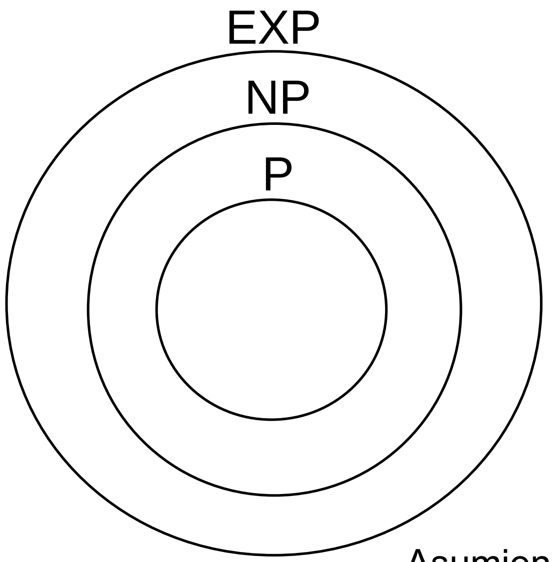
Clase Teórica 6. Problemas NP-completos.

- La clase pasada llegamos al siguiente nivel de detalle del mapa de la complejidad temporal:

L ∈ P sii existe una MTD que lo acepta en tiempo poly(n). O en otras palabras, la **construcción** de una solución para L requiere tiempo polinomial.

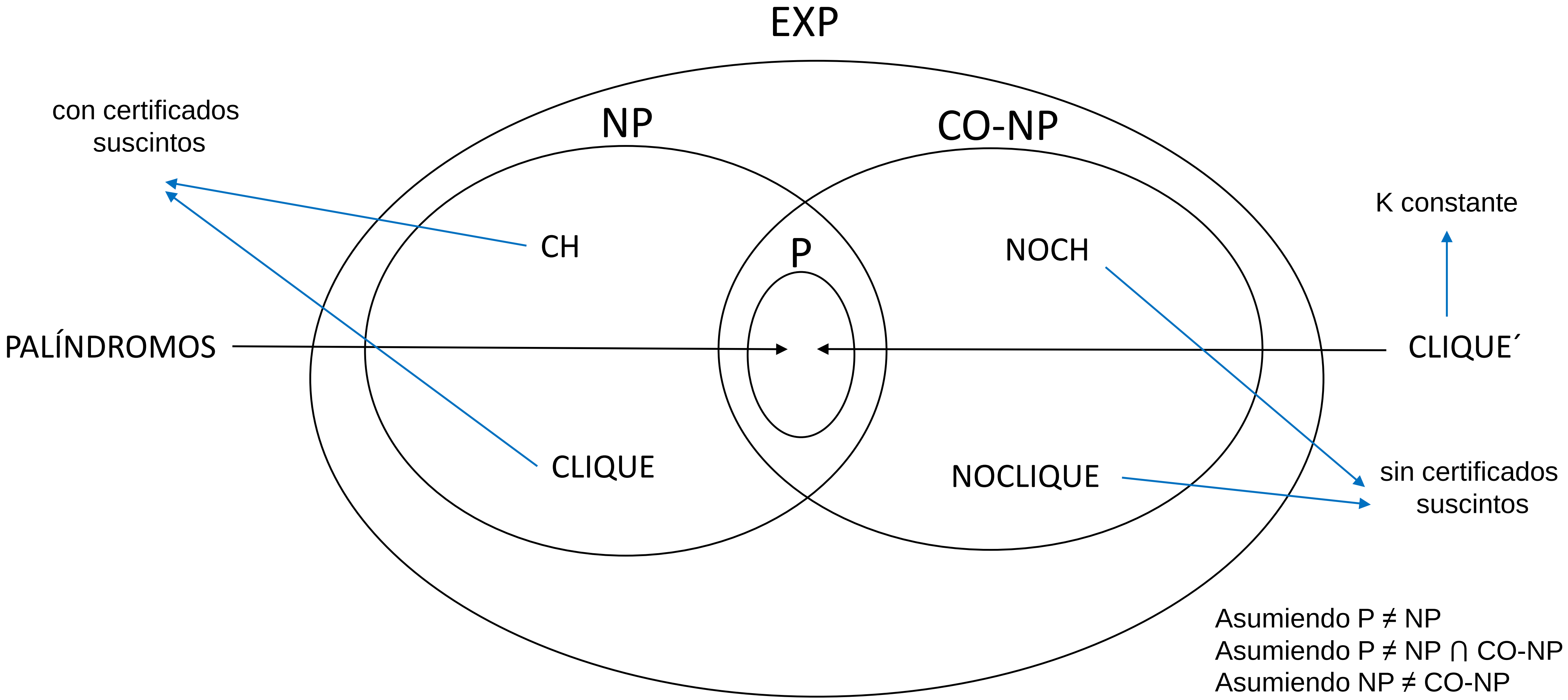


Asumiendo P ≠ NP

L ∈ NP sii existe una MTN que lo acepta en tiempo poly(n). O en otras palabras, la **verificación** de una solución para L requiere tiempo polinomial. También vimos que sólo los lenguajes de NP tienen **certificados suscintos**.

- Hemos encontrado primeros ejemplos en NP que **no estarían** en P: el problema del divisor que termina en 3, el problema CH (grafos con un circuito de Hamilton) y el problema CLIQUE (grafos con un subgrafo completo de tamaño K).
- Iremos refinando y poblando el mapa. Volveremos a recurrir a las **reducciones**.

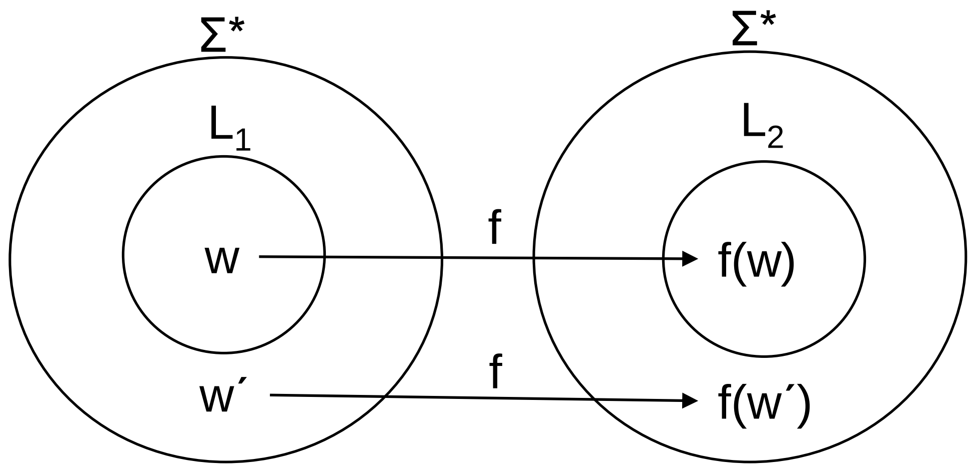
- Vimos que a diferencia del mapa de la computabilidad, en el mapa de la complejidad temporal **hay muchas asunciones** (preguntas abiertas que al día de hoy no pueden probarse):



- Vimos que $P \subseteq NP \cap CO\text{-}NP$ (P es cerrado con respecto al complemento). También se prueba que $NP \cup CO\text{-}NP \subset EXP$.

