8 INTRATABILIDAD

En capítulos anteriores se han estudiado herramientas para resolver problemas y calcular la complejidad de las diferentes soluciones encontradas. En particular, se estudió el problema de búsqueda en grafos y se plantearon alternativas para atacarlo, sobre todo en los casos en que encontrar la mejor solución implicaba una búsqueda exhaustiva de la misma sobre un espacio de tamaño exponencial, lo cual resulta impráctico.

Desde un punto de vista práctico, se suele calificar las soluciones de complejidad exponencial como "malas" y las soluciones de complejidad polinomial como "buenas". Más aun, se llega a afirmar que "si no se tiene una solución buena para un problema dado, no se tiene todavía una solución".

Sin embargo, resulta que los problemas para los que no se tienen soluciones "buenas" no son casos patológicos, ni mucho menos difíciles de encontrar, sino más bien problemas naturales y que se presentan con relativa frecuencia. Por tal razón es importante conocer ese tipo de problemas, tanto desde el punto de vista teórico, como sus implicaciones desde el punto de vista práctico.

En este capítulo se pretende establecer un conocimiento formal de problemas típicos para los cuales no se han logrado encontrar soluciones "buenas", de manera que se tome conciencia de su dificultad, así como de diferentes maneras de buscar soluciones.

8.1 CONCEPTOS BÁSICOS

Antes de emprender el estudio de problemas desde el punto de vista de su posible complejidad intrínseca, es necesario precisar con más detalle algunas de las nociones ya introducidas en capítulos anteriores, como son las de problema, solución y modelos de máquinas.

Las precisiones adicionales a las definiciones originales tienen que ver con la forma en que se plantean los problemas, se expresan sus soluciones, o en que algún modelo de máquina manipula sus datos. No obstante, aunque el suponerlas se puede pensar que se tiene una visión restringida de los conceptos involucrados, no parece difícil traducir a sus términos cualquier otra forma concreta de modelar los problemas a la que aquí se plantea, dentro de condiciones de análisis razonables. Así mismo, los modelos de máquinas no son esencialmente restringidos al pensar que los datos que se manipulan son de la forma específica que se desea.

8.1.1 Problemas

La noción de problema, introducida en el capítulo 1, se modifica ligeramente, de manera que ahora es una relación total sobre cadenas binarias:

Un *problema* X se entenderá como un conjunto de parejas ordenadas $\langle p,r \rangle$, donde p es una *pregunta* y r una *respuesta* . P_X es el *conjunto de preguntas* de X, i.e., $P_X = \{p \mid \langle p,r \rangle \in X\}$.

Llamando $\mathbf{B} = \{0,1\}$, se supondrá que $P_X = \mathbf{B}^*$, y que $r \in \mathbf{B}^*$. Esto quiere decir que toda cadena finita de 0's y 1's representa una pregunta posible, así como que todas las respuestas se codifican también con cadenas binarias.

Cualquier lenguaje formal puede traducirse en un lenguaje de cadenas binarias, es decir, un subconjunto de $\mathbf{B}^*$. En particular, se pueden traducir lo valores booleanos, los números naturales, etc. De hecho, se seguirá llamando *lenguaje* a cualquier subconjunto de $\mathbf{B}^*$. Se usarán las siguientes convenciones de traducción:

bool : false ≅ 0 (no), true ≅ 1 (sí)

nat : notación binaria, a menos que se diga otra cosa.

En el capítulo 1 se clasificaban los problemas de acuerdo al carácter de las preguntas y las respuestas:

- *funcional* : para cada $p \in \mathbf{B}^*$ existe un único r tal que $\langle p,r \rangle \in X$
- *de decisión* : las únicas respuestas son 1 ó 0 (sí o no)
- *de conteo* : las respuestas son números naturales
- etc.

Un algoritmo A *soluciona* un problema X si, para cualquier $p \in P_X$, si A recibe como entrada p, produce como resultado un r, tal que $\langle p,r \rangle \in X$.

En ocasiones es suficiente contemplar soluciones aproximadas para los problemas. Para esto es necesario definir una noción de vecindad o aproximación en el espacio de los posibles resultados, digamos, $d: \mathbf{B}^* \times \mathbf{B}^* \to \mathbf{R}$. Para un $\varepsilon > 0$, el algoritmo A ε–*soluciona* el problema X (i.e., lo soluciona aproximadamente, con precisión ε) si, para cualquier $p \in P_X$, cuando A recibe como entrada p, produce como resultado un r', tal que existe un r para el que $<p,r> \in X$, con $d(r,r') \leq \varepsilon$.

Los problemas de decisión son particularmente importantes; la mayoría de las clases de complejidad que se consideran se refieren a problemas de decisión. Cuando se tiene un problema de otra clase, siempre se puede pensar en un problema de decisión asociado, correspondiente al hecho de establecer si una pregunta de problema original tiene una solución[1]. Nótese que el problema original es, por lo menos, tan difícil como el problema de decisión asociado. Por otra parte,

---

[1] Con la definición actual, *toda* pregunta tiene una respuesta, puesto que un problema es una relación total sobre $\mathbf{B}^*$. Sin embargo, es claro que algunas de las preguntas pueden representar planteos absurdos del problema, para los que la respuesta debe entenderse como una codificación del hecho de que no haya solución adecuada.

ocasionalmente, el saber solucionar un problema de decisión puede llevar a saber solucionar ciertos problemas más generales.

Hay una correspondencia natural entre los lenguajes los problemas de decisión, que queda establecida en la siguiente definición:

Definición A

i. Para un lenguaje L, el *problema de decisión correspondiente* es

$$X_L = \{\langle p, sí\rangle \mid p \in L\} \cup \{\langle p, no\rangle \mid p \notin L\}$$

ii. Para un problema de decisión X, el *lenguaje correspondiente* es

$$L(X) = \{p \in \mathbf{B}^* \mid \langle p, sí\rangle \in X\}$$

iii. Para un lenguaje L, el *lenguaje complementario correspondiente* es

$$L^c = \mathbf{B}^* \setminus L$$

▯

Con la presente definición de problema, parece natural definir |p|, el tamaño de una pregunta p, como el número de símbolos que la conforman. Los modelos de máquinas considerados en 1.4 pueden restringirse, sin pérdida de generalidad, a manipular cadenas binarias.

