UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA MÉCANICA E INDUSTRIAL

Apuntes de Complejidad Computacional

Mayra Elizondo Cortés

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO FACULTAD DE INGENIERÍA

Apuntes de
Complejidad Computacional

Mayra Elizondo Cortés

APUNTES DE COMPLEJIDAD COMPUTACIONAL

ELIZONDO CORTÉS, Mayra
Universidad Nacional Autónoma de México
Facultad de Ingeniería
2023, 73 págs.

APUNTES DE COMPLEJIDAD COMPUTACIONAL

Primera edición impresa Julio 2014
Primera edición electrónica de un ejemplar (29 MB) en formato PDF
Publicado en línea: en Marzo de 2023

FACULTAD DE INGENIERÍA
http://www.ingenieria.unam.mx/

Hecho en México.

Unidad de Apoyo Editorial
Cuidado de la edición: Unidad de Apoyo Editorial
Diseño y formación editorial : Luis Enrique Vite Rangel

Prólogo

Uno de los conceptos más importantes e interesantes relacionados con el diseño y uso de algoritmos es la *complejidad computacional*. Con base en ella se evalúa la bondad de un algoritmo y el esfuerzo computacional que requerirá su aplicación, pero más aún, la complejidad computacional ayuda a diseñar mejores estrategias para solucionar problemas de naturaleza combinatoria. En este sentido, mejorar dichas estrategias para resolver problemas que por ese carácter combinatorio son muy difíciles de solucionar, es de enorme importancia dado que la mayoría de los problemas de la vida real presentan esas características; nos referimos a problemas logísticos, problemas financieros, problemas relacionados con producción, y en general, problemas relacionados con la mejor distribución de recursos.

Estos apuntes fueron concebidos para apoyar a los alumnos del posgrado de Ingeniería de Sistemas en sus maestrías de Investigación de Operaciones, Optimación Financiera e Ingeniería Industrial, en asignaturas específicas como Programación lineal, Programación no lineal, Programación entera y Métodos heurísticos, entre otras; sin embargo, también son útiles para alumnos de maestrías tales como la Maestría en Ciencias e Ingeniería de la Computación e inclusive para licenciaturas relacionadas con Ciencias e Ingeniería de la Computación, Matemáticas Aplicadas y Computación e Ingeniería Industrial. Adicionalmente, los presentes apuntes de Complejidad computacional pueden ser de utilidad para alumnos de doctorado.

El objetivo de este documento es exponer en forma concisa y lo más clara posible, conceptos como computabilidad y eficiencia de algoritmos, abordados en el capítulo 1; Máquinas de Turing deterministas y no deterministas, así como clases de complejidad computacional, tratadas en el capítulo 2; Problemas de decisión en el tercero; Clases P y NP vistas en el capítulo 4; Reducciones y transformaciones polinomiales, y NP-completez atendidos en el quinto y último capítulo. También, se incluyeron actividades sugeridas para evaluar, en cierto grado, el aprovechamiento del lector en la comprensión de los conceptos plasmados.

En general, se abarcan tópicos importantes e interesantes que comprenden el amplio tema de la *complejidad computacional*, así como una bibliografía para que los alumnos puedan tener mayor y más completa información de esta área, cuyo conocimiento y uso pueda apoyar al mejor diseño de estrategias de solución de problemas reales de toma de decisiones, problemas como los que agobian a nuestro país.

Sobre la autora

La doctora Mayra Elizondo Cortés estudió la Licenciatura en Matemáticas Aplicadas y Computación en la Facultad de Estudios Superiores Acatlán de la UNAM, y la Maestría y el Doctorado en Investigación de Operaciones en la Facultad de Ingeniería de la misma universidad. Se graduó del doctorado con mención honorífica. Es Profesora de Carrera Asociada "C" de Tiempo Completo en la División de Ingeniería Mecánica e Industrial de la Facultad de Ingeniería y obtuvo la definitividad en el área de Campo de Conocimientos: Sistemas; Campo Disciplinario: Investigación de Operaciones en el Departamento de Ingeniería de Sistemas. Cuenta con una trayectoria docente de 20 años.

Su principal línea de investigación es la optimización y la simulación aplicada esencialmente a procesos logísticos y de cadena de suministro. Con relación a dicha línea, ha presentado ponencias en seminarios y congresos nacionales e internacionales y dirigido varias tesis de maestría y doctorado.

Ha publicado apuntes para la materia de Simulación y tiene artículos de divulgación en las revistas arbitradas *Journal of Applied Research and Technology* y en la Revista de Ingeniería Investigación y Tecnología. Cuenta con un capítulo en un libro de Ingeniería de Sistemas. Es revisora de artículos en revistas tales como la Revista de Ingeniería Mecánica, Tecnología y Desarrollo y la Revista de Ingeniería Investigación y Tecnología. Actualmente se desempeña como Jefa de la Sección de Investigación de Operaciones e Ingeniería Industrial del Departamento de Ingeniería de Sistemas de la UNAM.

Índice

Prólogo 3
Sobre la autora 4

1. Problemas y algoritmos 6
1.1 Computabilidad 6
1.2 Problema 7
1.3 Algoritmo 8
1.4 Eficiencia de algoritmos 9
1.5 Análisis asintótico 10

2. Máquinas de Turing y complejidad computacional 22
2.1 Máquinas de Turing 22
2.2 Máquinas de Turing no deterministas 28
2.3 Clases de complejidad computacional 30
2.4 Problemas sin solución 32

3. Problema de decisión, problema de Satisfactibilidad y otros problemas fundamentales 35
3.1 Problema de decisión 35
3.2 Problemas de lógica booleana 36
3.3 Problemas de grafos 38

4. Clases P y NP 46
4.1 Algoritmos no-deterministas 46
4.2 Clase P 48
4.3 Clase NP 49

5. Reducciones polinomiales y NP-completez 54
5.1 Reducciones y transformaciones polinomiales 54
5.2 Problemas completos 62
5.3 NP-completez 63
5.4 Algunos problemas NP-completos 66

Apéndice 68
Bibliografía 72

1 Problemas y algoritmos

1.1 Computabilidad

La extendida aplicación del algoritmo simplex de programación lineal y sus variantes, a problemas que involucran miles de variables y restricciones, no podría ser posible sin la existencia de las computadoras actuales. Esto mismo ocurre con numerosas técnicas que se utilizan usualmente en la solución de problemas numéricos, problemas de simulación de sistemas físicos o sociales, problemas de manejo y administración de información, etc. Este tipo de problemas pueden resolverse manualmente, siempre y cuando los problemas sean de tamaño muy pequeño pero no tengan significancia práctica. Actualmente, es evidente que los límites del esfuerzo de cómputo humano se han superado por la necesidad actual de la ciencia y la tecnología. La pregunta que surge es si el potencial de las computadoras electrónicas tendrá algún límite.

Dos conceptos importantes y diferentes en computación son: problemas y algoritmos.

1.2 Problema

Un *problema* es un conjunto de *instancias* (casos particulares del problema) al cual le corresponde un conjunto de soluciones, además de una relación que asocia precisamente, a cada una de esas instancias con un subconjunto de soluciones, subconjunto que podría ser vacío.

Para el contexto en que nos encontramos existen dos tipos de problemas: los *problemas de decisión* en los que la respuesta es "sí" o "no"; y los *problemas de optimización* en los que se busca una respuesta a la pregunta "¿cómo se obtiene el mejor valor posible?".

Por ejemplo, el problema de decidir si una gráfica tiene o no un ciclo hamiltoniano queda completamente determinado por su conjunto de soluciones "sí" o "no". Por otro lado, el problema del agente viajero no solamente exige determinar si una gráfica tiene o no un ciclo hamiltoniano, sino que además pregunta cuál es el ciclo hamiltoniano más corto.

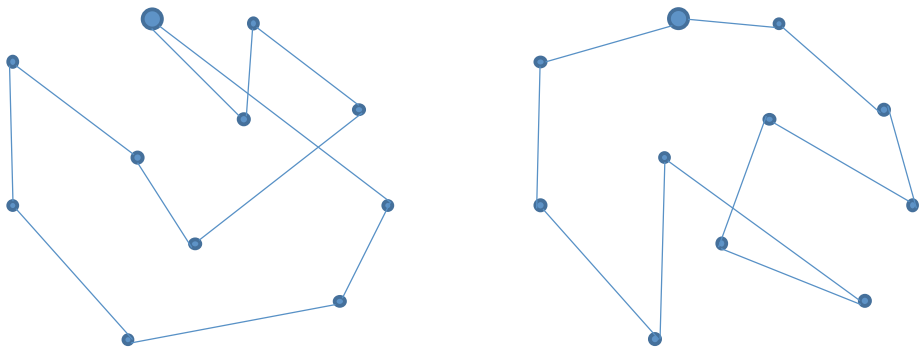


Figura 1. A partir de una gráfica completa, se observa la existencia de al menos dos ciclos hamiltonianos para un conjunto de nodos dado, ahora bien, ¿cuál es el más corto?

Para problemas de optimización, la instancia está compuesta por

i. un conjunto de configuraciones;
ii. un conjunto de restricciones; y además
iii. una *función objetivo* que asigna un valor real a cada instancia.

En el caso en que las configuraciones son discretas, el problema se considera *combinatorio.*

Estos problemas de optimización identifican, a partir de las configuraciones factibles (las que cumplen con todas las restricciones), la que tiene el mejor valor de la función objetivo, considerando que "mejor", puede significar máximo o mínimo. Esta configuración mejor, es la *solución óptima* de la instancia.

1.3 Algoritmo

Un *algoritmo* es un método de solución para resolver una instancia específica de un determinado problema. Por ejemplo, son algoritmos los métodos sistemáticos para resolver problemas como:

- encontrar una palabra en el diccionario;
- encontrar la ruta para llegar de la casa a la Universidad; o
- resolver una ecuación de segundo grado.

Al definir un algoritmo se tienen dos conjuntos de elementos primordiales:

- el conjunto ε de las entradas del algoritmo y que representan las configuraciones de la instancia; y
- un conjunto S de las *salidas*, que son los posibles resultados de la ejecución del algoritmo.

Los algoritmos son sucesiones de *instrucciones* que procesan la entrada $\rho \in \varepsilon$ para generar el resultado $\xi \in S$. Cada instrucción es una operación simple que genera un resultado intermedio único. La sucesión de instrucciones tiene que ser finita y sería deseable que para cualquier entrada, la ejecución de las instrucciones terminará después de un tiempo también finito.

En general, los algoritmos se implementan como programas de cómputo, expresados de manera general como pseudocódigos, para codificarlos luego en algún lenguaje de programación. Evidentemente, puede existir más de un algoritmo para resolver el mismo problema.

Un tipo especial son los algoritmos *deterministas*, en los cuales la salida está completamente determinada por la entrada. Otro tipo son los algoritmos *probabilistas o aleatorizados*, en los cuales no pasa lo anterior.

En un algoritmo *recursivo*, una parte de sí mismo se usa como subrutina. Si un algoritmo, en lugar de llamarse a sí mismo, repite una y otra vez el mismo código, se llama *iterativo.* Generalmente,

el pseudocódigo de un algoritmo recursivo es más corto que el de uno iterativo para el mismo problema. Un resultado importante es que cada algoritmo recursivo se puede convertir en un algoritmo iterativo (pero no viceversa), sin embargo, debe considerarse que tal conversión perjudica la eficiencia del algoritmo. Depende del problema, cuál manera es más eficiente, si recursiva o iterativa.

1.4 Eficiencia de algoritmos

Para la solución de un problema se pueden desarrollar diferentes algoritmos que variarán en cuanto a su *eficiencia*. Existen dos medidas importantes de la eficiencia de un algoritmo:

i. el tiempo total de cómputo, medido por el número de operaciones realizadas durante la ejecución de un algoritmo resolviendo la misma instancia, y
ii. la cantidad de memoria usada.

En el proceso de contar las operaciones que ejecuta un algoritmo, deben especificarse antes las operaciones calificadas como *operaciones básicas*. Comúnmente se consideran básicas a las siguientes:

i. operaciones aritméticas simples (+, -, x , /, mod);
ii. operaciones simples lógicas ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$);
iii. comparaciones simples ($<$, $>$, $=$, $\neq$, $\leq$, $\geq$);
iv. asignaciones de variables (:=).

Un elemento importante al hablar de la eficiencia de algoritmos es el *tamaño de la instancia*. Para un problema existen número infinito de instancias. Para definir el tamaño de una instancia, se debe fijar la unidad básica de medida. Comúnmente se usan la cantidad de bits, bytes, variables enteras, variables binarias, etc., que se requiere ocupar para representar todo el problema en la memoria de la computadora.

Por ejemplo, para un algoritmo que ordena números de una lista, el tamaño de la instancia es la cantidad de números en esa lista.

Sin embargo, al hablar de eficiencia de los algoritmos, los rápidos avances en la teoría y la tecnología de las computadoras hacen irrelevantes las medidas físicas de tiempo de corrida o requerimientos de memoria.

De tal forma, una medida más estandarizada es el número de operaciones básicas de cómputo que se requieren para resolver un problema. A esta medida se le llama *función de complejidad* y caracteriza la calidad de un algoritmo, cuyas soluciones incluyen el uso del *peor caso* y *caso promedio.*

La función del peor caso es tal que para un valor n, el valor de la función $f(n)$ representa el número de operaciones básicas para la más difícil de todas las instancias de tamaño n. En este caso, una dificultad es precisamente, identificar la instancia que representa el peor caso posible.

La función del caso promedio es tal que para un valor n, el valor de la función $f(n)$ representa el número promedio de operaciones básicas para todas las instancias de tamaño n. Esta función es muy útil en la práctica ya que el peor caso es muy raro, pero en el caso promedio, la posible dificultad es la estimación de la probabilidad con que ocurren los diferentes tipos de instancias. Además, el desempeño promedio no es seguro, de hecho, puede haber casos particulares que se comportan mucho peor que el promedio y nadie puede dar confianza en la suerte de intentar resolverlo.

1.5 Análisis asintótico

La *complejidad computacional* es un área muy activa de las ciencias de la computación, cuyos resultados tienen implicaciones trascendentes en el desarrollo y uso de algoritmos. Su meta es evaluar la calidad de un algoritmo en comparación con otros algoritmos o en comparación con la complejidad del problema o alguna cota de complejidad conocida.

El número de operaciones básicas obviamente depende de la cantidad de datos del problema. Clasificar 1 billón de números es completamente diferente de clasificar 10; de tal forma, expresamos el número de operaciones elementales (suponiendo el escenario del peor caso) como una función de algún o algunos números característicos suficientes, para calcular el volumen del trabajo que será realizado.

Por ejemplo, supongamos un algoritmo que resuelve un problema de tamaño n en un máximo de $7n^3 + 5n^2 + 27$ operaciones. Para tal función, estamos primeramente interesados en su tasa de crecimiento conforme n se incrementa. Queremos distinguir entre tasas de crecimiento "suaves"

y tasas de crecimiento "explosivas", por lo tanto, diferencias entre valores como $7n^3$ y n^3 no son realmente importantes, además, en la práctica no surgen grandes diferencias en las constantes. Es importante que también podemos descartar los términos de orden bajo, ya que es el grado mayor quien determina la tasa de crecimiento.

El resultado final es que la complejidad de este algoritmo puede ser suficientemente descrita por la función $f(n)=n^3$. Formalmente, decimos que este algoritmo es "de orden $O(n^3)$". Es también usual decir que este algoritmo "toma un tiempo $O(n^3)$".

El tiempo de corrida $T(n)$ es $O(f(n))$ si existen dos constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ para todo $n > n_0$ (véase figura 2).

Este simbolismo $O(f(n))$ es un recordatorio de que esta función representa el comportamiento en el peor caso para instancias suficientemente grandes. En realidad, no nos interesan los tiempos para instancias pequeñas ya que con ellas, normalmente todos los algoritmos producen resultados rápidamente.

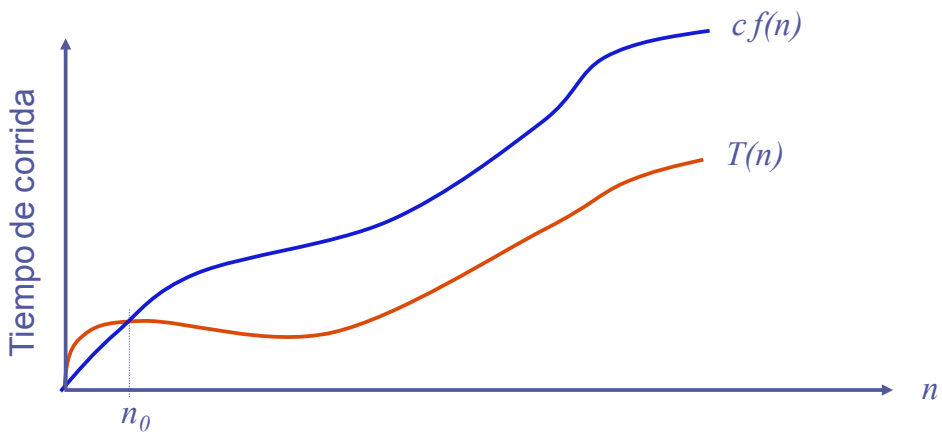


Figura 2. $T(n) = O(f(n))$

Entonces, de manera general, el orden de los algoritmos es una función $f(n)$ que acota asintóticamente, ya sea superior (O), inferior (Ω) o superior e inferiormente (Θ), a la función estimada de complejidad de un algoritmo para una entrada determinada, a partir de un valor n_0.

