Clase teórica 9

Misceláneas de
Teoría de la Computación II

Aproximaciones polinomiales

- **Asumiendo P ≠ NP**, ante la imposibilidad de resolver los problemas NP-completos (y FNP-completos) de manera eficiente, surgen técnicas de **remediación.**

- Una de ellas se dedica a los problemas de búsqueda de **óptimos** (máximos o mínimos).

- Se trata de las **aproximaciones polinomiales:** la idea es construir algoritmos eficientes que produzcan soluciones "buenas", **cercanas al óptimo según un criterio determinado.**

- **Ejemplo.** Problema de búsqueda del cubrimiento de vértices mínimo de un grafo (MCV):

  CV = {(G, K): G tiene un cubrimiento de vértices de tamaño K (K vértices de G tocan todos sus arcos)}.

  CV es **NP-completo.** Así, si P ≠ NP, **no existe un algoritmo eficiente para encontrar el CV mínimo**:

  Si hubiera una MT M_1 que **obtiene** un CV mínimo C de un grafo G en tiempo poly(n),
  también habría una MT M_2 que **decide** si G tiene un CV de tamaño K en tiempo poly(n):

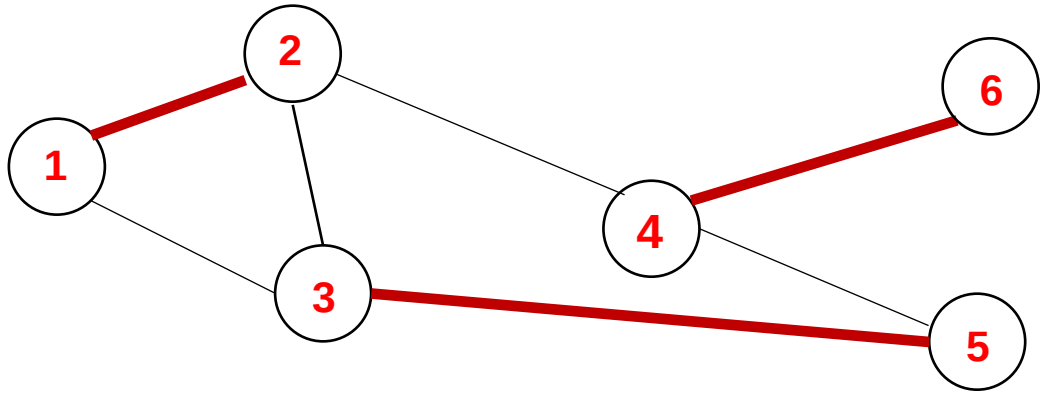
  1. Ejecuta M_1 a partir de G, y así obtiene un CV mínimo C de G.
  2. Acepta sii K ≥ |C|.

  De esta manera, recurrimos a una **aproximación polinomial** para resolver el problema (slide siguiente):

- La siguiente MT, dado el input G = (V, E), genera en tiempo poly(n) un cubrimiento de vértices de G **cercano al mínimo** (luego indicamos **cuán cercano**):

1. V' := ∅ y E' := E.
2. Si E' = ∅, acepta.
3. E' := E' – {(v_1, v_2)}, siendo (v_1, v_2) algún arco de E'.
4. Si $v_1 \notin V'$ y $v_2 \notin V'$, entonces V' := V' ∪ {v_1, v_2}.
5. Vuelve al paso 2.

Es decir, la MT genera un CV V´ con los vértices de los arcos disjuntos que va encontrando. Por ejemplo:



V´ = {1, 2, 3, 5, 4, 6}

- Llamando A al conjunto de arcos disjuntos detectado, naturalmente cualquier cubrimiento C tiene que incluir como mínimo un vértice de todo arco de A (si no, C no sería un cubrimiento).
- De esta manera, |C| ≥ |V'|/2, es decir, |V´| ≤ 2.|C|, y así, |V'| es **a lo sumo el doble del óptimo** (notar en el ejemplo que el cubrimiento mínimo mide 3, p.ej. {2, 3, 4}, mientras que V´ mide 6, es todo V).
- La MT construida trabaja en tiempo O(|E|.|V|) = **poly(n).**

