Capítulo 4. Intratabilidad

Diego Munguía Molina *

Octubre, 2017

El propósito de este capítulo es brindar una breve introducción al estudio de los límites de la computación.

El conocimiento y comprensión de estos límites tiene mucha importancia en el diseño de algoritmos pues nos ayudará a determinar cuándo un problema no se puede resolver de manera eficiente, y qué hacer en esta situación.

Vamos a empezar nuestro estudio caracterizando una serie de conceptos relacionados con la clasificación de problemas computacionales[1].

Problema Tratable Es un problema computacional que se puede resolver con un algoritmo que tiene *utilidad práctica* al hacer un uso eficiente de los recursos computacionales.

La noción de utilidad práctica se refiere a que el algoritmo hace un uso eficiente del tiempo y el espacio. Es decir, su ejecución debería tardar un tiempo manejable y debería consumir una fracción manejable del total de memoria disponible.

Notemos que nuestra caracterización de tratabilidad carece de rigorusidad matemática pues hablamos de usos *manejables* de los recursos computacionales. El significado de *manejable* dependerá del contexto del problema.

Por ejemplo en el contexto de un sistema empotrado para el monitoreo de signos vitales en un hospital el tiempo de ejecución de un algoritmo, para ser manejable, tendría que ser de fracciones de segundo.

En contextos más generales podríamos estar de acuerdo en que algoritmos que tarden semanas, meses o años corriendo, o que requieran utilizar más del 100% de la memoria no son algoritmos tratables. En estos escenarios, aunque exista una solución teórica para el problema, la puesta en práctica de la solución no será de mayor utilidad precisamente por la larga espera o porque en algún momento va a necesitar más memoria que la que hay disponible.

Se sigue entonces que los problemas **intratables** son aquellos para los que no existen algoritmos que tengan utilidad práctica.

[1] Para una caracterización de *problema computacional* ver Capítulo 0.

Comúnmente se acepta que los algoritmos tratables son aquellos que se pueden resolver en tiempo y espacio polinomial[2], es decir, aquellos que tienen complejidad $\Theta(n^k)$ para cualquier constante k. Por otro lado, se considera a los algoritmos de complejidad super-polinomial $\mathcal{O}(n!)$ o $\mathcal{O}(k^n)$ como intratables.

Podemos poner en duda la anterior afirmación si pensamos por ejemplo en un algoritmo con complejidad temporal $T(n) = \mathcal{O}(n^{100})$. Este algoritmo tardaría mucho tiempo en ejecutar, perdiendo de esta forma utilidad práctica. Sin embargo, es posible contra-argumentar que los problemas reales típicamente presentan exponentes mucho menores; al día de hoy no se conoce un algoritmo con dicha complejidad. Además, históricamente se ha registrado la tendencia de que una vez encontrada una solución de complejidad polinómica para un problema, rápidamente comienzan a aparecer soluciones más eficientes.

$\mathcal{P}$ versus $\mathcal{NP}$

Los problemas computacionales se pueden clasificar de acuerdo con su complejidad. Esta clasificación nos permite llevar cuenta de adónde se encuentran los límites de la computación y las grandes incógnitas aún no resueltas.

La primer clase de problemas de nuestro interés será la clase $\mathcal{P}$.

Clase $\mathcal{P}$ Problemas que se pueden resolver con algoritmos de complejidad temporal $\mathcal{O}(n^k)$. También se le conoce como $PTIME$.

De acuerdo con nuestra discusión anterior, esta clase entonces contiene a todos los problemas que son tratables, es decir problemas para los que conocemos soluciones eficientes.

Los algoritmos no sólo sirven para resolver problemas computacionales, sino también para verificar soluciones.

Por ejemplo, dado el siguiente problema computacional:

(1) **Encontrar un camino en un grafo**.
Entrada. Grafo G, y una tupla (a, b) que representa un vértice de salida a y un vértice de llegada b.
Salida. Una secuencia de vértices $(a, v_1, v_2, ..., v_n, b)$ que representa un camino entre a y b o $\varnothing$ si los vértices de entrada no están conectados por un camino en G.

Es posible escribir un algoritmo que busque un camino entre a y b. Pero también es posible derivar un nuevo problema a partir de este que podríamos llamar problema de verificación.

(2) **Verificar camino en G**.
Entrada. Grafo G, y una secuencia de vértices $C = (a, v_1, v_2, ..., v_n, b)$ que representa un camino entre los vértices a y b.

[2]Para más detalle sobre esta afirmación ver tesis Cobham–Edmonds.

Salida. *Verdadero* si C representa un camino en G, *Falso* si C no representa un camino en G.