De manera formal,

$T(n) = O(f(n))$ $T(n) \leq c\, f(n)$ para alguna constante c y valores suficientemente grandes de n.

$T(n) = \Omega(f(n))$ $T(n) \geq c\, f(n)$ para alguna constante c y valores suficientemente grandes de n.

$T(n) = \Theta(f(n))$ $c'|f(n)| \leq |T(n)| \leq c|f(n)|$ para algunas constantes c, c' y valores suficiente suficientemente grandes de n.

Lo anterior puede verse gráficamente en las siguientes figuras:

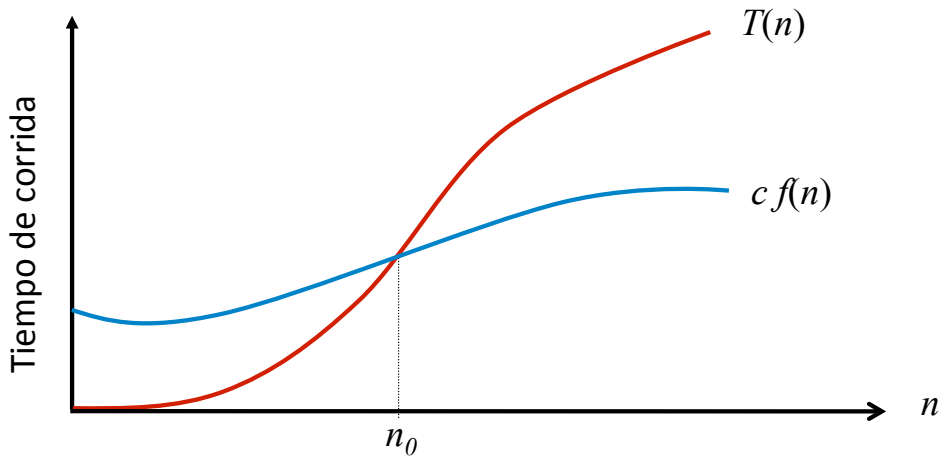


Figura 3. $T(n) = \Omega(f(n))$

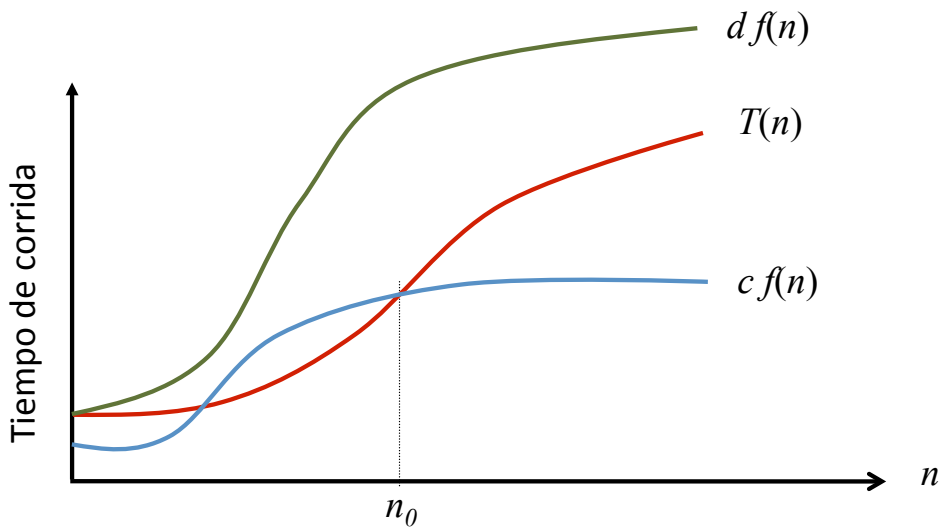


Figura 4. $T(n) = \Theta(f(n))$

Lo más usual es utilizar la notación O para referirse al orden de un algoritmo.

Algunas propiedades de las funciones O son:

- Transitividad

  $T(n) = O(f(n))$ y $f(n)=O(g(n))$ implica $T(n)=O(g(n))$;

- Reflexividad

  $f(n)=O(f(n))$;

- Si $T_1(n)=O(f(n))$ y $T_2(n)=O(g(n))$ entonces $T_1(n)+T_2(n)=max\{O(f(n)), O(g(n))\}$

  En palabras *"la tarea más difícil eventualmente siempre domina"*

- Si $T_1(n)=O(f(n))$ y $T_2(n)=O(g(n))$ entonces $T_1(n)\ x\ T_2(n)=O(f(n)\ x\ g(n))$.

Por otro lado, consideremos que para n suficientemente grande:

$$\log n < n < n^2 < n^3 < 2^n .$$

Esto a veces se establece como:

$$O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) .$$

Aquellos algoritmos que tienen una complejidad de tiempo polinomial o subpolinomial, esto es, aquellos que toman un tiempo $O(f(n))$ donde $f(n)$ es un polinomio o una función acotada por un polinomio, son *prácticos*.

Entonces, los algoritmos que tienen tiempos de corrida de orden $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, etc., son llamados *algoritmos de tiempo polinomial*.

Por otro lado, algoritmos de complejidades que no se pueden acotar por funciones polinomiales son llamados *algoritmos de tiempo exponencial*. Estos incluyen órdenes de crecimiento explosivo los cuales contienen también factores no exponenciales como $n!$. Para ilustrar el efecto del tiempo de ejecución, observe la siguiente figura:

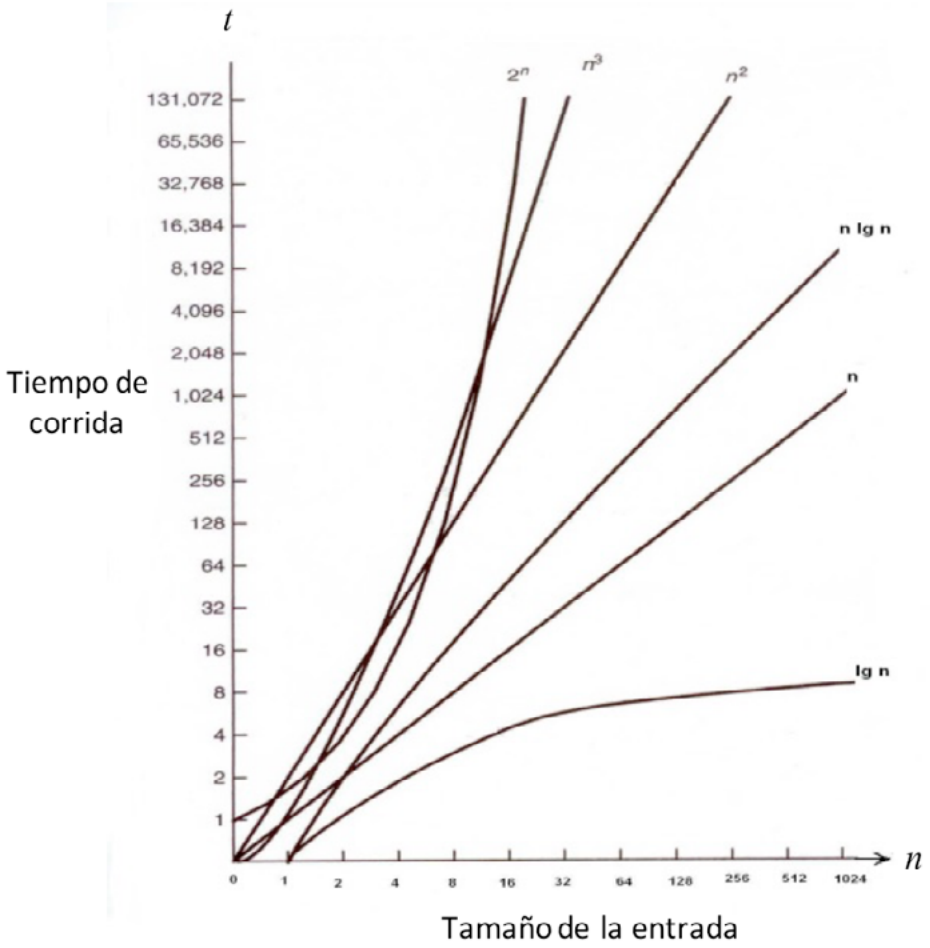


Figura 5. Comportamiento de funciones de complejidad

Hay varios argumentos que soportan la tesis de que "polinomial es un sinónimo para práctico": primero, los algoritmos de tiempo polinomial están en una situación más ventajosa para explotar mejoras tecnológicas de velocidad para computadoras. Esto puede mostrarse mejor en la siguiente tabla:

Tabla 1. Aprovechamiento de mejoras tecnológicas de velocidad para computadoras

Función de complejidad de tiempo	Tamaño de la mayor instancia del problema solucionable en 1 hora		
	Con una computadora actual	Con una computadora 100 veces más rápida	Con una computadora 1000 veces más rápida
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
2^n	N_4	$N_4 + 6.64$	$N_4 + 9.97$
3^n	N_5	$N_5 + 4.19$	$N_5 + 6.29$

Para comprenderla mejor, considere que,

- en el caso de $O(n^2)$: una computadora actual corre para
  an^2 segundos $\Rightarrow$ 1hr = aN_2^2;
  una computadora 100 veces más rápida corre para
  an^2 / 100 segundos $\Rightarrow$ 1hr = aN_X^2; / 100
  entonces, $aN_2^2 = aN_X^2 / 100$ por lo tanto $100N_2^2 = N_X^2$, $N_X = 10N_2$; y

- en el caso de $O(2^n)$: una computadora actual corre para
  $a2^n$ segundos $\Rightarrow$ 1hr = $a2^{N_4}$;
  una computadora 100 veces más rápida corre para
  $a2^n / 100$ segundos $\Rightarrow$ 1hr = $a2^{N_X} / 100$;
  entonces, $a2^{N_4} = a2^{N_X} / 100$ por lo tanto $100(2^{N_4}) = 2^{N_X}$,
  $N_X = N_4 + (\log 100 / \log 2) = N_4 + 6.64$.

Como puede verse, la mejora en tecnología es multiplicativa en algoritmos de tiempo polinomial y únicamente aditiva en algoritmos de tiempo exponencial. Adicionalmente, la situación es mucho peor que la mostrada en la tabla 1 si las complejidades involucran factoriales.

Un segundo argumento es que, en algoritmos de tiempo polinomial, una vez que son descubiertos, pasan por una serie de mejoras, esto es, reducciones en las constantes de sus funciones de

complejidad y en sus propios grados. Este es un resultado empírico y, en parte, explica por qué complejidades como $O(n80)$ u $O(10^{100}n)$ no aparecen en la práctica. Los algoritmos de tiempo polinomial típicamente tienen un grado de entre 2 o 3, y no incluyen coeficientes muy grandes. El inconveniente real para la solución de un problema, está en la definición del primer algoritmo de tiempo polinomial. Para problemas combinatorios, el "salto" a la clase polinomial requiere comúnmente un análisis profundo de la naturaleza del problema, si es que el salto puede hacerse.

Por supuesto, hay excepciones a estas reglas. El caso más famoso y muy interesante, es el excelente desempeño del método Simplex para resolver problemas de Programación lineal. Con el algoritmo Simplex, el número de pasos que se requieren para resolver un problema en una instancia de tamaño $m \times n$, max $c'x$, sujeto a: $Ax \leq b$, $x \geq 0$ puede variar considerablemente con respecto de los parámetros A, b y c, incluso si sus dimensiones se mantienen constantes. En un caso extremo, si $c \leq 0$ la solución factible inicial es óptima y no se requiere realizar ningún pivoteo, mientras que, para diferentes parámetros seleccionados, podrá requerirse de un número significativo de iteraciones para llegar a la solución óptima.

El algoritmo Simplex siempre ha trabajado muy bien en la práctica mostrando una complejidad empírica de $O(m^2n)$, sin embargo, probablemente esta complejidad es exponencial, de hecho, hay casos artificiales "patológicos" donde el Simplex muestra su comportamiento exponencial oculto. Podría suceder que el método captura alguna propiedad significativa del problema que aún debe ser descubierta.

Considerando lo anterior, hasta hace poco, una de las preguntas más desconcertantes era si se podía tener un algoritmo de tiempo polinomial para la programación lineal y se tuvieron pruebas contradictorias sobre la respuesta a esta pregunta. Por un lado, se conocía sin duda que la programación lineal era uno de los problemas sobre el cual no se podía avanzar en el desarrollo de un algoritmo de tiempo polinomial. Por otro lado, la programación lineal tiene dos ventajas que lo hace completamente diferente de otros problemas clásicos de optimización. Primero, la programación lineal posee una fuerte teoría de dualidad, de la cual carecen el resto de los problemas duros de optimación combinatoria. Y en segundo lugar, si bien el método Simplex se vuelve exponencial para el peor de los casos, trabaja empíricamente en instancias de tamaño aparentemente ilimitado.

Pero en este punto es importante considerar que en el estudio de la complejidad de un algoritmo, realmente sólo estamos interesados en el comportamiento del algoritmo cuando se trabaja con entradas muy grandes, porque estas entradas son las que van a determinar los límites de la aplicación del algoritmo. La conclusión general es que un problema puede considerarse "eficientemente resuelto" cuando se ha encontrado un algoritmo de tiempo polinomial para su solución.

Algunos ejemplos de problemas de optimización que son considerados "bien resueltos" en este sentido son:

- Programación lineal (existen algoritmos polinomiales);
- Algunos problemas de Ruta más corta;
- Algunos problemas de Ruta más larga (por ejemplo los cálculos de PERT/CPM simple);
- El problema del Árbol de mínima expansión;
- Algunos problemas de flujo en redes;
- Problemas de pareamiento;
- Problemas de Transporte, Asignación y Transbordo;
- Varios casos especiales de problemas duros (difíciles).

Tomemos en cuenta que esta lista, a pesar de todos los problemas que incluye, es muy corta comparada con la de problemas para los cuales, los algoritmos de tiempo polinomial no son conocidos, por ejemplo:

- Problema Mochila 0-1;
- Problema de Cobertura;
- Problema del Agente viajero;
- Problema de Localización sin restricciones de capacidad;
- Problema de Programación entera;
- Problema del Árbol de Steiner.

A lo largo de estos apuntes, veremos algunos de estos problemas más detenidamente.

Ejemplo 1.1

El problema del agente viajero (*Traveling Salesman Problem*, TSP) consiste en que "un viajero debe visitar cada ciudad exactamente una vez y debe regresar al punto de partida". Se conoce el costo de viajar de cada ciudad a todas las demás. De tal forma, el viajero debe planear su itinerario de viaje, logrando que el costo total sea mínimo. En términos generales, el problema es encontrar el costo mínimo del circuito (ciclo Hamiltoniano) de longitud n.

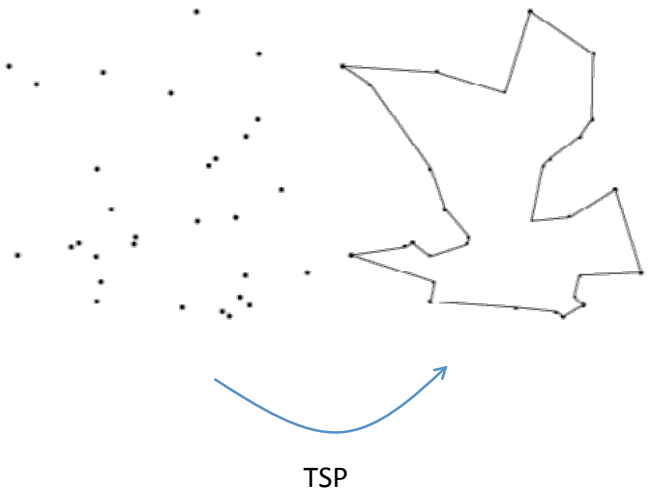


Figura 6. Problema del Agente viajero

De manera formal el problema se define como:

Instancia : Sea un conjunto C de m ciudades, la distancia $d(c_i, c_j) \in Z^+$ para cada par de ciudades $c_i, c_j \in C$, y un número entero positivo B.

