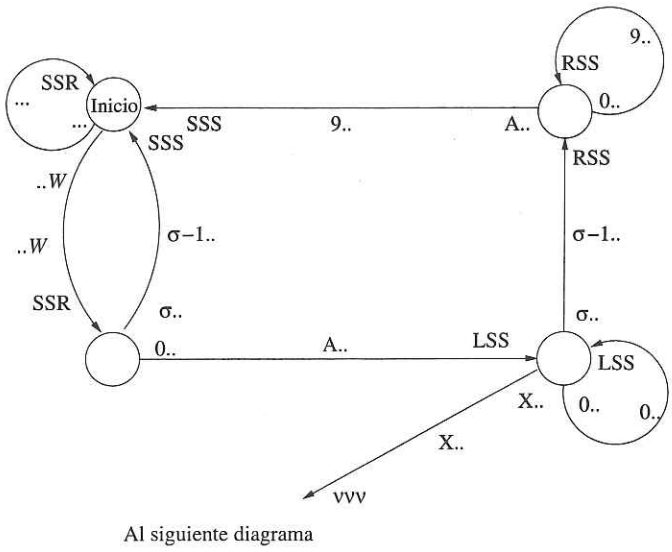


Figura 8.5: Leer hacia la derecha para "registrar" 12.

la izquierda hasta encontrar una *B*. Recuerde que el registro 0 se ha llenado de símbolos *B* en el segundo diagrama (figura 8.4). Estas se reemplazan por los dígitos que ocupan el registro 12, un dígito cada vez, en orden de derecha a izquierda. Así, cuando la máquina de Turing entra al tercer estado en el diagrama, reemplaza cada *B* por el dígito que "recuerda", y entonces se mueve hacia la derecha sobre la cinta 3 hasta que se encuentra de nuevo con los símbolos *A*. Continua leyendo hacia la derecha hasta que descubre ya sea un dígito o un marcador *W*. En el primer caso, procede a copiarlo al registro 0 como antes. En el segundo caso, sale del diagrama.

Al salir de este último diagrama, la instrucción LDA ha sido ejecutada: los contenidos del registro 12 se han transferido al registro 0. Un diagrama adicional, que no se muestra aquí, es suficiente para colocar las cabezas de las cintas 1 y 2 de nuevo en su posición inicial adecuada.

La operación de corrimiento no fue requerida en el sub-programa para LDA, pero se requiere para los sub-programas de instrucciones como ADD y SUB. En ciertos puntos críticos de estas operaciones, la máquina de Turing puede requerir una celda más en la secuencia del registro 0, de modo que la máquina de Turing suspende su operación actual e invoca el sub-programa de corrimiento. Cuando esto sucede, la máquina de Turing alterna entre dos estados. Estos estados son suficientes para "recordar" cada símbolo en la cinta 3 y entonces escribirlos un paso de tiempo después, siempre y cuando la cinta de trabajo 1 se utilice de manera adecuada.

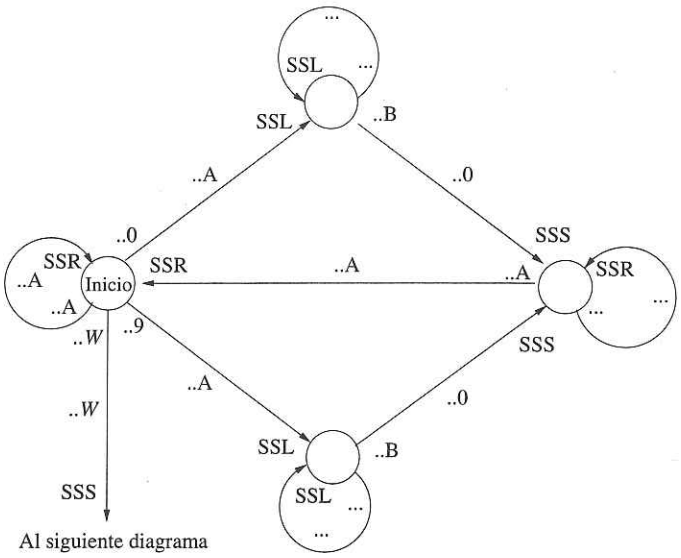


Figura 8.6: Copiar el registro 12 al registro 0.

Esto completa la descripción de un solo sub-programa de los doce requeridos por una máquina de Turing para simular una RAM. El programa total es bastante largo, pero esto no es sorpresivo, dado que las máquinas de Turing son conceptualmente más sencillas que las RAM. En cualquier caso, la construcción completa comprueba que una máquina de Turing es capaz de realizar cualquier cómputo que una RAM puede hacer. Esto es tan solo apenas un trozo más de evidencia que soporta a la tesis de Church.

# Bibliografía


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

[2] B. Bosworth. *Codes, Ciphers, and Computers: An Introdution to Information Security.* Hayden, Rochelle Park, NJ, 1980.

[3] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academia Press, 1973.

[4] D.E.R. Denning. *Cryptography and Data Security.* Addison-Wesley, 1983.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, 1979.

[6] D.H. Greene and D.E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhauser, 1982.

[7] R.W. Hamming. *Coding and Information Theory.* Prentice-Hall, 1980.

[8] F. Hennie. *Intorduction to Computability.* Addison-Wesley, 1977.

[9] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1960.

[10] D.E. Knuth. *The Art of Programming.* vol. 2, Addison-Wesley, 1967.

[11] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[12] E.C. Posner. *Combinatorial Structures in Planetary Reconnaissance.* Error Correcting Codes (H.B. Mann, ed.) Wiley, New York, 1969.

[13] W.H. Press, B.D. Flannery, S.A. Teukalsky, and W.T. VeHerling. *Numerical Recipies: The Art of Scientific Computing.* Cambridge University Press, 1986.

[14] D.F. Stanat and D.F. McAllister. *Discrete Mathematics in Computer Science.* Prentice-Hall, 1977.

[15] D. Wood. *Theory of Computation.* Harper and Row, 1987.