8.1.2 Ejemplos de problemas

Considérense los siguientes problemas:

- *Verificación de primalidad* (VP): Dado un entero positivo n, decidir si n es primo o no.
- *Factorización* (FAC): Dado un entero positivo n, encontrar uno de sus divisores (o todos ellos).
- *Problema del agente viajero* (AV): Dadas n ciudades en un plano y un número natural d, decidir si existe un *tour* (camino cerrado que no repite nodos) que visite las n ciudades, cuya longitud sea menor o igual a d. Las distancias se miden en números enteros.
- *Coloración de grafos* (CG): Dados un grafo G y un entero positivo k, decidir si G es k-*coloreable*, i.e, si se puede colorear con k o menos colores.
- *Conjuntos independientes* (CI): Dados un grafo G y un entero k, decidir si G tiene un subconjunto independiente de tamaño k.
- *Empaquetamiento* (inglés: *bin packing*) (BP): Dados un conjunto finito S de enteros positivos, un entero k (capacidad de una caja) y un entero n (número de cajas), decidir si se pueden empacar los enteros de S en a lo sumo n cajas, cada una de capacidad k, de manera que la suma de los enteros empacados en cada caja no sobrepase la capacidad de la misma.
- *Clique* (CQ): Dado un grafo G, con n vértices, y un número natural k, decidir si existe un subgrafo completo de G, con k vértices.
- *Partición* (PT): Dado un conjunto finito A de enteros positivos, determinar si existe un subconjunto A' de A, tal que: $\langle +: i \in A' : i\rangle = \langle +: i \in A \setminus A' : i\rangle$.

- *Morral* (inglés: *knapsack*) (KS) Dado un conjunto finito U, un "tamaño" $s(u) \in \mathbf{nat}^+$ y un "valor" $v(u) \in \mathbf{nat}^+$, para cada $u \in U$, una cota de tamaño $B \in \mathbf{nat}^+$ y un objetivo de valor $K \in \mathbf{nat}^+$, determinar si hay un subconjunto U' de U, para el que $\langle +: u \in U' : s(u) \rangle \leq B$ y también $\langle +: u \in U' : v(u) \rangle \geq K$.

Estos problemas tienen una característica en común: No se han encontrado aún algoritmos polinomiales para resolverlos, pero tampoco se ha podido probar que no los haya.

Por poseer esa característica, a dichos problemas se les clasifica como "difíciles" o, utilizando un término que se introduce un poco más adelante, NP-completos.

Lo anterior no quiere decir que no existan instancias fáciles de esos problemas. Por ejemplo, si a los datos de CG se añade que G es bipartito, la solución del problema es muy sencilla: si $k \geq 2$, G se puede colorear con k colores. De igual manera sucede con BP, si se sabe que todos los enteros de S son iguales.

¿Qué es lo difícil de esos problemas? Para responder esta pregunta se puede formular otra: ¿Sería posible diseñar un algoritmo para resolver uno de esos problemas -por ejemplo, CG- de tal manera que se pueda expedir una garantía de funcionamiento como: "el programador garantiza que, si un grafo G de n vértices y un entero positivo k son dados como datos, el programa responderá correctamente si el grafo G puede ser coloreado con k colores o menos y, además, lo hará en menos de $3476n^{25}$ ms" ?

En realidad, la dificultad del problema está en la imposibilidad de ofrecer una garantía como la mostrada y no en la obtención de una solución. En otras palabras, es posible dar algoritmos cuya respuesta al problema sea siempre la correcta, pero que el tiempo usado para darla sea exponencial. También se pueden dar algoritmos que dan una respuesta aproximada en tiempo polinomial, cuya confiabilidad no sería total[2].

Es importante aclarar que "tiempo polinomial" significa que la complejidad del algoritmo es $O(p(x))$, donde x es la medida de los datos del problema y $p(x)$ es un polinomio en x, es decir, que su grado es constante con respecto a x. Por ejemplo, si para resolver CG se encuentra un algoritmo de orden $O(n^{346})$, o $O((nk)^{346})$, se dirá que CG se resuelve en tiempo polinomial; pero si el algoritmo es de orden $O(n^k)$, no se puede afirmar lo mismo.

La representación de los datos puede influir en la estimación del tamaño de la entrada. Por ejemplo, para el problema CQ, el grafo G se puede representar con listas de adyacencias, matrices booleanas, etc. La entrada es una cadena binaria que denota la representación en cuestión. Así, si se elige representar los grafos con matrices binarias, la descripción de un grafo de n vértices consta, por lo menos, de n^2 bits. Como la entrada de CQ también incluye el número k, y éste se representa en binario, una pregunta para CQ mide $m = O(n^2 + \log k)$.

---

[2] Por ejemplo, cuando el algoritmo da una respuesta afirmativa, con seguridad G se puede colorear con k colores, pero si la respuesta es negativa no se tiene la certeza de que no se pueda colorear.

Continuando con CQ, si $k>n$ la respuesta es, trivialmente, no. Es decir, el problema es complicado sólo si $k\leq n$. En este caso, es claro que $m = O(n^2)$. El algoritmo intuitivo sugiere probar cada uno de los posibles $\binom{n}{k}$ subconjuntos de k vértices. Cuando $k = \frac{n}{2}$, es fácil ver que hay más de $2^{n/2}$ subconjuntos posibles, de modo que debe efectuarse un número de pruebas del orden de $2^{\sqrt{m}/2}$. En otras palabras, la complejidad temporal de este algoritmo es exponencial en el tamaño de la entrada. Por otra parte, no se conoce un algoritmo que sea esencialmente mejor que el intuitivo, una de las características que distinguen -por ahora- los problemas NP-completos.