Pregunta : ¿Hay una ruta de C que tenga una longitud B o menor, por ejemplo una permutación $< c_{\Pi(1)}, c_{\Pi(2)}, \ldots, c_{\Pi(K)}, c_{\Pi(K+1)}, \ldots, c_{\Pi(m)} >$ de C tal que

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B?$$

Considerando este problema como uno de teoría de grafos, se puede considerar que es un grafo a cuyas aristas entre los vértices, que representan las ciudades que debe visitar el agente viajero, se ha asociado un peso, que representa la distancia, costo, tiempo o alguna otra cantidad, que se pretenda minimizar. Así, el objetivo es encontrar la ruta mínima que pase por cada ciudad exactamente una vez y regrese a la ciudad inicial. Nuevamente en términos de teoría de grafos, la meta es encontrar un ciclo hamiltoniano que minimice la suma de los pesos de las aristas.

El TSP es un problema de decisión, en el sentido de que una instancia del TSP puede resolverse encontrando el mejor entre un conjunto de recorridos. Por lo tanto, una computadora puede resolver cualquier instancia del TSP al examinar sistemáticamente y evaluar todos los recorridos, para posteriormente seleccionar el recorrido más corto. El número de recorridos de n ciudades es $(n\text{-}1)!/2$. Así que la implementación del algoritmo mencionado, en una computadora requerirá alrededor de $n!$ pasos (operaciones elementales). La solución por medio de este algoritmo para una instancia del TSP de tamaño moderado, por ejemplo, encontrar el mejor recorrido de las capitales de los estados de Estados Unidos, requeriría muchos billones de años para resolverlo, incluso bajo las suposiciones más optimistas sobre la velocidad de las computadoras del futuro (50! posee alrededor de 65 dígitos decimales).

Actividades sugeridas

1. De un ejemplo de un problema de optimización, enuncie dos instancias de él y cinco configuraciones de cada instancia.

2. Dé un ejemplo de un algoritmo recursivo. Explique por qué es recursivo.

3. Dé un ejemplo de un algoritmo iterativo. Explique por qué es iterativo.

4. Considere el ejemplo recursivo que construyó en la actividad 1 y conviértalo en un algoritmo iterativo, ¿qué algoritmo es más eficiente? ¿por qué?

5. Describa el algoritmo de Búsqueda binaria y aplíquelo en algún ejemplo. ¿De qué orden es el algoritmo?

6. Indique mediante la notación de O, la complejidad del algoritmo para encontrar una palabra en un diccionario.

7. Seleccione la mejor notación de O como una función de N para el tiempo de ejecución de cada programa parcial. El tiempo de ejecución está definido como el número de veces en que se ejecuta la instrucción:

$$X = X + 1$$

($\lfloor x \rfloor$ denota "el entero más cercano $\leq x$" .)

a)

```
FOR I = 1 TO N
        FOR J = 1 TO N
                X = X + 1
        NEXT J
NEXT I
```

b)

```
FOR I = 1 TO N
        FOR J = 1 TO I
                X = X + 1
        NEXT J
NEXT I
```

c)

```
      J = N
10    X = X + 1
      J = ⌊J / 2⌋
   IF J = 1 THEN END
   GOTO 10
```

8. Suponga $T(N) = 30n^3 + 20nlogn + 10$, entonces $T(N) = O(n^3)$
   ¿Qué constante c y n_0 se pueden utilizar para que el resultado anterior sea cierto?

9. Para cada una de las siguientes expresiones, seleccione la mejor notación de *O*.
   a) $7n - 1$
   b) $8n^3 - 12n^2 - 1$
   c) $2\log n - 4n + 6n\log n$
   d) $2 + 4 + 6 + \ldots + 2n$
   e) $\dfrac{(n^2 + \log r - (n \square 1)}{n + n^2}$

10. El costo de cambiar una máquina de manufactura de un trabajo a otro es:

		Al trabajo A	B	C	D
Del trabajo	A	--	310	260	502
	B	775	--	131	441
	C	982	360	--	113
	D	716	689	931	--

Se debe encontrar la secuencia de cambio de un trabajo a otro, de manera que en total se tenga el costo mínimo. Cada trabajo se realiza una sola vez y el trabajo final deberá coincidir con el trabajo inicial.

Observe que este problema se puede plantear como un problema de agente viajero. Construya una red asociada al problema y resuélvalo mediante un algoritmo de fuerza bruta (enumeración exhaustiva).

2 Máquinas de Turing y complejidad computacional

2.1 Máquinas de Turing

Consideremos que en la solución de problemas con ayuda de las computadoras, éstas sólo pueden seguir algoritmos. Métodos como los que se enseñan en la escuela primaria para resolver operaciones aritméticas con números enteros son algoritmos típicos; se pueden aplicar a cualquier entero y son correctos considerando que al terminar el algoritmo se obtendrá la respuesta correcta. El matemático británico Alan M. Turing, inventó en 1936, un objeto matemático, conceptual, que representa a los algoritmos y al que se conoce como Máquina de Turing.

El modelo de la máquina de Turing se originó a partir de una cuestión planteada por David Hilbert en 1928, acerca de si las matemáticas eran *decidibles*, es decir, si existe un método definido que se pueda aplicar a cualquier sentencia matemática y que nos diga si es cierta o falsa. Turing ideó un modelo formal y abstracto de una computadora, que puede simular cualquier algoritmo utilizando una sola estructura de datos, la cual consiste en una sucesión de símbolos escrita en una cinta (infinita) dividida en cuadros, y un puntero que se mueve hacia adelante o hacia atrás de la cinta borrando o imprimiendo símbolos y se detiene al final de un cálculo.

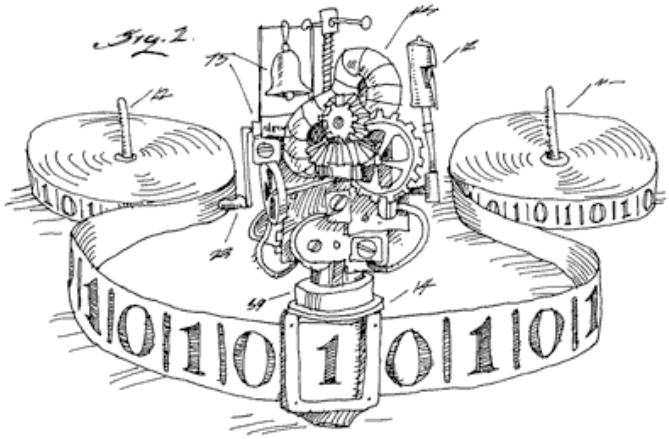


Figura 7. Máquina de Turing por Tom Dunne, Científico Americano (2002)
(Fuente: www.ecs.syr.edu/.../coretechnologies.htm)

Una consecuencia importante del desarrollo de este modelo fue la respuesta a la pregunta de Hilbert, ya que Turing demostró que existen problemas que una máquina no puede resolver.

2.1.1. Definición formal

Formalmente, se define una máquina de Turing determinista (MTD) $M = (Q, \Sigma, \Gamma, b, \delta, q_0, q_{sí}, q_{no})$ como:

i. un conjunto finito de estados Q;
ii. un conjunto finito de símbolos Σ que no contiene el espacio en blanco, que se llama alfabeto de entrada o de máquina;
iii. un conjunto finito de símbolos de cinta, llamada alfabeto de cinta G que contiene al espacio en blanco ($\Sigma \subseteq \Gamma$);
iv. b Î G es un símbolo denominado blanco, y es el único símbolo que se puede repetir un número de veces infinito;
v. una función de transición δ: $Q \times \Sigma \rightarrow Q \times \Gamma \times \{I, D\}$ donde I es un movimiento a la izquierda y D es un movimiento a la derecha;
vi. un estado inicial $q_0 \in Q$;
vii. un estado aceptador $q_{sí} \in Q$;
viii. un estado no aceptador $q_{no} \in Q$, $q_{sí} \neq q_{no}$.

En la literatura existe un gran número de definiciones alternativas, todas ellas con el mismo poder computacional.

2.1.2. Funcionamiento

La función δ define el "programa" de la máquina. De manera general, opera de la siguiente forma:

(estado, valor)→(nuevo estado, nuevo valor, dirección)

lo cual puede leerse como: "si estamos en el estado *e* leyendo la posición *n* , en la que está escrito el símbolo *s*, entonces este símbolo es reemplazado por uno nuevo y se continua leyendo la celda siguiente, ya sea a la izquierda o a la derecha".

Se dice que la máquina se *detiene* si se llega a uno de los tres estados de alto que son {"alto", $q_{sí}$, q_{no}}. Si la máquina se detuvo en $q_{sí}$, la máquina *acepta* el símbolo de entrada. Si la máquina se detuvo en q_{no}, la máquina *rechaza* el símbolo de entrada.

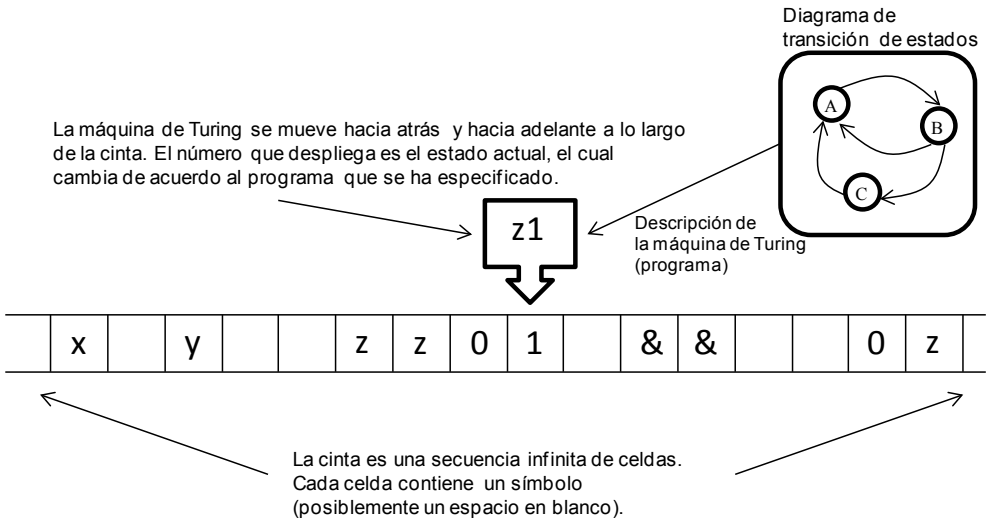


Figura 8. Modelo de la Máquina de Turing

De tal forma, la máquina *M* con la entrada *x* da una *salida* $M(x)$ que puede ser:

i. $M(x) = q_{sí}$ si la máquina acepta *x*;
ii. $M(x) = q_{no}$ si la máquina rechaza *x*;
iii. $M(x) = y$ si la máquina llega a "alto";
iv. $M(x) = /$ si *M* nunca se detiene con la entrada *x*.

Como puede percibirse, las máquinas de Turing son representaciones bastante naturales para resolver problemas que tienen que ver con sucesiones, por ejemplo, reconocer lenguajes: una máquina de Turing M reconoce el lenguaje dado L si y sólo si, para toda sucesión $x \in L$, $M(x)$= "sí" y si $x \notin L$, $M(x)$ "no". Así, la máquina de Turing puede considerarse como un autómata[1] capaz de reconocer lenguajes formales.

Las máquinas de Turing que paran con cualquier entrada se llaman *decidores.* Cuando un decidor reconoce algún lenguaje, también podemos decir que lo decide. La clase de lenguajes decididos por alguna máquina de Turing son lenguajes *recursivos*, esto es, se caracterizan porque para cada uno de ellos existe una máquina de Turing que aceptará cualquier palabra (sucesión x) que pertenezca al lenguaje y parará siempre.

Los lenguajes aceptados por alguna máquina de Turing son *recursivamente numerables* (de acuerdo con la jerarquía de Chomsky[2]), si existe una máquina de Turing que acepta y se detiene con cualquier palabra del lenguaje, pero que puede iterar indefinidamente, con una palabra que no pertenece al lenguaje, en otras palabras, una máquina de Turing M *acepta* un lenguaje recursivamente numerable dado L si y sólo si, para toda sucesión $x \in L$, $M(x)= q_{sí}$ pero si $x \notin L$, $M(x)$=/.

Al resolver un problema de decisión mediante una máquina de Turing, lo que se hace es decidir un lenguaje compuesto de representaciones de las instancias del problema que corresponden a la respuesta "sí". Los problemas de optimización se resuelven por máquinas de Turing logrando hacer el cómputo de una función apropiada que va de sucesiones a sucesiones, las cuales representan tanto la entrada como la salida (una sucesión de entrada da alguna o algunas sucesiones de salida), en un formato con un alfabeto adecuado.

Ejemplo 2.1

En este ejemplo se describe una máquina de Turing $M1$ que reconoce el lenguaje que consiste de todas las cadenas formadas por ceros, cuya longitud es una potencia de 2. Entonces, se dice que esta máquina de Turing decide el lenguaje $L = \{0_2^n \mid n \geq 0\}$.

1 Máquinas abstractas que resuelven problemas reconociendo lenguajes formales son estudiadas por la teoría de autómatas que es una rama importante de las ciencias de la computación.

2 Es una clasificación jerárquica en lingüística, de distintos tipos de gramáticas formales que a su vez generan lenguajes formales.

$M1$ = "Sobre la cadena entrada w:

1. Recorrer la cinta de izquierda a derecha, marcando un cero sí y otro no (intermitentemente).
2. Si en el paso 1 la cinta contiene un solo cero, aceptar.
3. Si en el paso 1 la cinta contiene más de un cero y la cantidad de ceros es impar, rechazar.
4. Regresar la cabeza a la cinta hasta la posición más a la izquierda.
5. Ir al paso 1."

Ahora, daremos la definición formal de la máquina de Turing:

$M1 = (Q, \Sigma, \Gamma, b, \delta, q_0, q_{si}, q_{no})$:
$Q = \{q_0, q_2, q_3, q_4, q_5, q_{si}, q_{no}\}$
$\Sigma = \{0\}$
$\Gamma = \{0, x, b\}$
δ se describe en el diagrama de estados de la figura 9 .

Los estados inicial, de aceptación y de rechazo son $\{q_0, q_{si}, q_{no}\}$, respectivamente.

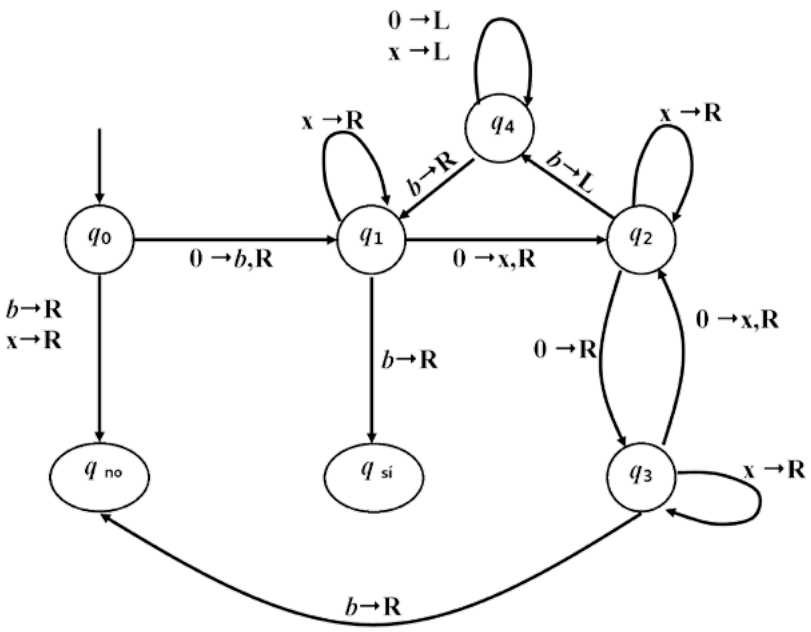


Figura 9. Diagrama de estados para la máquina de Turing $M1$

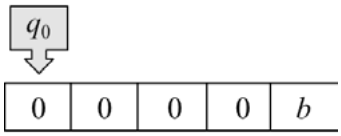
El funcionamiento es el siguiente:

En el diagrama de estados, la etiqueta 0 → b, R aparece en la transición de q_0 a q_1, esto significa que cuando $M2$ se encuentra en el estado q_0 con la cabeza en la cinta leyendo un 0, la máquina va al estado q_1, escribe en la cinta b y mueve la cabeza de la cinta a la derecha (R). Esto se expresa como $\delta(q_0, 0) = (q_1, b, R)$.

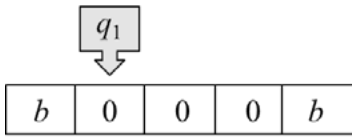
También se usa 0 → R en la transición de q_2 a q_3 lo cual significa que $M1$ se mueve a la derecha cuando lee un 0 en el estado q_2 , pero no se altera la cinta, expresado como $\delta(q_2, 0) = (q_3, 0, R)$.