- Dado un problema de optimización con instancias w, y una MT M planteada para resolverlo en tiempo poly(n),
  - si **opt(w)** es la medida de la solución óptima para w,
  - y **m(M(w))** es la medida de la solución obtenida por la MT M para w,
  - se define el **error relativo** ε_w de M con respecto a w del siguiente modo:

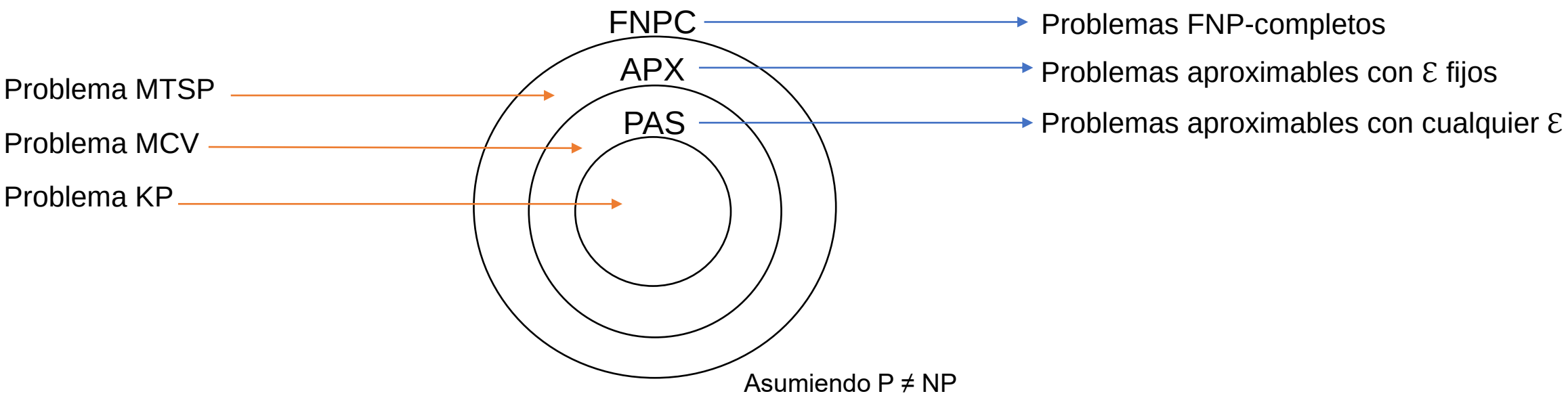
$$\varepsilon_w = \frac{|m(M(w)) - opt(w)|}{\max(m(M(w)), opt(w))}$$

  - ε_w siempre varía en el intervalo [0, 1).
  - Si ε es el máximo de todos los ε_w, se dice que M es una **aproximación polinomial con umbral** ε.
  - P.ej., la MT construida para el problema MCV cumple que $\varepsilon = |6 - 3|/\max(6, 3) = 3/6 = 1/2$, es decir que es una aproximación polinomial con umbral 1/2 (también se usa 1/2-aproximación polinomial).

- El objetivo es encontrar aproximaciones polinomiales con un umbral **lo más chico posible**.

- Asumiendo **P ≠ NP**:
  - Hay problemas aproximables con **cualquier** ε, problemas aproximables con ε **fijos** (como el que vimos), y problemas no aproximables (**no existen MT que los resuelvan con ningún** ε).
  - La **jerarquía** de problemas correspondiente es la siguiente:
    1. La clase **PAS** contiene los problemas aproximables con cualquier ε.
    2. La clase **APX** contiene los problemas aproximables con ε fijos.
    3. Se cumple **PAS** $\subset$ **APX** $\subset$ **FNPC**.

- Algunos problemas representativos de la jerarquía mencionada son los siguientes:



- El **problema de la mochila** (o KP por *Knapsack Problem*) plantea optimizar el armado de una mochila:
  - Se define el peso máximo que soporta un individuo.
  - Se considera un conjunto de objetos para colocar en la mochila, cada uno con peso y volumen.
  - El problema consiste en maximizar el volumen ocupado sin exceder el peso máximo.
  - KP es más fácil que MCV, a su vez más fácil que el problema de optimización del viajante de comercio (MTSP), en términos de las aproximaciones polinomiales.
  - También en este caso se plantean problemas completos, considerando reducciones de determinadas características.

