Notas de Curso

Introducción a los Problemas NP-Completos
(Teoría)

Dra. María De Luz Gasca Soto

Licenciatura en Ciencias de la Computación

Facultad de Ciencias, UNAM.

Revisión, Enero 2021

1. Introducción.

En cursos anteriores se revisan diferentes técnicas para resolver problemas algorítmicamente y se aplican a problemas específicos. Sería ideal si todos los problemas tuvieran un algoritmo elegante y eficiente que pudiera ser descubierto por un pequeño conjunto de técnicas. Pero la vida no es tan simple. Hay muchos problemas que no parecen "sucumbir" ante las técnicas que hemos aprendido. En este documento describiremos técnicas para identificar algunos de estos problemas.

El tiempo de ejecución de la mayoría de los algoritmos que hemos visto está acotado por algún polinomio que depende del tamaño de la entrada. A tales algoritmos los llamamos **Algoritmos Eficientes** y denominaremos como **Problemas Tratables** a los correspondientes problemas. En otras palabras:

Definición 1. Un algoritmo es **eficiente** si su desempeño computacional requiere tiempo ***O(P(n))***, donde ***P(n)*** es un polinomio sobre el tamaño de la entrada[1], ***n***.

Definición 2. La **Clase** de todos los problemas que pueden ser resueltos por algoritmos eficientes se denota por **P**, de tiempo **P**olinomial.

Definición 3. En Ciencias de la Computación, un problema es llamado **intratable** si resulta imposible resolverlo con un algoritmo polinomial.

Esta definición de eficiencia puede parecer extraña, ya que un algoritmo de orden $O(n^{10})$ no es *eficiente* o uno de complejidad $O(10^7 n)$ tampoco lo es, aunque sea lineal. Sin embargo se da esta definición por dos razones:

1) Permite el desarrollo de la teoría, la cual estamos explorando.
2) Funciona en la práctica, que es muy importante. Los problemas tratables tienen soluciones practicas. Por supuesto, algunas mejores que otras.

En otras palabras, los algoritmos de tiempo polinomial que encontramos en la práctica tienen grado pequeño, generalmente cuadrático. Inversamente, los algoritmos para los cuales se tiene que el tiempo de ejecución es más grande que cualquier polinomio, usualmente, no son prácticos para ejemplares muy grandes.

Hay muchos problemas para los cuales se conocen algoritmos de tiempo no polinomial. Algunos de ellos pueden ser resueltos por algoritmos eficientes aún no conocidos o descubiertos. Sin embargo, se tiene la certeza de que muchos problemas no pueden ser resueltos con algoritmos eficientes. Nos gustaría ser capaces de identificar y clasificar tales problemas, para de esta forma no tener que gastar tiempo buscando algoritmos inexistentes.

1.1 Clasificación de Problemas.

Los problemas se pueden clasificar, según su complejidad, en tres categorías generales que a continuación describiremos.

Categoría 1. ***Problemas para los cuales sí se han encontrado algoritmos de tiempo polinomial.*** Cualquier problema para el cual se tiene un algoritmo polinomial cae en esta categoría. Hemos encontrado algoritmos de tiempo $\Theta(n \log n)$ para el problema de ordenamiento, algoritmos de búsqueda que requieren tiempo $\Theta(\log n)$ y podríamos listar muchos ejemplos como estos.

[1] Generalmente el tamaño de la entrada se define como el número de bits requeridos para representarla.

Categoría 2. ***Problemas que se han demostrado ser intratables.*** Hay dos tipos de problemas en esta categoría. El primer tipo consiste de los problemas que requieren una salida, respuesta, no polinomial, como lo es el determinar circuitos hamiltonianos: si tuviéramos una arista de un vértice a cada uno de los otros vértices, habría ***(n-1)!*** circuitos hamiltonianos, el algoritmo que solucione este problema debería mostrar todos estos circuitos, lo que significa que tal requerimiento no es razonable.

El problema del circuito hamiltoniano que pregunta únicamente por un circuito claramente no es un problema de este tipo. Aunque es importante reconocer este tipo de intratabilidad, problemas como estos no representan dificultad. Usualmente, es posible reconocer de manera directa que la salida requerida no posee un tamaño de orden polinomial y una vez hecho esto, se replantea el problema simplemente preguntando por más información que podamos usar.

El segundo tipo de intratabilidad ocurre cuando los requerimientos del problema son razonables y podemos probar que el problema no puede ser resuelto en tiempo polinomial. Se han encontrado relativamente pocos de estos problemas. éstos fueron los problemas no-decidibles, *undecidable problems*, que fueron llamados así ya que se puede probar que para ellos no puede existir un problema que los resuelva. El más famoso de este tipo es el *Halting Problem*. En este problema tomamos como entrada cualquier algoritmo A y cualquier entrada E para A, el problema consiste en decidir si el algoritmo A se interrumpirá a aplicarlo sobre E. En 1936 Turing demuestra que este problema es no-decidible. Durante los años 50 y 60 se construyeron problemas de forma artificial y con características especificas para los cuales se demostró que eran intratables. A principios de la década de los 70's se demuestra que algunos problemas de decisión, no construidos artificialmente, son intratables.

