Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Complejidad

Encontrar un algoritmo para resolver un problema no necesariamente significa que podamos alcanzar la solución. La razón son los recursos y un ejemplo clásico para ilustrar esto es el ***Traveling Salesman Problem*** (TSP).

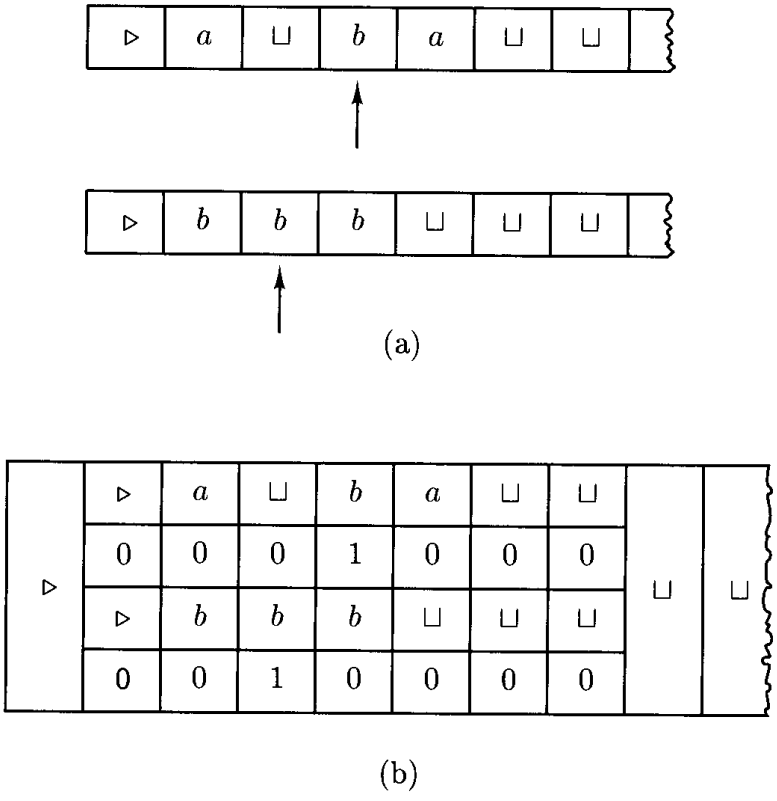
Este problema nos presenta una cantidad n de ubicaciones y las distancias entre ellas. Esto se representa con un grafo cuyos vértices son las ciudades y cuyas aristas ponderadas son los caminos entre ellas, conectando cada par de vértices. Un vendedor ambulante desea visitar dichas ubicaciones siguiendo un itinerario óptimo (que minimice la trayectoria total recorrida) y que termine en la ciudad de partida.

Ejercicio de práctica A: identifique la forma en la que se debe presentar una solución a este problema y plantee por qué no es escalable un algoritmo *nave* para resolverlo, apoyándose en la notación *big-Oh*.

Este es el tipo de problemas que no son prácticamente solucionables, aunque exista un algoritmo para resolverlos. Argumentos empíricos han llevado a la asociación de los algoritmos considerados aplicables en la práctica con los que están **acotados polinomialmente**. Estos algoritmos se caracterizan como aquellos donde una máquina de Turing que los implementa no realiza más de $p(n)$ pasos en su computación, donde $p(n)$ es un polinomio. El tiempo de ejecución de algoritmos pertenecerá a $O(n^k)$ si k es el grado del polinomio. Aquellos problemas cuyos algoritmos están acotados polinomialmente caen dentro de la **clase de complejidad P**.

Hay que observar que un algoritmo de éstos podría pertenecer a $O(n^{100})$, lo cual es claramente irrealizable. Sin embargo, este tipo de algoritmos raramente se ve en la práctica; este es uno de los argumentos empíricos que apoyan la clasificación antes mencionada. La complejidad de un problema se determina, naturalmente, con respecto al algoritmo más rápido conocido para resolverlo.

