Análisis y Complejidad de Algoritmos

Completitud NP

Arturo Díaz Pérez
Sección de Computación
Departamento de Ingeniería Eléctrica
CINVESTAV-IPN
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco
México, D. F. CP 07300

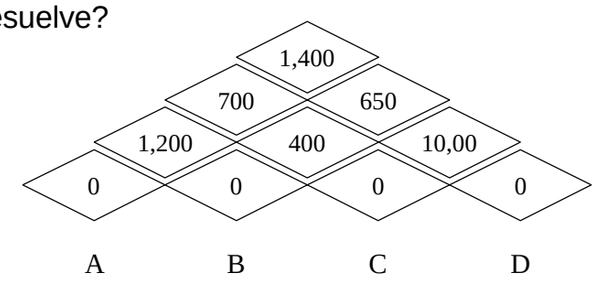
Tel. (5)747 3800 Ext. 3755
e-mail: adiaz@cs.cinvestav.mx

Intratabilidad

- Informalmente:
  - Si una computadora tiene dificultades para resolver un problema, se dice que el problema es intratable.
  - Intratable significa “difícil de tratar o de trabajar”
- Un algoritmo de tiempo polinomial es aquel cuya complejidad en el peor caso está acotada por arriba por un polinomio sobre el tamaño de su entrada.
  - Si n es el tamaño de la entrada, entonces, existe un polinomio $p(n)$ tal que $T(n)$ es $O(p(n))$.
- Ejemplo:
  - ¿El algoritmo MergeSort es de tiempo polinomial?

Intratabilidad

- Un problema se dice intratable si NO se conoce un algoritmo de tiempo polinomial para resolverlo.
  - El algoritmo de “fuerza bruta” para resolver el problema de parentización óptima de matrices toma un tiempo $\Omega(4^n/n^{3/2})$.
  - ¿Cuánto tiempo toma el algoritmo de programación dinámica que lo resuelve?



Análisis y Diseño de Algoritmos *Np-Completeness-3*

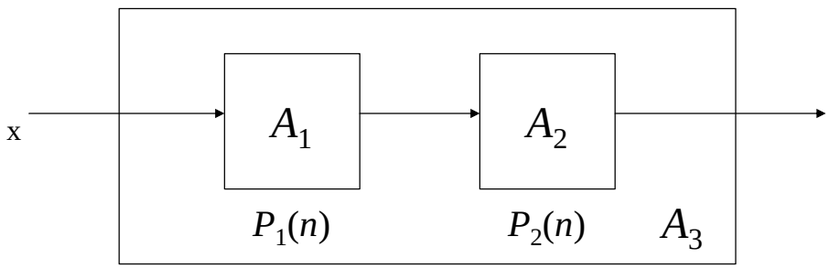
Tiempo Polinomial

- Tratabilidad ~ Tiempo Polinomial
  - ¿Es $\Theta(n^{100})$ tratable?
    - En la práctica existen pocos problemas que requieren tiempo polinomial de un grado muy alto.
  - Si un algoritmo toma tiempo polinomial en un modelo de cómputo, también toma tiempo polinomial en algún otro modelo de cómputo.
    - Máquinas de Turing
    - Máquinas de Acceso Aleatorio (RAM)

Análisis y Diseño de Algoritmos *Np-Completeness-4*

Tiempo Polinomial

- La clase de problemas solubles en tiempo polinomial tiene propiedades de cerradura interesantes.
  - La suma, multiplicación y composición de polinomios son cerradas



Categorías de Problemas

- Problemas para los cuales se ha encontrado algoritmos polinomiales
  - Ordenamiento, búsqueda, caminos más cortos, componentes fuertes.
  - Circuito Euleriano
  - Multiplicación de Matrices
  - La mayoría de los problemas que hemos analizado hasta ahora
- Problemas que se han probado ser intratables
  - Problemas no decidibles: Halting problem.
  - Todos los circuitos Hamiltonianos de una gráfica

Categorías de Problemas

- Problemas cuya intratabilidad no se ha probado, pero para los cuales tampoco se conoce un algoritmo de tiempo polinomial
  - Circuito Hamiltoniano, TSP
  - Knapsack
  - Satisfactibilidad
  - Cliqué
  - Calendarización Optima (Job Scheduling)
  - Cubierta Mínima
  - Corte Máximo