A continuación se puede ver la ejecución de las tres primeras transiciones de $M1$ con la cadena de entrada $w = 0000$, el movimiento de la cabeza y los cambios en la cinta.

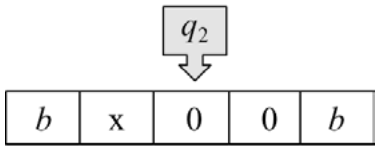
q_0000 corresponde a



bq_1000 corresponde a



bxq_200 corresponde a



La ejecución completa de *M1* es como sigue:

q_00000b
bq_1000b
bxq_200b
$bx0q_30b$
$bx0xq_2b$
$bx0q_4xb$
bxq_40xb
bq_4x0xb

$bxq_1 0xb$
$bxxq_2 xb$
$bxq_2 xxb$
$bxxq_2 xb$
$bxxxq_2 b$
$bxxq_4 xb$
$bxq_4 xxb$
$bq_4 xxxb$
$bxq_1 xxb$
$bxxq_1 xb$
$bxxxq_1 b$
$bxxxq_1 b_{sí}\ \gamma$

Es interesante considerar que técnicamente, cualquier "objeto matemático finito" puede representarse mediante una sucesión finita con un alfabeto adecuado. Por ejemplo, los números siempre podrían representarse como números binarios; para representar grafos, se puede construir una sucesión con información del número de nodos y después los elementos de su matriz de adyacencia.

Existen muchos tipos de máquinas de Turing. Un tipo de ellas son las máquinas de Turing con muchas cintas, sin embargo, máquinas de este tipo son polinómicamente equivalentes en cuanto a tiempo, a las que tienen una sola cinta.

2.2 Máquinas de Turing no deterministas

Otro modelo de máquina de Turing menos realista es la máquina de Turing *no determinista* (MTND), no obstante, representan un concepto importante para definir la complejidad computacional.

Se puede simular cada MTND mediante una máquina de Turing determinista pero con un costo de tiempo exponencial, esto es, con una pérdida exponencial de eficiencia.

El no determinismo puede consistir en que:

- La máquina no se encuentre totalmente definida;
- Para un estado actual y el símbolo actual de la cinta, puede haber un número finito de movimientos a elegir, o ninguno disponible, en cuyo caso, la máquina parará los cálculos.

El grado de no determinismo de una MTND, *N*, es la cantidad máxima de movimientos para cualquier par actual estado-símbolo (combinaciones), precisamente de aquí se desprende también la idea de costo de tiempo exponencial.

Se dice que una máquina de Turing no determinista *M* aceptará una cadena *x* si existe alguna secuencia de selecciones no deterministas que resulta en el estado de aceptación "sí". Una máquina de Turing no determinista rechaza su entrada si no existe ninguna secuencia de selecciones no deterministas que resultará en "sí". En otras palabras, se dice que una máquina de Turing no determinista *M* aceptará una cadena *x* si es *posible* que *M* llegue a su estado de parada después de iniciar sus cálculos con la entrada *x* y se utiliza el término "posible", ya que si una máquina no determinista no alcanza el estado de parada, puede ser debido a una mala decisión de la máquina, y no a que la cadena *x* no pertenezca al lenguaje.

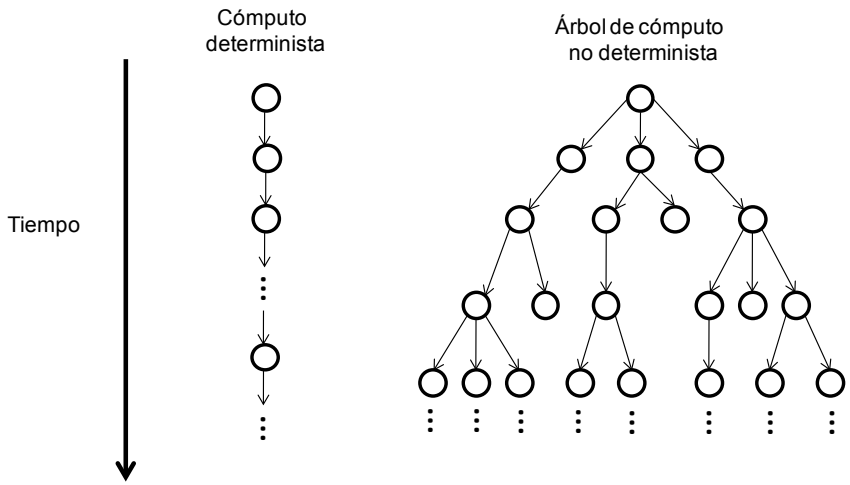


Figura 10. Ejecución de una MTND. Note que el árbol de cómputo no determinista acepta (decide) si alguna rama alcanza la configuración aceptadora y note también que el tamaño de un árbol es exponencial en su profundidad

Note que cualquier máquina de Turing determinista es también no determinista. Además, es lógico que una máquina de Turing determinista se puede simular por medio de una no determinista. También una máquina de Turing determinista puede simular a una no determinista. Por lo tanto, note que no se gana ninguna potencia adicional a causa del no determinismo.

Una afirmación importante sobre la robustez del modelo de máquina de Turing es que según la tesis de Church-Turing (ver Wegener (2005)), formulada por Alan Turing y Alonzo Church de forma independiente a mediados del siglo XX, se podría probar matemáticamente que para cualquier algoritmo es posible crear una máquina de Turing equivalente.

Figura 11. La tesis de Church-Turing

En otras palabras, la tesis clásica de Church-Turing dice que todos los modelos de computación se pueden simular unos a otros, así que el conjunto de problemas que se puede resolver algorítmicamente es independiente del modelo de computación (el cual incluye tanto a la computadora como al lenguaje de programación).

Es importante considerar que aunque se asume como cierta, la tesis de Church-Turing no puede ser probada ya que no se poseen los medios necesarios, por eso es una tesis. La evidencia de su verdad es abundante pero no definitiva. Precisamente, la tesis de Church establece que la definición de algoritmo o procedimiento efectivo, es una máquina de Turing.

2.3 Clases de complejidad computacional

El concepto, abstracto, de máquinas de Turing, al principio puede parecer un poco difícil de comprender, sobre todo para aquellos que no están familiarizados con temas de Teoría de la Computación (lenguajes naturales, lenguajes formales, autómatas, etc.); sin embargo, por ahora, es conveniente afirmar que existen diferentes clases de complejidad computacional, las cuales se pueden definir, precisamente, a través de máquinas de Turing deterministas y no deterministas.

En general, es posible definir diferentes tipos de clases de complejidad computacional mediante las siguientes características:

i. Tipo de máquina de Turing a utilizar
ii. Modalidad de computación: determinista, no determinista, etc.
iii. Recurso a controlar: tiempo, espacio de memoria, etc.
iv. Cota que se impone al recurso: una función de complejidad como $c, \sqrt{n}, n, \lceil \log n \rceil, n\log n, n^2, n^3+3n, 2^n, n!$; donde n es el parámetro de la función $f(n)$ y c es una constante

Recordemos que dadas dos funciones de complejidad f y g, también $f + g$, $f \times g$ y 2^f son funciones de complejidad.

Así, formalmente, una *clase de complejidad* es el conjunto de todos los lenguajes decididos por alguna máquina de Turing M del tipo i) que opera en el modo de computación ii), tal que para toda entrada x, M necesita el máximo de $f(|x|)$ unidades del recurso definido en iii), donde la cota del uso del recurso es una función f de iv).

Dada una función de complejidad f, se tienen las clases siguientes:

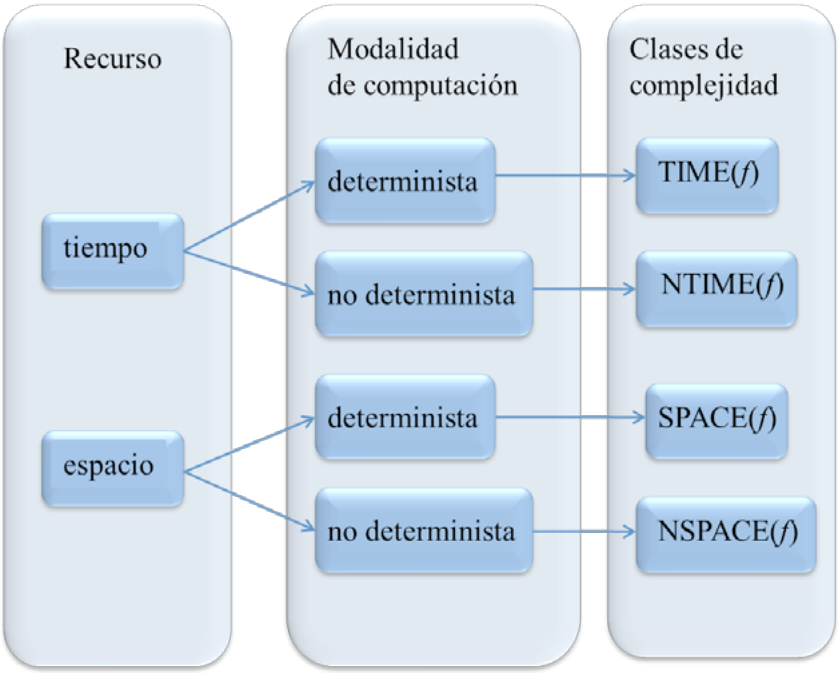


Figura 12. Clases de complejidad

Nos ocuparemos especialmente de las clases de complejidad de tiempo por ser las más utilizadas.

Considere ahora las siguientes definiciones importantes:

Definición 2.3.1. Una clase de complejidad **TIME**($f(n)$) es el conjunto de lenguajes L tales que una máquina de Turing (determinista) decide L en un tiempo $f(n)$.

Definición 2.3.2. Una clase de complejidad **NTIME**($f(n)$) es el conjunto de lenguajes L tales que una máquina de Turing no determinista decide L en un tiempo $f(n)$.

Definición 2.3.3. El conjunto **P** contiene todos los lenguajes decididos por máquinas de Turing (deterministas) en tiempo polinomial,

$$\mathrm{P} = \bigcup_{k>0} \mathrm{TIME}(n^k) \tag{2.1}$$

Definición 2.3.4. El conjunto **NP** contiene todos los lenguajes decididos por máquinas de Turing no deterministas en tiempo polinomial,

$$\mathbf{NP} = \bigcup_{k>0} \mathrm{NTIME}(n^k) \tag{2.2}$$

Definición 2.3.5. El conjunto **EXP** contiene todos los lenguajes decididos por máquinas de Turing no deterministas en tiempo exponencial,

$$\mathrm{EXP} = \bigcup_{k>0} \mathrm{TIME}(2^{n^k}) \tag{2.3}$$

Por ejemplo, algunas tareas de la clase **P** son:

- verificar si un número es par o impar, tiempo $O(1)$;
- búsqueda de un elemento entre n elementos, tiempo $O(n)$;
- ordenación de n elementos, tiempo $O(n\log n)$;
- multiplicación de matrices, tiempo $O(n^3)$;
- los problemas de la sección 1.5, considerados como bien resueltos.

Por último, daremos una definición que será importante para el tema de Reducciones polinomiales y NP-completez.

Definición 2.3.6. Sea L un lenguaje. Decimos que L está en la clase de complejidad de espacio SPACE($f(n)$) si existe una máquina de Turing con entrada y salida que decide a L y opera dentro de un espacio acotado por $f(n)$.

2.4 Problemas sin solución

Una pregunta que surge naturalmente es ¿existen problemas matemáticos que estén bien definidos y para los cuales aún no existe un algoritmo? Turing mostró que existen más lenguajes que máquinas de Turing, entonces, existen lenguajes que no son decididos por ninguna máquina de Turing.

Así, un problema que no cuenta con una máquina de Turing se llama un problema *sin solución*, en inglés *undecidable problem*, que puede traducirse como *problema indecidible* o un problema *intratable* o un problema *irresoluble.* Podemos aplicar estos conceptos a problemas de decisión para los cuales existe un conjunto al que no le podemos asignar una respuesta, ni afirmativa ni negativa, esto es, no existe un algoritmo que nos permita determinar si el problema tiene solución.

2.4.1. Problema Halting

El problema Halting (o problema de paro o problema de la detención) es un problema de decisión para máquinas de Turing. Es el ejemplo de problema irresoluble más conocido. Consiste en decidir si una máquina de Turing se detendrá con cierta entrada, o bien quedará en un ciclo infinito.

Problema Halting

Dados: una descripción M_n de una máquina de Turing M y una entrada x;

Pregunta: ¿va a parar la máquina de Turing M cuando se ejecuta con x?

Alan Turing demostró en 1936 que un algoritmo general para resolver el problema Halting para todos los posibles pares de programas-entradas, no puede existir. Esto significa que no existe un algoritmo que se pueda aplicar a cualquier par programa-entrada arbitario para decidir si el programa parará cuando se ejecute con esa entrada. Este fue el primer problema para el que se demostró formalmente que no tenía solución, esto es, que era indecidible o irresoluble sobre las máquinas de Turing.

Una de las razones por la que es importante conocer que el problema Halting no tiene solución, es que nos permite decidir si otros problemas son resolubles o no. El método típico para probar si un problema es decidible es la técnica de reducción[3]. Para hacer esto se muestra que si se encontró una solución para el nuevo problema, esto podría ser usado para decidir al problema Halting que se sabe es irresoluble (por medio de transformar instancias del problema Halting en instancias de problema nuevo). Así, ya que sabemos que el método no puede decidir el problema Halting, el método tampoco puede decidir el problema nuevo.

3 Es el contexto de complejidad computacional, es una transformación de un problema en otro problema. El tema de reducción será tratado más a fondo en el último apartado de estos apuntes.

Actividades sugeridas

Conteste las siguientes preguntas:

1. ¿Si un lenguaje L es recursivo, también es recursivamente numerable? Conteste sí o no y justifique su respuesta.

2. ¿Si un lenguaje L es recursivamente numerable, también es recursivo? Conteste sí o no y justifique su respuesta.

3. Explique por qué cualquier máquina de Turing determinista es también no determinista.

4. Indique cuántas configuraciones distintas puede alcanzar una máquina de Turing que use N casillas.

5. Corrobore el resultado del ejemplo 2.1 y ejecute la máquina de Turing $M1$ con w=00000.

3 Problema de decisión, problema de Satisfactibilidad y otros problemas fundamentales

3.1 Problema de decisión

Un marco contextual apropiado para el análisis de problemas es el de los problemas de optimización, los cuales consideran minimizar o maximizar una función (la función objetivo), sujeta a una o más restricciones.

Una clase especial de problemas de optimización requiere únicamente de una respuesta "sí" o "no". Estos son llamados *problemas de decisión* o *problemas de reconocimiento.*

La teoría de complejidad computacional no dedica estricta atención a los problemas de decisión. Sin embargo, su uso implica una situación muy ventajosa ya que los resultados obtenidos para un problema de decisión, pueden ser fácilmente generalizados, considerando que:

1. Para cada problema de optimización, existe una versión de decisión.

2. Cualquier resultado de complejidad para la versión de decisión se mantiene también para el problema original, esto es, el problema original no es "mucho más duro" que la versión de decisión.

Observe que lo anterior puede entenderse como: un problema de optimización original no es más difícil que su versión de decisión.

Ejemplo 3.1

Sea un IKP (Problema Mochila entero, *Integer Knapsack Problem*): Tenemos una cantidad no limitada de n artículos de tamaños S_i, i=1,…,n y valores V_i, i=1,…,n y también tenemos un contenedor de capacidad C. El objetivo es cargar el contenedor con X_1 piezas del artículo 1, X_2 piezas del artículo 2, etc., de tal forma que el valor total se maximice:

$$\text{Maximizar} \quad X_1V_1 + X_2V_2 + \ldots + X_nV_n$$

$$\text{sujeto a: } X_1S_1 + X_2S_2 + \ldots + X_nS_n \leq C$$

donde, $X_1, X_2, \ldots, X_n$ son variables de decisión que representan la cantidad de un artículo en particular que será cargado en el contenedor, y toman valores no negativos.

La correspondiente versión de decisión IKP-dec es: Dados dos enteros positivos K, C y dados S_i, V_i, i=1,…,n; ¿existe una asignación de enteros no negativos para las variables $X_1, X_2, \ldots, X_n$ tal que

$$X_1V_1 + X_2V_2 + \ldots + X_nV_n \geq K \quad \text{y}$$

$$X_1S_1 + X_2S_2 + \ldots + X_nS_n \leq C\,?$$

La respuesta a este problema sólo puede ser sí o no.

3.2 Problemas de lógica booleana

3.2.1. Problema de Satisfactibilidad (SAT)

Otro problema importante en el contexto es el *problema de satisfactibilidad* (SAT), el cual es un problema de lógica booleana. El problema SAT puede definirse como sigue:

Se puede mostrar que, dada una expresión booleana general (también llamada proposición), podemos construir una equivalente en Forma Normal Conjuntiva (FNC), esto es, una fórmula como:

$$C_1 \text{ Y } C_2 \text{ Y } C_3 \text{ Y} \ldots \text{ Y } C_m$$

donde C_i ; i=1,2,…,m son cláusulas que consisten de disyunciones de variables booleanas, simples o negadas.