Reducciones Polinomiales

- Vamos a volver a recurrir a las reducciones, ahora **reducciones polinomiales**: las funciones de reducción f se computan por una MT M_f en tiempo polinomial:



La MT M_f que computa la función de reducción f lo hace en tiempo polinomial

- Se define que **f ∊ FP**, es decir que f se computa en tiempo poly(n). Se usa FP en lugar de P para diferenciar funciones de lenguajes.

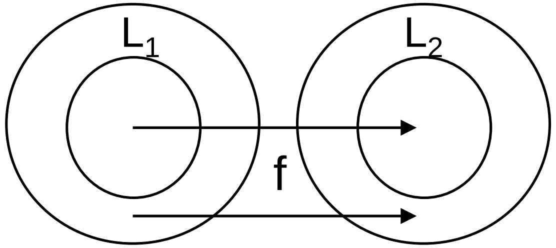
- La expresión $\mathbf{L_1\ \alpha_P\ L_2}$ establece que existe una reducción polinomial de L_1 a L_2.

- La utilidad de las reducciones polinomiales es similar a lo estudiado en Computabilidad, como veremos a continuación.

Teorema.

(a) Si $L_1 \; \alpha_P \; L_2$, entonces $L_2 \in P \longrightarrow L_1 \in P$

(b) Si $L_1 \; \alpha_P \; L_2$, entonces $L_2 \in NP \longrightarrow L_1 \in NP$

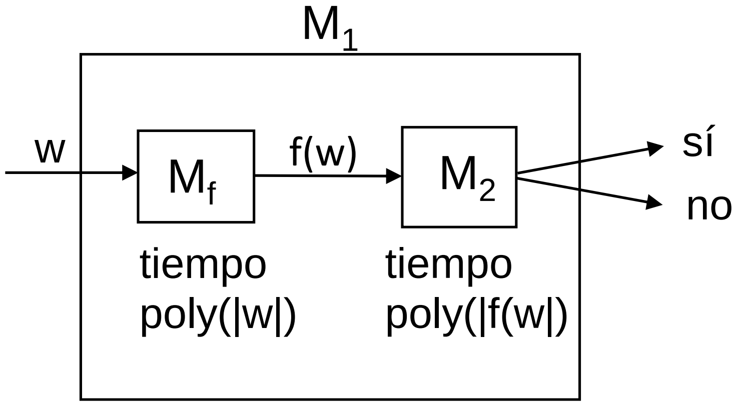


- Notar que es la misma idea que en Computabilidad, pero en lugar de tratar con R y RE tratamos con P y NP.

Prueba. Veamos el caso (a). Por hipótesis, $L_1 \; \alpha_P \; L_2$ y $L_2 \in P$, es decir que existe una MT M_f que reduce L_1 a L_2 en tiempo poly(n) y existe una MT M_2 que acepta L_2 en tiempo poly(n). Por lo tanto, la siguiente MT M_1 acepta L_1 también en tiempo poly(n), y así se cumple que $L_1 \in P$:

$L(M_1) = L_1$

$L(M_2) = L_2$



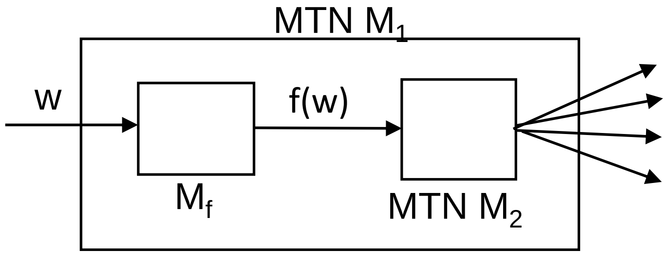
M_1 acepta L_1 (lo vimos en la parte de Computabilidad). **Además lo hace en poly(|w|) pasos:**

✓ Dado w, M_f hace **poly(|w|) pasos**, y así genera un output f(w) que mide poly(|w|) (¿por qué?).

✓ M_2 hace poly(|f(w)|) pasos, es decir poly(poly(|w|)) = **poly(|w|) pasos**.

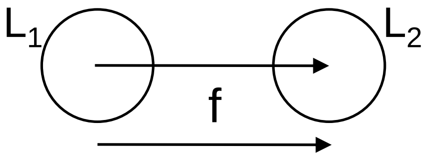
✓ Sumando, nos queda que M_1 hace poly(|w|) + poly(|w|) pasos, es decir **poly(|w|) pasos**.

El caso (b), es decir, **si $L_1 \alpha_P L_2$ y $L_2 \in$ NP entonces $L_1 \in$ NP**, queda como ejercicio (Es la misma prueba que en el caso (a), considerando ahora cada una de las computaciones de la MTN M_2. De esta manera, la MTN M_1 tendrá todas sus computaciones de tiempo poly(|w|).)



- Similar a lo visto en Computabilidad, se puede recurrir a $L_1 \alpha_P L_2$ para:

  ✓ Obtener una **solución eficiente de L_1** conociendo una **solución eficiente de L_2**.
  ✓ O bien demostrar **no pertenencia** a las clases P o NP (corolario del teorema anterior):
  (a) Si $L_1 \alpha_P L_2$, entonces $L_1 \notin$ P $\longrightarrow$ $L_2 \notin$ P (en vez de $L_2 \in$ P $\longrightarrow$ $L_1 \in$ P).
  (b) Si $L_1 \alpha_P L_2$, entonces $L_1 \notin$ NP $\longrightarrow$ $L_2 \notin$ NP (en vez de $L_2 \in$ NP $\longrightarrow$ $L_1 \in$ NP).
  Es decir, se mantiene la relación establecida por las reducciones, ahora considerando el tiempo: **si $L_1 \alpha_P L_2$, L_2 es "tan o más difícil" que L_1** (p.ej. no puede ser $L_2 \in$ P y $L_1 \notin$ P).