Un Problema Intratable, Soluble

- Problema: **Encontrar todos los circuitos Hamiltonianos de una gráfica dirigida.**
  - En el peor caso habría un arco entre cada par de vértices, así que habría (*n*-1)! ciclos Hamiltonianos.
  - Existe un número intratable (mayor que polinomial) de soluciones. Por tanto, no puede existir un algoritmo de tiempo polinomial para resolver este problema.
- Problema: **Encontrar un ciclo Hamiltoniano en una gráfica dirigida**
  - No se conoce un algoritmo de tiempo polinomial para resolverlo
  - No se sabe si tal algoritmo no puede existir

Un Problema Intratable, Insoluble

- Problema: **¿Podemos examinar un programa arbitrario y decidir si su ejecución se detiene sobre una entrada arbitraria?**
  - Dada la función definida por $f(x) = 2x + 1$, se detiene para x=17.
  - De hecho se detiene para todos los números naturales
  - Considere ahora la función definida por
    - *h*(*x*) ***if*** *x* = 7 ***then*** *return* 2 ***else while***(*true*) ***do*** ;
  - *h*(7) se detiene, pero para cualquier otro valor de *x*, *h*(*x*) no se detiene.

Un Problema Intratable, Insoluble

- ¿Existe un algoritmo que puede decidir si la ejecución de un programa arbitrario se detiene en una entrada arbitraria?
  - NO. Este problema es insoluble.
  - Demostrado por Alan Turing en 1936.

Halting Problem

- Supongamos que existiera tal algoritmo.
  - Definimos la función "*Halt*" suponiendo que f_x toma un solo argumento:

    Halt(x) = ***if*** f_x *se detiene en la entrada x* ***then*** *1* ***else*** *0*

  - *Halt* es computable dado que hemos supuesto que existe una función que puede evaluar a condición.
  - Construyamos lo siguiente

    g(x) = ***if*** *Halt(x) = 1* ***then while*** *true* ***do*** *;* ***else*** *0*
  - *g* es computable dado que hemos asumido que *Halt* es computable.

Halting Problem

- Supongamos que existiera tal algoritmo.
  - . . .
  - *g* es una de las posibles f_x funciones de un argumento, digamos f_n.
  - Examinemos $g(n) = f_n(n)$.
  - Si *Halt*(*n*) = 1, entonces, *g*(*n*) corre por siempre pero *g*(*n*) se detiene en *n*. #
  - Si *Halt*(*n*) = 0, entonces, *g*(*n*) se detiene pero *g*(*n*) corre por siempre. #
- Por lo tanto, no puede existir tal función *Halt*.

Problemas de Decisión

- Problemas de Decisión
  - Un ***problema de decisión*** consta de un subconjunto $A \subset N^m \times N^n$ y consiste en decidir cuando una instancia de él pertenece o no a *A*. Es pues de la siguiente forma:
    - **Instancia**: Un punto $(\mathbf{x},\mathbf{y}) \in N^m \times N^n$
    - **Solución**: una respuesta: 1 si $(\mathbf{x},\mathbf{y}) \in A$, 0 si $(\mathbf{x},\mathbf{y}) \notin A$
  - Un algoritmo que resuelve el problema de decisión es propiamente un *reconocedor* del conjunto *A* o, en otras palabras, un *comprobador* de parejas
    - (*Instancia*, *Solución*)

Problemas de Solución

- Problemas de Solución
  - Un problema de solución consta de un subconjunto $A \subset N^m \times N^n$ y consiste en localizar una solución dada una instancia.
    - **Instancia**: Un punto $\mathbf{x} \in N^m$
    - **Solución**: Una respuesta: **y** si $(\mathbf{x}, \mathbf{y}) \in A$, $\perp$ si $\forall\ \mathbf{y} \in N^n$: $(\mathbf{x}, \mathbf{y}) \notin A$
  - Un algoritmo que resuelve el problema de solución es propiamente un *resolvedor* o *calculador* de soluciones de acuerdo a la regla "*A*"