La clasificación de un problema como P es independiente del modelo de computación elegido. No importa si usamos máquinas multicinta, o con muchos lectores, o con una cinta multidimensional. Transferir un algoritmo de un modelo a otro toma tiempo polinomial, por lo que el tiempo de ejecución en general sigue siendo polinomial. Para ilustrar esto, consideremos la simulación de máquinas de Turing multicinta con máquinas de una única cinta.



La multicinta de (a) representada en una unicinta de cuatro pistas en (b). Cada columna contiene, en las pistas impares, los símbolos de una cinta y, en las pares, los símbolos que determinan dónde está el lector en esa cinta.

Llamemos a una máquina multicinta arbitraria M y a la máquina unicinta que la simule T_M. La simulación se realiza arrejumbrando todas las cintas de M en la cinta de T_M, representadas como "pistas" o *tracks*. El lector de T_M lee una "columna" a la vez, y la idea es que T_M en realidad no está leyendo varios símbolos sino un único "símbolo compuesto" $[t_1, t_2, \dots, t_k]$ (donde los t_i son símbolos del alfabeto de M, uno en cada cinta de M). Es como que le hubiéramos puesto nombre a cada posible tupla $[t_1, \dots, t_k]$ y la hubiéramos agregado al alfabeto de T_M.

El problema que nos queda es que M tiene un lector por cinta y los lectores de cada cinta no necesariamente están todos en la misma posición, y T_M, que solo tiene un lector, necesita representar las posiciones de los lectores de M. Por otra parte, mantengamos la condición de que una máquina multicinta recibe y mantiene su *input* en una única cinta. Si el *input* dado a la multicinta es $x = (a_1, \dots, a_n)$ sea $|x| = n$ la cantidad de símbolos en dicho *input*. El primer paso de T_M debe ser transformar cada a_i a $[a_i, 1, B, 1, \dots]$, lo cual conlleva un recorrido completo del *input* de M, incurriendo en un costo $O(|x|) = O(n)$.

Ejercicio de práctica B: explique cómo la máquina unicinta puede encontrar y representar la posición de cada lector en la multicinta. Luego plantee los pasos que tendrá que seguir para simular la operación de un paso completo de la multicinta. Finalmente, responda: ¿qué información de la multicinta necesitamos para simular su operación, y como representamos esa información en cada paso de la unicinta?

La cantidad de pasos realizada por T_M es proporcional a los pasos de M, ya que por cada paso de M se recorre la cinta de T_M una cantidad constante de veces. El contenido en cada cinta de M puede incrementar su tamaño solo una casilla en cada paso. Eso significa que T_M añadirá, como máximo, una "columna" a sus pistas en cada paso. El costo inicial correspondiente a la transformación del *input* es $O(|x|)$ y el costo de simular m pasos de M será $O\big(m * (|x| + m)\big)$, suponiendo que con cada paso se añade una columna, incrementando el tiempo de cada recorrido del contenido en la cinta de T_M durante la simulación.

Como el costo incurrido por T_M en todo momento no es calculado con una función exponencial o factorial o algo similar, sino con un polinomio sobre costo m para M (porque $|x|$ es una constante), la solución implementada por M no cambia de clase de complejidad al implementarse en T_M. Además, demostramos que todas las máquinas multicinta se pueden simular en tiempo polinomial con máquinas unicinta, demostrando que no son significativamente más poderosas.

Demostrar que no existen algoritmos acotados polinomialmente para resolver algún problema se dificulta. Aunque casi todos los tipos de máquinas de Turing son equivalentes a una máquina estándar de una cinta, hay una máquina que puede resolver en tiempo polinomial los problemas que otras máquinas no: la **máquina de Turing no-determinística**. Estas máquinas de Turing son capaces de realizar más de una acción a partir de una configuración determinada. Es decir que, en lugar de desarrollar una computación de forma de cadena lo pueden hacer como un árbol, donde cada nivel corresponde a una o (simultáneamente) más operaciones en una máquina determinística. Estas máquinas, para proveer un resultado, realizan todas las computaciones posibles de forma no-determinística y luego revisan los resultados para ver cuál provee lo deseado.