Categoría 3. ***Problemas para los cuales no se ha podido demostrar que son intratables, pero que no se ha encontrado un algoritmo polinomial que lo solucione.*** Esta categoría incluye cualquier problema para el cual un algoritmo de tiempo polinomial no ha sido descubierto y que además no se ha podido demostrar que tal algoritmo no es posible. Algunos de estos problemas, que requieren una solución, son: el problema del agente viajero, el problema de la ***m***-coloración, para ***m> 2***, el problema de los circuitos hamiltonianos. Se han encontrado algoritmos de aproximación y heurísticas para estos problemas, tales resultan ser eficientes para ejemplares grandes. Estos es, existe un polinomio en ***n*** que acota el número de veces el número de operaciones elementales que se efectúan sobre ejemplares tomados de un conjunto restringido. Sin embargo, tal cota polinomial no existe para todo el conjunto de ejemplares. Para mostrar esto, es necesario encontrar una secuencia infinita de ejemplares para los que ningún polinomio en ***n*** acote el número de operaciones elementales que se ejecutan durante el algoritmo.

Existe una interesante relación entre muchos de los problemas en esta categoría. El desarrollo de tales relaciones llevo a la creación de la Teoría de NP, la cual revisaremos durante este curso.

1.2 Problemas de Decisión

Es mas conveniente desarrollar la teoría si nos restringimos a los problemas de decisión. Un problema de decisión es aquel cuya salida es simplemente si o no. Cada problema de optimización tiene un correspondiente problema de decisión. A continuación presentamos una lista de problemas de optimización y su versión como problemas de decisión.

Ejemplo 1. **Problema del Agente Viajero.** Sea ***G*** una gráfica dirigida y con pesos sobre las aristas. Se define un ***tour*** en la gráfica como una trayectoria que inicia en un vértice, visita todos los vértices de la gráfica, una sola vez, y termina en el vértice inicial.
El problema de optimización del agente viajero consiste en determinar un tour con el mínimo peso total sobre sus aristas.

Dado un número positivo ***d***, el problema de decisión del agente viajero consiste en determinar un tour cuyo peso total no sea mayor a ***d***.

Ejemplo 2. **Problema de la Mochila.** El problema de optimización de la mochila consiste en determinar la ganancia total máxima de los elementos que pueden ser puestos en la mochila, dado que cada elemento tiene un peso y un beneficio (ganancia) y que solo un peso máximo ***W*** puede ser cargado en la mochila.

Dada una ganancia ***p***, el problema de decisión de mochila consiste en determinar si es posible cargar la mochila con un peso no mayor a ***W***, mientras el total de las ganancias sea menor o igual a ***p***.

Ejemplo 3. **Problema de Coloración en Gráficas.** El problema de optimización para la coloración consiste en determinar el mínimo número de colores necesarios para colorear una gráfica de tal forma que dos vértices adyacentes no tengan el mismo color, a una coloración con estas características se llama **coloración válida**. Al mínimo número de colores utilizado en una coloración válida se le denomina el **número cromático** de la gráfica.

Dado un número positivo ***m***, el problema de decisión de coloración consiste en determinar si existe una coloración válida que utilice a lo más ***m*** colores.

Ejemplo 4. **Problema del Clan.** Un **clan** en una gráfica no dirigida ***G=(V, E)*** es un subconjunto ***W*** de ***V*** tal que cada vértice en ***W*** es adyacente a los demás vértices en ***W***. Un clan maximal es un clan de tamaño máximo.

El problema de optimización del Clan consiste en determinar el tamaño de un clan maximal en para gráfica dada.

Dado un entero positivo ***k***, el problema de decisión del Clan consiste en determinar si existe un clan que contenga al menos ***k*** vértices.

Ejemplo 5. **Problema del Conjunto Independiente.** Un **conjunto independiente** en una gráfica no dirigida ***G=(V, E)*** es un sub-conjunto de vértices ***W*** de ***V*** tal que cada vértice en ***W*** no es adyacente a los demás vértices en ***W***. Un conjunto independiente maximal es un conjunto independiente de tamaño máximo.

El problema de optimización del conjunto independiente consiste en determinar el tamaño de un conjunto independiente maximal para una gráfica dada.

Dado un entero positivo ***k***, el problema de decisión del conjunto independiente consiste en determinar si existe un conjunto independiente que contenga al menos ***k*** vértices.

1.3 Sobre la Teoría de P y NP

En este material se discutirá sobre problemas que no se sabe si están en **P**. En particular, revisaremos una clase especial de problemas llamada Problemas **NP-Completos**. Podemos agrupar a estos problemas en una clase ya que en un sentido estricto son equivalentes:

Existe un algoritmo eficiente para cualquier problema NP-Completo si
y sólo si existe un algoritmo eficiente para todo problema NP-Completo.

Existe una creencia general de que no existe un algoritmo eficiente para cualquier problema NP-Completo, pero esto no ha podido ser demostrado. Aún si hubiera algoritmos eficientes para problemas NP-Completos éstos serían, seguramente muy complicados, pues esta clase de problemas han sido estudiados por diversos investigadores durante muchos años. Cientos de problemas han sido clasificados como NP-Completos, por lo cual este material resulta ser muy importante

Revisaremos este material en tres partes:

1) Se define formalmente el concepto de Transformación o Reducción de Problemas.
2) Se define la clase de los problemas NP-Completos, mostrando la forma cómo se demuestra que un problema pertenece a ésta clase.
3) Se presentan varias técnicas y ejemplos para resolver de forma aproximada problemas NP-Completos. Tales soluciones posiblemente no sean las óptimas y no siempre funcionan, pero son mejor que nada.

2. Transformaciones o Reducciones.

En esta sección describiremos a los conceptos básicos sobre transformaciones o reducciones de problemas.

Sean **P** y **Q** dos problemas. Supóngase que un ejemplar arbitrario **E** de **P** puede solucionarse convirtiendo al ejemplar **E** en un ejemplar **E'** de **Q**, resolviendo para **E'** y reduciendo la solución **S'** en la solución **S** para **E**. Este proceso es fácil de visualizar en un diagrama, la Figura 1 lo muestra.