Comprobadores y Resolvedores

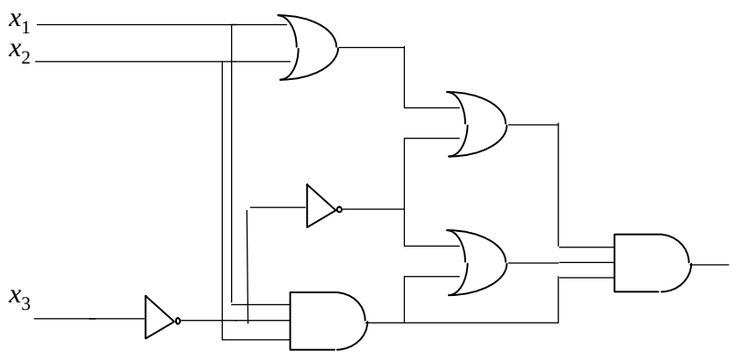
- Las diferencias principales en los problemas de decisión y solución:
  - ***Comprobación***: En un problema de decisión se verifica que una pareja hipotética $(\mathbf{x},\mathbf{y}) \in N^m \times N^n$ es efecto, una pareja de la forma (*Instancia*, *Solución*)
  - ***Resolución***: En un problema de solución se construye una correspondiente *Solución* para cada *Instancia* dada
  - Un problema de solución puede ser visto como la *proyección* de un problema de decisión, pues
    - $\mathbf{x} \in dom(A)\ \exists\ \mathbf{y} \in N^n$: $(\mathbf{x},\mathbf{y}) \hat{I}\ A$, donde
    - $dom(A) = \{\mathbf{x} \in N^m | \exists\ \mathbf{y} \in N^n : (\mathbf{x},\mathbf{y}) \hat{I}\ A\}$, el *dominio* del conjunto *A*.

Observaciones

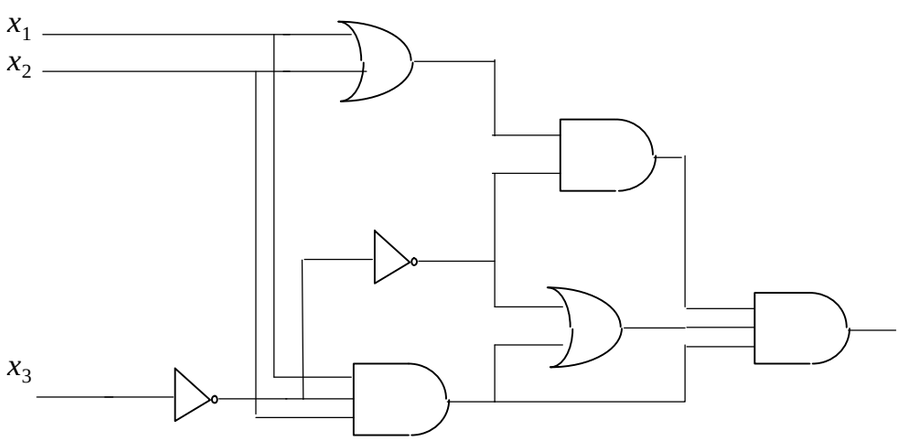
- Observaciones
  - Sea D_A un resolvedor del problema *A*. Podemos construir un comprobador N_A como sigue:
    - Dado $(x,y) \in N^m \times N^n$, hacemos $y_0 = D_A(x)$
    - Si $y = y_0$, entonces se reconoce positivamente. Se rechaza en otro caso
  - Sea N_A un comprobador del problema *A*. Podemos construir un resolvedor como D_A sigue:
    - Enumeramos el espacio de posibles soluciones, y_i
    - Para cada *i*, verificamos con N_A si acaso $(x,y_i) \in A$ y suspendemos esta iteración la primera vez que se obtiene una respuesta positiva.

Satisfactibilidad de Circuitos

- CIRCUIT-SAT
  - Considere un circuito combinacional booleano compuesto de compuertas AND, OR y NOT. ¿El circuito es satisfactible?
  - ¿Existe una asignación de señales digitales que hace que la salida del circuito sea 1?

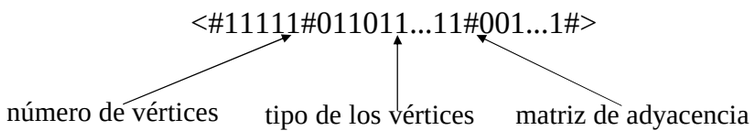


Satisfactibilidad de Circuitos



Satisfactibilidad de Circuitos

- El problema de satisfactibilidad puede ser codificado
- Dado un circuito *C*, el circuito puede ser codificado en una cadena binaria <*C*> cuya longitud no es mayor que el tamaño del circuito mismo



Análisis y Diseño de Algoritmos *Np-Completeness*-19

Lenguajes y Problemas

- Sea Σ un alfabeto finito arbitrario. Un problema de decisión *P* se define por un conjunto de instancias $I \subseteq \Sigma^*$ del problema y una condición ϕ_P: $I \rightarrow \{0,1\}$ tal que tiene un valor 1 en instancias “si” y un valor 0 en instancias “no”.
  - ← $I_{si} = \{ w \in I \mid \phi_P(w) = 1\}$,
  - ← $I_{no} = I - I_{si}$
- El complemento de un problema de decisión *P*, denotado como *coP*, es el problema de decisión en el cual las instancias “si” de *coP* son las instancias “no” de *P* y viceversa.