Este nuevo problema tiene como propósito verificar las soluciones del problema original. Al ser este un problema computacional, sabemos que los podemos resolver algorítmicamente. Y con esto volvemos a nuestra afirmación original; los algoritmos no sólo resuelven problemas, sino que también podemos utilizarlos para verificar soluciones. Por tanto, también es posible clasificar el problema original con base en la complejidad de su correspondiente algoritmo de verificación. A esta nueva clase la llamaremos $\mathcal{NP}$ siempre y cuando el algoritmo de verificación tenga complejidad de tiempo polinómica.

Clase $\mathcal{NP}$ Problemas que se pueden verificar con algoritmos de complejidad temporal $\mathcal{O}(n^k)$.

A partir de estas dos definiciones podemos deducir que todo problema que esté en $\mathcal{P}$ también está en $\mathcal{NP}$, tal que $\mathcal{P} \subseteq \mathcal{NP}$.

Ejercicio. Explique porqué la afirmación anterior es verdadera.

En este punto podemos introducir uno de los problemas más importantes aún no resueltos de la computación[3] conocido como $\mathcal{P}$ **versus** $\mathcal{NP}$.

Este problema se pregunta si $\mathcal{P} \subseteq \mathcal{NP}$ realmente corresponde a $\mathcal{P} \subset \mathcal{NP}$ o más bien a $\mathcal{P} = \mathcal{NP}$.

Si se logra demostrar que todo problema en $\mathcal{NP}$ también está en $\mathcal{P}$ quiere decir que si un problema se puede *verificar* en tiempo polinomial, entonces también se puede *resolver* en tiempo polinomial.

Esta demostración tendría consecuencias muy significativas para las ciencias de la computación. Existen problemas en $\mathcal{NP}$ para los que no se conoce una solución eficiente. Si se demostrara que $\mathcal{P} = \mathcal{NP}$ entonces podríamos concluir que estas soluciones existen, a pesar de que no las conocemos, y que por lo tanto vale la pena seguir buscándolas. La solución eficiente de estos problemas afectaría, positiva o negativamente, diversos campos de acción y del saber incluyendo a la filosofía, la economía, la criptografía, la medicina, la biología y la inteligencia artificial.

Para acercarnos más a este problema debemos aprender sobre problemas de decisión, y sobre las clases $\mathcal{NP} - dificil$ y $\mathcal{NP} - completo$.

[3]Este problema es uno de los llamados siete problemas del milenio, y el Instituto Clay de Matemáticas en EE.UU. ofrece un millón de dólares a quien lo resuelva.

Problemas de decisión

Los problemas de decisión aparecen en la base de las ciencias de la computación. Se utilizan para determinar si un problema es tan si quiera computable[4] o no. Por esta razón, no es de extrañar que toquemos el tema en nuestra discusión sobre los límites de la computación.

Problema de decisión Un problema que dada una secuencia de entradas, produce una salida booleana: *falso* o *verdadero*. En otras palabras, es un problema que se puede plantear como una pregunta que se responde con si o no.

Determinar la disivilidad de un par de números es un ejemplo de problema de decisión.

(3) **¿Es x divisible entre y?**.
Entrada. $x, y \in \mathbb{R}$.
Salida. *Verdadero* cuando $x \mod y = 0$, *Falso* cuando $x \mod y \neq 0$.

Los problemas de decisión pueden ser *decidibles* o *indecidibles*. Los problemas decidibles son aquellos para los cuales se puede enunciar un algoritmo que los resuelva. Por otro lado, los problemas indecidibles no se pueden resolver computacionalmente porque no se puede encontrar un algoritmo que no se encicle, o que no llegue a un estado de fallo, o no produzca salidas incorrectas o indefinidas al tratar de resolverlo.

Un ejemplo de problema indecidible es el *problema de la parada*[5].

(4) **Problema de la parada**.
Entrada. La descripción P de un programa computacional, y una tupla de argumentos $E = (e_1, e_2, ..., e_n)$ para P.
Salida. *Verdadero* si P eventualmente se va a detener cuando se ejecute con las entradas E, *Falso* si P se va a enciclar cuando se ejecute con las entradas E.

El problema de la parada establece la pregunta ¿es posible determinar si un programa P se va a enciclar cuando se ejecute con una serie de entradas particular? Alan Turing demostró en 1936[6] que no existe un algoritmo que pueda resolver este problema, por lo tanto es *indecidible*.

Los problemas de verificación se pueden establecer como preguntas de si o no. Volviendo a nuestro ejemplo anterior sobre encontrar un camino en un grafo, podemos replantear el algoritmo de verificación como una pregunta: ¿es $(a, v_1, v_2, ..., v_n, b)$ un camino en G? Podemos concluir entonces que los problemas en la clase $\mathcal{NP}$ son problemas de decisión.

[4] Computable: resoluble computacionalmente, es decir a través de un algoritmo (también llamado método efectivo)

[5] *Halting problem* en la literatura en inglés.