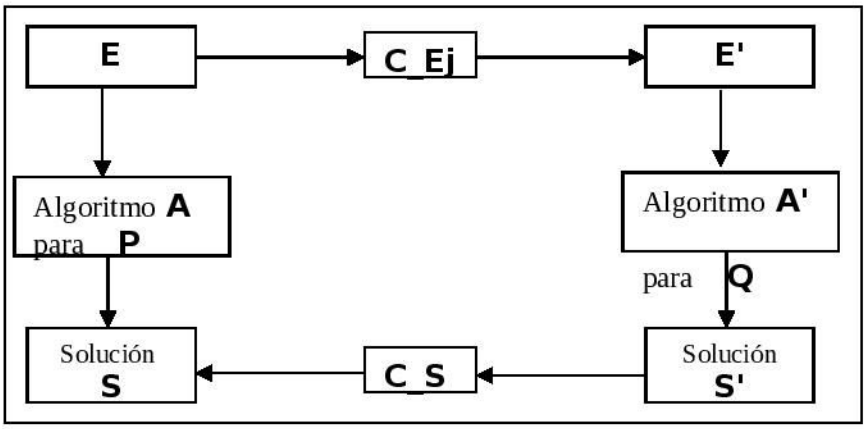


Figura 1. Reducción de **P** en **Q**: **P** $\propto$ **Q**

Definición 4.

Si existen algoritmos, **C_Ej**, para convertir un ejemplar **E** en otro **E'** y, **C_S**, para convertir la solución **S'** en **S**, se dice entonces que existe una **reducción** de **P** en **Q**, la cual se denota como **P** $\propto$ **Q.**

Generalmente, se realiza una transformación de P en Q, cuando P es muy difícil de resolver de manera directa y Q es más fácil. Por ello, en Ciencias de la Computación, a este proceso se le denomina Algoritmo de Reducción o Algoritmo de Simplificación, el cual es expresado explícitamente en la Figura 2.

```
Algoritmo de Simplificación ( E: Ejemplar )
Inicio
        E' ←−  C_Ej ( E );          // Convertir Ejemplar
        S' ←−  A' ( E' );           // Resolver para Q
        S  ←−  C_S ( S' )           // Transformar Solución
Fin
```

Figura 2. Algoritmo de Simplificación.

Ejemplo 1. Considere los siguiente problemas,

P: Encontrar la media en un conjunto de ***n*** números; y

Q: Ordenar una lista de ***n*** números.

El problema **P** es equivalente a buscar el $\lceil n/2 \rceil$–ésimo elemento más pequeño del conjunto de ***n*** datos. Si el conjunto se encuentra ordenado, resulta muy fácil encontrar tal elemento y entonces podemos construir la reducción **P** $\propto$ **Q.**

La Figura 3 ilustra el algoritmo de reducción para este ejemplo y la Figura 4 muestra la esquematización gráfica de la transformación.

```
SimplificaMedia ( E: Ejemplar )
Inicio
        E' ←−  E ;                  // Convertir Ejemplar
        S' ←−  Ordenar( E' );       // Ordena los datos
        S  ←−  S' ( ⌈n/2⌉ )         // Toma al elemento ⌈n/2⌉
        Regresa ( S )
Fin
```

Figura 3. Algoritmo de Reducción para Simplificar la media.

Para este ejemplo, los algoritmos de conversión son triviales, ya que E=E' y S sólo extrae al elemento en la posición $\lceil n/2 \rceil$ de la secuencia ordenada S'.

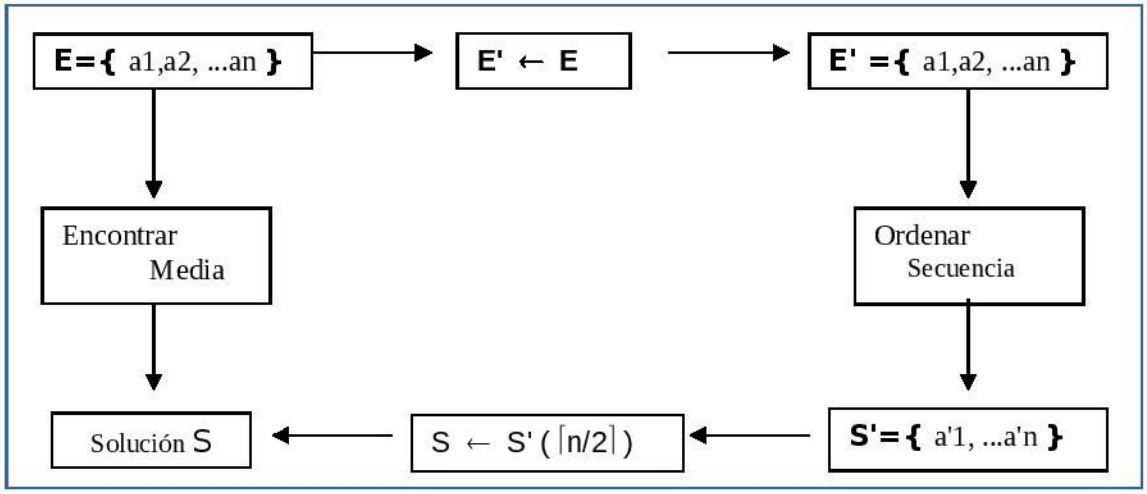


Figura 4. Reducción para encontrar la media

Ahora necesitamos una forma de garantizar el desempeño computacional del algoritmo de simplificación para **P**, considerando que ya sabemos cual es el tiempo requerido para efectuar cada conversión y resolver **Q**. Tal garantía nos la da Kingston[2] en el siguiente resultado, el cual se ilustra en la Figura 5.