Análisis y Diseño de Algoritmos *Np-Completeness*-20

Lenguajes y Problemas

- Las instancias “SI” de un problema de decisión se codifican como cadenas binarias por una función de codificación $\sigma: \Sigma^* \rightarrow \{0,1\}^*$ que asigna para cada $w \in I$ una cadena $\sigma(w) \in \{0,1\}^*$.
- Con respecto a σ, el lenguaje $L(P)$ asociado con un problema de decisión P es el conjunto $L(P) = \{\sigma(w) \mid w \in I_{si}\}$.
- Similarmente, con respecto a σ, el lenguaje $L(coP)$ asociado con coP es el conjunto $L(coP) = \{\sigma(w) \mid w \in I_{no}\}$.

Lenguajes y Problemas

- El complemento de un lenguaje, denotado por $\overline{L}$, es $B^* - L$; esto es, $\overline{L}$ consiste de las cadenas que no están en L.
- Un problema de decisión puede ser generalizado a un problema P caracterizado por una función $f: B^* \rightarrow B^*$ descrita como un conjunto de pares ordenados $(x, f(x))$, donde cada cadena $x \in B^*$ aparece solo una vez como lado izquierdo en un par.
  - ← Un lenguaje se define por el problema $f: B^* \rightarrow B^*$ y consiste de las cadenas en las cuales f tiene un valor 1

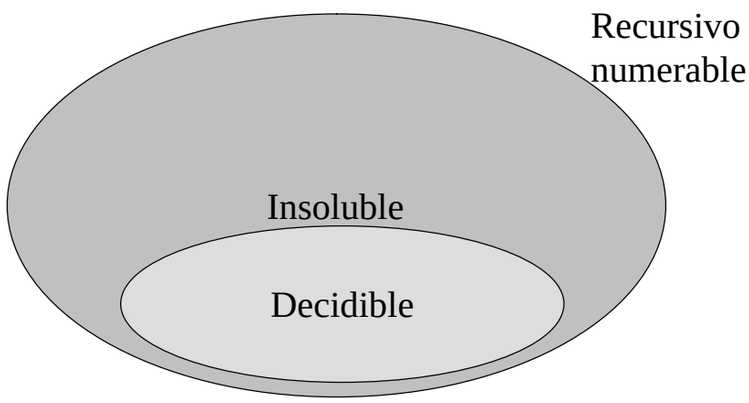
Lenguajes y Problemas

- Existen muchas formas de codificar las instancias de los problemas
  - ← Un autómata finito es suficiente para determinar si una cadena binaria esta en σ(*I*).
- Las cadenas en *L*(*P*), son cadenas ó de *L*(coP) o cadenas de σ(Σ*-*I*).
  - ← Ya que probar la membresía en σ(Σ*-*I*) es fácil, probar la membresía en *L*(*coP*) y *L*(*P*) requiere más o menos el mismo espacio y tiempo.

Decidibilidad

- Un lenguaje L que es *decidible* (también llamado recursivo) tiene un algoritmo que se detiene sobre todas las entradas posibles aceptando aquella que están en L
- Un lenguaje para el cual existe un algoritmo que acepta aquellas cadenas de L pero que posiblemente no se detiene sobre cadenas que no están en L se denomina un *lenguaje recursivo numerable*
- Un lenguaje que es recursivo numerable pero no es decidible es *insoluble*

Decidibilidad



Complejidad

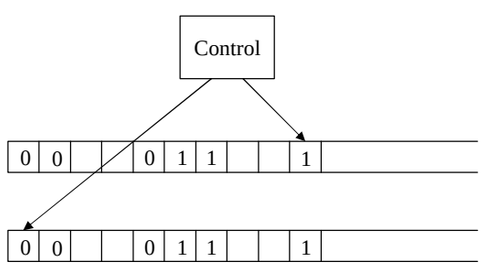
- Un problema decidible es
  - computacionalmente soluble en principio, pero
  - no necesariamente soluble en la práctica
- La dificultad para resolver un problema se debe al consumo de recursos en
  - tiempo, y
  - espacio