Ejemplo 3.2

Sea la siguiente expresión booleana en FNC (¬ es el símbolo que se establece para negación):

$$(X_1 \text{ O } X_2 \text{ O } X_3) \text{ Y } (X_1 \text{ O } \neg X_2) \text{ Y } (X_2 \text{ O } \neg X_3) \text{ Y } (X_3 \text{ O } \neg X_1) \text{ Y } (\neg X_1 \text{ O } \neg X_2 \text{ O } \neg X_3)$$

Entonces el problema de decisión asociado es:

Dada una expresión booleana en FNC, ¿es ésta satisfactible? Esto significa: ¿existe un conjunto de valores "verdadero-falso" que asignados a las variables de la expresión, componen un resultado "verdadero" para toda la proposición?

Utilizando tablas completas de verdad, el problema SAT se puede resolver en tiempo $O(n^2 \cdot 2^n)$. Entonces, aplica que SAT Î NP, pero no se sabe si está también en P.

3.2.2. Problema Complemento SAT

Evidentemente, complemento SAT es también un problema de lógica booleana:

Dada una expresión booleana en FNC, ¿es ésta no satisfactible? Esto significa: ¿existe un conjunto de valores "verdadero-falso" que asignados a las variables de la expresión, componen un resultado "falso" para toda la proposición?

3.2.3. Problema Hornsat

Un problema del tipo SAT de interés, es una versión en la cual solamente se permiten *cláusulas Horn.* Una cláusula Horn es una disyunción que tiene a lo más una variable booleana positiva y cualquier número de variables negadas.

Entonces, el problema Hornsat dice:

Dada una expresión booleana que es una conjunción de cláusulas Horn, ¿es satisfactible?

El problema Hornsat sí tiene un algoritmo polinomial, de hecho, es actualmente uno de los problemas más "duros" para los que se sabe que es computable en tiempo polinomial, así que, Hornsat ∈ P.

3.3 Problemas de grafos

3.3.1. Problema de alcance (Reachability)

Un problema básico de grafos es el problema de alcance, el cual dice:

Dado un grafo $G=(V,E)$ y dos vértices $v,u \in V$, ¿existe un camino de v a u ? en otras palabras ¿se puede alcanzar a v desde u?

Existe un algoritmo básico para resolver Reachability, llamado algoritmo de Floyd-Warshall[4], que además, cuando se aplica a grafos ponderados, encuentra los caminos más cortos (*shortest path*); el tiempo del algoritmo es $O(n^3)$.

3.3.2. Complemento Reachability

Evidentemente, es el complemento del anterior y dice:

Dado un grafo $G=(V,E)$ y dos vértices $v,u \in V$, ¿es verdad que no existe ningún camino de v a u ?

3.3.3. Problema Hamilton path

En este problema se debe decidir si un grafo determinado, contiene un camino Hamiltoniano. El problema Hamilton path dice:

Dado un grafo $G=(V,E)$, ¿existe un camino C en G tal que visite cada vértice exactamente una vez?

4 También conocido como algoritmo de Floyd o algoritmo de Roy-Warshall, define la ruta más corta en un grafo ponderado, con pesos positivos o negativos. La ejecución del algoritmo definirá las longitudes de las rutas más cortas entre todo par de vértices. Este algoritmo es un ejemplo de aplicación de la programación dinámica .

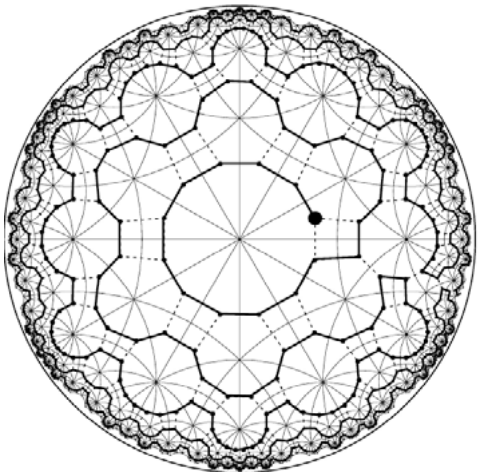


Figura 13. Un grafo cayley con un hamiltonian path
(Fuente: http://www.D.Umn.Edu/~ddunham/notices/paper.Html)

3.3.4. Problema Hamilton cycle

Ahora, se debe decidir si un grafo determinado, contiene un ciclo hamiltoniano. El problema Hamilton cycle dice:

Dado un grafo $G=(V,E)$, ¿existe un ciclo C en G tal que visite cada vértice exactamente una vez?

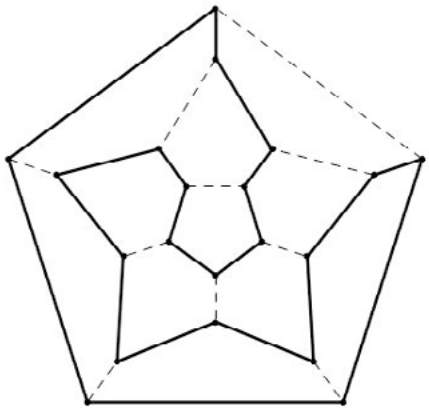


Figura 14. Hamilton cycle de la gráfica
(Fuente: http://jwilson.coe.uga.edu/EMAT6680/Yamaguchi/emat6690/essay1/GT.html)

Un problema de optimización importante que está basado en el problema de decisión Hamilton cycle es precisamente el problema del agente viajero (TSP, por sus siglas en inglés). El TSP es precisamente la versión ponderada del problema de decisión Hamilton cycle, en el cual se debe encontrar un ciclo hamiltoniano para el cual, la suma total de los pesos de las aristas incluidas en el ciclo, es mínima. Este problema fue tratado en el ejemplo 1.1.

El problema de decisión asociado a TSP es TSP-dec, que se describe a continuación.

3.3.5. Problema TSP de decisión

El problema del agente viajero de decisión o TSP-dec se describe de la siguiente forma, dado un grafo ponderado $G=(V,E)$ con pesos en las aristas y una constante c, ¿existe un ciclo C en G tal que C visite cada vértice exactamente una vez y que la suma de los pesos de las aristas de C sea menor o igual a c? Para comprender la relación entre el TSP y su versión de decisión, observe el siguiente ejemplo.

Ejemplo 3.3

Sea el TSP: "Dadas las coordenadas de n ciudades, defina el tour cerrado más corto que visita cada ciudad exactamente una vez". La correspondiente versión de decisión, TSP-dec es:

> "Dado un número de n ciudades, $n \geq 3$ y entero, una matriz n x n de enteros que representan distancias no negativas $D = [\ d_{ij}\ ]$, y un entero no negativo L : ¿existe un tour cerrado que pase por cada ciudad exactamente una vez, con una distancia total $\leq L$?

Una solución para el problema TSP original es un tour con una longitud específica que se puede comparar inmediata y fácilmente con L y así, resolver el TSP-dec, que sólo requiere una respuesta sí o no. Ahora, inversamente suponga que tenemos un algoritmo llamado A, y que a partir de una matriz de distancias obtiene la longitud de un tour óptimo para el problema TSP-dec. Entonces, el siguiente procedimiento da una solución para el problema original. Suponga primero que mediante el algoritmo de búsqueda binaria es posible obtener la longitud óptima del tour.

Procedimiento para obtener la solución óptima del TSP

```
begin
     Use búsqueda binaria para definir la longitud óptima, a la
     que llamaremos Lmin (La secuencia de visitas, o sea, el
     tour óptimo es aún desconocido)
     for i=1,...,n-1
      for j=i+1,…,n !comentario: este ciclo doble
                      examina cada arco (i,j) entre  ciudades
           guardado = d_ij
           d_ij = (número suficientemente grande en
                comparación con los elementos de la
                matriz de distancias)
                !comentario:esta acción implica borrar
                el arco de la matriz
           call A (con la nueva matriz de distancias,
                  definir si existe un tour óptimo con
                  costo £ Lmin)
           if respuesta = "no" then d_ij = guardado
           !comentario: si después de borrar el arco
           (i,j) perdemos la solución óptima, este
           arco está en el tour óptimo
        next j
     next i
end
```

Suponga que Lmin=50, si se borra el arco (i,j) y no se puede obtener un tour con costo ≤ 50, entonces quiere decir que al borrarlo, sólo puedo encontrar tours con costo >50, por tanto, se pierde el óptimo, lo cual nos indica que el arco (i,j) debe ser incluido nuevamente para mantener el óptimo, o sea, el arco está en el tour óptimo. En caso de que borrando el arco, puedo obtener tours con costo ≤ 50, entonces el arco no es necesario y puede borrarse definitivamente. Al final del procedimiento, el conjunto de arcos que no tiene sus longitudes modificadas, conforma el tour óptimo.

Si el algoritmo A es polinomial, el procedimiento completo es también polinomial ya que, primero, la complejidad del algoritmo de búsqueda binaria indica que a lo más se requieren $log_2(n+1)$ pasos; y el ciclo es de orden n^2 veces el orden de A. Por lo tanto, un algoritmo de tiempo polinomial para el TSP-dec produciría un algoritmo de tiempo polinomial para el TSP. Este es el tipo de resultados que se usan para otros problemas de interés.

Observe, como ya se mencionó, que una solución para el TSP original es un tour de longitud específica, la cual puede compararse fácilmente con L, y así, también se resuelve el TSP-dec, entonces decimos que TSP se reduce al TSP-dec y decimos que TSP no es más difícil que su versión de decisión.

Finalmente para este ejemplo, recuerde que precisamente la gran dificultad para resolver el problema TSP-dec es que el algoritmo A de tiempo polinomial, no existe.

Se ha puntualizado que un problema de optimización original no es más difícil que el problema de decisión asociado, cualquier resultado probado sobre la complejidad del problema de decisión será aplicable también al problema de optimización.

Por supuesto, hay problemas que en su forma original son ya problemas de decisión, tal como el problema Halting.

3.3.6. Problema de Clique (Camarilla)

Este problema trata de la existencia de subgrafos completos, esto es, conjuntos de vértices $C \subseteq V$ tales que para todo par de vértices $v, u \in C$, existe una arista $\{v, u\} \in E$, es decir, existe una arista que los conecta.

En otras palabras, un clique es un subgrafo en que cada vértice está conectado a cada otro vértice del grafo. Esto equivale a decir que el subgrafo inducido por V es un grafo completo. El tamaño de un clique $|C|$ es el número de vértices que contiene.

El problema de Clique dice:

Dado un grafo no dirigido $G=(V,E)$ y un entero $k > 0$, ¿existe un subgrafo completo inducido por el conjunto $C \subseteq V$ tal que $|C| = k$?

En otras palabras, el problema del Clique consiste en que dado un grafo, decidir si existe en él un Clique con un tamaño particular.

El problema de Máximo Clique consiste en definir un Clique con el máximo número de vértices.

Aplica que Clique $\in$ NP.

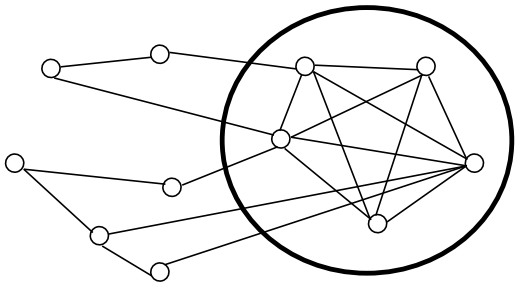


Figura 15. Máximo clique de 5 vértices en un grafo de 11 vértices

3.3.7. Problema Independent set

Lo opuesto de un Clique es un *conjunto independiente* (*independent set*), en el sentido que cada Clique corresponde a un conjunto independiente del grafo complemento.

El problema de Independent set dice:

Dado un grafo no dirigido $G=(V,E)$ y un entero $k > 0$, ¿existe un subgrafo inducido por el conjunto $I \subseteq V$ tal que $|I| = k$ y que no contenga ninguna arista? En otras palabras, ¿existen un subconjunto de k vértices en el grafo, tales que ninguno de ellos está conectado a otro (considere, por supuesto, que ellos pueden estar conectados a vértices fuera del conjunto)?

Un Maximum independent set es el mayor conjunto independiente para un grafo dado G. El problema de definir tal conjunto se llama problema Maximum independent set y es un problema de optimización NP. Como tal, es improbable que exista un algoritmo eficiente para definirlo.

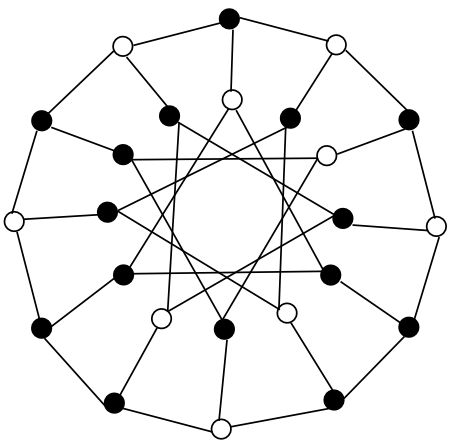
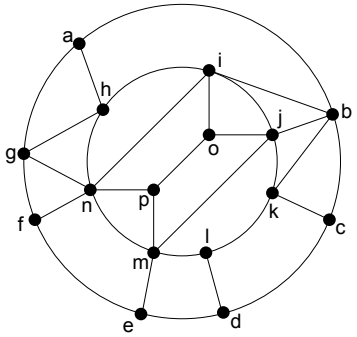


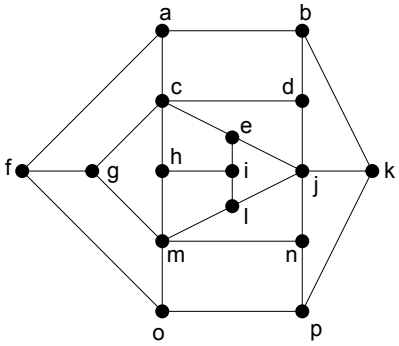
Figura 16. Los nueve vértices blancos forman un conjunto independiente máximo para el grafo Generalizado de Petersen
(Fuente: http://medlibrary.org/medwiki/Independent_set_(graph_theory))

Actividades sugeridas

1. Encuentre el ciclo hamiltoniano en la gráfica.



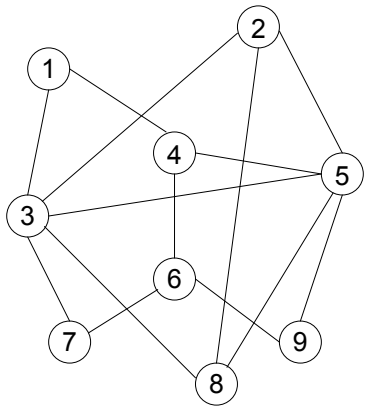
2. Demuestre que la gráfica no contiene un ciclo hamiltoniano.



3. Partiendo de un tablero de ajedrez de ocho por ocho, y siguiendo los movimientos del caballo, empezar en cualquier posición del tablero y recorrer todas las posiciones del tablero, sin repetir alguna de ellas. Véase cada posición como un nodo de una red, y considere que dicha red está conectada por arcos que sigan el movimiento del caballo. ¿Cuántos movimientos son necesarios efectuar para recorrer todo el tablero? Ahora, suponga que tiene un tablero de *n*x*n* ¿Cuántos movimiento requerirá para recorrer todo el tablero con el movimiento del caballo?

4. Explique qué es un código Gray, cómo se construye y cómo se relaciona con un ciclo hamiltoniano.

5. Dado el siguiente grafo, encontrar el Clique de 4 nodos y el clique de 3 nodos. ¿Cuál sería el Clique máximo y cuál el Clique parcial?



6. Describa una aplicación práctica para el problema Clique.

7. Describa una aplicación práctica para el problema Independent set.

4 Clases P y NP

4.1 Algoritmos no-deterministas

Un algoritmo no-determinista es una herramienta teórica, no existe en la realidad. Es como un algoritmo ordinario, pero está permitido usar la siguiente instrucción que en la realidad es imposible:

```
goto both label 1, label 2
```

Esta instrucción realiza el cómputo de dos procesos paralelos en un solo paso, o sea, con el mismo procesador, no en procesamiento paralelo. De aquí la imposibilidad de ejecutar la instrucción, el algoritmo no-determinista debe decidir en cada paso de la ejecución, entre varias alternativas y explorarlas todas antes de encontrar la solución.

Para tener una idea del poder de cómputo no-determinista, consideremos la conocida versión del problema de Programación Lineal Entera (ILP) 0-1 que es un problema "difícil", esto es, no se conoce un algoritmo determinista para él:

Dada una matriz A de $m \times n$ y un m *-vector* entero b, ¿existe un n *-vector* $\boldsymbol{x}$ con elementos 0-1, tal que $Ax=b$?