MT probabilísticas

- Las **MT probabilísticas (PT)** son otra alternativa para la remediación en el marco de los problemas NP-completos, además de constituir por sí mismas una manera eficaz y de amplio espectro para resolver problemas.

- Son MT que trabajan en **tiempo poly(n)** y que en ciertos pasos eligen **aleatoriamente** una de entre dos o más continuaciones, por lo que en cada ejecución pueden generar un output distinto.

- El **criterio de aceptación** es distinto al considerado hasta ahora: aceptan/rechazan según el cociente entre las aceptaciones/rechazos y el total de computaciones (a más computaciones que aceptan, mayor **probabilidad** de aceptación, y lo mismo para el caso de los rechazos).

- Se definen **distintos tipos de PT**, y correspondientemente distintas clases de problemas (de igual nombre).

- Por ejemplo, un tipo de PT es la **máquina RP** (por *randomized polynomial time*, o tiempo polinomial aleatorio). Para todo input w:

  1. La máquina **acepta en al menos la mitad de sus computaciones** o **rechaza en todas**.
  2. La máquina **acepta w sii lo hace en al menos la mitad de sus computaciones**.

<u>Ejemplo</u>. Construcción de una PT para el problema de decidir si un número es compuesto (no primo).

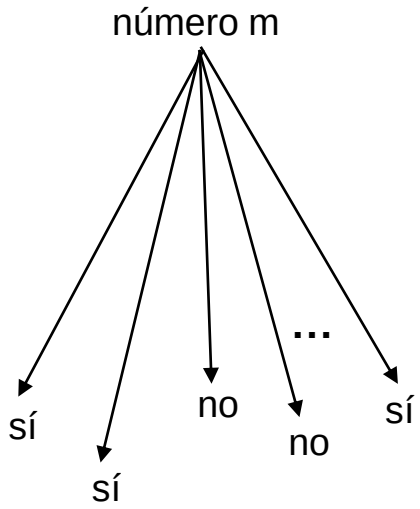
- La PT se basa en un teorema de Fermat, que establece que todo número compuesto impar m cumple que **al menos la mitad de los números entre 1 y m son testigos de la composicionalidad de m**: permiten verificar en tiempo poly(n) que m es compuesto (es decir, son **certificados suscintos**).

- En base al teorema, la siguiente máquina RP M, dado un input natural m, hace:

  1. Si m es par, acepta.
  2. Asigna aleatoriamente a la variable x un número entre 1 y m.
  3. Verifica si m es compuesto por medio del testigo x y responde adecuadamente.

- Notar que:
  a. Si m es compuesto, **al menos la mitad de las computaciones de M aceptan.**
  b. Si m es primo, **todas las computaciones de M rechazan.**

- De esta manera:
  a. Si m es compuesto, **la probabilidad de error** de M es: ≤ 1/2
  b. Si m es primo, **la probabilidad de error** de M es: 0



La máquina nunca acepta mal. Puede rechazar mal pero con baja probabilidad (≤ 1/2).

- M trabaja en tiempo **poly(n). Veamos cómo se puede achicar la probabilidad de error:**

- Sea la siguiente máquina M´ que ejecuta varias veces, por ejemplo 100, la máquina M que construimos recién (se asume que las ejecuciones son independientes). M´ hace:

  1. Si en alguna ejecución M acepta, entonces acepta.
  2. Si en cambio en todas las ejecuciones M rechaza, entonces rechaza.

- Notar ahora que:

  a. Si m es compuesto, la **probabilidad de error** de M´ es: $\leq (1/2)^{100}$
  b. Si m es primo, **la probabilidad de error** de M' es: 0

- Lo habitual para reducir la probabilidad de error de una PT es ejecutarla una cantidad poly(n) de veces.

- Otro tipo de PT es **la máquina BPP** (por *bounded probabilistic polynomial time*, o tiempo polinomial probabilístico acotado). Para todo input w, en tiempo poly(|w|):

  1. La máquina **acepta/rechaza en al menos 3/4 de sus computaciones.**
  2. La máquina acepta w sii **lo hace en al menos 3/4 de sus computaciones.**

  (se puede reemplazar 3/4 por cualquier otra fracción mayor que 1/2).