Complejidad

- Ejemplo. Considere el lenguaje $A = \{ 0^n1^n \mid n \geq 0 \}$. Claramente el lenguaje es decidible.
  - ¿Cuánto tiempo le tomaría a un programa decidir si una palabra pertenece al lenguaje?
  - El número de pasos puede depender de factores diversos.
    - Por ejemplo, para una gráfica puede depender del número de vértices, arcos, del grado máximo, de todos los anteriores o de ninguno de ellos!
- La complejidad se mide como una función de la longitud de la cadena de entrada

Complejidad

- Ejemplo. Considere el lenguaje $L = \{ 0^n1^n \}$
  - Una MT M_1 de una sola cinta puede decidir el lenguaje en tiempo $O(n^2)$ (Tarea)
  - Una MT M_2 de una sola cinta puede decidir el lenguaje en tiempo $O(n\log n)$ (Tarea)
  - Una máquina M_3 de dos cintas puede decidir el lenguaje en tiempo $O(n)$ (Tarea)



Complejidad

- Complejidad y computabilidad difieren en un aspecto importante
- Computabilidad: todos los modelos razonables son equivalentes (Tesis de Church).
- Complejidad: la elección del modelo afecta la complejidad
- ¿Cómo el modelo afecta a la complejidad?

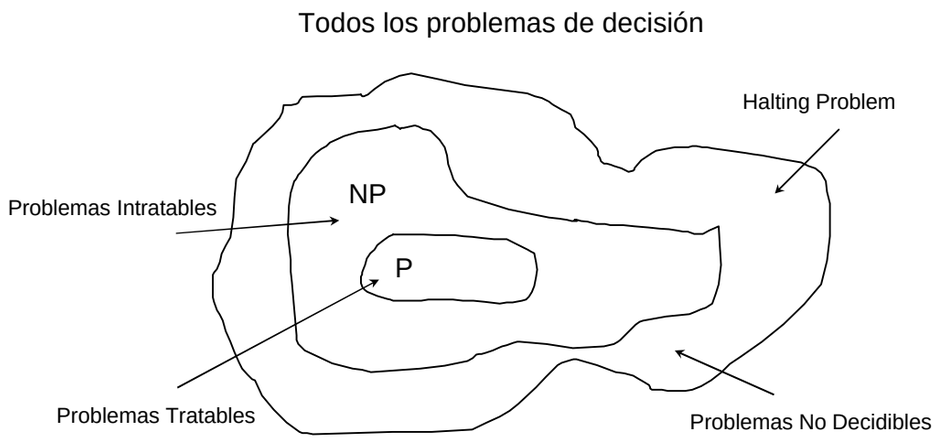
Modelo No Determinista

- Extendiendo el poder de una computadora
  - ← Supongamos que existe una computadora que adivina una solución y otro mecanismo (no necesariamente diferente) verifica si la solución es correcta (SI/NO).
  - ← Este es un algoritmo no determinista
  - ← ***No-determinismo***: En un problema de decisión se parte de una pareja hipotética (**x**,**y**) $\in N^m \times N^n$. Queda indeterminada la manera en la que se obtiene **y** dada **x**.
  - ← ***Determinismo***: En un problema de solución se construye algorítmicamente la correspondiente *Solución* de una *Instancia* dada.

Modelo No Determinista

- P: El conjunto de todos los problemas de decisión que pueden ser ***resueltos*** en tiempo polinomial.
- NP: El conjunto de todos los problemas de decisión que pueden ser ***verificados*** en tiempo polinomial.
  - ← Se requiere que cualquier instancia "SI" se verifique en tiempo polinomial.
  - ← La adivinanza es no determinista, pero el algoritmo de verificación es determinista.