- También se mantienen las mismas propiedades demostradas en Computabilidad:
  ✓ **Reflexividad**. $L_1 \alpha_P L_1$. La función identidad se computa en tiempo lineal (¿por qué?).
  ✓ **Transitividad**. Si $L_1 \alpha_P L_2$ y $L_2 \alpha_P L_3$, entonces $L_1 \alpha_P L_3$ (composición de funciones poly(n)).
  ✓ **Asimetría**. Vimos antes que todo L recursivo cumple $L \alpha L_U$ y no $L_U \alpha L$. La reducción es lineal (transforma un input w en un output (<M>,w)). Así, para todo lenguaje L de EXP = R se cumple $L \alpha_P L_U$ y no $L_U \alpha_P L$ (naturalmente L_U es más difícil que todo lenguaje de EXP).

<u>Ejemplo de reducción polinomial</u>.

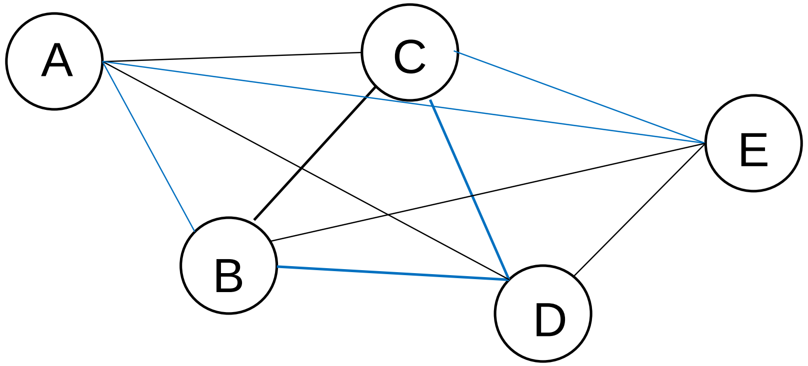
CH = {G | G es un grafo que tiene un circuito de Hamilton}

TSP = {(G,K) | G es un grafo completo con arcos con longitudes y tiene un circuito de Hamilton con una longitud ≤ K}

- TSP (por *travelling salesman problem*) representa el problema del viajante de comercio, en que un vendedor debe recorrer varias ciudades y volver a la inicial optimizando su recorrido:

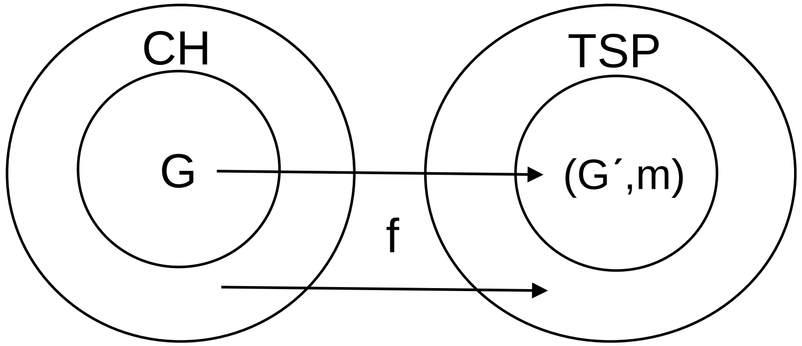
Por ejemplo, sea el siguiente grafo G, y sea K = 600 Km. Si la suma de los arcos (A,B), (B,D), (D,C), (C,E) y (E,A) es 550 Km, entonces el vendedor puede recorrer A, B, D, C, E, A en menos de 600 Km, y así el par (G, K) ∈ TSP.

grafo G



- Vamos a probar que **existe una reducción polinomial de CH a TSP.**

1. Definición de la función de reducción f.



f(G) = (G´,K), tal que:

- ✓ G´ tiene los mismos vértices que G.
- ✓ G´ es un grafo completo (todos sus vértices están conectados entre sí). Los arcos que están en G miden 1 y los que no miden 2.
- ✓ K = m, es decir que K es la cantidad de vértices de G.
- ✓ Y si G no representa un grafo, entonces f(G) = 1, es decir que f le asigna una cadena inválida.

2. La función f es total computable y se computa en tiempo poly(n).

- ✓ Validar la sintaxis de G vimos que tarda tiempo $O(n^2)$.
- ✓ Escribir el V de G en G´ tarda tiempo O(n).
- ✓ Escribir el E de G en G´ agregando a cada arco una longitud de 1 tarda tiempo O(n) + O(1) = O(n).
- ✓ Agregar los arcos faltantes de E de G en G´ asociando a cada arco una longitud de 2 tarda tiempo $O(|V|^2.|E|) + O(1) \leq O(|G|^3) = O(n^3)$.
- ✓ Escribir K, es decir la cantidad m de vértices de G, tarda tiempo O(n). Y escribir el 1 tarda tiempo O(1).

Así, la MT M_f que computa la función f tarda tiempo $O(n^2) + O(n) + O(n) + O(n^3) + O(n) = O(n^3)$.

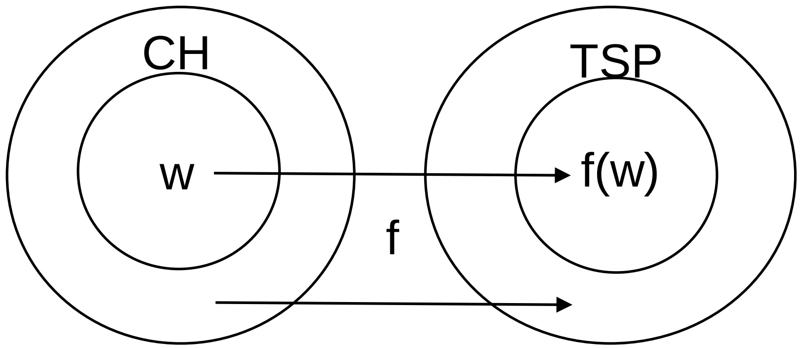
3. Prueba de que G ∈ CH ↔ f(G) ∈ TSP.

G ∈ CH ↔ G tiene un circuito de Hamilton ↔ G´ tiene un circuito de Hamilton de longitud ≤ m ↔ (G´, m) ∈ TSP.

- Hemos probado CH α_P TSP.