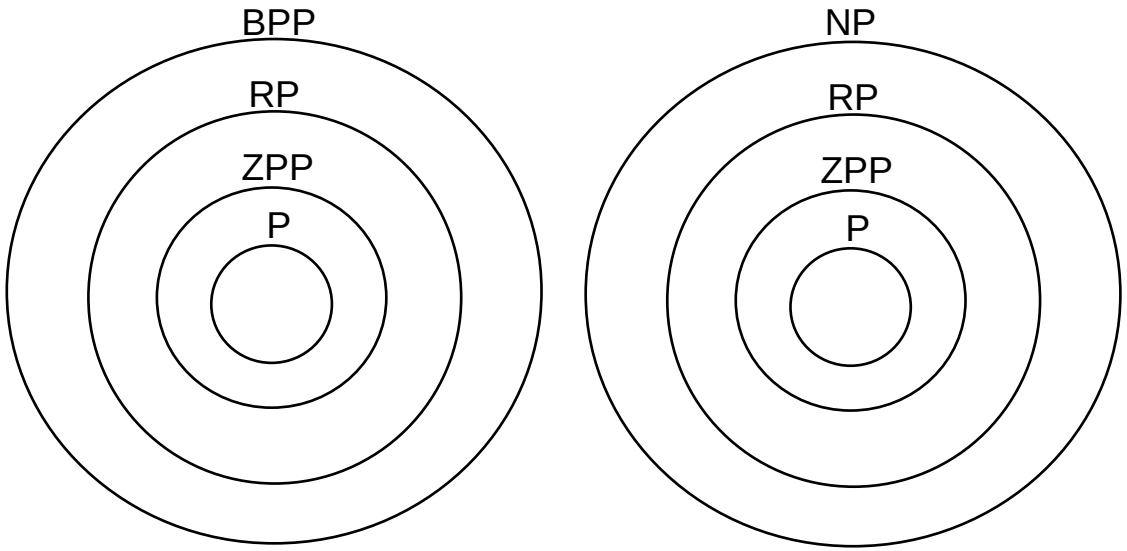
- A la clase de problemas BPP se la considera **la clase P probabilística**.

- También en este caso se puede **achicar la probabilidad de error** por repetición de ejecuciones.

- Una tercera clase que se considera es **ZPP** (*zero probabilistic polynomial time,* o tiempo polinomial probabilístico cero), que se obtiene de RP ⋂ CO-RP. Se cumple:

No se sabe si las inclusiones son estrictas.

Se desconoce si BPP $\subseteq$ NP.



La sospecha es que los problemas de ZPP, RP y BPP son tratables (P = ZPP = RP = BPP).

De esta manera, no se espera que ninguna máquina ZPP, RP o BPP resuelva un problema NP-completo.

- En la práctica, las PT apuntan a resolver problemas de una manera más simple o eficiente, en comparación con las MT clásicas. Un ejemplo es el problema de primalidad, que está en P.

- Se asume que los algoritmos son **implementables**, es decir que cuentan con **generadores aleatorios confiables**. Un área muy interesante en la teoría de la complejidad computacional trata los **generadores pseudoaleatorios** (trabajos iniciados por Kolmogorov y Chaitín):
  - Definición histórica: una cadena w es aleatoria si toda MT M que la genera cumple que **|<M>| ≥ |w|.**
  - Un generador pseudoaleatorio genera largas cadenas aleatorias a partir de pequeñas cadenas aleatorias, conocidas como **semillas aleatorias**.

Máquinas cuánticas

- En 1985, David Deutsch sugirió un modelo alternativo para definir y ejecutar algoritmos, un dispositivo que pudiera simular **cualquier sistema físico** considerando las leyes de la mecánica cuántica, que en definitiva son las que rigen el universo. Así surge la noción de **máquina cuántica**.

- Con las máquinas cuánticas se pone en duda la **tesis fuerte de Church-Turing**.

- En lugar de celdas con estados clásicos 0 o 1 (bits), en estas máquinas las celdas pueden contener **estados cuánticos superpuestos** 0 y 1 (**qubits**).