[6] Alan Turing, "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2, 42 (1936), pp 230 - 265.

La clase $\mathcal{NP} - difcil$

Informalmente la clase de problemas $\mathcal{NP} - difcil$ se define como la clase que contiene problemas que son por lo menos tan difíciles como los problemas más difíciles en $\mathcal{NP}$.

Un ejemplo de problema $\mathcal{NP} - difcil$ es el *problema del vendedor viajero*.

(5) **Problema del vendedor viajero**.
Entrada. Un grafo G no dirigido, pesado y completo.
Salida. Una secuencia de vértices $(v_1, v_2, ..., v_n)$ que representa un camino cíclico de costo mínimo en el que aparecen todos los vértices de G exactamente una vez.

Este problema también se puede plantear como una pregunta (que no es de decisión): Dada una lista de ciudades y las distancias entre cada par de ciudades, ¿cuál es la ruta más corta que visita a cada ciudad exactamente una vez y retorna luego a la ciudad de origen?

El problema del vendedor viajero se puede resolver con un algoritmo de fuerza bruta que examina todas las posibles permutaciones de caminos para encontrar el camino de mínimo costo. Este algoritmo tiene complejidad $T(n) = \mathcal{O}(n!)$. Otra posible solución, el algoritmo de programación dinámica *Held-Karp*, resuelve el problema en $T(n) = \mathcal{O}(n^2 2^n)$.

De la definición también se desprende que los problemas en $\mathcal{NP} - difcil$ no necesariamente tienen que ser parte de $\mathcal{NP}$. El problema de la parada es un ejemplo de un problema $\mathcal{NP} - difcil$ que no está en $\mathcal{NP}$ (pues es indecidible).

Ejercicio. Explique porqué la afirmación "los problemas en $\mathcal{NP} - difcil$ no necesariamente tienen que ser parte de $\mathcal{NP}$" es verdadera.

La clase $\mathcal{NP} - completo$

Conocer esta clase nos permitirá establecer algunas relaciones de equivalencia entre problemas, y con esto plantear de manera más formal las consecuencias de resolver el problema de $\mathcal{P} versus \mathcal{NP}$.

Clase $\mathcal{NP} - completo$ Problemas de decisión que están en $\mathcal{NP}$ y en $\mathcal{NP} - difcil$.

Esta relación indica que los problemas $\mathcal{NP} - completo$ se pueden verificar fácilmente, al mismo tiempo que no se conocen algoritmos eficientes (tiempo polinomial) que los solucionen.

Siguiendo la definición anterior y si fuera el caso que $\mathcal{P} = \mathcal{NP}$ entonces inmediatamente se seguiría que $\mathcal{P} = \mathcal{NP} - completo$ tal como lo ilustra la figura 1.

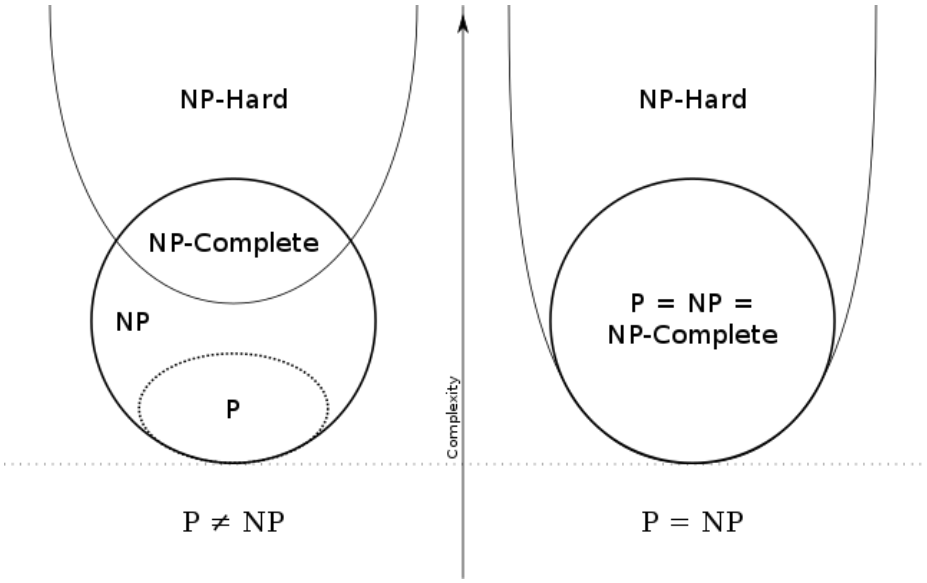

Figura 1: Diagrama de Euler para P versus NP (Behnam Esfahbod CC BY-SA 3.0 via Wikimedia Commons)

Esto significaría que existen soluciones en tiempo polinomial para algunos de los problemas más complejos de la computación como lo son el vendedor viajero, los ciclos hamiltonianos, el problema de los cliques, o el problema de la suma de subconjuntos.