- Así, como indicamos antes, **TSP es "tan o más difícil" que CH.**

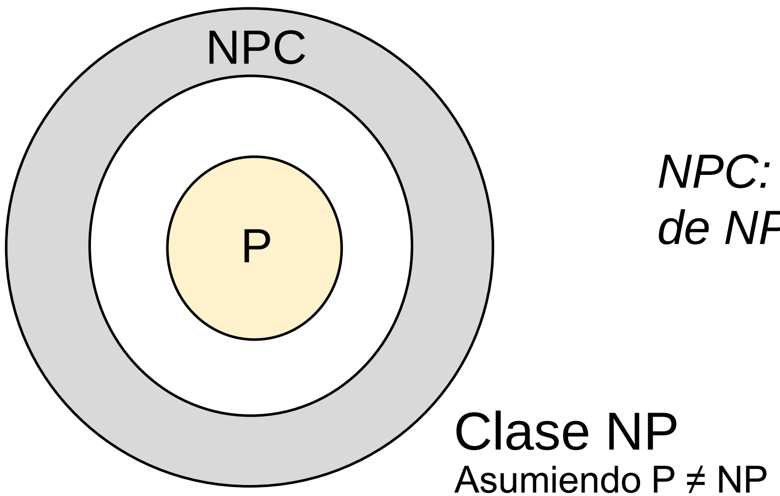
- La clase pasada dijimos que CH **no estaría** en P. Por lo tanto, **tampoco TSP estaría** en P (si TSP $\in$ P, entonces CH $\in$ P).



- El **no estaría** en P proviene de la imposibilidad (actual) de probar que efectivamente no existe una MT que resuelva eficientemente CH, como así también es el caso de TSP, CLIQUE, etc.

- De todos modos, hay una manera **más efectiva, más contundente, sin ser una prueba formal** (por la pregunta abierta ¿P $\subset$ NP?), de "condenar" en la práctica a un problema de NP a no estar en P (a menos que P = NP). Esto ocurre cuando se demuestra que el problema es **NP-completo**:

Los problemas NP-completos son los más difíciles, desde el punto de vista de la complejidad temporal, de la clase NP.



NPC: clase de los problemas de NP que son NP-completos.

Problemas NP-completos

Definición. L ∈ NPC (o bien L es NP-completo) sii:

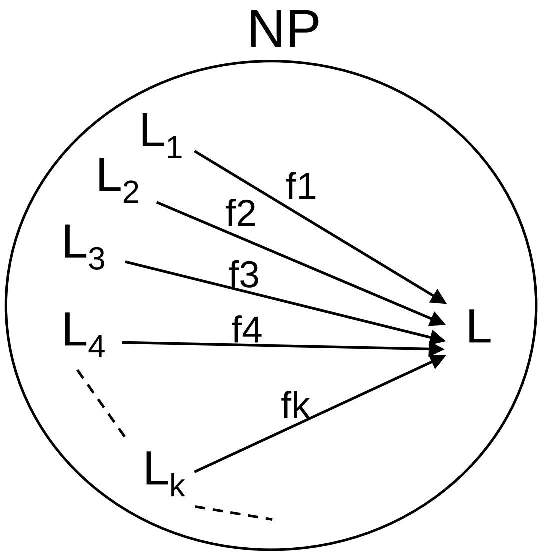
(a) L ∈ NP

(b) Para todo L´ ∈ NP se cumple L´ α_P L (se dice en este caso que L es NP-difícil).

- En otras palabras, **L es NP-completo sii está en NP y además todos los lenguajes de NP se reducen polinomialmente a él.** O lo mismo: **L está en NP y además es NP-difícil**.

- Notar entonces que por propiedad de las reducciones polinomiales, **si un lenguaje L es NP-completo entonces es "tan o más difícil" que cualquier lenguaje L_i de NP.** Integra la subclase de problemas más difíciles de la clase NP:

Todas las funciones f_i están en FP, es decir que se computan en tiempo polinomial.

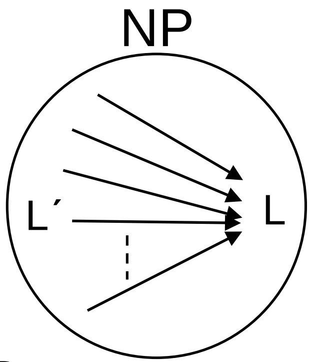


Veamos a continuación qué significa que un lenguaje L sea NP-completo, en relación a su pertenencia a la clase P. Adelantándonos: ¿qué sucedería si L estuviera en P?

Teorema. Si L ∈ NPC y L ∈ P, entonces P = NP.

Prueba. P ⊆ NP (una MTD es un caso particular de una MTN). Veamos que también NP ⊆ P:

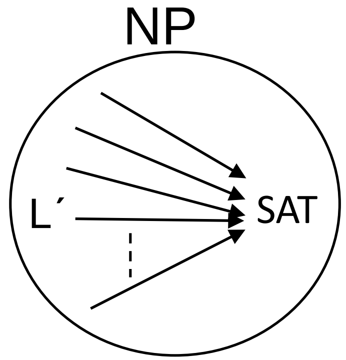
- ✓ Sea L´ ∈ NP. Veamos que también, asumiendo las hipótesis, vale L´ ∈ P.
- ✓ Como L es NP- completo, entonces por definición: L´ α_P L.
- ✓ Y como L ∈ P, entonces por propiedad de α_P se cumple que también L´ ∈ P.
- ✓ Como L´ es cualquier lenguaje de NP, llegamos entonces a que NP ⊆ P.



- En palabras, los problemas NP-completos están "condenados" a no estar en P, a menos que P = NP. **Asumiendo P ≠ NP, un problema NP-completo no está en P.**

- Existen númerosísimos problemas en la clase NPC. Históricamente Cook (EEUU), e independientemente Levin (Rusia), en 1971, probaron que **el problema de satisfactibilidad en la lógica proposicional (SAT) es NP-completo:**

SAT = {φ | φ es una fórmula booleana (sin cuantificadores) satisfactible}

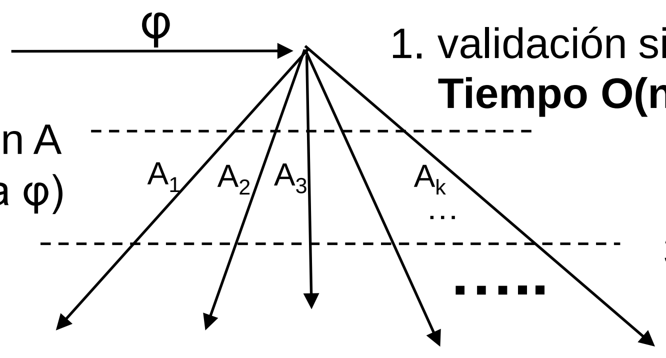


- **El teorema de Cook-Levin** establece entonces:
  (a) SAT ∈ NP.
  (b) Todos los lenguajes de NP se reducen polinomialmente a SAT.