- Los estados cuánticos 0 y 1 se denotan con **|0>** y **|1>**. Un qubit, en el caso más general, puede estar en el estado de superposición $\boldsymbol{\alpha_0|0\rangle + \alpha_1|1\rangle}$**,** tal que:
  - Las α_i son **amplitudes** (probabilidades positivas o negativas).
  - Se cumple $\boldsymbol{\alpha_0^2 + \alpha_1^2 = 1}$ (lo que se conoce como probabilidad de norma 2).
  - Al leer un qubit, la probabilidad de que se lea |0> es α_0^2 y de que se lea |1> es α_1^2. Luego de la lectura se destruye la **coherencia cuántica**, y el qubit queda con |0> o |1>, según lo que se haya leído.

- En el caso más general, las amplitudes α_i son números **complejos**. Con números **reales** se simplifican las descripciones, al tiempo que el poder de lo cuántico ya se percibe utilizando dicho dominio.

- Por ejemplo, en un sistema de 2 qubits pueden existir 4 estados cuánticos:

  |00>, |01>, |10> y |11>

  que se describen de la siguiente manera:

  $\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$, con **$\alpha_{00}^2 + \alpha_{01}^2 + \alpha_{10}^2 + \alpha_{11}^2 = 1$**

  Cuando el sistema es observado, el estado revelado es:

  |00> con probabilidad α_{00}^2, |01> con probabilidad α_{01}^2, …

  y luego se destruye la coherencia cuántica.

- Generalizando, un estado cuántico se representa por la expresión:

  $\alpha_1|s_1\rangle + \ldots + \alpha_k|s_k\rangle$

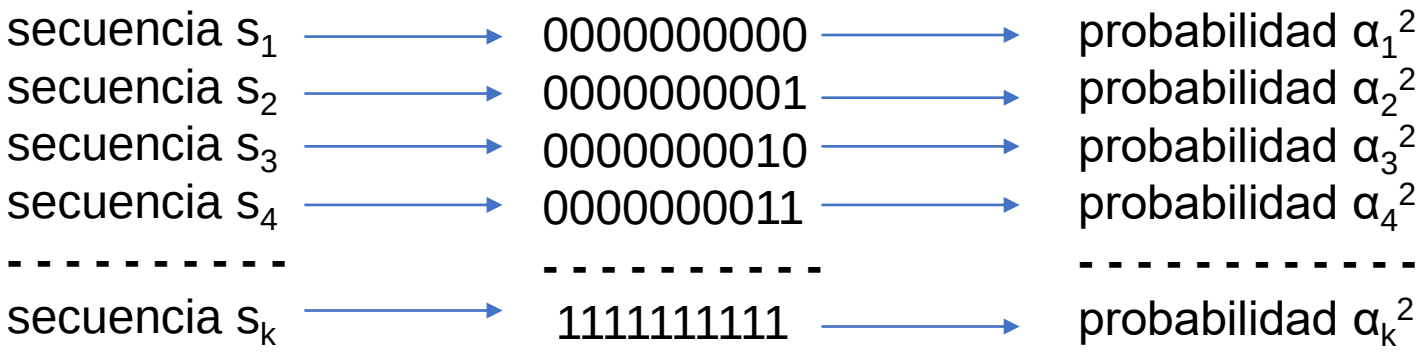
  siendo las s_i secuencias de 0 y 1, y las α_i amplitudes que cumplen la ecuación $\alpha_1^2 + \ldots + \alpha_k^2 = 1$

- Notar entonces que 1 qubit puede representar la misma información que 2 bits,
  2 qubits pueden representar la misma información que 4 pares de bits,
  3 qubits pueden representar la misma información que 8 ternas de bits,
  y generalizando: **n qubits pueden representar la misma información que 2^n n-tuplas de bits.**

- Esto último explica la esencia de la potencia computacional de las máquinas cuánticas, algo así como la posibilidad de procesar en **paralelo** secuencias de 0 y 1 en lugar de hacerlo secuencia por secuencia (pero con una cierta **probabilidad de error**, lo mismo que vimos antes con las PT).