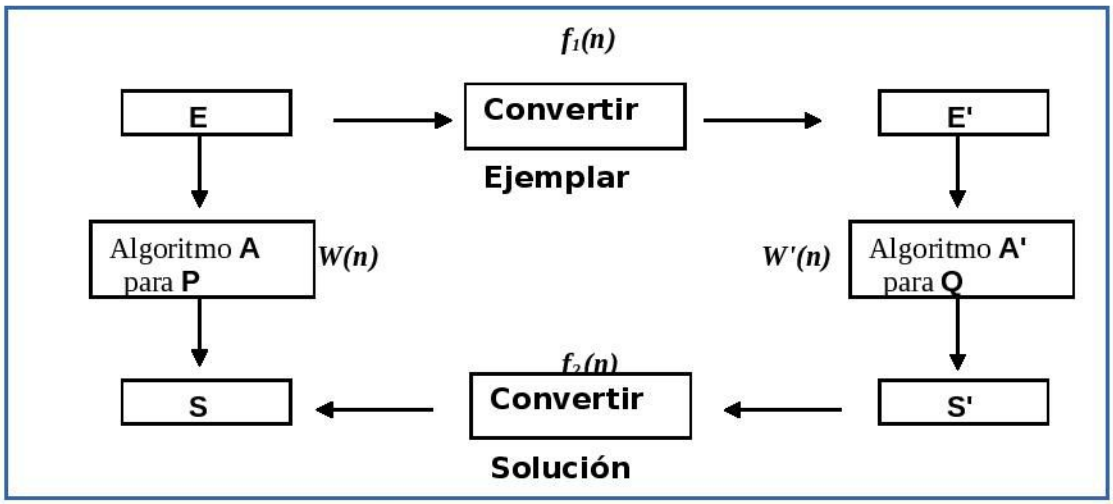


Figura 5. Teorema TLB

Teorema TLB ***(Transformation Lower Bound).*** Suponga que **P** $\propto$ **Q** y que los desempeños computacionales, en el peor de los casos, para los algoritmos que convierten a E en E' y a S´en S son $f_1(n)$ y $f_2(n)$, respectivamente, donde ***n*** es el tamaño del ejemplar E. Entonces,

a) Por cada algoritmo con desempeño computacional $W´(n)$ que solucione **Q**, existe un algoritmo para **P** con desempeño: $W(n) = f_1(n) + W´(n) + f_2(n)$.

b) Si $g(n)$ es una cota mínima sobre la complejidad de **P**, en el peor de los casos, entonces una cota mínima sobre la complejidad de **Q** resulta ser:

$$g'(n) = g(n) - (f_1(n) + f_2(n))$$

Demostración. Se supone que todos los algoritmos y cotas están basados en un mismo modelo de cómputo.

(a) Obvio.

(b) Por Contradicción. Si un algoritmo para **Q** de complejidad menor que

$$g'(n) = g(n) - (f_1(n) + f_2(n))$$

existiese, por (a), se tendría un algoritmo para **P** de complejidad menor que $g(n)$ existe, lo cual contradice el hecho de que $g(n)$ es una cota mínima para la complejidad de **P**. □

Nótese que el algoritmo para **P** que menciona el Teorema TLB resulta ser el algoritmo de simplificación. Así que el algoritmo de simplificación, para el peor de los casos, requiere un número de operaciones de orden

$$W(n) = f_1(n) + W´(n) + f_2(n) + \Theta(n).$$

que es lo que indica el Teorema TLB en el inciso (a).

La Figuras 5 presenta el diagrama de la transformación y la Figura 6 muestra el algoritmo de reducción, ambas figuras ilustran el Teorema de cotas mínimas.

Si los algoritmos de conversión, para los ejemplares y las soluciones, tienen desempeño computacional de orden polinomial, se dice que la **transformación** es **polinomial**.

```
Algoritmo de Simplificación ( E: Ejemplar )
Inicio                                  // Complejidad:
        E' ←−  C_Ej ( E );              //  f1 (n)
        S' ←−  A' ( E' );               //  W' (n)
        S  ←−  C_S ( S' )               //  f2 (n)
        Regresar ( S )                  //  Θ (n)
Fin
```

Figura 6. Complejidad del Algoritmo de Simplificación

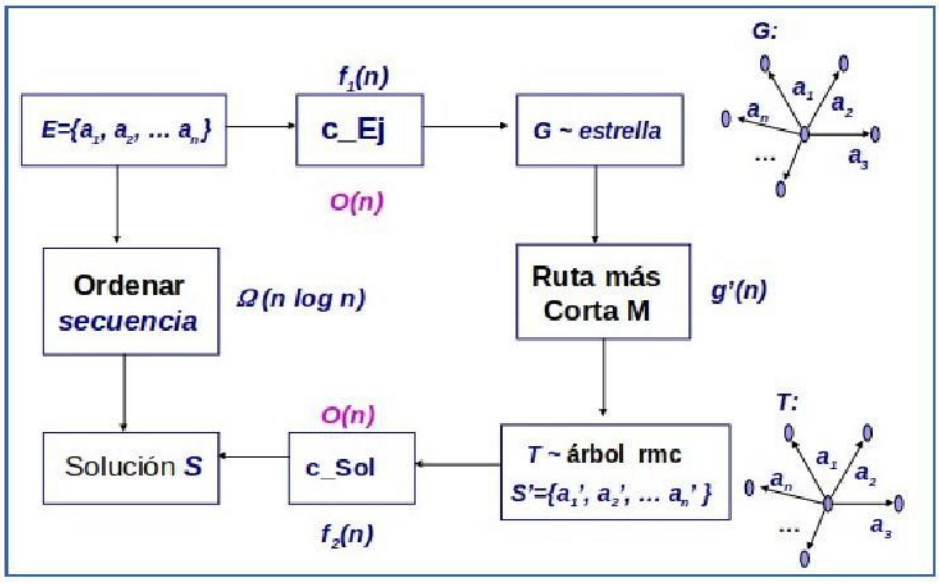
Ejemplo 2. Sea **P** el Problema de ordenamiento y **Q** el Problema de la ruta más corta. Al problema **Q** lo podemos ver como dos subproblemas:

Q_1: Determinar la arborescencia de rutas más corta con raíz v_o.

Q_2: Ordenar los vértices según sus distancias.

Observemos que los ejemplares del problema **P** son listas de datos mientras que los ejemplares para **Q** son gráficas. ¡Hay que transformar listas en gráficas! El truco está en considerar una familia muy especial de gráficas denominada gráficas estrella. La transformación es la siguiente: Dada la lista de datos $L=\{ a_1, a_2, a_3, ..., a_n\}$, un ejemplar para **P**, se construye la gráfica estrella G con conjunto de vértices $V=\{ v_o, v_1, v_2, ..., v_n\}$ y conjunto de arcos, con costos, $A= \{ (v_o,v_1,a_1), (v_o,v_2,a_2), (v_o,v_3,a_3) , ... , (v_o,v_n, a_n) \}$:

Para solucionar el subproblema Q_1 utilizaremos el Algoritmo de Dijkstra para rutas más cortas. El Algoritmo de Dijkstra aplicado a la gráfica $G(V, A)$ a partir de v_o, pondrá en una cola de prioridades a los vecinos de v_o, es decir a todos los demás vértices e irá tomando, en cada iteración, el de menor peso. Así que para solucionar el subproblema Q_2 basta con ir almacenando en una cola simple **q** el dato que obtiene, en cada iteración, de la cola de prioridades. Entonces esta modificación del algoritmo de Dijkstra regresa el árbol de rutas más cortas enraizado en y la cola **q**, que contiene los costos a_i en orden. De esta forma, la transformación de las soluciones resulta trivial, ya que basta regresar la cola simple **q**. La siguiente figura ilustra la transformación.



3. Reducciones en Tiempo Polinomial.

En esta sección nos restringiremos a trabajar con los **Problemas de Decisión**, esto es, consideraremos únicamente problemas cuya respuesta es **sí** o **no**. Esta restricción hace la discusión y la teoría más simple. La mayoría de los problemas pueden ser convertidos fácilmente a problemas de decisión.

Un problema de decisión puede ser visto como un problema de reconocimiento de lenguaje, *language recognation problem*.

Sea U el conjunto (universo) de todas las posibles entradas de un problema de decisión. Sea $L \subseteq U$ el conjunto de toas las entradas para las cuales la respuesta al problema es **sí**. Al conjunto L se le denomina: ***el lenguaje correspondiente al problema***, usaremos indistintamente problema y lenguaje. El problema de decisión consiste en reconocer si una entrada dada pertenece o no al conjunto L.

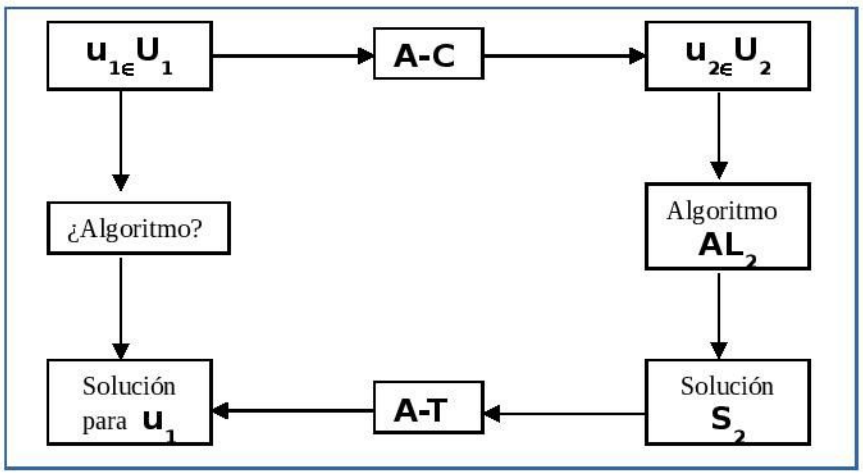
Ahora introducimos la noción de transformación en tiempo polinomial entre lenguajes.

Definición 5. Sean L_1 y L_2 dos lenguajes para los cuales los espacios de entradas U_1 y U_2. El lenguaje L_1 es **polinomialmente reducible** a L_2 su existe un algoritmo que en tiempo polinomial convierte cada entrada $u_1 \in U_1$ en una entrada $u_2 \in U_2$ tal que entrada $u_1 \in L_1$ si y sólo si entrada $u_2 \in L_2$. El algoritmo es polinomial en el tamaño de la entrada u_1.

Suponemos que la notación de tamaño está bien definida en los espacios entrada U_1 y U_2. En particular, el tamaño de entrada u_2 también es polinomial con respecto al tamaño de u_1.