- La prueba de que SAT está en NP es sencilla. La siguiente MTN acepta SAT en tiempo poly(n):

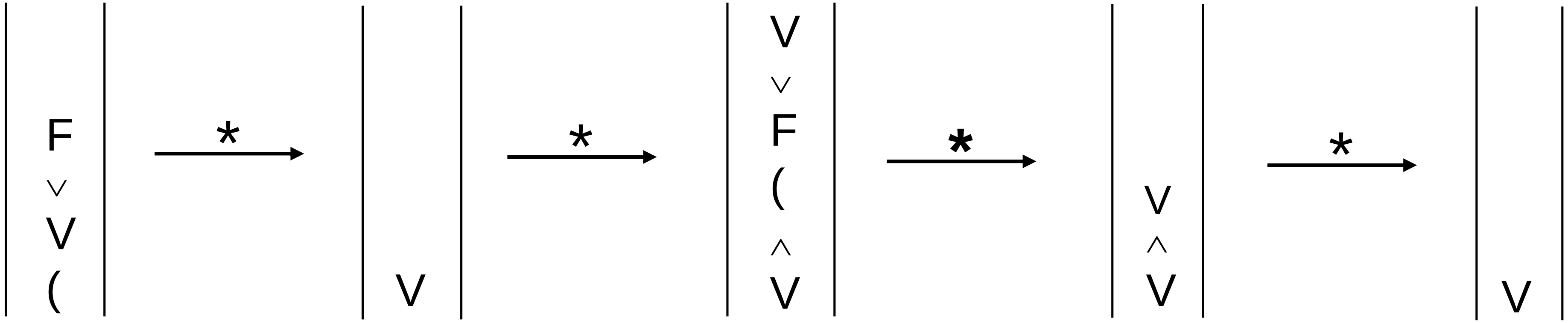
1. validación sintáctica del input
   Tiempo O(n) (ejercicio)

2. Generación de una posible solución A
   (asignación de valores de verdad a φ)
   Tiempo O(n) (ejercicio)



3. Verificación de si efectivamente
   la asignación A_i satisface φ
   Tiempo O(n²)

- La verificación de si una asignación A satisface una fórmula φ se puede hacer empleando una pila. Por ejemplo, sean $\varphi = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$ y A = (V, F, F, V). En el peor caso, por cada símbolo leído se puede recorrer la pila completa:

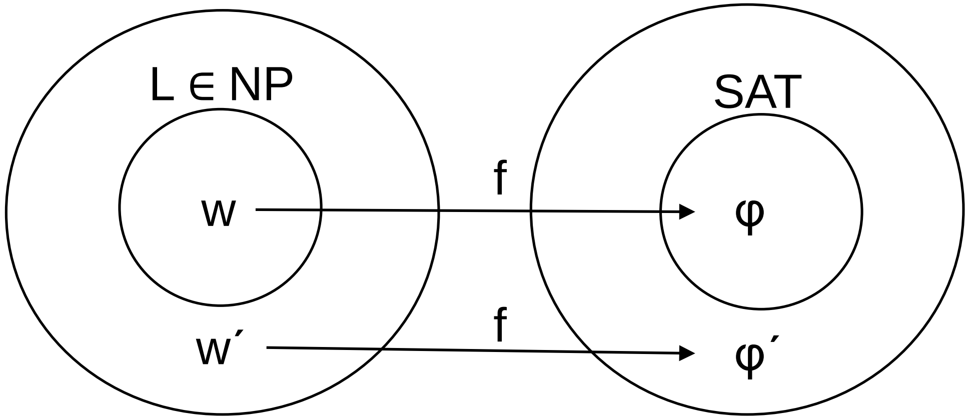


- De esta manera el tiempo de verificación es $O(|\varphi|.|\varphi|) = O(n^2)$.

- La prueba de que todos los lenguajes de NP se reducen polinomialmente a SAT es muy ingeniosa, en un sentido similar a la que utilizó Turing en 1936 para probar que la lógica de predicados no es decidible (reduciendo desde HP). Se puede ver en el libro de la materia.

- La idea general es:

Dada una MTN M que acepta L en tiempo poly(n):



se construye la fórmula booleana φ en términos de M y el input w

$w \in L \leftrightarrow \varphi$ es satisfactible. La función de reducción f consiste en construir, dado un input w y la MTN M que acepta L en tiempo poly(n), una fórmula booleana φ a partir de las formas que pueden tener las computaciones de M y de la forma de w. Sólo cuando existe una computación de M que acepta w se va a cumplir que la fórmula φ construida es satisfactible. Además, f se computa en tiempo poly(n) (se basa en que las computaciones de M tienen poly(|w|) pasos).

- Otro problema NP-completo clásico es $L_{U\text{-}K}$, el problema de **pertenencia acotada universal**: $L_{U\text{-}K} = \{(\langle M \rangle, w, 1^k) \mid M \text{ acepta } w \text{ en } k \text{ pasos}\}$. La prueba es más sencilla (ver el libro de la materia).

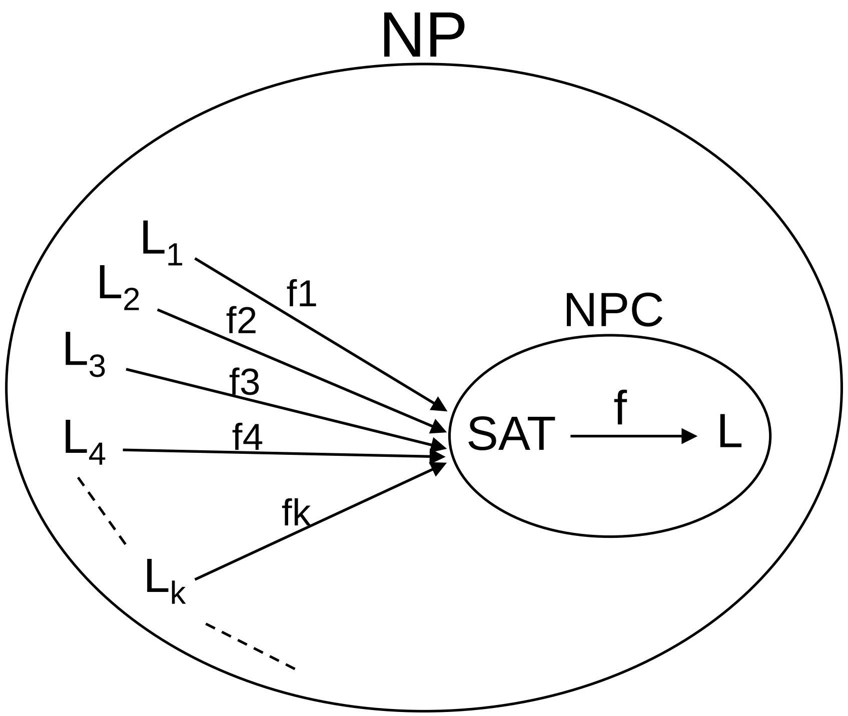
- En Computabilidad, para poblar el mapa tuvimos que recurrir a una técnica difícil, la **diagonalización**, para encontrar un primer ejemplar en el conjunto RE – R (por ejemplo HP).