Más detalle de lo anterior

- Dado un sistema, supongamos de 10 qubits, los posibles estados cuánticos varian desde |0000000000> hasta |1111111111>. Podríamos suponer entonces como que en todo momento la máquina mantiene en paralelo a lo sumo $k = 2^{10}$ secuencias de 0 y 1:



  - La sumatoria de los α_i^2 en todo momento es 1.
  - En la lectura final se detecta la secuencia con mayor probabilidad (el mayor valor α_i^2) y se destruye la coherencia cuántica.
  - En cada paso se opera una **compuerta cuántica** sobre 1, 2 o 3 qubits.

- El tiempo se mide en **cantidad de ejecuciones sobre compuertas cuánticas**.

Computaciones y compuertas cuánticas

- Las computaciones cuánticas se estructuran en tres pasos:

  1. **Inicio.** Preparación del input en el estado cuántico inicial del sistema.
  2. **Evolución.** Computación cuántica propiamente dicha.
  3. **Lectura** del output obtenido.

- Las operaciones se modelizan con **álgebra matricial** con determinadas restricciones (p.ej., las matrices utilizadas son **unitarias**, lo que permite que cada operación sea **reversible**), propio de la mecánica cuántica, y se implementan, como en el caso de las máquinas clásicas, por circuitos booleanos, ahora cuánticos (sus compuertas se conocen como **compuertas cuánticas**).

- Cada tipo de compuerta se aplica a uno, dos o tres qubits. Ejemplos de compuertas son:

  - La compuerta **NOT** trabaja así: $|0\rangle \longrightarrow |1\rangle$ y $|1\rangle \longrightarrow |0\rangle$.
  - La compuerta **Hadamard** trabaja así: $|0\rangle \longrightarrow \sqrt{1/2}\,|0\rangle + \sqrt{1/2}\,|1\rangle$ y $|1\rangle \longrightarrow \sqrt{1/2}\,|0\rangle - \sqrt{1/2}\,|1\rangle$.
  - La compuerta **Toffoli** se aplica a 3 qubits, y los modifica sólo si los primeros dos son $|11\rangle$, de la siguiente manera: $|110\rangle \longrightarrow |111\rangle$ y $|111\rangle \longrightarrow |110\rangle$, es decir que invierte el tercer qubit.

- Se prueba que las compuertas Hadamard y Toffoli constituyen un **conjunto universal** de compuertas cuánticas.

- La clase de problemas que pueden ser resueltos por algoritmos cuánticos con poly(n) operaciones y con una probabilidad de error menor que 1/4, se denomina **BQP** (por *Bounded error, Quantum, Polynomial time).* En realidad, por la posibilidad de iterar ejecuciones, el valor puede ser cualquiera menor que 1/2 (como en las máquinas probabilísticas).

- BQP incluye problemas de interés **de los que no se conocen soluciones polinomiales utilizando máquinas clásicas**. Se cumple que: **P** $\subseteq$ **BPP** $\subseteq$ **BQP** $\subseteq$ **PSPACE.** No se sabe si las inclusiones son estrictas. Tampoco se conoce la relación entre BQP y NP (se conjetura que **BQP no incluye a los problemas NP-completos**).

- **El algoritmo cuántico de Deutsch y Jozsa** fue uno de los primeros algoritmos cuánticos. Determina si una función booleana sobre n argumentos es constante (0 o 1) o equilibrada (igual cantidad de 0 y 1) de una manera mucho más rápida que un algoritmo clásico (en lugar de considerar secuencialmente 2^n secuencias de 0 y 1 las trata simultáneamente).

- **El algoritmo cuántico de Shor** para factorizar números trabaja en **tiempo polinomial** (si se materializaran las computadoras cuánticas habrá entonces que descartar los sistemas de criptografía usados masivamente en la actualidad).

- Otro algoritmo que se destaca es el **algoritmo cuántico de Grover**. Mientras que por medio de los algoritmos clásicos encontrar un dato en una base de datos desordenada de tamaño N requiere en promedio N/2 búsquedas, este algoritmo lo hace en tiempo O($\sqrt{N}$) - **aceleración cuadrática** -.