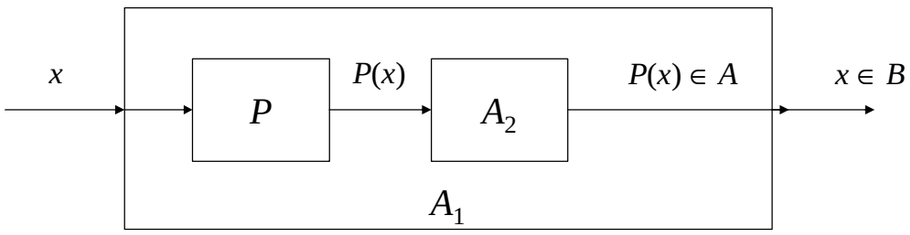
Problemas P y NP



Reducibilidad

- Sean *A* y *B* dos conjuntos en *N*. Diremos que
  - ← *B* se *reduce en tiempo polinomial* a *A*, $B \prec_{tp} A$, si existe un algoritmo *P* de tiempo polinomial, tal que,

$$\forall x \in N: x \in B \Leftrightarrow P(x) \in A$$

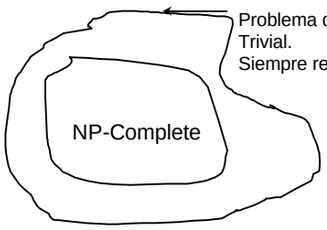


Reducibilidad

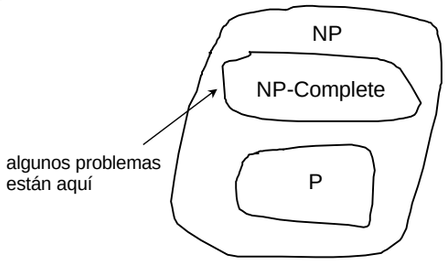
- Un problema *B* se dice que es *NP*-completo si
  - está en la clase NP, y
  - todos los problemas NP se reducen en tiempo polinomial a *B*
- Un problema *B* se dice que es *NP*-difícil (NP-hard) si cumple con la propiedad 2 pero no necesariamente con la propiedad 1.

Problemas P, NP y NP-Completos

☞ NP-completo es el subconjunto “más difícil” de NP.



P = NP



P ≠ NP

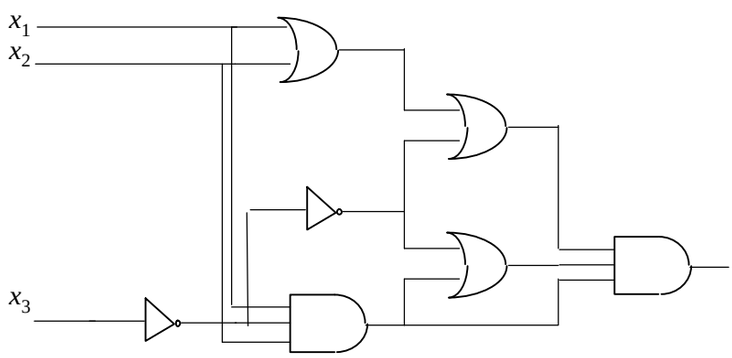
- Si un problema NP-completo estuviera también en la clase P, entonces, necesariamente P = NP.
- Para mostrar que un primer lenguaje L_0 es NP-completo, se debe dar una reducción en tiempo polinomial para cada lenguaje en NP a L_0.
- Una vez que se tiene un lenguaje L_0 NP-completo, se puede probar que otro lenguaje L_1 en NP es también completo mostrando una reducción en tiempo polinomial de L_0 a L_1

Satisfactibilidad de Circuitos

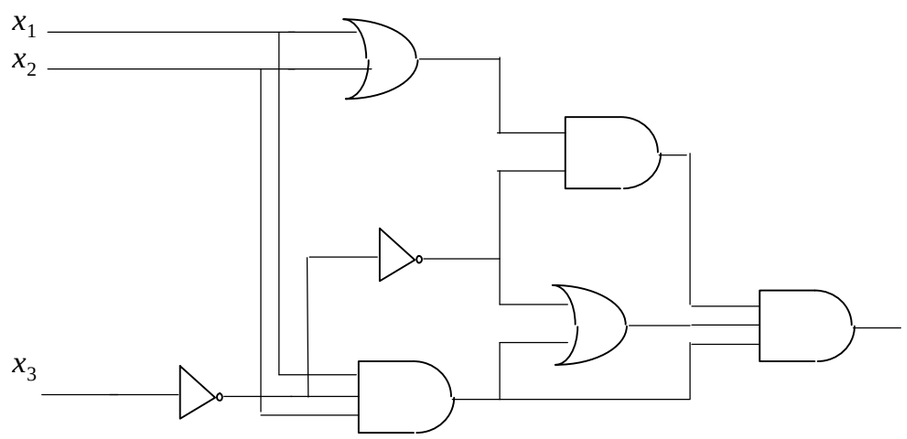
☞ CIRCUIT-SAT

← Considere un circuito combinacional booleano compuesto de compuertas AND, OR y NOT. ¿El circuito es satisfactible?