Lo sorprendente es que la familia de problemas NP-completos no es solo una colección de problemas "difíciles" para los cuales no se ha encontrado una solución que tome tiempo polinomial y tampoco se ha podido probar que no exista. Además cumple otras dos propiedades:

- Si se descubre un algoritmo rápido (i.e., tiempo polinomial) para resolver un problema NP-completo, entonces se tendrá inmediatamente un algoritmo rápido para solucionar cualquier problema NP-completo.
- Si se prueba que no puede existir un algoritmo rápido para resolver un problema NP-completo, entonces se concluirá que no puede existir un algoritmo rápido para resolver ningún problema NP-completo.

8.1.3 Problemas de optimización y de búsqueda

El estudio de intratabilidad se enfoca, fundamentalmente a problemas de decisión. Sin embargo, es frecuente que interese, además de saber si una pregunta para un problema dado tiene respuesta afirmativa o no, sino precisamente cuál es una respuesta a la pregunta. Por ejemplo, el problema del agente viajero se plantea usualmente de la siguiente forma:

> (AV_O) : Dadas n ciudades en un plano, encontrar la longitud mínima de un *tour* (camino cerrado que no repite nodos) que visite las n ciudades.

De esta forma, AV_O es un problema de *optimización*. En rigor, la teoría de intratabilidad no contempla estos problemas; sin embargo, en este y muchos otros casos, hay una relación sencilla entre las soluciones de un problema de optimización y las de un problema decisión relacionado.

Por ejemplo, si en el planteo de AV_O se define `m = <max u,v: u,v ciudades: dist(u,v)>`, cualquier *tour* deberá medir, a lo sumo, nm. Supóngase un algoritmo `av(G,dist,d)`, que decida AV con el grafo completo `G`, etiquetado con distancias `dist`, y la cota `d`. Entonces, el siguiente algoritmo resuelve el problema AV_O:

```
funct avo (G,dist):nat
{Pre:  m = <max u,v: u,v ciudades: dist(u,v)> }
{Pos: avo = "longitud de un tour mínimo" }
var d: nat
⌈  min, max:= 0,nm;
   do max-min≥1     →    d:= ⌊(max + min)/2⌋;
                         if  av(G,dist,d)  then max:= d
                                           else min:= d+1
                         fi
```

```
    od;
    avo:= d
  ⌋
```

Este algoritmo efectúa una búsqueda binaria en el rango `0..nm`, de manera que tendrá como complejidad

$$T_{AVo}(G,dist) \leq O(\langle \max d:\ 0 \leq d \leq nm:\ T_{av}(G,dist,d)\rangle \log nm)$$

Ahora, la entrada de AV_o consta de n^2 números, que denotan las distancias entre ciudades, más un número adicional que denota el tamaño de la cota `d`. Sea `q` el tamaño de la codificación binaria de las n^2 distancias. Sea `r` el tamaño de una entrada para AV. Puesto que $d \leq nm$ y `m` es la distancia más grande entre ciudades:

$$r \leq q + \log d \leq q + \log n + \log m \leq 3q$$

La búsqueda binaria utiliza, a lo sumo, $O(2q)$ operaciones. Entonces:

$$T_{AVo}(q) \leq O(T_{av}(3q)\ 2q).$$

En otras palabras, si AV es soluble en tiempo polinomial, también lo será AV_o.

Por otra parte, cuando se sabe resolver problemas de optimización, es usual preguntar, adicionalmente, cómo se consigue el valor óptimo calculado. Para el caso del agente viajero, se puede plantear el problema:

> (AV_b) : Dadas `n` ciudades en un plano, encontrar un *tour* (camino cerrado que no repite nodos) que visite las `n` ciudades, de longitud mínima.

Este es un problema de *búsqueda*. Supóngase algoritmos `av(G,dist,d)` y `avo(G,dist)` para resolver los problemas de decisión y optimización correspondientes. El siguiente algoritmo resuelve el problema AV_b. Sin pérdida de generalidad, el conjunto de las ciudades se identifica con `1..n`; la variable `tourmin` es de tipo **`set of`** `1..n`.

```
proc  avb (G,dist; var tourmin)
{Pos: "tourmin es tour de costo mínimo" }
var  dist1: 1..n x 1..n → nat;
       t0: nat
⌈  t0:= avo(G,dist);
   tourmin:= ∅;
   dist1:= dist;
   for ⟨i,j⟩∈1..n x 1..n → i≠j
             →   dist1(i,j):= ∞;
                 if   ¬av(G,dist1,t0)
                      then tourmin:= tourmin ∪ {<i,j>};
                           dist1(i,j):= dist(i,j)
                 fi
```

```
    rof
  ⌋
```

Es fácil constatar que avb tiene complejidad polinomial, si av también es polinomial. Por lo tanto, AV_b es polinomial si AV también lo es.

8.2 CLASES DE COMPLEJIDAD

En 1.3 se introdujeron las siguientes definiciones:

- Un problema X tiene orden de R-complejidad $O(f)$, si se puede mostrar un algoritmo A que lo resuelve, para el que $R_A = O(f)$. Es decir, se establece una cota superior para la complejidad de los algoritmos que solucionan el problema.
- Un problema X tiene orden de R-complejidad $\Omega(f)$, si para cualquier algoritmo A que lo solucione, se tiene que $f = O(R_A)$.

En 1.3.3 se habló de clases de complejidad para algoritmos. Estas nociones se pueden extender para definir clases de complejidad de problemas. Así, para un problema X, cuyo orden de complejidad es $O(f)$, se dirá que éste es

- *Constante* : si $f = O(1)$
- *Logarítmica* : si $f = O(\log n)$
- *Polilogarítmica* : si $f = O(\log^k n)$, para una constante $k>0$
- *Lineal* : si $f = O(n)$
- *Polinomial* : si existe una constante $k>0$, tal que $f = O(n^k)$
- *Exponencial* : si existe una constante $k>0$, tal que $f = O(2^{n^k})$.

Los calificativos anteriores se pueden anteceder de la frase "*por lo menos* ", si se remplaza $O(\ldots)$ por $\Omega(\ldots)$ en las definiciones.

Por ejemplo, el problema de averiguar el máximo de un conjunto de n números es $\Omega(n)$, ya que cualquier algoritmo que se conciba para resolverlo deberá, al menos, efectuar $n-1$ comparaciones. En otras palabras, es un problema por lo menos lineal.