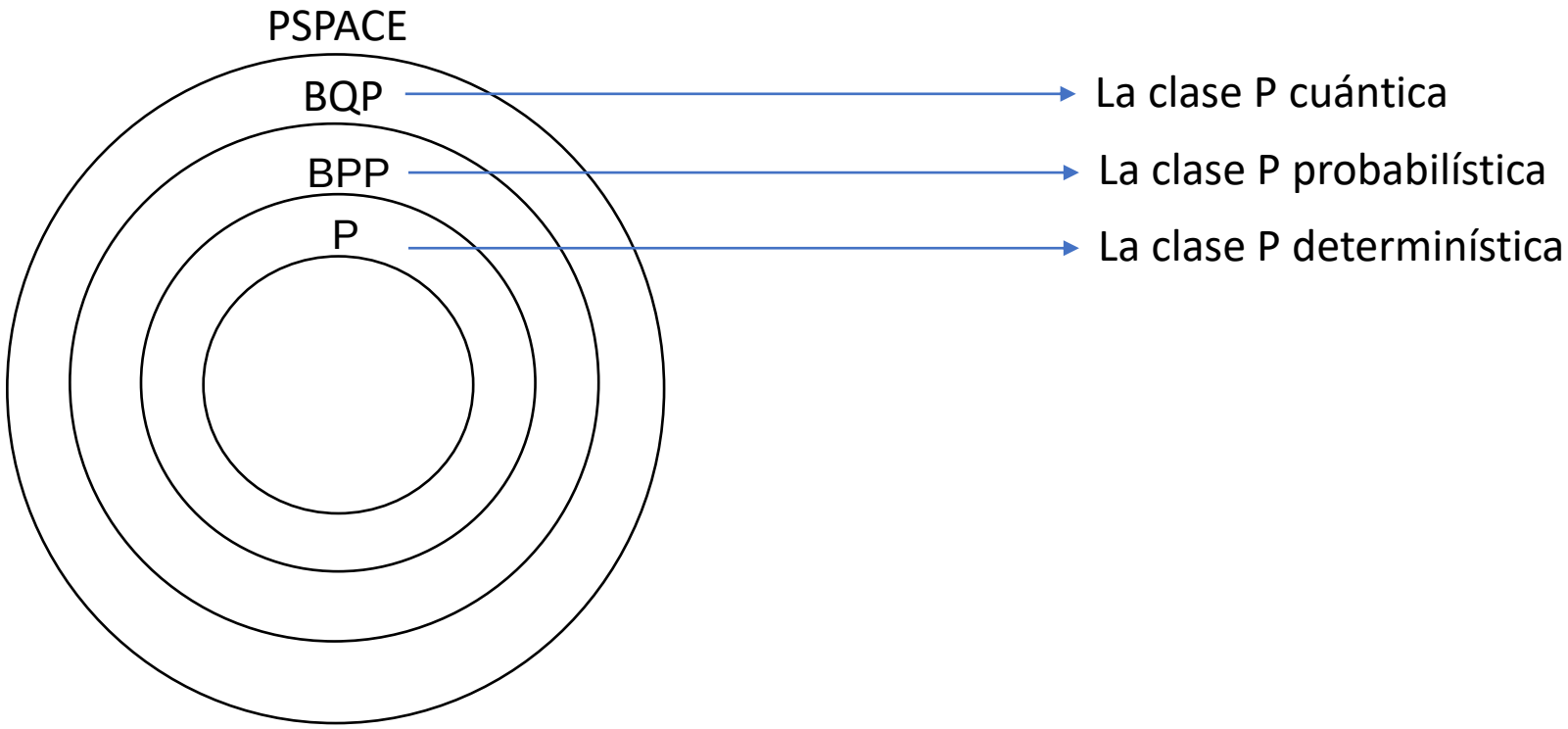
- La ubicación de la **clase BQP** en la jerarquía de la complejidad temporal sería entonces la siguiente:

- BQP reúne a todos los problemas que se pueden resolver en tiempo polinomial cuántico con una probabilidad de error menor que 1/4.

- La clase BQP no incluiría los problemas NP-completos.

- No se conoce la relación entre BQP y NP.

- Podría cumplirse P = BPP.