La siguiente rutina resulta tan poderosa que permitiría resolver el problema en tiempo polinomial, precisamente por medio de la exploración de todas las posibles combinaciones de valores de los elementos del vector ***x***:

```
begin
      for j=1,…,n
            goto both A,B
            A: x_j = 0
            goto again
            B: x_j = 1
            : again
      next  j
      if x=(x_1,…,x_n) satisfies Ax=b then output "yes" else output
      "no"
end
```

Mediante un árbol de soluciones podríamos ver que en cada paso o nivel del árbol, podría resolverse más de un problema, con el mismo procesador.

De hecho, si tuviésemos computadoras capaces de ejecutar esta rutina extra, no sería necesaria la teoría de complejidad computacional.

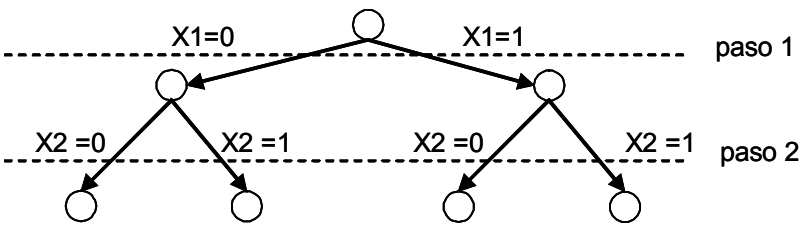


Figura 17. Árbol de soluciones no-determinista

Por otro lado, vimos en la sección anterior cómo, dado un problema de optimización, podemos definir un problema de decisión (o reconocimiento) asociado, esto es, una pregunta que puede ser respondida por "sí" o "no". Sin embargo, varios problemas computacionales bien conocidos son, en principio, problemas de decisión, como los problemas tradicionalmente estudiados por la teoría de la computación, a continuación se mencionan algunos de ellos.

En al capítulo anterior mencionamos el problema Halting (problema de la parada o detención o terminación): Dado un algoritmo y su entrada, ¿alguna vez se detendrá?

También introdujimos el problema de la Satisfactibilidad: Dada una fórmula booleana, ¿es ésta satisfactible? (¿la asignación de valores a sus variables hace que la expresión sea verdadera?).

Tratamos ya también con una versión del problema Hamilton cycle: Dado un grafo G, ¿existe un circuito en G que visite todos los nodos exactamente una vez?

Estos tres problemas son problemas de decisión. Nuestra definición de versiones de decisión de los problemas de optimización nos permite estudiar ambos tipos de problemas de una manera uniforme. Además, desde que hemos señalado que la versión de decisión de un problema no es más difícil que el problema de optimización original, cualquier resultado probado sobre la complejidad de la versión de decisión se aplicará también a la versión de optimización original.

Estamos ahora en posición de definir formalmente dos clases importantes de problemas de decisión, las clases P y NP; para así, poder clasificarlos de acuerdo con su complejidad.

4.2 Clase P

Podemos denotar por P, la clase de los problemas de decisión que pueden ser resueltos por un algoritmo en tiempo polinomial. La clase P puede ser definida con precisión en términos de algún formalismo matemático para los algoritmos, tal como el modelo de la máquina de Turing. Esto provoca que tales modelos de computación tengan una característica excepcional: si un problema puede ser resuelto en tiempo polinomial por alguno de dichos modelos, puede ser resuelto en tiempo polinomial por todos los demás. Por lo tanto, se puede decir que la clase P, es extremadamente estable bajo variaciones en los detalles de nuestros supuestos.

Entonces, estamos satisfechos con definir P informalmente, como la clase de los problemas de decisión que tiene algoritmos de tiempo polinomial; en otras palabras, P es la clase de problemas de decisión relativamente sencilla, para la cual existen algoritmos eficientes. Mencionamos en el capítulo 1 varios problemas representativos de la clase P. Listamos otros a continuación.

- Conectividad de un grafo: Dado un grafo G, ¿G es conexo?
- Reachability: Dado un grafo $G=(V,E)$ y dos vértices $v,u \in V$, ¿existe un camino de v a u ?

- Emparejamiento máximo: Dado un grafo G y un entero k, ¿existe un emparejamiento en G con k o más aristas?
- Árbol de expansión mínima: Dado un grafo $G=(V,E)$ conexo, una función de costo d definida en E, y un entero L, ¿existe un árbol de expansión de G con costo L o menos?

Todos estos problemas pertenecen a la clase P, en otras palabras, dada una instancia de cada uno, tenemos una forma eficiente para decir si la respuesta es "sí" o "no".

4.3 Clase NP

El matemático y científico de la computación norteamericano Stephen Arthur Cook introdujo formalmente la clase NP, aparentemente la clase más abundante de los problemas de decisión. Para que un problema esté en NP, no requerimos que cada instancia pueda ser solucionada en un tiempo polinomial por algún algoritmo. Simplemente requerimos que alguna respuesta "sí" sea "fácilmente" verificable, esto es, que si x es una instancia "sí" del problema, entonces exista un certificado breve (de longitud limitada por un polinomio) para x, el cual puede ser verificado en un tiempo polinomial; dicho de otro modo, para la clase NP, tanto la codificación de la respuesta, como el tiempo que toma verificar su validez deben ser "cortos", esto es, de tiempo polinomial o de tiempo polinomialmente acotado. Formalmente, lo anterior se expresa como: cualquier instancia "sí" del problema tiene la propiedad de "certificado sucinto", y a esto último lo podríamos comprender como, tiempo de verificación breve de una respuesta "sí".

El término NP se establece para "Polinomial No-determinista", debido a una definición alternativa y equivalente, que está basada en la noción de algoritmos no-deterministas. Una observación de la definición anterior es que: con la finalidad de mostrar que un problema pertenece a la clase NP, no es necesario mostrar de qué manera puede producirse una respuesta "sí" (que además es fácilmente verificable); en otras palabras, no es necesario resolver el problema; es suficiente con mostrar que tal verificación (también llamado certificado) existe.

Ejemplo 4.1

Para clarificar lo anterior, consideremos nuevamente la versión de decisión del problema del agente viajero (TSP-dec). Si imaginamos que tenemos una respuesta "sí", ésta es una secuencia de ciudades, digamos $c_1, c_2, c_3, \ldots, c_n$, tal que la suma de las ligas $(c_1,c_2),(c_2,c_3),\ldots,(c_n,c_1)$ es menor o igual a un valor dado L. La forma en que se obtuvo esta solución es irrelevante. Sin embargo, es claro que podemos verificar fácilmente la longitud del ciclo, por lo tanto, podemos decir que sabemos que TSP-dec $\in$ NP. Por otro lado, no conocemos si el problema está también en la clase P, debido a que ningún algoritmo polinomial ha sido aún encontrado para resolver el problema, esto es, un algoritmo que defina la secuencia de ciudades de mínima longitud.

La clase NP consiste de todos los problemas "razonables", i.e., con respuesta "sí" o "no", de importancia teórica y práctica. Para problemas que no están en la clase NP, la propia verificación de que una solución es válida, esto es, la propia verificación de una respuesta "sí", puede ser extremadamente difícil.

Ejemplo 4.2

Considerar la versión de decisión del problema de Clique: Dado un grafo $G=(V,E)$ y un entero k, ¿existe un clique (esto es, un subconjunto de V completamente conexo, en el que cada vértice está conectado a cada otro vértice del grafo) de tamaño k?

No es claro que Clique esté en P. La forma obvia de solucionar este problema podría ser someter todos los $\binom{|V|}{K}$ subconjuntos de V con cardinalidad k a la prueba de si ellos satisfacen el requerimiento del problema. El punto es, por supuesto, que existe un número exponencial de tales conjuntos. De hecho, no se conoce ningún algoritmo polinomial para resolver este problema, Clique está en NP. Para comprender esto, suponga que nos dan una instancia "sí" del Clique; en otras palabras, nos dan $G=(V,E)$, un grafo y un entero k tal que G tiene un clique C de cardinalidad k. Entonces esta instancia tiene un certificado absolutamente conciso, esto es, una lista de los nodos en el clique C. Este certificado puede ser revisado eficientemente para validación, porque uno solamente tiene que verificar que C consiste de k nodos, y que todos estos nodos son conectados por las aristas en E.

Podemos formalizar todas las ideas anteriores como sigue. Sea Σ un alfabeto finito fijo y sea $\$$ un símbolo especial en Σ (el símbolo $\$$ es especial porque marca el fin de la entrada y el inicio del certificado). Si x es una secuencia de símbolos de Σ, su longitud, es decir, el número de símbolos que hay en ella, se denota por $|x|$.

Entonces tenemos la siguiente definición:

Definición 4.3.1. Decimos que un problema de decisión A está en la clase NP si existe un polinomio $p(n)$ y un algoritmo a (el algoritmo de certificar - revisar) tal que lo siguiente es verdadero:

La secuencia x es una instancia "sí" de A si y sólo si existe una secuencia de símbolos en Σ, la ceritificación $c(x)$, $|c(x)|\leq p(|x|)$, con la propiedad que si se provee la entrada $x\$c(x)$ al algoritmo a, éste alcanza una respuesta "sí" después de a lo más $p(|x|)$ pasos.

Ejemplo 4.2 *(continuación)*

En nuestro ejemplo, si x es la codificación de una instancia "sí" en el problema de Clique (G,k), entonces el certificado de x, $c(x)$, podría ser un código de la lista de los nodos en un Clique C apropiado. El algoritmo de certificar-revisar examinaría si x es claramente la codificación de un grafo G y un entero k, si $c(x)$ es un conjunto de vértices C, si $|C| = k$, y si existe una arista en G para cada par de vértices u,v de C. De hecho, se puede tomar al polinomio $p(n)$ como $2n^2$.

Ejemplo 4.3

Considerar nuestro bien conocido problema del agente viajero (TSP) :Dado un entero n, una matriz simétrica n x n no negativa de enteros $[d_{ij}]$, y un entero L, encontrar si existe un ciclo C tal que $\sum_{j=1}^{n} d_{iC(j)} \leq L$.

Este problema está en la clase NP porque, dada una instancia $(n, [d_{ij}], L)$ del TSP con una respuesta "sí", podemos demostrar este hecho de forma absolutamente práctica, exhibiendo un tour apropiado. El certificado $c(x)$ de tal instancia podría entonces ser una codificación de un ciclo C que satisfaciera $\sum_{j=1}^{n} d_{iC(j)} \leq L$. El algoritmo a podría revisar si n, $[d_{ij}]$ y L son apropiadas, si C es claramente un ciclo, y si su tamaño total es L o menos.

Ejemplo 4.4

Un problema de programación lineal entera (integer linear programming, ILP) es el siguiente problema de optimización.

$$\begin{aligned} &\text{Minimizar } c'x \\ &\text{sujeto a:} \\ &\quad Ax = b \\ &\quad x \geq 0 \\ &\quad x \text{ entero} \end{aligned}$$

Los valores de A, b y c son enteros.

Considerar la siguiente versión de decisión del ILP:

Dada una matriz A entera $m \times n$ y un *m-vector* entero b, ¿existe un *n-vector* x tal que $Ax=b$, $x \geq 0$?

Note que, en la transformación de un ILP a su versión de decisión, no necesitamos incluir explícitamente el entero L y la desigualdad $c'x \leq L$, como normalmente lo hicimos en otras transformaciones. Esto es porque esta desigualdad puede ser transformada en una igualdad al añadir una variable de holgura y entonces incorporarla al sistema $Ax=b$.

El problema ILP de decisión está en NP. Para ver esto, consideremos el teorema que establece que si un problema entero tiene solución factible, entonces tiene una solución factible $x \in \{0,1,\ldots,M\}^n$ donde $M = n(ma_1)^{2m+3}(1+a_2)$. Tal solución concisa podría servir como una verificación breve de una instancia "sí" del ILP, ya que la longitud de su representación binaria es polinomial con respecto al tamaño de la entrada.

Es importante observar que, para establecer que un problema está en NP, no tenemos que explicar cómo se puede obtener eficientemente el certificado $c(x)$ dada una entrada x. Simplemente debemos mostrar la *existencia* de al menos *una* secuencia (cadena de símbolos) para cada x.

Ahora, note que P es un subconjunto de NP, P $\subseteq$ NP. En otras palabras, cada problema resuelto eficientemente es también brevemente certificable.

Pero, ¿por qué NP es una clase interesante de problemas de decisión? Primero, porque contiene a P, problemas con cuyos resultados estamos satisfechos. Segundo, contiene muchos problemas de nuestro interés, y cuya pertenencia a P está en duda. Ejemplo de esto son el TSP, ILP, Clique, y Hamilton cycle. De hecho, podemos decir que las versiones de decisión de todos los problemas de optimización combinatoria están en NP. Para fortalecer esta idea, recordemos que los problemas de optimización combinatoria apuntan al *diseño óptimo* de objetos, tales como ciclos, rutas, conjuntos de nodos, particiones y listas de enteros. Por lo tanto, es razonable esperar que, una vez en-

contrada, la solución óptima puede ser especificada de forma precisa y de este modo servir como un certificado para la versión de decisión. De ninguna manera podemos esperar diseñar algo óptimamente si no podemos diseñarlo dentro de un tiempo razonable y limitado. Por lo tanto, la clase NP surge naturalmente en el estudio de la complejidad de los problemas de optimización combinatoria.

Pero, ¿es P un subconjunto propio de NP, o es el caso de que P y NP son los mismos? Si lo último fuera cierto, muchos problemas, ahora notorios por su dificultad (TSP, Clique, ILP, y Satisfactibilidad, por nombrar algunos) podrían estar en P, y por consiguiente ser (supuestamente) fáciles.

Es ampliamente aceptado ahora, que P es un subconjunto propio de NP, $P \subseteq NP$. Esta es una conjetura muy importante en complejidad computacional, ya que en otras palabras dice: "no todos los problemas pueden resolverse eficientemente".

Actualmente, la pregunta teórica más destacada que enfrentan los científicos de la computación es el llamado problema P = NP. Es interesante considerar que, si bien no existe la prueba formal, existe una fuerte evidencia para la validez de esta última conjetura.

La principal herramienta matemática de este tipo de investigaciones ha sido la noción de *reducción*, la cual examinaremos en el capítulo siguiente.

5 Reducciones polinomiales y NP-completez

5.1 Reducciones y transformaciones polinomiales

Las herramientas básicas para relacionar las complejidades de varios problemas son las *reducciones* y las *transformaciones*.

Una clase de complejidad contiene una colección infinita de problemas, por ejemplo, la clase NP contiene problemas como Sat, Reachability, Clique, etc. Sin embargo, no todos son igualmente difíciles, a pesar de pertenecer a la misma clase.

Para establecer una categorización de problemas con respecto a su dificultad, existe la construcción de reducciones, que nos dice: un problema A es por lo menos tan difícil como otro problema B si existe una *reducción* del problema B al problema A. En otras palabras

$$(\text{dificultad de } A) \geq (\text{dificultad de } B) \text{ si } B \text{ se reduce a } A.$$

Básicamente, el término *reducción* se utilizó en el sentido de transformar las instancias de un problema en instancias de otro. Un problema B reduce al problema A si existe una transformación R tal que para toda entrada x del problema B produce una entrada equivalente y de A, tal que $y=R(x)$.

Equivalente aquí significa que, la respuesta para la entrada y de A, "sí" o "no", es una respuesta correcta para la entrada x de B, esto es, si la respuesta al problema A con la entrada y es la misma que la del problema B con la entrada x, esto es, $x \in B$ si y sólo si $R(x) \in A$.

Entonces, observemos que para resolver el problema B con la entrada x, habrá que construir $R(x)$ y después resolver el problema A para obtener la respuesta que puede aplicarse a los dos problemas A y B, con sus entradas respectivas. Esto es, si tenemos un algoritmo para resolver el problema A y un algoritmo para construir $R(x)$, la conjunción de ambas situaciones nos da un algoritmo para resolver el problema B (véase la figura 18).

Si además, $R(x)$ se construye fácilmente, se entiende que A es por lo menos tan difícil como B; en otras palabras, B no puede ser más difícil de resolver que A.

Para que el concepto de reducción sea útil, debe involucrar la mayor debilidad computacional posible. Adoptaremos las *reducciones de espacio logarítmico* como nuestra noción de "reducción eficiente". Consideremos las siguientes definiciones.