← ¿Existe una asignación de señales digitales que hace que la salida del circuito sea 1?



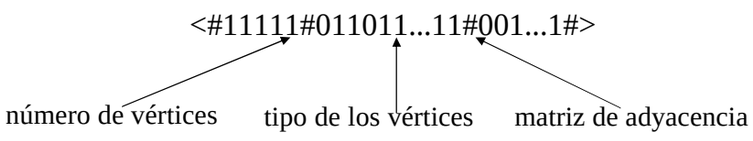
Satisfactibilidad de Circuitos



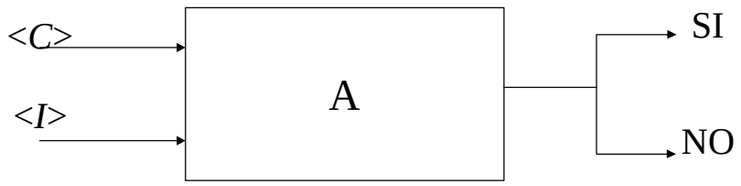
Satisfactibilidad de Circuitos

- El problema de satisfactibilidad puede ser codificado

- Dado un circuito *C*, el circuito puede ser codificado en una cadena binaria <*C*> cuya longitud no es mayor que el tamaño del circuito mismo



CIRCUIT-SAT ∈ NP



- **Cada una de las compuertas (vértices) es excitada con las señales de entrada <*I*>.**
  - ← Si la salida del circuito completo es 1, la salida es SI
  - ← Si la salida del circuito completo es 0, la salida es NO
- La verificación toma un tiempo proporcional a la cantidad de compuertas.
  - ← Si el número de compuertas es polinomial, la verificación se realiza en tiempo polinomial.

CIRCUIT-SAT ∈ NP

- Si un circuito *C* es satisfactible, entonces, existe al menos una asignación *I** que lo satisface y el algoritmo A verificaría esa posibilidad.
- Si un circuito *C* no es satisfactible, el algoritmo A no puede dar una salida afirmativa

Configuraciones

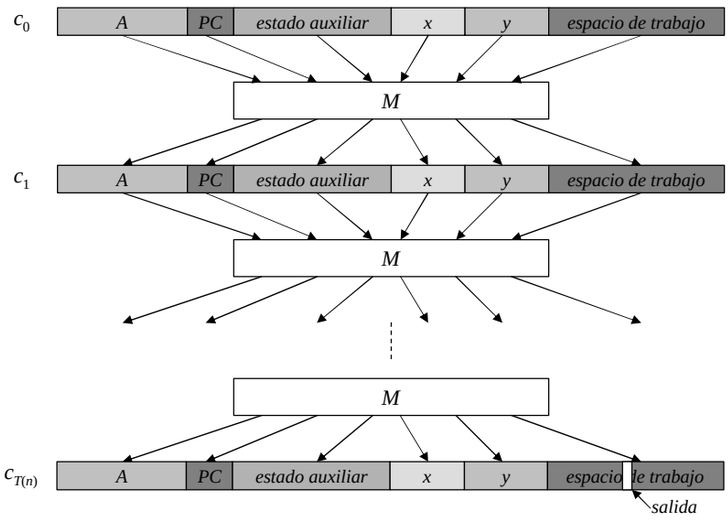
- **Se debe mostrar que cualquier lenguaje en NP es reducible en tiempo polinomial a CIRCUIT-SAT.**
  - Un programa se almacena en memoria como una secuencia de instrucciones
  - Una instrucción típica codifica
    - operación que será ejecutada
    - dirección de los operandos de memoria
    - dirección de memoria en donde se almacena el resultado
    - el contador de programa, una localidad de memoria especial, mantiene la dirección de la siguiente instrucción a ejecutar
      - la ejecución de una instrucción puede alterar el contenido del contador de programa

Configuraciones

- **Se debe mostrar que cualquier lenguaje en NP es reducible en tiempo polinomial a CIRCUIT-SAT.**
  - . . .
  - En cualquier punto de la ejecución de un programa, el estado completo de la computación se representa por la memoria de la computadora
    - El programa mismo, el contador de programa, el espacio de trabajo y los bits de estado
    - Un estado particular de la memoria se le llama una ***configuración***.
    - La ejecución de una instrucción puede ser vista como pasar de una configuración a otra.