Estas nociones dependen, además, del modelo de máquina que se utilice para ejecutar los algoritmos. Los ejemplos más importantes se refieren a los casos en que el recurso considerado es el tiempo y, en segundo término, el espacio. El modelo de máquina que se supone es usualmente determinístico y secuencial (v.gr., MTD, RAM), aunque, precisamente la noción de problema NP se puede entender también con respecto a máquinas secuenciales no determinísticas (v.gr., MTND).

En próximas secciones se definen algunas clases de problemas, estableciendo con precisión el modelo de máquina y las cotas esperadas para los algoritmos que los resuelvan.

8.2.1 La clase P

A la clase P pertenecen los problemas de decisión que tienen una solución sobre una MTD, cuyo tiempo de ejecución es de orden polinomial.

Informalmente, se acostumbra denominar con P a la clase de todos los problemas de búsqueda solubles en tiempo polinomial sobre una MTD. A falta de una denominación formal estándar para esta clase de problemas, se utilizará PB para denotarla.

Muchos problemas importantes -o corrientes- son polinomiales. Curiosamente, hay problemas polinomiales que parecen formas "duales" de problemas difíciles, de manera que puede resultar relativamente fácil pensar que un problema es más difícil de lo que en realidad es.

A continuación, algunos problemas pertenecientes a P:

- *Recubrimiento de arcos* (RA): Dado un grafo $G(V,E)$ y un número entero $k \geq 0$, decidir si hay un conjunto de arcos $E' \subseteq E$, con $|E'| < k$, tal que todo vértice es comienzo o final de algún arco de E'.
- *Camino de Euler* (CE): Dado un grafo $G(V,E)$, con $|E|=m$, decidir si hay un camino en G de longitud m, que no repite arcos.
- *Ecuaciones diofantinas lineales* (EDL): Dados tres números naturales a, b, c, decidir si existen dos números naturales x, y, tales que $ax+by = c$.
- *Partición unaria* (PU): Dado un conjunto A de n números enteros, escritos en notación unaria, decidir si hay un subconjunto A' de A, tal que: $\langle +: a \in A' : a \rangle = \langle +: a \in A \backslash A' : a \rangle$.
- *Programación lineal* (PL): Dada una matriz entera $A[1..m,1..n]$, un vector $b[1..m]$, un vector $c[1..n]$ y un entero k, decidir si hay un vector de números racionales $x[1..n]$, tal que[3] $Ax \leq b,\ cx^T \geq k$.

El problema PL es, históricamente, quizás el más importante de los problemas clasificados como polinomiales. Durante mucho tiempo se pensó que era un problema no polinomial que, curiosamente tiene en el algoritmo simplex (el método de solución más utilizado para PL) una solución exponencial, pero práctica en la mayoría de los casos.

8.2.2 Reducciones

Para mostrar que un problema está en una clase, basta mostrar que hay un algoritmo que satisfaga lo requerido en la definición de la clase (modelo de máquina, cotas de recursos). En cambio, para mostrar que un problema *no* pertenece a una clase, se usan, comúnmente, métodos indirectos. Hay una técnica fundamental que, curiosamente, sirve tanto para llevar a cabo ambos tipos de prueba: la reducción.

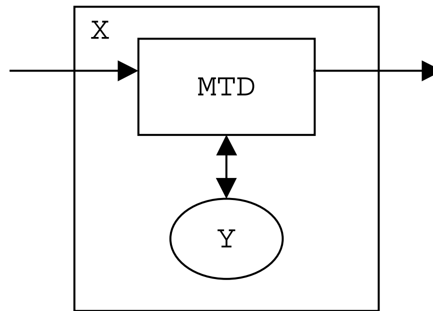
La idea básica tras de una reducción es el uso de subrutinas. En términos de máquinas de Turing, las subrutinas corresponden a oráculos que saben responder preguntas específicas. Supóngase que un problema X se puede resolver en tiempo polinomial en una MTO (máquina de Turing con oráculo) que usa un oráculo que soluciona un problema Y. Si Y es soluble en una MTD en tiempo polinomial, entonces X también debe ser soluble en una MTD en tiempo polinomial.

---

[3] Las comparaciones entre vectores se efectúan por componentes.

Definición A

Si X, Y son problemas, una (*Turing-*) *reducción de* X *a* Y es una MTO que resuelve X, usando un oráculo para Y. En este caso, X es *reducible a* Y (*mediante la reducción*).



▯

Se supone que la consulta al oráculo consume recursos $O(1)$, v.gr., en tiempo y en espacio.

Cuando se trata de problemas de decisión, la definición anterior puede especializarse:

Definición B

Si X, Y son problemas de decisión, una *transformación de* X *a* Y es una función (calculable en una MTD) $f: B^* \to B^*$, tal que

$$\langle x,r\rangle \in X \quad \Leftrightarrow \quad \langle f(x),r\rangle \in Y$$

▯

La definición de transformación corresponde a una reducción en la que el oráculo solo es consultado una vez, y su respuesta debe coincidir con la respuesta del primer problema.

Las reducciones son herramientas útiles para la clasificación de problemas cuando se agrupan en clases que también obedecen cotas para los recursos utilizables. Por ejemplo, RP, la clase de las *reducciones polinomiales* , son las que obedecen cotas de tiempo polinomiales. En términos de la definición, la MTO resuelve X en tiempo polinomial, suponiendo que el oráculo es consultado en $O(1)$. Esto no quiere decir que X sea polinomial, porque para ello se necesita que Y sea solucionable polinomialmente en una MTD.

Para una clase de reducciones R y dos problemas X, Y, la notación $X \leq_R Y$ se usa para significar que hay una reducción en R, de X a Y. Por conformar las reducciones polinomiales quizás la clase de reducciones más importante, se usará la notación $X \leq Y$ para significar que X reduce polinomialmente a Y.