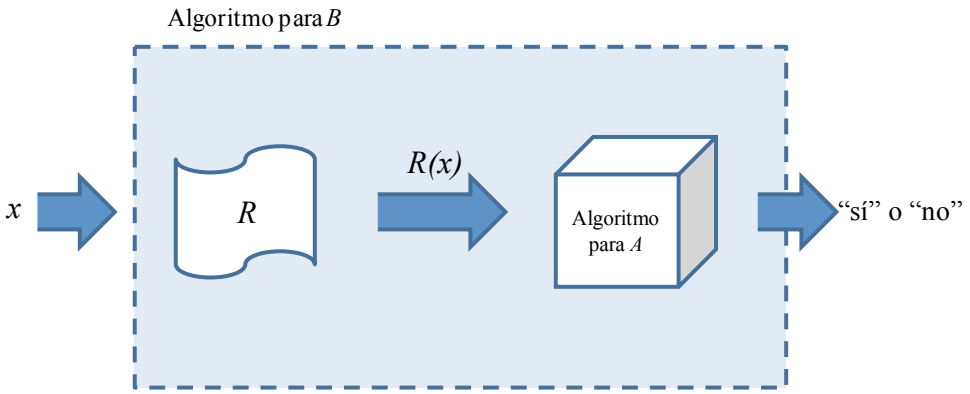


Figura 18. Reducción de B a A
(Fuente: papdimitriou, christos h. (1994), *Computational complexity.* Addison wesley longman, united states of america)

Definición 5.1.1. Decimos que un lenguaje L_1 es reducible a un lenguaje L_2, si y sólo si existe una función R de cadenas a cadenas computada por una máquina de Turing determinista en espacio $O(\log n)$ tal que para toda entrada x se cumple que $x \in L_1$ si y sólo si $R(x) \in L_2$. A la función R se le llama una reducción de L_1 a L_2.

Definición 5.1.2. Sea L un lenguaje. Decimos que L está en la clase de complejidad de espacio $\log n$ (SPACE($\log n$)) si existe una máquina de Turing con entrada y salida, que decide L y opera dentro de una cota espacial $\log n$.

A continuación se mostrarán algunos ejemplos de reducciones.

Ejemplo 5.1

Recordemos que el problema Hamilton path es un problema muy difícil. Mostraremos que Sat es como mínimo tan difícil como Hamilton path, esto es, demostraremos que Hamilton path puede reducirse a Sat, es decir, Hamilton path $\propto$ Sat.

Suponga que tenemos un grafo G. Construiremos una expresión booleana $R(G)$, tal que $R(G)$ es satisfactible si y sólo si G tiene un camino hamiltoniano.

Además, suponga que G tiene n nodos 1,2,….,n. Entonces, $R(G)$ tendrá n^2 variables booleanas. Informalmente, la variable x_{ij} representará el hecho: "el nodo j es el *i-ésimo* nodo en el camino hamiltoniano". Lo cual por supuesto puede ser verdadero o falso y lo podemos ver también como

$$x_{ij} = \begin{cases} 1 \text{ si el nodo } j \text{ es el } i\text{-ésimo en el ciclo hamiltoniano} \\ 0 \text{ c.o.c.} \end{cases}$$

La expresión booleana $R(G)$ estará en forma normal conjuntiva, así describiremos sus cláusulas.

Las cláusulas detallarán todos los requerimientos de las x_{ij} que son suficientes para garantizar que ellas codifican un camino hamiltoniano verdadero.

Para comenzar, el nodo j debe aparecer en el tour, esto es representado por la cláusula

$$(x_{ij} \vee x_{2j} \vee \ldots \vee x_{nj})$$

que dice que j debe aparecer en el ciclo hamiltoniano, esto es, j debe estar en la posición 1 o en la 2, o en la 3, …, o en la n , y tendremos una cláusula como esta para cada j.

Pero el nodo j no puede aparecer en dos posiciones del camino, la *i-ésima*, y *k-ésima,* al mismo tiempo, lo cual se expresa con la cláusula

$$(\neg x_{ij} \vee \neg x_{kj})$$

que nos dice que j no está en la posición i o no está en la posición k o no está en ninguna de las dos, esta cláusula se repetirá para todos los valores de j, y de $i \neq k$.

Inversamente, algún nodo debe ser el *i-ésimo*, esto es, en la posición 1 debe ir el nodo 1 o el nodo 2…o el nodo *n;* así que agregamos la cláusula

$$(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$$

para cada *i*, y dos nodos no deberían ser el *i-ésimo* , esto es

$$(\neg x_{ij} \vee \neg x_{ik})$$

para todo *i*, y todo $j \neq k$.

Finalmente, para cada par (i, j) que no es una arista de *G*, no debe ocurrir que *j* siga después de *i* en el ciclo hamiltoniano; por lo tanto, las siguientes cláusulas se agregan para cada par (i, j) que no está en *G* y para $k = 1,\ldots,n - 1$:

$$(\neg x_{ki} \vee \neg x_{k+1,j})$$

Esto completa la construcción. La expresión $R(G)$ es la conjunción de todas estas cláusulas.

Aseveramos que *R* es una reducción de Hamilton path a Sat. Para probar esta aseveración establecemos dos puntos:

1. Que para cualquier grafo *G*, la expresión $R(G)$ tiene una asignación que satisface el valor de verdad si y sólo si *G* tiene un camino hamiltoniano;

2. Que *R* se puede calcular en espacio log *n*.

Suponga que $R(G)$ tiene una asignación *T* que satisface el valor de verdad.

Dado que *T* satisface todas las cláusulas de $R(G)$, debe darse el caso de que, para cada *j* existe una única *i* tal que $T(x_{ij})$ = verdadero, de otro modo, las cláusulas de la forma $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$ y $(\neg x_{ij} \vee \neg x_{kj})$ no pueden ser todas satisfechas.

Similarmente, las cláusulas $(x_{i1} \vee x_{i2} \vee \ldots v\ x_{in})$ y $(\neg x_{ij}\ v \neg x_{ik})$ garantizan que para cada *i* existe una única *j* tal que $T(x_{ij})$ = verdadero.

Por consiguiente, *T* realmente representa una *permutación* $\pi(1),\ldots,\pi(n)$ de los nodos de *G*, donde $\pi(i)=j$ sí y sólo si $T(x_{ij})$ = verdadero.

Sin embargo, las cláusulas $(\neg x_{k,i} \vee \neg x_{k+1,j})$ donde (i,j) no es una arista de G y $k = 1, \ldots, n-1$, garantizan que, para todo k, $(\pi(k), \pi(k+1))$ sí es una arista de G.

Esto significa que $(\pi(1), p(2), \ldots, \pi(n))$ es un camino hamiltoniano de G.

Inversamente, suponga que G tiene un camino hamiltoniano $(\pi(1), \pi(2), \ldots, \pi(n))$ donde π es una permutación.

Entonces, es claro que la asignación verdadero $T(x_{ij})$=verdadero si $\pi(i) = j$, y $T(x_{ij})$=falso si $\pi(i) \neq j$, satisface todas las cláusulas de $R(G)$, y se cumple el primer punto de nuestra aseveración.

Aún tenemos que mostrar que R puede ser computada en un espacio log n. Dado G como una entrada, una máquina de Turing M da como salida $R(G)$ de la siguiente forma: primero escribe n, el número de nodos de G, en binario y con base en n genera en su cinta de salida, una por una, las cláusulas que no dependen del grafo (los primeros cuatro grupos en la descripción de $R(G)$).

Para este fin, M sólo necesita tres contadores i, j, y k, para ayudar a construir los índices de las variables de las cláusulas. Para el último grupo, el único que depende de G dado que son las aristas que no están en G, M de nuevo genera una por una en su cadena de trabajo, todas las cláusulas de la forma

$$(\neg x_{k,i} \vee \neg x_{k+1,j}).$$

Para $k = 1,\ldots,n-1$, después de que tal cláusula se genera, M busca en su entrada si (i,j) es una arista de G y si no lo es, entonces da como salida la cláusula. Todas estas tareas se computan en espacio log n (cuya demostración no expondremos con detalle) y así, se cumple el segundo punto de la aseveración. Esto completa nuestra prueba de que el ciclo hamiltoniano puede ser reducido a Sat.

Esta reducción, tomada de Papadimitriou (1994), es una de las reducciones más importantes ya que es una de las más simples y claras que se pueden encontrar.

Ejemplo 5.2

Considere los dos siguientes problemas de decisión:

Problema de Partición: Dados n enteros positivos $x_1,\ldots,x_n$, ¿existe un $S \subset \{1,\ldots,n\}$ tal que $\sum_{i \in S} x_i = \sum_{i \notin S} x_i$?

Problema Mochila: Dados t+1 enteros positivos $a_1,\ldots,a_t$ y b, ¿existe un $S \subset \{1,\ldots,t\}$ tal que $\sum_{i \in S} a_i = b$?

Suponemos que Partición $\propto$ Mochila. Para ver esto, suponga que $I = (x_1,\ldots,x_n)$ es una instancia para el problema de Partición. Construyamos la siguiente instancia para el problema Mochila: Tomemos $t=n$, $a_i = 2x_i$ para i=1,2,…,t y $b = \sum_{i=1}^{n} x_i$. Podemos decir que transformamos el problema de Partición al problema Mochila. Entonces claramente, la respuesta a la instancia I del problema de Partición es "sí", si y sólo si se tiene una respuesta "sí" del problema Mochila. Es claro que esta transformación toma un tiempo polinomial.

Ahora, mostraremos que también Mochila $\propto$ Partición. Sea $(a_1,\ldots,a_t , b)$ una instancia del problema Mochila. Construyamos la siguiente instancia para el problema de Partición: Tomemos $n=t+1$, $x_i = a_i$ para i=1,2,…, n-1, y $x_n = |\sigma - 2b|$, donde $\sigma = \sum_{i=1}^{t} a_i$. Claramente esta transformación toma tiempo polinomial. Note que x_n se elige de tal forma que b ó σ – b = $\frac{1}{2}\sum_{i=1}^{n} x_i$. Por lo tanto, la respuesta a este problema Mochila es "sí", si y sólo si la respuesta al problema de Partición es "sí".

El ejemplo anterior nos dice, de manera aproximada, que Mochila es un caso especial de Partición o de forma general, que un problema B es un caso especial de un problema A. A este tipo de transformaciones se les conoce como *reducciones Karp* o *reducciones muchos a uno* (*many-one*).

Existe otro tipo de reducciones, las *reducciones Cook*, las cuales establecen que un problema B es reducible en tiempo polinómico de Turing a A si dada una función que resuelve a A en tiempo polinómico, podría escribirse un programa que llamando a la subrutina anterior resuelva a B en tiempo polinómico.

Para comprender mejor las reducciones Cook, considere un algoritmo para resolver el problema B, que en su ejecución llama a una subrutina que resuelve a A, a lo más un número de veces polinomialmente acotado (figura 19).

Decimos que un problema B reduce en tiempo polinomial a otro problema A si y sólo si:

1. hay un algoritmo para B que usa una subrutina para A;
2. cada llamado a la subrutina para A cuenta como un único paso, y
3. el algoritmo para B corre en tiempo polinomial.

Si el algoritmo para B es polinomial $\Rightarrow$ B se reduce polinomialmente a A.

La implicación práctica viene de la siguiente proposición:

Si B se reduce polinomialmente a A y existe un algoritmo de tiempo polinomial para A, entonces hay un algoritmo de tiempo polinomial para B. La prueba está basada en el hecho de que la composición de dos polinomiales es polinomial.

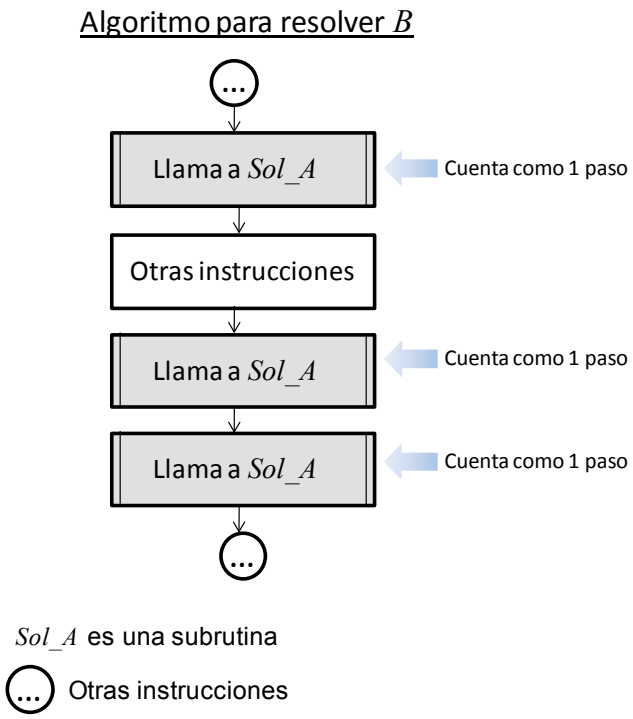


Figura 19. Algoritmo para resolver B cuya subrutina es otro algoritmo para resolver A

Es importante considerar que para poder establecer resultados formales, hay que clasificar las reducciones según los recursos computacionales utilizados por sus transformaciones.

Ejemplo 5.3

En este ejemplo se utilizará la versión de decisión del Problema de Programación Entera (IP) y se mostrará que SAT$\propto$IP-dec.

Consideremos la fórmula booleana en CNF:

$$(X_1 \text{ O } X_2 \text{ O } X_3) \text{ Y } (X_1 \text{ O } \neg X_2) \text{ Y } (X_2 \text{ O } \neg X_3) \text{ Y } (X_3 \text{ O } \neg X_1) \text{ Y } (\neg X_1 \text{ O } \neg X_2 \text{ O } \neg X_3)$$

A partir de ella podemos construir un IP equivalente si asignamos 1 a "verdadero", 0 a "falso" y si

representamos $\neg X_i$ como $(1\text{-}X_i)$:

$$
\begin{aligned}
&X_1 + X_2 + X_3 \geq 1\\
&X_1 + (1\text{-}X_2) \geq 1\\
&X_2 + (1\text{-}X_3) \geq 1\\
&X_3 + (1\text{-}X_1) \geq 1\\
&(1\text{-}X_1) + (1\text{-}X_2) + (1\text{-}X_3) \geq 1
\end{aligned}
$$

Para entender cómo funciona, considere la segunda cláusula (X_1 O $\neg X_2$). Esta cláusula es verdadera si X_1 es verdadera o X_2 es falsa o ambas son verdaderas. Esto es equivalente a requerir que $X_1 = 1$ o $(1\text{-}X_2) = 1$ o $X_1 + (1\text{-}X_2) = 2$, que es exactamente lo que exige la restricción correspondiente $X_1 + (1\text{-}X_2) \geq 1$.

Para completar la construcción de este IP, elegimos una restricción, digamos la primera, $X_1 + X_2 + X_3 \geq 1$, y la convertimos en una función objetivo de maximización, entonces tenemos el IP

$$\text{Maximizar } X_1 + X_2 + X_3$$

sujeto a

$$
\begin{aligned}
&X_1 + (1\text{-}X_2) \geq 1\\
&X_2 + (1\text{-}X_3) \geq 1\\
&X_3 + (1\text{-}X_1) \geq 1\\
&(1\text{-}X_1) + (1\text{-}X_2) + (1\text{-}X_3) \geq 1
\end{aligned}
$$

X_1, X_2, X_3 son binarias

Si en la solución óptima, $X_1 + X_2 + X_3$ toma un valor ≥ 1, entonces la fórmula booleana original es satisfactible. Por lo tanto, un algoritmo para el SAT es:

1. Construir un BIP-dec equivalente en tiempo polinomial;
2. Llamar una vez al algoritmo para BIP-dec y obtenga una respuesta “sí” o “no”;
3. Regrese la respuesta anterior como respuesta de la instancia de SAT.

Esto significa que SAT$\propto$ IP-dec. Aquí finaliza el ejemplo.

A continuación se dan formalmente propiedades importantes de reducciones.

Proposición 5.1

i. Si $B \propto A$ y $A \propto C$, entonces $B \propto C$ (decimos que la relación $\propto$ es transitiva).
ii. Si $B \propto A$, y A está en P, entonces B está en P.
iii. Si B y A están en P, entonces $B \propto A$ y $A \propto B$, y se dice que son equivalentes.

En otras palabras, podemos considerar los siguientes casos básicos derivados de lo anterior:

- (B se reduce a A) y (A es "fácil") $\Rightarrow$ B es fácil;
- (B se reduce a A) y (B es "difícil") $\Rightarrow$ A es difícil (debido a que si A fuera fácil B sería difícil y eso es una contradicción);
- (B se reduce a A) y (A es "difícil") $\Rightarrow$ (no hay conclusión para B, un caso muy común).

De aquí, podemos justificar el decir que "A es al menos tan difícil como B":

Si B se reduce polinomialmente a A, entonces es posible implicar que B puede verse como un caso especial de A (como lo indicamos antes) y en consecuencia, "A es al menos tan difícil como B".

Por ejemplo, el problema de TSP se reduce a TSP-dec, así, esta versión es al menos tan difícil como el problema original TSP.

5.2 Problemas completos

Las relaciones transitiva y reflexiva de reducciones, originan un orden de problemas. Dentro de ese orden de complejidad existen elementos maximales que son de especial interés.