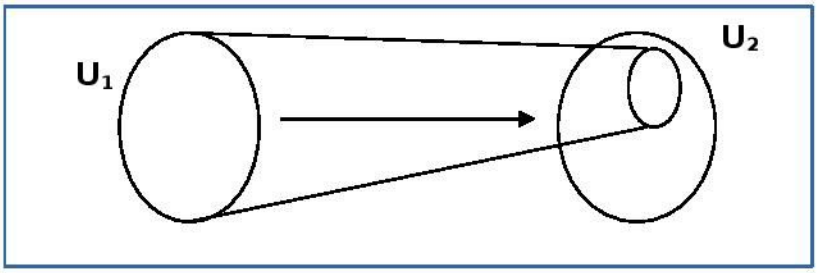
El algoritmo mencionado en la definición anterior convierte o ***transforma*** un problema en otro. Si tenemos un algoritmo para L_2, entonces podemos componer los dos algoritmos para producir un algoritmo para L_1. Denotamos a tal conversión o transformación como **A-C**, al algoritmo para L_2, como AL_2 y a la transformación de la solución como **A-T.**

Dada una entrada arbitraria $u_1 \in U_1$ podemos usar **A-C** para convertir u_1 en u_2 entonces usamos el algoritmo AL_2 para determinar si u_2 pertenece a L_2, lo cual nos indicará si u_1 pertenece a L_1.



Teorema 1. Si L1 es polinomialmente reducible a L_2 y existe un algoritmo polinomial para L_1, entonces existe un algoritmo polinomial para L1. □

La noción de trasformación (reducción) **no** es simétrica. Si L_1 es polinomialmente reducible en L_2 no implica que L_2 sea polinomialmente reducible en L_1. Esta asimetría se da porque la definición de reducibilidad requiere que ***cualquier*** entrada de L_1 pueda ser convertida en ***una*** entrada para L_1 pero no al revés. Es posible que las entradas de L_2 involucradas en la reducción son tan sólo una pequeña parte de todas las posibles entradas para L_2. Así pues, si L_1 es polinomialmente reducible en L_2 entonces se considera que L_2 es un problema ***más difícil.***



Definición 6. Dos lenguajes, L_1 y L_2, son **polinomialmente equivalentes**, o simplemente **equivalentes,** si cada uno es polinomialmente reducible en el otro.

En particular, todos los problemas no trivialmente tratables son equivalentes ya que todos tienen algoritmos de tiempo polinomial. La relación de ser polinomialmente reducible es transitiva.

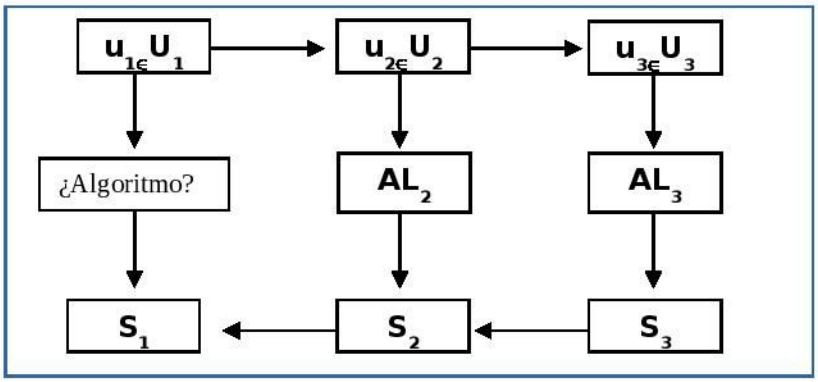


Figura 7: Propiedad de Transitividad

Teorema 2. Si L_1 es polinomialmente reducible en L_2 y L_2 es polinomialmente reducible en L_3 entonces L_1 es polinomialmente reducible en L_3.

Prueba. Podemos hacer la composición entre dos algoritmos de conversión y formar una transformación de L_1 a L_3. Υνα εντραδα u_1 ¤ L_1 πυεδε σερ χοντωερτιδα πριμερο εν υνα εντραδα u_2 ¤ L_2 λα χυαλ σε χοντωιερτε α υνα u_3 ¤
L3. Ya que estamos usando reducciones polinomiales, la composición también es polinomial. Finalmente, el resultado es un algoritmo de conversión en tiempo polinomial. □
La Figura 7 ilustra este teorema.

El objetivo de este método es buscar problemas equivalentes cuando no es posible encontrar algoritmos eficientes, tiempo polinomial. Cuando tenemos un problema que no podemos resolver eficientemente, tratamos de ver si éste es equivalente a otros problemas conocidos como difíciles. La clase de los Problemas NP-Completos abarca cientos de problemas de este tipo, que resultan ser equivalentes.

4. El No Determinismo y El Teorema de Cook.

La teoría de los problemas NP-Completos inicia con el famoso Teorema de Cook en 1971. Es parte de una teoría denominada **Complejidad Computacional**. Limitaremos la discusión a algunas partes que nos ayuden a ***usar*** la teoría.

No se describió con gran detalle ni en términos precisos y matemáticos la noción de algoritmo. Esto no ha sido importante para describir algoritmos prácticos, con un número razonable de pasos y tan grandes como los pueda soportar una computadora. La definición precisa de un algoritmo es muy importante para probar cotas mínimas. El modelo fundamental de la computación es la Máquina de Turing, otro modelo – comúnmente usado – es el de la máquina de acceso aleatorio, *Random Access Machine*. Ambos modelos, e incluso otros, resultan ser equivalentes para los propósitos de este material. Ya que podemos transformar un algoritmo de un modelo a otro sin alterar el tiempo de ejecución más que por un factor constante. El Teorema de Cook, por ejemplo, fue probado usando la máquina de Turing, pero es válido también para otros modelos. No usaremos un modelo especifico para no perdernos en detalles innecesarios en este curso introductorio.