Las siguientes definiciones técnicas son importantes para trabajar con clases de reducciones:

Definición C

Sean C es una clase de problemas y R una clase de reducciones

i. R es C-*compatible* si para cualquier para de problemas X, Y:

$$X \leq_R Y \quad \wedge \quad Y \in C \quad \Rightarrow \quad X \in C$$

ii. R es *transitiva* si para cualesquiera tres problemas X, Y, Z:

$$X \leq_R Y \quad \wedge \quad Y \leq_R Z \quad \Rightarrow \quad X \leq_R Z$$

□

A manera de ejemplo, las reducciones polinomiales son transitivas y, además, P-compatibles.

8.2.3 Completitud

Las reducciones pueden servir para probar que un problema Y no pertenece a una clase C. Para esto, basta mostrar una reducción de una clase de reducciones que sea C-compatible, que muestre a un problema X, para el que se sabe que $X \notin C$, como reducible a Y. Así, si valiera que $Y \in C$, la definición de compatibilidad llevaría a concluir que $X \in C$, lo que sería contradictorio.

La argumentación anterior se vuelve más importante si se conocen ciertos problemas que actúan como representantes de las clases, en el siguiente sentido:

Definición A

Sean X un problema, C una clase de problemas, R una clase de reducciones.

i. X es R-*difícil para* C :⇔ $Y \leq_R X$, para todo $Y \in C$.
ii. X es R-*completo para* C :⇔ X es R-*difícil para* C y $X \in C$.

□

Un problema R-completo para C es un representante típico de los problemas más difíciles de C. Por ejemplo, se puede probar que PL es un problema completo para P bajo reducciones que son calculables usando espacio logarítmico adicional (*reducciones log-space*).

Para una clase de problemas y una clase de reducciones dadas, hay una dificultad significativa en encontrar problemas completos bajo las reducciones. Pero una vez que se encuentra uno, la demostración de que otros también lo son es más sencilla, si además se tiene que el conjunto de reducciones es transitivo:

Lema B

Sean C una clase de problemas, R una clase transitiva de reducciones, X un problema R-completo para C.

$$X \leq_R Y \quad \wedge \quad Y \in C \quad \Rightarrow \quad Y \text{ es R-completo para } C$$

Demostración:

Sea $Z \in C$. Como X es R-completo para C, entonces X es R-difícil para C, de modo que $Z \leq_R X$. Por transitividad, debe valer que $Z \leq_R Y$, i.e., Y es R-difícil para C.

□

8.2.4 La clase NP

A la clase NP pertenecen los problemas de decisión (o bien, los lenguajes reconocibles) que tienen una solución (o bien, un algoritmo de reconocimiento) sobre una MTND, cuyo tiempo de ejecución es de orden polinomial.

Una MTND es igual a una MTD, con la posibilidad suplementaria de tomar más de una decisión -mover una cabeza o escribir algo- a partir de una misma configuración de sus componentes. El número de selecciones factibles es, no obstante, finito. La decisión se toma de manera no-determinística, i.e., arbitrariamente. Si una MTND se enfrenta en dos ocasiones a una misma configuración, la secuencia de operaciones efectuadas puede diferir de una a otra ejecución.

Las MTNDs se utilizan sólo para resolver problemas de decisión, de suerte que la cinta de respuesta puede limitarse a una celda. Si la cinta de respuesta contiene un "sí", debe entenderse que hay una forma de calcular la respuesta "adivinando" las decisiones correctas en las selecciones no-determinísticas. Si la respuesta es "no", significa que no hay ninguna posible computación que termine en una respuesta positiva.

Una computación de una MTND puede visualizarse como un árbol de ramificación finita, cuyos nodos son configuraciones o estados de computación y cuyos arcos representan las decisiones arbitrarias que pueden tomarse en cada configuración. Cuando la computación termina, el árbol es finito y, sin pérdida de generalidad, se puede suponer que todas las hojas tienen la misma altura. Como *tiempo* se considera la altura de este árbol, lo que equivale a estimar el tiempo por el de la posible ejecución más corta, si la respuesta es afirmativa, o el de la más larga, si la respuesta es negativa.

8.2.4.1 Máquinas de Turing de adivinación y verificación (MTAV)

Una máquina de Turing de adivinación y verificación (MTAV) es como una máquina de Turing determinística en cuanto a las cintas y al repertorio de instrucciones que puede ejecutar, pero su control se divide en dos componentes: una componente de adivinación y una de verificación. Ante una entrada dada x, la componente de adivinación actúa como una MTO y "adivina" un dato c, que se denominará un *certificado*; el par $\langle x,c \rangle$ sirve de entrada a la componente de verificación, que actúa como una MTD que resuelve un problema de decisión. Así, la componente de verificación puede terminar con una respuesta afirmativa, o bien con una respuesta negativa, o, eventualmente, no parar.

Es conveniente aclarar el significado de la adivinación que lleva a cabo la primera componente de una MTAV. En realidad, el que se pueda pensar que la verificación del certificado producido no tenga, necesariamente, un resultado afirmativo, da pie para pensar -como en efecto se debe hacer- que es factible producir certificados inútiles. Así, esta componente no es tan MTO como se podría desear, ya que hay la posibilidad de que se equivoque. Esto, por otro lado, puede interpretarse más "humanamente", en el sentido de que la componente de adivinación actúa como una persona podría hacerlo; unas veces con más intuición que otras, aventura maneras de verificar un resultado, que después deben ser verificables en un tiempo razonable.

Una MTAV resuelve un problema `X` si, para cualquier pregunta $x \in P_X$, `<x,sí>`$\in$`X` si y sólo si existe un certificado `c` que, después de ser producido por la componente de adivinación, lleva a que la componente de verificación se detenga con `sí` cuando se le da como entrada `<x,c>`.

Una MTAV resuelve un problema `X` en tiempo `O(f)` si, para cada $x \in P_X$, la componente de verificación tarda `O(f(|x|))` en llevar a cabo su ejecución. Nótese que esto impone una cota sobre el tamaño del certificado, porque el tiempo para examinarlo -junto con la entrada- está acotado. Un caso especial importante se da cuando la MTAV resuelve un problema en orden polinomial; entonces el tamaño del certificado para un `x` está acotado por un polinomio fijo `p(|x|)`.

Lema A

Los lenguajes reconocidos en tiempo polinomial por MTNDs son exactamente los mismos reconocidos en tiempo polinomial por MTAVs.