Con esto se da paso a la **clase de complejidad NP** (*Non-deterministic polinomially bounded*): los problemas que pueden ser resueltos en tiempo polinomial por una máquina de Turing no-determinística. El TSP es NP porque un algoritmo no-determinístico propondría, por cada ubicación visitada, todas las ubicaciones (excepto la actual) como la siguiente, y recorrería todos los posibles caminos en una cantidad de iteraciones igual n.

Los problemas se clasifican como **problemas de decisión** y **problemas de búsqueda** (también llamados **abstractos** o **de optimización**). Los problemas de decisión tienen una respuesta positiva o negativa, como un **predicado** (función con contradominio binario). Reconocer un *input* como perteneciente a un lenguaje es un problema de decisión.

Los problemas abstractos son asociaciones específicas entre el conjunto de instancias (posibles *inputs*) del problema y un conjunto de soluciones que puede ser distinto a {0,1}. Sin embargo, es importante notar que todo problema de búsqueda se puede asociar con un problema de decisión bajo la imposición de ciertas restricciones. Dicha asociación suele implicar la posible solución del problema de búsqueda en términos de soluciones al problema de decisión.

Ejercicio de práctica C: agregue los recursos que considere necesarios y explique cómo podría lograrse ejecutar el algoritmo *naïve* para resolver el TSP en tiempo polinomial. Responda: ¿qué problema de decisión podría plantearse asociado al TSP?

Es importante conocer la diferencia entre los tipos de problema porque, formalmente, las clases de complejidad vistas comprenden únicamente problemas de decisión. Hablamos de medir la complejidad de un problema abstracto asociándolo con uno o más problemas de decisión adecuados. Los problemas en NP son también caracterizados por ser **verificables** en tiempo polinomial. Si bien no se ha podido resolver un problema NP en tiempo polinomial, sí se puede verificar en tiempo polinomial si una solución propuesta es válida o no. Esa solución propuesta es llamada un **certificado**.

Un algoritmo de verificación para el TSP tomaría una instancia del problema (grafo conexo, ponderado y no-dirigido) y una secuencia de todas las ubicaciones, sin repetir, como certificado. El algoritmo revisaría que el certificado tenga todas las ubicaciones y que visite cada una únicamente una vez. Terminaría comparando la distancia recorrida con la distancia d dada. Esto es claramente realizable en tiempo polinomial.

Ejercicio de práctica D: explique por qué $\boldsymbol{P \subseteq NP}$**.**

Nadie ha demostrado al mundo que no existe un algoritmo eficiente para resolver algún problema NP, aunque la creencia general es que no lo hay. La igualdad entre dos conjuntos se define como la contención mutua: $A = B \Leftrightarrow A \subseteq B$ y $B \subseteq A$. Quien demuestre $NP \nsubseteq P$ o $NP \subseteq P$ será mi héroe o heroína… además de que se ganará un pistal.

Notemos que demostrar que un problema es NP no implica que no sea P. Demostrar que un problema pertenece a P requiere la propuesta de un algoritmo eficiente que lo resuelve, pero no encontrar dicho algoritmo no nos permite concluir nada. Si un problema no es verificable en tiempo polinomial significa que es, de hecho, más difícil que cualquiera en NP. Estos problemas caen dentro de una clase de complejidad nueva: la **clase de complejidad NP-*hard***. Lo que clasifica a un problema X como NP-*hard* es que cualquier problema de NP es **reducible en tiempo polinomial** a X. La reducibilidad de la que hablamos plantea que si podemos resolver X podemos resolver todos los problemas reducibles a él. En otras palabras, si proponemos una solución a un problema A que requiera o use la solución a otro problema B, estamos reduciendo el problema A al problema B. Lo importante es que la reducción se pueda hacer con un algoritmo con tiempo de ejecución acotado polinomialmente.