- Lo mismo sucede en Complejidad Temporal: una vez encontrado un primer ejemplar en el conjunto NPC con una **reducción polinomial muy ingeniosa y nada simple**, vamos a poder seguir poblando NPC de una manera más sencilla. La idea general es, p.ej. tomando SAT:

1. Sea L un lenguaje de NP, tal que SAT se reduce en tiempo poly(n) a él.

2. Como SAT es NP-completo, todo L_i de NP se reduce en tiempo poly(n) a SAT.

3. Por lo tanto, por transitividad, todo L_i de NP se reduce en tiempo poly(n) a L. Y como L está en NP, entonces L también es NP-completo.



Todas las f_i están en FP

Formalizando:

Teorema. Sean $L_1 \in$ NPC y $L_2 \in$ NP, tales que $L_1 \, \alpha_P \, L_2$. Entonces $L_2 \in$ NPC.

Pueba. Queda como ejercicio.

Otros ejemplos clásicos de problemas NP-completos.

- CSAT = {φ | φ es una fórmula booleana satisfactible y está en la forma normal conjuntiva o FNC}
  En este caso φ es una conjunción de cláusulas de variables o variables negadas, como p.ej.:
  $$(x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_5 \vee \neg x_1) \wedge x_3$$
  Se prueba que SAT α_P CSAT.

- 3-SAT = {φ | φ es una fórmula booleana satisfactible y está en la forma 3-FNC}
  Es un caso particular de CSAT con 3 variables o variables negadas por cláusula, como p.ej.:
  $$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\neg x_5 \vee \neg x_3 \vee x_1) \wedge (x_1 \vee x_3 \vee x_6)$$
  Se prueba que CSAT α_P 3-SAT. 3-SAT es muy útil para las reducciones polinomiales, como lo es L_U para las reducciones generales.

- CV = {(G,K) | G es un grafo y tiene un cubrimiento de vértices de tamaño K}. G tiene un cubrimiento de vértices de tamaño K si con K de sus vértices se tocan todos los arcos de G.
  Se prueba que 3-SAT α_P CV.

- 3-COLOR = {G | G es un grafo y sus vértices pueden colorearse con 3 colores de modo tal que ningún par de vértices vecinos tenga el mismo color}
  Se prueba que 3-SAT α_P 3-COLOR.

- Hay toda una heurística para poblar la clase NPC con reducciones polinomiales (un compendio fantástico se presenta en el libro de **Garey y Johnson de 1979** - lo referenciamos en la clase 1 -).

- Luego de que Cook y Levin demostraran que SAT es NP-completo, a partir de dicho problema **Karp en 1972 introdujo 21 problemas NP-completos**, que impulsó sobremanera este área de la complejidad computacional - lo publicaremos como artículo interesante -.

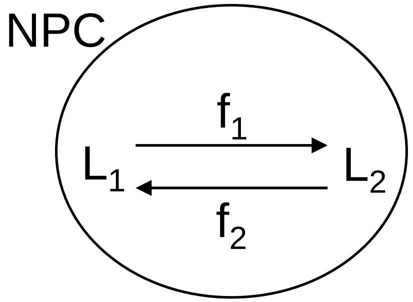
- Levin fue el primero en llamar a los problemas NP-completos **problemas universales**. La idea subyacente es que todos los problemas NP-completos son un **único problema**, codificado en términos de grafos, lógica, aritmética, etc. De hecho, notar que para todo par de lenguajes NP-completos L_1 y L_2 se cumple $L_1 \alpha_P L_2$ y $L_2 \alpha_P L_1$ (ejercicio).

- **Hay un fenómeno curioso con el número 3**. Por ejemplo, problemas como 3-SAT y 3-COLOR son NP-completos. Sin embargo 2-SAT y 2-COLOR están en P. Esto ocurre con numerosos problemas (*Nota: a mediados de 1970, Appel y Haken probaron que alcanza con 4 colores para pintar grafos y mapas sin vecinos con el mismo color*). Curiosidades similares ocurren con problemas como la programación lineal (soluciones reales vs enteras), determinante vs permanente de una matriz, etc.

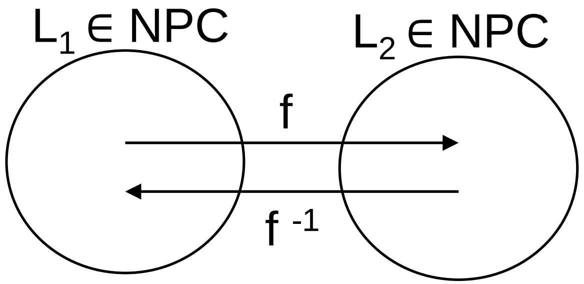
- El concepto de completitud se extiende a toda la jerarquía temporal (y también espacial). De hecho se considera que una clase sin problemas completos identificados **no tiene mucha razón de existir**.

Dos caracterizaciones de los problemas NP-completos.

1. Ya vimos recién que para todo par de lenguajes L_1 y L_2 de NPC se cumple $L_1 \alpha_P L_2$ y $L_2 \alpha_P L_1$:



- No necesariamente f_2 es la función inversa de f_1. Sin embargo, **todos los lenguajes conocidos de NPC cumplen dicha propiedad**. Es decir, existe una biyección entre todo par de lenguajes NP-completos conocidos:



En este caso se dice que L_1 y L_2 son p-isomorfos (la p se debe a que las funciones f y f^{-1} están en FP)

- La **Conjetura de Berman-Hartmanis** plantea que efectivamente todos los lenguajes NP-completos son p-isomorfos. Se prueba fácilmente que si se cumple la conjetura … ¡P ≠ NP! (ejercicio - ayuda: los lenguajes finitos están en P -). Como contrapartida, la **Conjetura de Joseph-Young** plantea que existe algún lenguaje NP-completo que no es p-isomorfo con SAT.