Demostración:
Ejercicio.

◻

El lema anterior, junto con la definición A de 8.1.1, permite concluir que los problemas de decisión solucionables con MTNDs en tiempo polinomial también lo son con MTAV en tiempo polinomial. Es decir, los problemas de la clase `NP` son exactamente los mismos que se pueden resolver con MTAVs en tiempo polinomial.

En realidad, la noción de solubilidad mediante MTAVs corresponde, básicamente, a la de verificabilidad (determinística) en tiempo polinomial. En lugar de tenerse una única computación posible para cada entrada, hay en realidad muchas diferentes: una para cada posible adivinación.

8.2.4.2 Ejemplos de problemas `NP`

Es usual mostrar que un problema está en la clase `NP` exhibiendo un método de verificación de certificados que trabaje en tiempo polinomial, como la componente de verificación de una MTAV. A continuación se muestran algunos ejemplos: se describe un certificado para cada uno, cuya bondad para mostrar el resultado es fácilmente verificable en tiempo polinomial.

AV: Dadas n ciudades en un plano y un número natural d, decidir si existe un *tour* (camino cerrado que no repite nodos) que visite las n ciudades, cuya longitud sea menor o igual a d. Las distancias se miden en números enteros.
El certificado es un *tour* para el que se debe verificar si tiene longitud menor o igual a d.

CG: Dados un grafo $G(V,E)$ y un entero positivo k, decidir si G es k-*coloreable*, i.e, si se puede colorear con k o menos colores.
El certificado es una asignación $c:V \rightarrow \{1,2,_,k\}$, para la que se debe verificar si es una coloración (i.e., cada color corresponde a un número en $\{1,2,\ldots,k\}$; no debe haber vértices vecinos del mismo color).

CQ: Dado un grafo G, con n vértices, y un número natural k, decidir si existe un subgrafo completo de G, con k vértices.
El certificado es un conjunto de k vértices, para el que se debe verificar que corresponde a un grafo completo.

BP: Dados un conjunto finito S de enteros positivos, un entero k (capacidad de una caja) y un entero n (número de cajas), decidir si se pueden empacar los enteros de S en a lo sumo n cajas, cada una de capacidad k, de manera que la suma de los enteros empacados en cada caja no sobrepase la capacidad de la misma.
El certificado es una partición de los enteros de S en, a lo sumo n conjuntos. Se debe verificar si cada conjunto suma menos de k.

El siguiente problema reviste especial importancia dentro de los problemas NP. Antes de enunciarlo, conviene introducir algo de terminología:

Dado un conjunto de n símbolos sentenciales (i.e., símbolos que representan proposiciones lógicas) $\{u_1, \ldots, u_n\}$, el conjunto de los literales que se pueden construir a partir de ellos es $L = \{u_1, \neg u_1, \ldots, u_n, \neg u_n\}$. Una *cláusula* es un subconjunto de L, y su interpretación en lógica corresponde a la disyunción de sus elementos. Un conjunto de cláusulas se interpreta como la conjunción de las cláusulas que lo componen. Un conjunto de cláusulas es satisfacible si existe una valuación o asignación de verdad para los símbolos sentenciales que hace que la fórmula lógica correspondiente sea verdadera.

SAT (*Satisfacibilidad de cláusulas*): Dado un conjunto de m cláusulas, construidas a partir de n símbolos sentenciales, decidir si es satisfacible.

El certificado es una asignación de verdad para los símbolos sentenciales. La verificación es, trivialmente, polinomial. Así, SAT$\in NP$

Para $k\geq 1$, el siguiente problema es una particularización de SAT:

k-SAT (*Satisfacibilidad de cláusulas*): Dado un conjunto de m cláusulas, cada una hasta de k literales, construidas a partir de n símbolos sentenciales, decidir si es satisfacible.

8.2.4.3 NP-completitud

Un problema X es NP-*completo* si pertenece a la clase NP y cualquier problema Y∈NP es reducible a X. En otras palabras, si llamamos RP a la clase de las reducciones polinomiales (que son, además, NP-compatibes), un problema es NP-completo si es RP-completo para NP.

Un problema NP-completo es un problema difícil por excelencia, dentro de los de su clase, puesto que todos los demás se pueden refrasear en él, de manera polinomial. Así, los problemas en NP se pueden resolver polinomialmente -en forma determinística- si y sólo si existe un algoritmo polinomial -determinístico- para resolver algún problema NP-completo.

El siguiente resultado es quizás la herramienta más importante para mostrar que un problema es NP-completo:

Lema A

Sean X,Y problemas. Entonces:

$$X \text{ es NP-completo} \;\wedge\; Y \in NP \;\wedge\; X \leq Y \;\Rightarrow\; Y \text{ es NP-completo}$$

□

Naturalmente, para poder utilizar este lema, es necesario contar con algún problema NP-completo. El Teorema de Cook (cf. 8.1.3) proporciona como resultado un problema NP-completo, con lo cual se cuenta con una piedra angular para el desarrollo posterior de la teoría.

8.2.5 La relación entre P y NP

Trivialmente, $P \subseteq NP$, porque las MTD son casos especiales de MTND. Por el momento es un problema abierto el determinar si $NP \subseteq P$, aunque la evidencia experimental parece indicar que esto es improbable.

De hecho, el mejor resultado general que se tiene, para comparar las dos clases de problemas es el siguiente:

Teorema A

Si $X \in NP$, entonces existe un polinomio p tal, que X puede ser resuelto determinísticamente en tiempo $O(2^{p(n)})$.

Demostración:

Sea M una MTAV que resuelve X en, a lo sumo, tiempo polinomial $q(n)$. Para una entrada x, con $|x|=n$, que tenga como resultado sí, debe existir un certificado c, con $|c| \leq q(n)$. De este modo, la componente de verificación puede llegar a la respuesta, para la entrada <x,c>, utilizando no más de $q(n)$ pasos. El número de posibles certificados es, a lo sumo, $2^{q(n)}$. Una MTD que efectúe la

verificación para cada uno de los posibles certificados, consumirá, en el peor caso -ensayar todos los certificados- $q(n)2^{q(n)}$ pasos. Para un polinomio apropiado p, es claro que $q(n)2^{q(n)} = O(2^{p(n)})$.