Una reducción en tiempo polinomial (denotada con $\leq_p$) se define formalmente entre dos lenguajes arbitrarios L_1 y L_2 como una función $f: L_1 \rightarrow L_2$ computable en tiempo polinomial, de modo que si $L_1 \leq_p L_2$ (L_1 es reducible en tiempo polinomial a L_2, o "no es más difícil que" L_2) entonces:

$$\forall x, x \in L_1 \Leftrightarrow f(x) \in L_2$$

Lo que esto nos dice es que un lenguaje es reducible a otro si podemos transformar las instancias de uno en instancias del otro en tiempo polinomial y estas instancias relacionadas tienen las mismas respuestas de pertenencia a sus respectivos lenguajes.

Como mencionado, formalmente se tratan las clases P y NP como conjuntos de lenguajes que son decidibles en tiempo polinomial por máquinas de Turing ordinarias y no-determinísticas, respectivamente. De la forma en que se han definido las reducciones en tiempo polinomial arriba es un tipo restringido de reducciones llamadas **reducciones de Karp**, caracterizadas por transformar el *input*, meterlo al algoritmo que resuelve otro problema y dar la respuesta obtenida. Un tipo más general de reducciones se llama **reducciones de Cook** o **de Turing**. Plantean que la solución a un problema se puede alcanzar con una o más llamadas a la solución de otro problema.

Dado que una reducción de Cook plantea que la solución a A se puede alcanzar con un algoritmo que llama una o más veces a la solución de B, es necesario garantizar que la cantidad de veces que se llama a la solución de B sea polinomial, así como el tiempo de ejecución del algoritmo entre o alrededor de las llamadas a la solución de B. Intuitivamente, si esto se cumple es porque la reducción es "fácil", por lo que la dificultad de A dependerá únicamente de la dificultad de B.

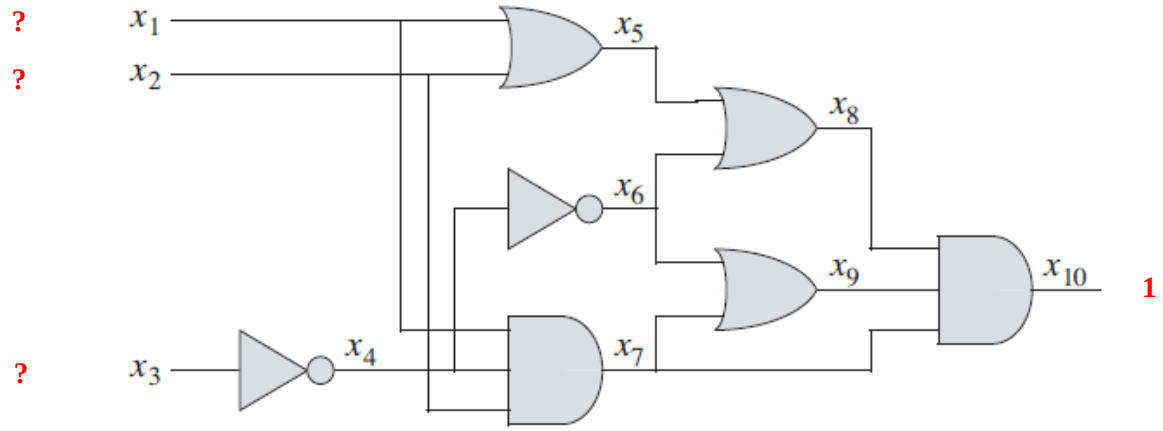
La clase NP-*hard* es la clase de problemas más difíciles que trataremos, pero hay bastantes otros problemas que son tan difíciles como los NP-*hard* y que, además, también son NP. Esta intersección entre NP y NP-*hard* es la importante **clase de complejidad NP-*complete***.