4.1 El No Determinismo

Ahora discutiremos la noción de No Determinismo, la cual no es muy intuitiva. Esto lleva a la gente a creer que la Teoría de los Problemas NP-Completos es algo misterioso. Debemos pensar en un algoritmo no determinista como una noción abstracta y no como una meta realista. El concepto de no determinismo es más importante para el desarrollo de la teoría y la explicación de la existencia de la clase de problemas NP-Completos que para las técnicas que usa la teoría.

Un **Algoritmo No Determínistico** tiene ***todas*** las operaciones clásicas de uno determínistico y posee *una primitiva* muy importante denominada **elección-nd**, *nd-choice*, que es usada para manipular selecciones de una forma poco usual. La primitiva elección-nd también es llamada primitiva adivinadora, *guessing*.

La primitiva elección-nd está asociada con un número fijo de selecciones, tal que para cada elección el algoritmo sigue una diferente ruta computacional. Supondremos, sin pérdida de generalidad que el número de elecciones es siempre dos.

Sea L un lenguaje que nos sirva para reconocer. Dada una entrada x, un algoritmo no determínistico efectúa pasos determínisticos normales intercalando el uso de la primitiva elección-nd y al final decide si acepta o no a la entrada x. La diferencia clave entre un algoritmo determínistico y uno no determínistico esta en la forma en que ellos reconocen el lenguaje.

Definición 7. Un **algoritmo reconoce a un lenguaje L** si la siguiente condición se satisface: Dada una entrada x, es posible convertir cada elección-nd encontrada durante la ejecución del algoritmo en una elección real tal que la salida del algoritmo acepte a x si y sólo si $x \in L$.

En otras palabras, el algoritmo debe proveer al menos un posible camino para las entradas pertenecientes al lenguaje L para que sean salidas aceptables, y no debe generar o proveer ningún camino para que entradas que no pertenezcan a L lleguen a ser salidas aceptables.

Una entrada $x \in L$ puede tener varias rutas hacia salidas razonables. Requerimos que el algoritmo tenga al menos una *buena* secuencia de elecciones para $x \in L$. Por otro lado, para cada entrada $x \notin L$, debemos tener una salida ***rechazada*** sin importar cual selección sea sustituida por la primitiva elección-nd. El tiempo de corrida para una entrada $x \in L$ es la longitud de la mínima ejecución sobre las secuencias de elecciones que lleve a una salida ***aceptable***. El desempeño computacional de una algoritmo no determínistico se refiere al tiempo de ejecución en el peor de los casos, para las entradas $x \in L$, las entradas que no pertenecen a L son ignoradas.

Ejemplo de Algoritmo No Determínistico.

Un conjunto ***M*** de aristas es llamado un **apareamiento** o **acoplamiento** si cualesquiera dos aristas en ***M*** no tienen un vértice en común. Un apareamiento ***M*** en una subgráfica es **máximo** si no existe un otro apareamiento ***M'*** tal que $|M'| > |M|$. Un apareamiento en una gráfica es perfecto si cada vértice de la gráfica es incidente a una arista en el apareamiento.

Consideremos el problema de decidir si una gráfica dada ***G = (V,A)*** tiene un apareamiento perfecto. Ahora damos un Algoritmo no determínistico para este problema.

1. Mantenemos un conjunto ***M*** de aristas, el cual inicialmente está vacío.
2. Examinamos todas las aristas de ***G***, una arista ***e*** a la vez.
3. Usamos la primitiva elección-nd para decidir si incluimos o no a la ***e*** arista en el apareamiento ***M***.
4. Una vez examinadas todas las aristas de la gráfica, verificamos si ***M*** es un apareamiento perfecto.

La verificación puede hacerse en tiempo lineal, pues sólo tenemos que determinar si ***M*** contiene exactamente $|V|/2$ aristas y si cada vértice incide únicamente en una arista de ***M***. La salida del algoritmo será ***sí***, si ***M*** es un acoplamiento perfecto y ***no*** en otro caso. Este resulta ser un algoritmo no determínistico correcto para el problema del apareamiento perfecto, ya que:

1) Si un apareamiento perfecto existe, entonces hay una secuencia de elecciones que llevará a la salida exitosa en ***M***.
2) El algoritmo dice si solamente si la existencia de un apareamiento perfecto fue probada, por la verificación.

Observemos que la primitiva elección-nd ayuda a seleccionar un candidato a ser conjunto que forme el apareamiento perfecto, no construyó el apareamiento perfecto.

De hecho ***no*** estamos solucionando el problema, sólo ***estamos proponiendo un candidato a ser la solución.*** Si el candidato propuesto resulta ser la solución, el algoritmo de verificación deberá ser capaz de reconocerlo como una solución al problema. Los algoritmos determínisticos son muy poderoso pero su fuerza no es ilimitada. No todos los problemas pueden ser resueltos eficientemente por algoritmos no determínisticos.

Ejemplo. Suponga que el problema consiste en determinar si el máximo apareamiento en una gráfica es de tamaño ***k***. Usamos un algoritmo no determínistico para encontrar un máximo acoplamiento de tamaño ***k***, si existe. Pero no es fácil determinar, aun no deterministicamente, si no hay un acoplamiento mayor.