▯

Si el teorema anterior es lo mejor que se puede esperar, los problemas NP-completos son exponenciales, cuando se tratan en máquinas determinísticas.

8.3 PROBLEMAS NP-COMPLETOS

En 1971, S.A. Cook mostró que SAT es NP-completo. Este fue el primer resultado que proporcionó un problema NP-completo y fue de gran utilidad, mediante el Lema 8.2.4.3.A, para encontrar muchos otros problemas NP-completos.

Teorema A (Cook, 1971)

SAT es NP-completo

Demostración (esbozo):

Ya se vio que SAT está en NP. Para probar que SAT es NP-difícil, la demostración debe ser a bajo nivel, es decir, apoyarse directamente en las definiciones de máquinas de Turing y mostrar las reducciones polinomiales correspondientes.

Sea L un lenguaje reconocible por una MTAV M_L, en tiempo polinomial $p(n)$. Así, la componente de adivinación de M_L produce un certificado $c(x)$ para cada pregunta x, con $|x|=n$, y la componente de verificación decide determinísticamente la pregunta $<x,c(x)>$ en tiempo inferior o igual a $p(n)$.

Se pretende asociar, a cada $x \in B^*$, un conjunto de cláusulas $f(x)$, que se construya en tiempo polinomial, tal que $f(x)$ sea satisfacible si y sólo si $x \in L$. Como fórmula de la lógica, $f(x)$ describe el funcionamiento de la máquina M_L, v.gr., "en cada momento de la computación, M_L está en exactamente un estado", "en cada momento de la computación, hay exactamente un carácter del alfabeto en la cabeza lectora", etc. En especial, una de las afirmaciones contenidas en $f(x)$ es: "en la etapa $p(|x|)$, M_L se encuentra en un estado de aceptación".

Así, si se logra probar que $f(x)$ es satisfacible, querrá decir que $x \in L$, y que hay una manera de mostrarlo en tiempo $p(|x|)$. La afirmación recíproca es, así mismo, verdadera.

Una prueba detallada se encuentra en [Gar79].

▯

Una vez que se dispone de un problema NP-completo, el Lema 8.2.4.3.A es la herramienta más útil para encontrar otros problemas NP-completos. Ya en 1971, Cook mostró el siguiente resultado:

Teorema B

3-SAT es NP-completo

Demostración:

i. 3-SAT es una variante de SAT. Como éste está en NP, también lo está 3-SAT.

ii. Se mostrará que SAT ≤ 3-SAT.
Para esto, sea

$$\varphi = x_1 \lor x_2 \lor \ldots \lor x_n$$

una cláusula de n literales, n>3. Considérese la siguiente fórmula, cuyas cláusulas tienen, a lo sumo, tres literales:

$$\begin{aligned} \alpha \quad = \quad (x_1 \lor \neg y_1) \quad & \land \quad (y_1 \lor x_2 \lor \neg y_2) \\ & \land \quad (y_2 \lor x_3 \lor \neg y_3) \\ & \ldots \\ & \land \quad (y_{n-1} \lor x_n \lor \neg y_n) \quad \land\ y_n \end{aligned}$$

Ahora, dada una valuación

$$a: \{x_1, \ldots, x_n\} \rightarrow \mathbf{B}$$

sea i_0 el índice definido por[4]:

$$i_0 = \langle \min: 1 \le i \le n \land a(x_i)=1 : i \rangle$$

Defínase la valuación

$$\hat{a}: \{x_1, \ldots, x_n, y_1, \ldots, y_n\} \rightarrow \mathbf{B}$$

de la siguiente forma:

$$\begin{aligned} \hat{a}(x_i) &= a(x_i) && \text{, para } 1 \le i \le n \\ \hat{a}(y_i) &= 0 && \text{, para } i \le 1 < i_0 \\ \hat{a}(y_i) &= 1 && \text{, para } i_0 \le i \le n. \end{aligned}$$

Entonces, es fácil comprobar que

$$a(\varphi) = 1 \quad \Leftrightarrow \quad \hat{a}(\alpha) = 1$$

En otras palabras, cada una de las cláusulas de longitud n>3 de una fórmula pueden refrasearse en una fórmula en forma clausal, cuyas cláusulas tiene, a lo sumo, tres literales. La transformación es, evidentemente, polinomial.

▯

---

[4] Obsérvese que la definición incluye la posibilidad de que i_0 sea ∞, si es el caso que el rango de la minimización indicada sea vacío.

El anterior teorema sorprende, porque *a priori* se podría creer que las preguntas para 3-SAT son esencialmente más sencillas que las de SAT. En realidad, el resultado afirma que no se gana mucho en restringirse a usar sólo cláusulas con, a lo sumo, tres literales. El ejemplo en cuestión ilustra el hecho de que la situación es más complicada de lo que en principio se pudiera pensar: no es difícil mostrar que 1-SAT y 2-SAT están en P.

El resultado puede utilizarse ahora, para encontrar otros problemas NP-completos. En general, entre más problemas NP-completos se conozcan, se tendrán más posibilidades para reducir los candidatos a NP-completitud a un problema NP-completo.