2. Conjetura sobre la densidad de los lenguajes de NPC:

- **Todos los lenguajes NP-completos conocidos son densos**. Un lenguaje es **denso** si para todo n, la cantidad de sus cadenas de longitud a lo sumo n es exp(n).

- El opuesto a lenguaje denso es lenguaje **dispero** (en este caso la cota por cada n es poly(n)).

- Si se cumple la conjetura de Berman-Hartmanis (todos los lenguajes NP-completos son p-isomorfos), se comprueba fácilmente que no puede haber una mezcla de lenguajes densos y dispersos (ejercicio). **De esta manera, todos los lenguajes serían densos**.

- **Se prueba que si existe un lenguaje NP-completo disperso, entonces P = NP.**

- Notar que a diferencia de lo observado en Computabilidad, en Complejidad Temporal **el tamaño o la densidad de un lenguaje SÍ se relaciona con la dificultad para resolverlo** (intuitivamente, a más cadenas, más complejidad para encontrar una solución).

Hay otras caracterizaciones que podrían contribuir a diferenciar P de NP. P.ej. el tipo de lenguaje lógico necesario para describir a cada clase (se conoce como complejidad descriptiva). Hasta el momento de todos modos, el intento ha sido infructuoso.

Clase Práctica 6. P, NP, NPC, reducciones polinomiales.

Ejemplo 1. Sea DSAT = {φ | φ es una fórmula booleana sin cuantificadores, en la forma normal disyuntiva o FND, y es satisfactible}. Una fórmula booleana sin cuantificadores está en la forma FND si es una disyunción de cláusulas formadas por conjunciones de literales (variables o variables negadas), como por ejemplo:

$$(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_2 \wedge x_4 \wedge \neg x_4 \wedge x_5) \vee x_6 \vee (x_5 \wedge x_6)$$

Se cumple que DSAT ∈ P. Existe una MT M que acepta DSAT en tiempo polinomial.
Dada la fórmula φ, M hace:

1) Verifica la sintaxis de φ. Si la sintaxis es errónea, rechaza. **Tiempo O(n)**.

2) Chequea si existe una cláusula de la disyunción que no tenga al mismo tiempo variables y variables negadas x_i y $\neg x_i$. Si existe una cláusula así, significa que φ es satisfactible, y acepta; en caso contrario, rechaza. **Tiempo O(n^2).**

Ejemplo 2. Sea NODSAT = {φ | φ es una fórmula booleana sin cuantificadores en la forma FND y existe una asignación que no la satisface}.

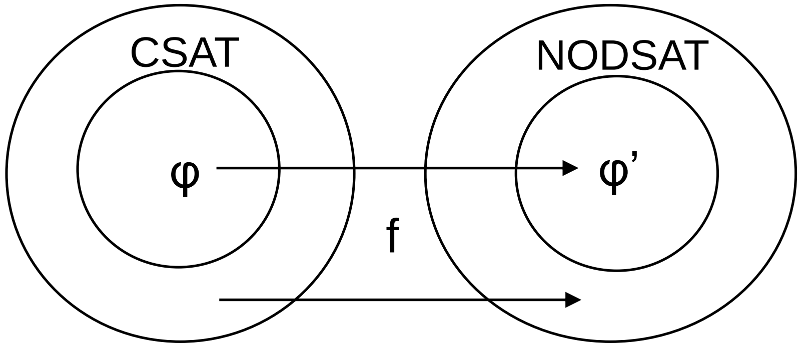
No pareciera que NODSAT ∈ P:
Si φ tiene m variables, en el peor caso deben probarse 2^m asignaciones de valores de verdad, por lo tanto $O(2^n)$ asignaciones, con n = |φ|.
Tiempo exp(n).

Más aún, se prueba que NODSAT ∈ NPC, es decir que es un lenguaje NP-completo:

1. Se prueba fácilmente que NODSAT ∈ NP (queda como ejercicio).

2. Y se cumple que todos los lenguajes de NP se reducen polinomialmente a NODSAT.
Esto lo vamos a probar a continuación encontrando una reducción polinomial de CSAT (es NP-completo) a NODSAT:

Sea la siguiente función de reducción $f(\varphi) = \varphi´$, tal que f niega la fórmula φ en base a las leyes de De Morgan para obtener la fórmula φ'. Por ejemplo:



Si $\varphi = (x_1 \vee x_2) \wedge (x_4 \vee \neg x_4) \wedge (\neg x_3 \vee x_5 \vee x_6)$,
entonces $\varphi' = (\neg x_1 \wedge \neg x_2) \vee (\neg x_4 \wedge x_4) \vee (x_3 \wedge \neg x_5 \wedge \neg x_6)$

Se cumple que:

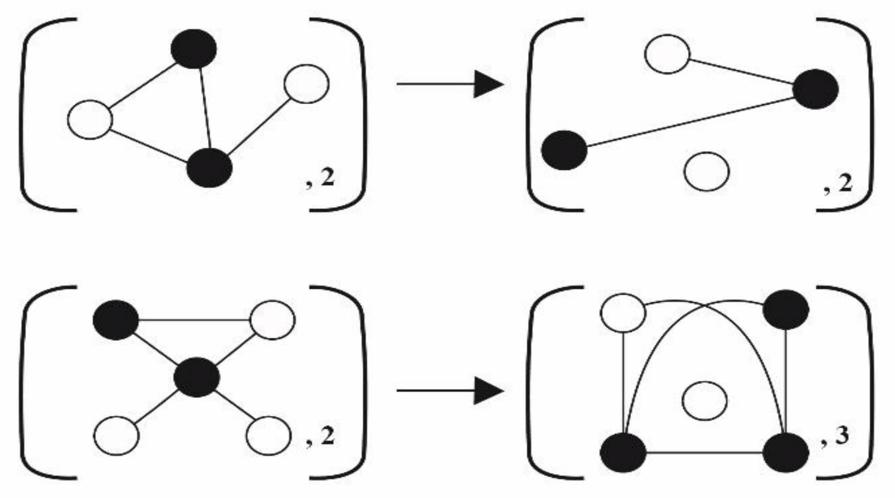
f es una función total computable en tiempo polinomial.
La MT M_f que computa f, dada la fórmula φ, primero verifica su sintaxis, y si hay incorrección genera como output la cadena 1.
En caso contrario, M_f transforma φ en φ' negando φ de acuerdo a las leyes de De Morgan.
M_f trabaja en tiempo polinomial, porque la verificación sintáctica de φ es lineal, imprimir 1 consume tiempo constante, y transformar φ en φ' según lo especificado es también lineal.