Definición 8. La clase de problemas para los cuales existe un algoritmo no determínistico cuyo desempeño computacional es polinomial, en el tamaño de la entrada, es llamada **Clase NP.**

Parece razonable creer que los algoritmos no determínisticos son más poderosos que los determínisticos, pero ¿en realidad lo son? Una manera de probarlo es exhibiendo un problema **NP** que no esté en **P**. Nadie ha sido capaz de hacerlo. En contraste, si queremos probar que las dos clases son iguales, **P = NP**, entonces tenemos que mostrar que todo problema que pertenece a **NP** puede ser resuelto por un algoritmo determínistico en tiempo polinomial. Nadie lo ha probado tampoco, además pocos creen que sea cierto. El problema de determinar la relación entre **P** y **NP** es conocido como **El Problema P = NP**.

Ahora definiremos dos clases, las cuales no sólo contienen importantes problemas (todos equivalentes unos con otros) que no se sabe si están en **P**, pero también contienen los problemas más difíciles en **NP**.

Definición 9. Un problema **X** es llamado **Problema NP-Difícil,** *NP-Hard*, si cada problema en **NP** es polinomialmente reducible en **X**.

Definición 10. Un problema **X** es llamado **Problema NP-Completo,** *NP-Complete*, si

1) el problema **X** pertenece a **NP**

2) el problema **X** es **NP-Difícil**

La definición de **NP-Dificultad,** *NP-Hardness*, implica que si se prueba que cualquier problema **NP-Difícil** pertenece a **P**, entonces la prueba debería implicar que **P=NP**. Cook en 1971 demuestra que sí existen problemas NP-completos. En particular, exhibe un problema que describiremos brevemente después. Una vez que encontramos un problema NP-Completo probar que otros también lo son, llega a ser un trabajo fácil. Dado un problema **Y** es suficiente probar que el problema de Cook, o cualquier otro problema NP-completo, es polinomialmente reducible a **Y**.

Lema 1. Un problema **X** es un problema **NP-Completo** si

1) **X** pertenece a **NP**

2') es polinomialmente reducible a **X,** para algún problema **Y** que sea **NP-completo.**

Prueba. Por la condición (2) de la definición de **NP-C**ompleto, cada problema **NP** es polinomialmente reducible a **X** y tal relación es transitiva; así cada problema **NP** es polinomialmente reducible a **X**, también.

Es mucho más fácil probar que dos problemas son polinomialmente reducibles que probar la condición (2) de la definición directamente. Con este lema Cook encontró un gancho para la teoría completa. Como se han encontrado más y nuevos problemas que son NP-completos, tenemos más oportunidades para probar la condición (2').

Poco después de que Cook dio a conocer su resultado, Karp en 1972 encontró y demostró 24 importantes problemas NP-completos. Desde entonces, a la fecha, cientos – quizá miles – de problemas han sido clasificados como NP-completos.

En el siguiente capítulo presentamos ejemplos de problemas NP-completos y sus respectivas pruebas y se mencionan otros problemas sin prueba. Usualmente, la parte más difícil de la demostración es la verificación de la condición (2) o (2').

4.2 El Teorema de Cook

Ahora describimos el problema que Cook probó como NP-Completo y mencionaremos la idea principal de la prueba. El problema es conocido como **SAT**, *Satisfiability*.

Sea **S** una expresión lógica en **Forma Normal Conjuntiva**, *Conjunctive Normal Form*, **CNF.** Esto es, **S** es el producto (operación lógica **and**) de múltiples sumas (operaciones **or**). Cualquier expresión lógica puede ser transformada a CFN.

Ejemplo. $S=(x+y+z^{-}) \cdot (x^{-}+y+z^{-}) \cdot (x^{-}+y^{-}+z^{-})$, donde la adición y multiplicación corresponden a las operaciones lógicas **or** y **and**. La variable x^{-} representa la negación de x. Cada variable tiene valor $0 \approx False$, falso, o $1 \approx True$, verdadero.

Definición 11. (Satisfiable). Se dice que una expresión lógica **se satisface** si existe una asignación de valores de verdad, ceros y unos, a sus variables tal que el valor de la expresión sea verdadero, 1.

El **Problema SAT** consiste en determinar si una expresión dada se satisface, sin necesidad de encontrar la asignación que la satisface. Para el ejemplo, S si se satisface ya que la asignación $x=1, y=1$ y $z=0$ la satisface.

Definición 12. (Asignación de Verdad). A una asignación de ceros y unos, valores de verdad, para las variables de una expresión lógica se le llama **Asignación de Verdad,** *truth assignment***.**

El problema del SAT está en NP ya que es posible adivinar una asignación de verdad y verificar en tiempo polinomial que la expresión se satisface.

La idea de la prueba de que el SAT es NP-difícil es que una máquina de Turing, aún una no determinista, y todas sus operaciones sobre una entrada dada, puede ser descrita como una expresión lógica. Por descrita queremos decir que la expresión será satisfecha si y sólo si la máquina de Turing termina con un estado aceptable para tal entrada. Esto no es fácil de hacer, pues tal expresión puede llegar a ser muy complicada, aunque su tamaño sea polinomial con respecto al número de pasos que la máquina de Turing realiza. Por lo tanto, cualquier algoritmo NP puede ser descrito por un ejemplar del problema SAT.

Teorema 3: Teorema de Cook. El problema SAT es NP-completo.

Bibliografía.

[1] M.R. Garey & D.S. Johnson
Computer and Intractability, A Guide to the Theory of NP-Completness
WH. Freeman, USA, 1979

[2] J. Kingston
Algorithms and Data Structures: Design, Correctness, and Analysis,
Addison Wesley, Pu. Co., USA 1990.

[3] U. Manber
Introduction to Algorithms. A Creative Approach.
Addison Wesley, Pu. Co., USA 1989.

[4] R. Neapolitan & K. Naimipour
Foundations of Algorithms,
DC Heath and Co., USA, 1996