Teorema C

CQ es NP-completo

Demostración:

i. CQ es NP.

ii. Se mostrará que 3-SAT $\le$ CQ.
Sea $\alpha = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ una fórmula en forma clausal, con cláusulas de, a lo sumo, tres literales. Para tener una notación genérica, se supondrá que cada C_i es de la forma

$$C_i = x_{i1}^{\beta i1} \vee x_{i2}^{\beta i2} \vee x_{i3}^{\beta i3}$$

con las convenciones:

$$\beta_{ih} \in \mathbf{B}, \qquad x^1 = x, \qquad x^0 = \neg x$$

Considérese el grafo $G(V,\rightarrow)$, definido así:

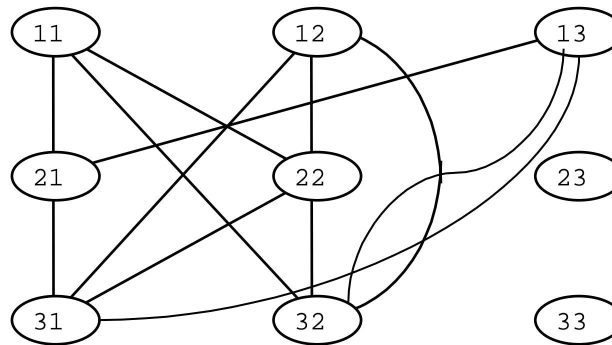
$$V = 1..k \times 1..3$$

$$\langle i,h\rangle \rightarrow \langle j,m\rangle \quad :\Leftrightarrow \quad i \neq j \wedge (x_{ih} \neq x_{jm} \vee \beta_{ih} = \beta_{jm})$$

Los nodos de G son los literales de α. Entre dos de estos literales hay un arco si y sólo si pertenecen a diferentes cláusulas y no corresponden a la misma variable con signos diferentes. Por ejemplo, para

$$\alpha = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2)$$

el grafo correspondiente es (arcos bidireccionales):



Entonces, es fácil comprobar que:

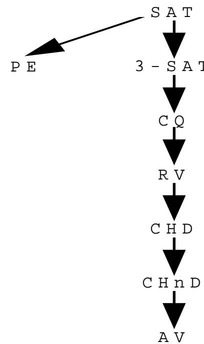
$$\alpha \text{ es satisfacible} \Leftrightarrow \text{G tiene una k-clique.}$$

En el ejemplo, puesto que α tiene tres cláusulas, debe buscarse un triángulo en el grafo. Por ejemplo, los nodos correspondientes a <1,2>, <2,2> y <3,1> conforman un triángulo. Si a los literales correspondientes se les hace corresponder 1 como valor de verdad, la fórmula toda es satisfecha por una asignación que, al menos, afirme esto. Es decir:

$$x_2 = 1, \quad x_3 = 1, \quad \neg x_1 = 1 \quad (x_1 = 0).$$

▯

Como los teoremas B y C, se puede continuar mostrando más y más resultados. Hay catálogos de problemas NP-completos, organizados por temas (grafos, lógica, aritmética, pasatiempos, etc.) y técnicas más o menos estándar para mostrar estos resultados [Gar79]. Una pequeña muestra de algunos resultados conocidos (.→. : "es reducible a"):



La descripción de los problemas aun no mencionados:

PE *(Programación entera)* : Dados una matriz C y un vector d cuyos elementos son números enteros. Decidir si hay un vector b, de 0's y 1's, para el que $Cb \leq d$.

CHD *(Camino Hamiltoniano dirigido)* : Dado un grafo dirigido G, decidir si hay un camino cerrado que pase por todos lo vértices exactamente una vez.

CHnD *(Camino Hamiltoniano no dirigido)* : Dado un grafo no dirigido G, decidir si hay un camino cerrado que pase por todos lo vértices exactamente una vez.

8.4 COMPLEJOS PERO …

Al solucionar problemas reales aparecen frecuentemente problemas NP-completos, lo cual no quiere decir que estos problemas sean, necesariamente, intratables. En realidad, se pueden dar diferentes tipos de soluciones que, aunque no resuelven el problema para cualquier pregunta de manera correcta y rápida, son aceptables para el uso que de ellas se quiere hacer. Estas soluciones se pueden clasificar en tres tipos:

Tipo 1: "Casi siempre se encuentra la respuesta"

Corresponde a desarrollar un algoritmo con las siguientes propiedades:

- Se ejecuta en tiempo polinomial.
- Si encuentra una solución, es la correcta.
- No siempre encuentra una solución, aunque casi siempre lo hace. Además, mientras más grande el tamaño de los datos iniciales, es mayor la probabilidad de encontrar una solución.

Tipo 2: "Casi siempre rápido"

En esta categoría se encuentran los algoritmos que siempre dan la respuesta correcta y casi siempre lo hacen de manera rápida, aunque existen preguntas para las que las respuestas se obtienen en tiempo exponencial. Un algoritmo así tiene las siguientes características:

- Siempre encuentra una solución y es la correcta.
- En promedio la complejidad es mejor que exponencial, pero en ocasiones es exponencial (v.gr., $n^{\log n}$).

Dentro de este tipo de algoritmos se encuentran, por ejemplo, los algoritmos de reintento y los de búsqueda heurística con heurística admisible.

Tipo 3: "Cerca del óptimo"

Este tipo de algoritmos son los que dan una respuesta que, aunque no es la correcta, está aproximadamente cerca de la correcta. Sus características son:

- Ejecución en tiempo polinomial (en promedio).
- Siempre produce una respuesta.
- Existe una garantía de que la respuesta no se desvía de la correcta más de un tanto por ciento, o sea, existe una medida de la aproximación.

Los métodos voraces y los de búsqueda heurística (cuya heurística no es admisible) son ejemplos de algoritmos que pueden estar dentro de esta categoría.

EJERCICIOS

1 Formule problemas CQ_o y CQ_b, análogos a CQ como los problemas de optimización y búsqueda correspondientes a AV. Establezca una relación de dificultad entre las soluciones de estos problemas.

2 Pruebe que 2-SAT está en P.

AYUDA: representar una cláusula $x_1 \vee x_2$ como la conjunción de dos implicaciones:

$$x_1 \Rightarrow x_2,\ x_2 \Rightarrow x_1$$

La satisfacibilidad de una fórmula depende de la existencia de un ciclo

$$x_i \Rightarrow \ldots \Rightarrow \neg x_i \Rightarrow \ldots \Rightarrow x_i.$$

3 Pruebe que los lenguajes reconocidos en tiempo polinomial por MTNDs son los mismos reconocidos en tiempo polinomial por MTAVs.

4 Pruebe que:

a CHD ≤ CHnD

b CHnD ≤ AV

5 **a** Pruebe que PT ≤ KS

b Se ha dado una solución de KS con programación dinámica, de complejidad $O(nB)$.
Por qué no es ésta una prueba de que $P = NP$?