¿Cómo demostramos que un problema está en $\mathcal{NP} - completo$?

Para responder esta pregunta necesitamos conocer el concepto de *reducción*.

Reducción Mecanismo para reducir un problema para el que no se conoce una solución a otro para el que si se conoce solución.

Decimos que existe un algoritmo de reducción en tiempo polinomial cuando se cumplen las siguientes condiciones:

1. A es un problema de decisión que se quiere resolver con alguna restricción, por ejemplo en tiempo polinomial.
2. B es un problema de decisión para el que se conoce un algoritmo de solución que cumple con las restricciones establecidas para A. Por ejemplo, se resuelve en tiempo polinomial.
3. α es una instancia de A y β es una instancia de B.
4. Existe un algoritmo que transforma cualquier α en β en tiempo polinomial, y la respuesta para α es R si y sólo si la respuesta para β es R. Este algoritmo lo llamaremos $Tr_\beta(\alpha) : \beta$.

Si se cumplen estas cuatro condiciones, entonces podemos resolver A ejecutando B sobre una transformación de las instancias tal como se plantea en la siguiente ecuación.

$$A(\alpha) = B(Tr_\beta(\alpha)) \tag{1}$$

Para demostrar que un problema que llamaremos *Objetivo* está en $\mathcal{NP}-$*completo* necesitamos de un problema que llamaremos $Referencia$, para el que sean hechos conocidos que (a) no se puede resolver en tiempo polinomial, y (b) se puede verificar en tiempo polinomial. Se cumple además que es posible reducir *Objetivo* a $Referencia$.

Buscando una prueba por contradicción, suponemos que *Objetivo* se resuelve en tiempo polinomial; dado que el problema se puede reducir a $Referencia$ tendríamos que aceptar entonces que $Referencia$ también se resuelve en tiempo polinomial por la condición (4) en la definición de reducción.

Un problema de referencia de uso común para este tipo de demostraciones es el *problema de satisfacibilidad de circuitos* (**SAT**). El teorema Cook-Levin[7] demuestra que **SAT** es $\mathcal{NP}-$*completo* y su prueba también muestra cómo se puede reducir cualquier problema de decisión en $\mathcal{NP}$ a **SAT**.

El problema SAT

Este problema se pregunta si existe un instanciamiento de variables que satisfaga una expresión booleana dada. En otras palabras, se pregunta si es posible encontrar una combinación de entradas para una expresión booleana que produzca *Verdadero* como resultado. Recordemos que las expresiones booleanas combinan variables y los operadores lógicos $\wedge$, $\vee$ y $\neg$.

Consideremos la expresión $E = \neg(x_1 \vee x_2) \wedge \neg(\neg x_3 \wedge x_4)$. Decimos que E es satisfacible cuando $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, y $x_4 = 1$.

(6) **SAT**.
Entrada. Una expresión booleana E.
Salida. Instanciamiento de variables que satisface la expresión.

Sólo se conocen algoritmos con complejidad exponencial para resolver este problema.

Este problema tiene aplicación en el campo diseño de circuitos, específicamente en la optimización de circuitos. Cuando se tienen combinaciones complejas de compuertas lógicas es deseable poder evaluar la satisfacibilidad del circuito. Si una sección de un circuito no es satisfacible se puede reemplazar el conjunto de compuertas por un 0, logrando de esta manera una simplificación del circuito.

[7] Cook, Stephen (1971). "The complexity of theorem proving procedures". Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158.

Otras posibles aplicaciones son en el área de demostración automática de teoremas y en inteligencia artificial.

Problemas de Optimización

Aún no existe una solución al problema $\mathcal{P} versus \mathcal{NP}$, por esta razón sólo conocemos soluciones super-polinomiales –y por tanto intratables– para los problemas $\mathcal{NP} - completo$.

¿Qué podemos hacer si nos tenemos que enfrentar a un problema $\mathcal{NP}-completo$? ¿Existe algún tipo de solución alternativa?

En este caso es posible utilizar algoritmos de optimización que se ejecutan en tiempo polinomial para generar soluciones aproximadas.

Problema de optimización Generar posibles soluciones a un problema de complejidad super-polinomial. Cada solución generada tiene un valor asociado y relacionado con las restricciones del problema a aproximar. Se busca obtener la solución con el mejor valor posible.

Las técnicas de diseño de algoritmos de programación probabilística, *hill-climbing*, *simulated annealing* y algoritmos genéticos son diferentes acercamientos a la resolución del problema de optimización.

Discutiremos estas técnicas con mayor detalle en el próximo capítulo.