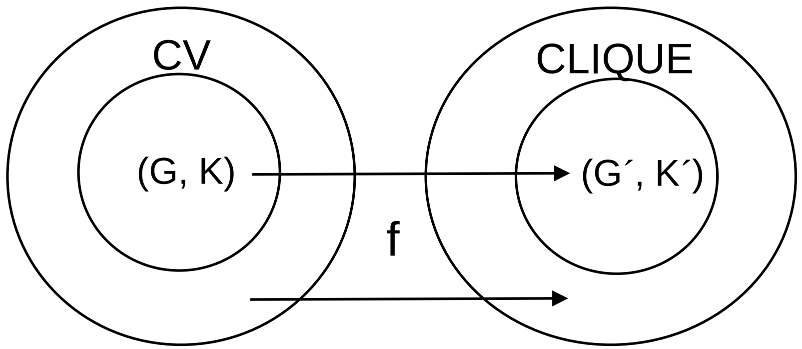
$\varphi \in$ CSAT $\leftrightarrow f(\varphi) = \varphi' \in$ NODSAT.
$\varphi \in$ CSAT $\leftrightarrow \varphi$ está en la forma FNC y existe una asignación A que la satisface $\leftrightarrow \varphi'$ está en la forma FND y existe una asignación A que no la satisface $\leftrightarrow \varphi' \in$ NODSAT.

Ejemplo 3. Sea CLIQUE = {(G, K) | G es un grafo y tiene un clique de tamaño K}.

Se prueba que CLIQUE es NP-completo. Ya probamos que CLIQUE ∈ NP. Falta probar que todos los lenguajes de NP se reducen a él. Encontraremos una reducción polinomial de CV a CLIQUE, siendo CV el lenguaje que representa el problema del cubrimiento de vértices ya referido:
CV = {(G, K) | G tiene un cubrimiento de K vértices, es decir que K de sus vértices tocan a todos los arcos de G}. Como CV es NP-completo, entonces también lo será CLIQUE.

Función de reducción f de CV a CLIQUE. Dado un grafo válido G (si es inválido se asigna un 1), con m vértices y un número natural K ≤ m, la función es: f((G, K)) = (G^C, m – K), siendo G^C el grafo "complemento" de G (tiene los mismos vértices que G y sólo los arcos que G no tiene). Abajo a la izquierda hay dos casos de aplicación de f:





f es una función total computable en tiempo polinomial. Queda como ejercicio**.**

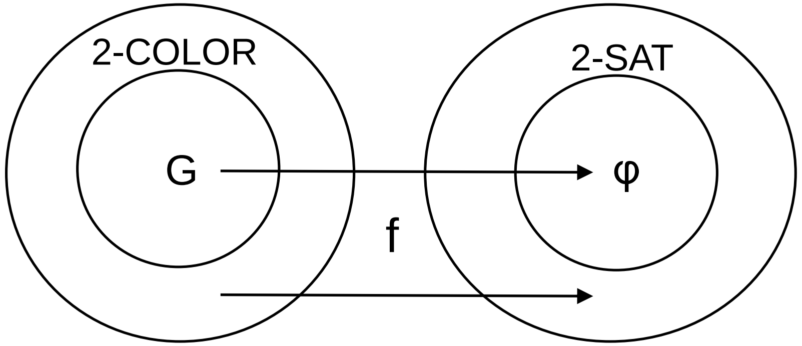
(G, K) ∈ CV sii (G^C, m – K) ∈ CLIQUE. Para mayor claridad, descomponemos la prueba en los dos sentidos:

Primero veremos que si (G, K) ∈ CV, entonces (G^C, m – K) ∈ CLIQUE. Sea (G, K) ∈ CV, y V' un cubrimiento de vértices de G de tamaño K. Veamos que V – V' es un clique de G^C de tamaño m – K, y así que (G^C, m – K) ∈ CLIQUE. Por un lado, el conjunto de vértices V – V' tiene tamaño m – K. Por otro lado, supongamos que G^C no es un clique, por ejemplo que no incluye un arco (i , h), siendo i y h vértices de V – V'. Entonces (i, h) es un arco de G, siendo i y h vértices que no están en V', por lo que V' no es un cubrimiento de vértices de G (absurdo).

Ahora veremos que si (G^C, m – K) ∈ CLIQUE, entonces (G, K) ∈ CV. Sea (G^C, m – K) ∈ CLIQUE, y V' un clique de G^C de tamaño m – K. Veamos que V – V' es un cubrimiento de G de tamaño K, y así que (G, K) ∈ CV. Por un lado, el conjunto de vértices V – V' tiene tamaño m – (m – K) = K. Por otro lado, supongamos que V – V' no es un cubrimiento de vértices de G, por ejemplo que existe un arco (i, h), con i y h vértices no pertenecientes a V – V' (y así pertenecientes a V´). Pero entonces V' no es un clique de G (absurdo).

Ejemplo 4. Existe una reducción polinomial de 2-COLOR a 2-SAT, siendo:
2-COLOR = {G | G es un grafo tal que sus vértices se pueden colorear con 2 colores de manera tal que dos vértices vecinos no tengan el mismo color}.
2-SAT = {φ | φ es una fórmula booleana sin cuantificadores satisfactible, en la forma FNC y con dos literales (variables o variables negadas) por cláusula}.

Función de reducción f de 2-COLOR a 2-SAT.
A todo grafo válido G, la función f le asigna una fórmula booleana φ en la FNC con dos literales por cláusula, de modo tal que por cada arco (i, k) de G, f construye una subfórmula $(x_i \vee x_k) \wedge (\neg x_i \vee \neg x_k)$.
A los grafos inválidos le asigna la cadena 1.



La función f es total y se computa en tiempo polinomial.
La validación de la sintaxis de un grafo es cuadrática. Escribir un 1 tarda tiempo constante. La generación de la fórmula booleana descripta tarda tiempo lineal.

G ∈ 2-COLORACIÓN ↔ φ ∈ 2-SAT.
Asociando dos colores c_1 y c_2 con los valores de verdad verdadero y falso, respectivamente, claramente los vértices de todo arco de G tienen colores distintos si y sólo si la conjunción de las dos cláusulas que se construyen a partir de ellos es satisfactible.