Configuraciones

- El hardware que pasa de una configuración a otra puede ser un circuito combinacional



Reducción a CIRCUIT-SAT

- **El algoritmo de reducción *F* de tiempo polinomial debe hacer lo siguiente:**
  - ← Construye un circuito combinacional que calcula todas las configuraciones producidas por una configuración inicial dada.
  - ← Dado una entrada *x*, debe calcular un circuito $C = f(x)$ que es satisfactible si y solo si existe un certificado *y* tal que $A(x, y) = 1$.
  - ← Cuando *F* obtiene una entrada *x*, calcula $n = |x|$ y construye un circuito combinacional *C*' que consiste de $T(n)$ copias de *M*.

Reducción a CIRCUIT-SAT

- **El algoritmo de reducción *F* de tiempo polinomial debe hacer lo siguiente:**
  - La entrada a *C'* es una configuración inicial que corresponde al cómputo de $A(x,y)$ y la salida es la configuración $c_{T(n)}$.
    - El circuito $C = f(x)$ que calcula *F* se obtiene modificando *C*' ligeramente.
      - Las entradas a *C*' que corresponden al contador de programa, la entrada *x*, y el estado inicial de la memoria son "alambradas" directamente a los valores iniciales.
      - Las entradas restantes corresponden al certificado *y*.
      - Todas las salidas de los circuitos son ignoradas excepto un bit de $c_{T(n)}$.
      - El circuito *C* construido calcula $C(y) = A(x,y)$ para cualquier *y* de longitud $O(n^k)$.
      - El algoritmo de reducción F, al encontrar una cadena de entrada *x*, calcula tal circuito *C* y lo da como salida.

CIRCUIT-SAT ∈ NP-Difícil

- **Se deben mostrar dos cosas:**
  - *F* calcula correctamente una función de reducción *f*.
    - *C* es satisfactible, si y solo si, existe un certificado y tal que $A(x,y) = 1$.
  - *F* se ejecuta en tiempo polinomial.

CIRCUIT-SAT ∈ NP-Difícil

- *F* calcula correctamente una función de reducción *f*.
  - *C* es satisfactible, si y solo si, existe un certificado y tal que $A(x,y) = 1$.

Prueba de 1.

⇒ Supongamos que existe un certificado *y* de longitud $O(n^k)$ tal que $A(x,y) = 1$.

- Si aplicamos los bits de *y* a las entradas de *C*, la salida de *C* es $C(y)=A(x,y) = 1$.
  - Si el certificado existe, *C* es satisfactible.

⇐ Supongamos que *C* es satisfactible.

- Existe una entrada *y* tal que $C(y) = 1$. Se puede concluir que $A(x,y) = 1$.
  - Si *C* es satisfactible, existe un certificado para *A*.

CIRCUIT-SAT ∈ NP-Difícil

***F* se ejecuta en tiempo polinomial.**

← **Prueba de 2.**

← Sea $|x| = n$. El número de bits para representar una configuración es polinomial en *n*.

- A es constante. La longitud de *x* es *n*. La longitud del certificado es $O(n^k)$.
- Ya que el algoritmo corre en a lo más $O(n^k)$ pasos, la cantidad de almacenamiento de trabajo es también polinomial sobre *n*.
- El circuito combinacional *M* que implementa el hardware de la computadora tiene un tamaño polinomial en la longitud de una configuración, el cual es polinomial sobre $O(n^k)$ lo que es también polinomial en *n*.

CIRCUIT-SAT ∈ NP-Difícil

- ***F* se ejecuta en tiempo polinomial.**
  - ← **Prueba de 2.**
    - . . .
    - El circuito *C* consiste de a lo más $t = O(n^k)$ copias de *M* y de aquí tiene un tamaño polinomial sobre *n*.
    - La construcción de *C* a partir de *x* se realiza en tiempo polinomial por el algoritmo de reducción *F*, dado que cada paso de la construcción toma un tiempo polinomial.

CIRCUIT-SAT ∈ NP-Difícil

- El lenguaje CIRCUIT-SAT es por lo tanto al menos tan difícil como cualquier lenguaje en NP, y dado que pertenece a NP, es NP-completo.