Ejercicio de práctica E: ¿Cuál es la importancia de la clase NP-*complete*? Dibuje un diagrama de Venn que explique la relación entre las clases de complejidad P, NP, NP-*hard* y NP-*complete*.

La reducibilidad polinomial es transitiva, por lo que encontrar un problema NP-*complete* "natural" permitiría encontrar más problemas NP-*complete* mediante reducciones en cadena. La idea es que si un problema X es conocido NP-*complete* es también NP-*hard*, y reducir X a cualquier problema Y demostrará que Y también es NP-*hard* porque todos los problemas NP se podrán reducir a Y, además de que resolver X será tan fácil o difícil como sea resolver Y.

En otras palabras, el algoritmo para resolver X sería reducir (en tiempo polinomial) X a Y y luego resolver Y. Si Y se reduce a un problema Z, resolver Z permitirá resolver Y y resolver Y permitirá resolver X. Y como X es NP-*complete* (lo que implica que es NP-*hard*), todo problema NP se puede reducir a él, lo que implica que todo problema NP se puede reducir a Y y a Z, haciéndolos NP-*hard*. Si Y y Z son, además, NP, entonces Y y Z serán NP-*complete*.

El primer problema NP-*complete* descubierto es el ***Satisfiability Problem*** (abreviado SAT). Este problema fue demostrado NP-*complete* por Stephen Cook, y dicha demostración es referida frecuentemente como el **Teorema de Cook-Levin**. SAT pregunta si, dada una expresión booleana, existe alguna asignación de valores a sus variables de forma que el resultado de la expresión sea 1 o verdadero; en palabras inventadas, si la expresión es **satisfactible**.



Una instancia de (*Circuit*) SAT

Es fácil ver que este problema pertenece a NP porque podemos asignar valores a las variables de forma no-determinística y luego evaluar las expresiones para ver si alguna resulta en 1. La evaluación es el método de verificación y sabemos que evaluar una expresión booleana toma tiempo polinomial. La parte divertida es demostrar que SAT es NP-*hard*; *i.e.*, probar que todo problema NP se puede reducir a SAT (ver "Apéndice Complejidad: demostración SAT NP-*complete*).

Para ejemplificar un problema de NP-*hard* (pero no NP-*complete*) podemos tomar el *halting problem*, el problema usado comúnmente para explicar no-computabilidad. El *halting problem* plantea la pregunta: ¿existirá algún algoritmo que, dado cualquier otro algoritmo y su respectiva entrada, nos diga (sin ejecutarlo) si ese otro algoritmo se detendrá al procesar esa entrada? Este problema no es NP porque no es computable, y los problemas NP son computables en tiempo polinomial por máquinas de Turing no-determinísticas. Podemos reducir SAT al *halting problem* en tiempo polinomial, con lo cual demostramos que todo problema en NP es reducible en tiempo polinomial al *halting problem*, por transitividad. Solo tenemos que suponer que existe un algoritmo mágico que resuelve el *halting problem*.

Ejercicio de práctica F: plantee la reducción de SAT al *halting problem*.

Fuentes:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. Massachusetts: The MIT Press.
- Goldreich, O. (2008). Computational Complexity: A Conceptual Perspective. New York: Cambridge University Press.
- Hopcroft, J. E., & Ullman, J. D. (1979). Introduction to Automata Theory, Languages and Computation. Addison-Wesley.
- Lewis, H. R., & Papadimitriou, C. H. (1988). Elements of the Theory of Computation. New Jersey: Prentice-Hall.
- Martin, J. C. (2003). Introduction to Languages and the Theory of Computation. McGraw-Hill.
- Sudkamp, T. A. (1997). Languages and Machines: An Introduction to the Theory of Computer Science. Addison-Wesley.
- http://en.wikipedia.org/wiki/NP_(complexity)
- http://www.cs.umsl.edu/~sanjiv/classes/cs5130/lectures/np.pdf
- https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/#limits-to-computing