Definición 5.2.1. Sea C una clase de complejidad computacional y $L \in C$ un lenguaje. El lenguaje L es C-completo si para todo $L' \in C$ aplica que $L' \propto L$.

Dicho en palabras, un lenguaje L es completo en su clase C si cada otro lenguaje $L' \in C$ se puede reducir a L.

Otra definición sumamente importante y utilizada es la siguiente:

Definición 5.2.2. Un lenguaje L es C-duro (también se dice C-difícil) si se puede reducir cada lenguaje $L' \in C$ a L , pero no se sabe si es válido que $L \in C$.

Las clases P y NP , así como otras consideradas como principales, contienen problemas completos naturales. De hecho, los problemas completos se utilizan para caracterizar una determinada clase de complejidad.

5.3 NP-completez

Demos ahora la definición de problema NP-completo.

Definición: Un problema de decisión Π se dice que es NP-completo, si Π está en NP y Π' $\propto$ Π para cada problema Π' en NP.

Consideremos la siguiente proposición.

Proposición 5.2

Si Π $\propto$ Π', Π' está en NP y Π es NP-completo, entonces Π' es NP-completo.

Un resultado importante, derivado de lo anterior indica que si pudiéramos demostrar que un problema NP-completo pertenece a la clase P, la implicación sería que P = NP.

Un problema NP-completo tiene las siguientes propiedades importantes:

i. Un problema NP-completo no puede resolverse con ningún algoritmo polinomial conocido (a pesar de los persistentes esfuerzos de investigadores brillantes durante muchas décadas);
ii. Si existiera un algoritmo polinomial (esto es, eficiente) para cualquier problema NP-completo, entonces P = NP, y por lo tanto existirían algoritmos polinomiales para todos los problemas en NP.

La segunda propiedad se deriva de que, si decimos que Π es este problema: todos los problemas

en NP se transforman polinomialmente a Π y dado que la transformación polinomial es una caso especial de reducibilidad polinomial, se aplica el caso 1: (todos los problemas en NP se reducen a Π) y (Π es "fácil") ⇒ todos los problemas en NP son fáciles.

Considere el revolucionario resultado que esto implicaría, sería como considerar que todos los problemas considerados hasta ahora como difíciles podrían resolverse en tiempo polinomial, de inicio la complejidad computacional no tendría razón de ser.

La clase de problemas NP-completo son los problemas más difíciles en NP, lo cual significa que, a menos que P=NP, no existe un algoritmo polinomial para un problema NP-completo. En ese sentido, una tarea importante resulta ser, establecer cuáles problemas son completos para cada clase.

En general, probar NP-completez de un lenguaje L por reducción, consiste de los siguientes pasos:

1. Muestre que el lenguaje L está en NP;
2. Elija un lenguaje L NP-completo a partir del cual se hará la reducción, esto es $L' \propto L$;
3. Describa la función de reducción y argumente que ésta es computable en tiempo polinomial;
4. Argumente que si una instancia x está en L', entonces $f(x) \in L$;
5. Argumente que si $f(x) \in L$, entonces $x \in L'$.

5.3.1 Utilidad de la NP-completez

El significado práctico de la noción de que un problema es NP-completo radica en la creencia generalizada de que tales problemas son *esencialmente intratables* desde el punto de vista computacional; esto significa que no son susceptibles al uso de algoritmos de solución eficientes; y también que dado que ningún algoritmo resuelve correctamente un problema NP-completo, entonces se requerirá, en el peor de los casos, una cantidad de tiempo *exponencial* para resolverlo, y por lo tanto será impráctico para todos los problemas, salvo para instancias muy pequeñas.

Entonces, un investigador que enfrenta un problema nuevo de optimización combinatoria y no puede encontrar un algoritmo eficiente para éste, tiene la opción de tratar de probar que el problema que combate es NP-completo. Por supuesto, en la mayoría de los casos esto no será tan excitante como encontrar un algoritmo eficiente. Sin embargo, esta alternativa tiene ventajas definidas y seguras. Primero, le ahorrará al investigador trabajo en esfuerzos futuros inútiles para resolver este problema algorítmicamente. Además, una vez que se sabe que un problema es NP-completo, uno está generalmente dispuesto a establecer metas menos ambiciosas que la de desarrollar un algoritmo que *siempre* encuentre un solución *exacta* y cuyos requerimientos en tiempo *nunca* excedan

un crecimiento polinomial. De esta forma, podríamos desear tomar una de las aproximaciones alternativas posibles.

Por esta razón, los problemas NP-completos son el principal objetivo en la búsqueda de algoritmos eficientes.

Es importante considerar que no todos los problemas NP son difíciles; muchos son sencillos y pueden resolverse por medio de algoritmos cuyas complejidades *temporales* son polinomiales.

La noción de los problemas NP-completos y nuestra sospecha que P ≠ NP sugiere la topografía de la clase NP mostrada en la figura 20, la cual es una sobre-simplificación del conocimiento actual en el campo.

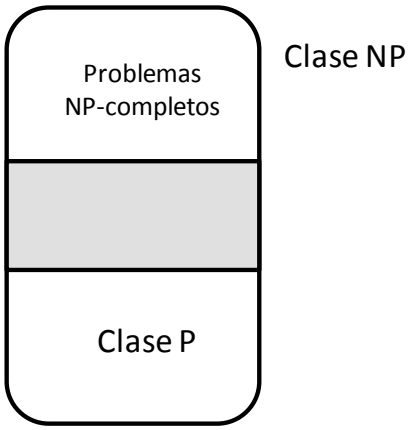


Figura 20. Clases P y NP

En la figura se asume que la complejidad aumenta mientras subimos. Por supuesto, no es para nada obvio en este punto que los problemas NP-completos existan; pero existen.

El conjunto de problemas NP-completos es muy grande y crece constantemente. Conviene también subrayar que la teoría de los problemas NP-completos no sostiene que éstos nunca puedan resolverse con algoritmos polinomiales; sólo afirma que todos ellos pueden resolverse en un número polinomial de pasos.

5.4 Algunos problemas NP-completos

5.4.1. Teorema de Cook

El teorema de Cook, debido al científico en computación Stephen Arthur Cook, es de importancia central para la complejidad computacional, de hecho, es uno de los resultados más importantes de la teoría de complejidad computacional. El teorema de Cook, desarrollado en 1971, fue la primera demostración de que SAT es un problema NP-completo, esto es, cualquier problema en NP se puede reducir en tiempo polinomial mediante una máquina de Turing determinística al problema de determinar si una fórmula booleana es satisfactible. Una consecuencia importante de este teorema es que si existe un algoritmo determinista de tiempo polinomial para resolver satisfactibilidad Booleana, entonces existe un algoritmo determinista de tiempo polinomial para resolver *todos* los problemas en NP. Decisivamente, el mismo razonamiento aplica para cualquier problema NP-completo.

A la pregunta de si tal algoritmo existe se le llama problema P vs. NP , que es el problema más famoso de las ciencias de la computación, el que pregunta si cada problema de optimización cuyas respuestas pueden verificarse eficientemente como correctas u óptimas, se pueden resolver con un algoritmo eficiente. Dada la abundancia de tales problemas de optimización en la vida diaria, una respuesta positiva a la pregunta P vs. NP, probablemente podría tener profundas consecuencias prácticas y hasta filosóficas.

5.4.2. Los 21 problemas completos de Karp

En 1972, Ricahrd Karp tomó la idea del teorema de Cook y extendió el conocimiento con su destacado artículo, "Reducibility Among Combinatorial Problems", en el cual demostró que 21 diversos problemas combinatorios y de teoría de grafos, cada uno de ellos particularmente perversos por su intratabilidad computacional, son NP-completo. Así, Karp introdujo la metodología estándar actual para probar que un problema es NP-completo. El resultado de su trabajo puede verse en la figura 21. Un planteamiento importante de dicho artículo es la muestra de que las diferentes apariencias de estos problemas son esencialmente, el mismo problema disfrazado.

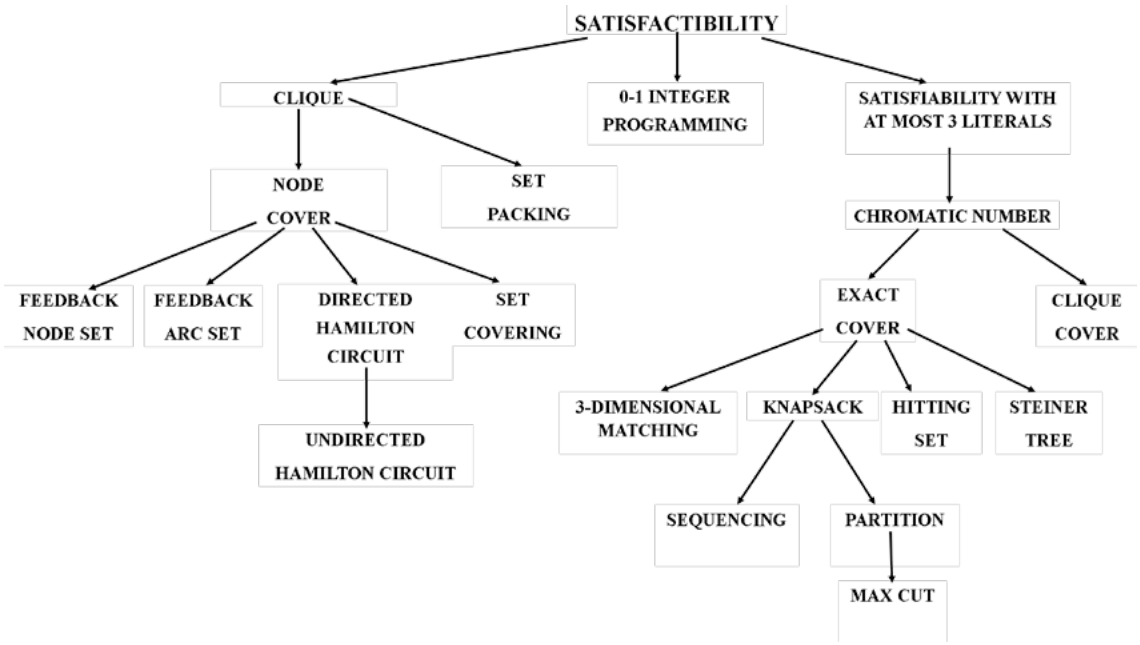


Figura 21. Los 21 problemas completos de Karp. Obsérvese la dirección de reducción que muestra la forma de la figura, por ejemplo, se demostró que KNAPSACK es NP-completo mediante la reducción de EXACT COVER a KPNAPSACK.

(Fuente: Richard M. Karp (1972). "Reducibility Among Combinatorial Problems". In R. E. Miller and J. W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum. pp. 85–103.)

Apéndice

Búsqueda binaria o dicotómica

Cuestiones generales

La búsqueda de un elemento dentro de un arreglo (*array*) es una de las operaciones más importantes en el procesamiento de la información, y permite la recuperación de datos previamente almacenados. Todos los algoritmos de búsqueda tienen dos finalidades:

i. Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.
ii. Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

Como principales algoritmos de búsqueda en arreglos tenemos la búsqueda secuencial, la binaria y la búsqueda utilizando tablas de *hash*.

Para utilizar el algoritmo de búsqueda binaria, el arreglo debe estar ordenado.

La búsqueda binaria consiste en dividir el arreglo por su elemento medio en dos subarreglos más pequeños, y comparar el elemento buscado con el del centro.

Si coinciden, la búsqueda se termina. Si el elemento es menor, debe estar (si es que está en el arreglo) en el primer subarreglo, y si es mayor está en el segundo.

Por ejemplo, para buscar el elemento 3 en el arreglo {1,2,3,4,5,6,7,8,9} se realizarían los siguientes pasos:

Se toma el elemento central y se divide el arreglo en dos:
{1,2,3,4} – 5 – {6,7,8,9}

Como el elemento buscado, 3, es menor que el central, 5, debe estar en el primer subarreglo:
{1,2,3,4}

Se vuelve a dividir el arreglo en dos:
{1} – 2 – {3,4}

Como el elemento buscado, 3, es mayor que el central, 2, debe estar en el segundo subarreglo: {3,4}

Se vuelve a dividir en dos:
{} – 3 – {4}

Como el elemento buscado coincide con el central, lo hemos encontrado.

Si al final de la búsqueda todavía no hemos encontrado al elemento, y el subarreglo a dividir está vacio {}, concluimos que el elemento no se encuentra en el arreglo. La implementación de este algoritmo sería:

```
int desde,hasta,medio,elemento,posicion; // desde y
     // hasta indican los límites del arreglo que se está mirando.
int array[N];

// Dar valor a elemento.

for(desde=0,hasta=N-1;desde<=hasta;)
{
   if(desde==hasta) // si el arreglo sólo tiene un elemento:
   {
      if(array[desde]==elemento) // si es la solución:
```

```
            posicion=desde; // darle el valor.
         else // si no es el valor:
            posicion=-1; // no está en el arreglo.
         break; // Salir del bucle.
       }
      medio=(desde+hasta)/2; // Divide el arreglo en dos.
      if(array[medio]==elemento) // Si coincide con el central:
      {
         posicion=medio; // ese es la solución
         break; // y sale del bucle.
       }
      else if(array[medio]>elemento) // si es menor:
         hasta=medio-1; // elige el arreglo izquierda.
      else // y si es mayor:
         desde=medio+1; // elige el arreglo de la derecha.
   }
```

En general, este método realiza $\log_2(N+1)$ comparaciones antes de encontrar el elemento, o antes de descubrir que no está. Este número es muy inferior que el necesario para la búsqueda lineal (o secuencial) para casos grandes.

Búsqueda secuencial

Consiste en recorrer y examinar cada uno de los elementos del arreglo hasta encontrar el o los elementos buscados, o hasta que se han mirado todos los elementos del arreglo.

```
for(i=j=0;i<N;i++)
   if(array[i]==elemento)
   {
      solucion[j]=i;
      j++;
    }
```

Este algoritmo se puede optimizar cuando el arreglo está ordenado, en cuyo caso la condición de salida cambiaría a:

```
for(i=j=0;array[i]<=elemento;i++)
```

o cuando sólo interesa conocer la primera ocurrencia del elemento en el arreglo:

```
for(i=0;i<N;i++)
   if(array[i]==elemento)
      break;
```

En este último caso, cuando sólo interesa la primera posición, se puede utilizar un centinela, esto es, dar a la posición siguiente al último elemento de arreglo el valor del elemento, para estar seguro de que se encuentra el elemento, y no tener que comprobar a cada paso si seguimos buscando dentro de los límites del arreglo:

```
array[N]=elemento;
for(i=0;;i++)
   if(array[i]==elemento)
      break;
```

Si al acabar el bucle, i vale N es que no se encontraba el elemento. El número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de $(N+1)/2$.

Bibliografía

BAZARAA, M. S., John Jarvis and Hanif Sherali (1990). *Linear Programming and Network Flows.* 2nd ed., John Willey & Sons, New York.

COOK, Stephen (1971). The Complexity of Theorem Proving Procedures in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. ACM, New York, pp. 151–158.

CORMEN, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2001). *Introduction to Algorithms*. McGraw-Hill Book Co., Boston, MA.

GAREY M. and D. Johnson (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York.

KARP, Richard M. (1972). Reducibility Among Combinatorial Problems in R. E. Miller and J. W. Thatcher (editors). *Complexity of Computer Computations*. Plenum, New York, pp. 85–103.

LEE, R.C.T., S.S. Tseng, R.C. Chang y Y.T. Tsai (2007). *Introducción al diseño y análisis de algoritmos.* McGraw Hill Interamericana, México.

PAPADIMITRIOU, Christos H. (1994). *Computational Complexity.* Addison Wesley Longman, Boston, MA.

PAPADIMITRIOU, Christos H. and Kenneth Steiglitz (1998). *Combinatorial Optimization. Algoritms and Complexity.* Dover Publications Inc., New York.

WEGENER, Ingo (2005). *Complexity Theory: Exploring the Limits of Efficient Algoritms*. Springer, Heidelberg.

ZUCKERMAN, David (1996). "On Unapproximable Versions of NP-Complete Problems" in *SIAM Journal on Computing*. Vol. 25, Issue 6, pp. 1293–1304.
doi:10.1137/S0097539794266407. http://citeseer.ist.psu.edu/192662.html.

APUNTES DE COMPLEJIDAD COMPUTACIONAL
se publicó digitalmente en el repositorio
de la Facultad de Ingeniería en marzo
de 2023. Primera edición electrónica
de un ejemplar (29 MB) en formato PDF.

El cuidado de la edición y diseño estuvieron a cargo
de la Unidad de Apoyo Editorial de la Facultad de
Ingeniería. Las familias tipográficas utilizadas fueron
Minion Pro y Courier New con
sus respectivas variantes.