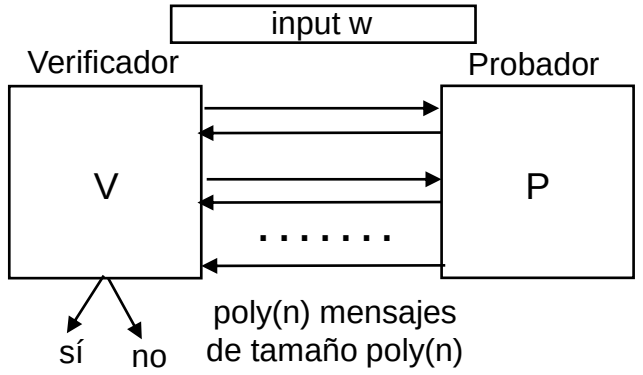
- **Los algoritmos cuánticos son un paso más allá de los algoritmos probabilísticos** (probabilidades positivas y negativas, superposición cuántica, lectura destructiva, etc). Por otro lado en ellos se mantienen nociones previas: tanto en BPP como en BQP la idea es que las máquinas trabajen en tiempo polinomial y se equivoquen con una probabilidad acotada, que tiende a cero iterando ejecuciones independientes.

Anexo de la clase teórica 9

Misceláneas de Teoría de la Computación II

Otras misceláneas

Sistemas de pruebas interactivas



- En el caso básico generalizan las MT que aceptan lenguajes de NP:
  - Están constituidos por 2 MT, P (probador) y V (verificador).
  - Dado un input, P y V se intercambian mensajes.
  - V inicia y termina la computación, aceptando o rechazando.
  - P no tiene restricciones de espacio ni de tiempo. V en cambio trabaja en tiempo poly(n).
  - La cantidad y longitud de los mensajes es poly(n).
  - La clase de lenguajes aceptados por este modelo se denomina **DIP** (por *deterministic interactions proofs*). Se cumple **DIP = NP.**
  - En síntesis, ahora, en lugar del envío de un solo mensaje suscinto del probador al verificador, se envían una cantidad polinomial de mensajes suscintos. Y el probador le permite al verificador consultas intermedias.

- En una variante del modelo, V es una máquina probabilística:
  - En este caso, la clase de lenguajes aceptados se denomina **IP**. Se cumple **IP = PSPACE**. Sorprendemente, haciendo probabilístico al verificador la potencia del sistema crece sobremanera.
  - Un ejemplo clásico de problema dentro de la clase IP es el problema de los grafos no isomorfos, que no estaría en NP.

- Conceptos importantes derivados de esta variante son:
  - **Pruebas de conocimiento cero** (*zero knowledge proofs*). De interés para la criptografía. La idea es que P lo convenza a V con mínima información, sin revelar la solución.
  - **Chequeo probabilístico de pruebas** (*probabilistic checking of proofs*). También ligado a la criptografía y a la limitación de la información compartida. En este caso P tiene una prueba completa x que V debe verificar (como vimos en NP). Pero ahora la cuestión es minimizar: (a) los bits de x, (b) los accesos a x, que V necesita para convencerse. En base a estos parámetros se definen las clases de problemas PCP(f, g), siendo O(f(n)) y O(g(n)) la cantidad de bits y la cantidad de accesos, respectivamente.

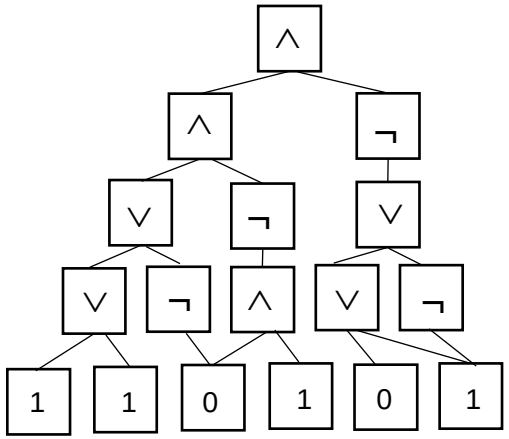
Máquinas para paralelismo

- El objetivo es estudiar problemas cuyas soluciones paralelas sean **significativamente** más rápidas que sus soluciones secuenciales.

- Un modelo muy utilizado para esto es la **familia uniforme de circuitos booleanos**, secuencias de circuitos booleanos c_n con n nodos de input y un nodo de output. El *tamaño* de c_n se denota con $H(c_n)$, su *profundidad* con $T(c_n)$, que representa el **tiempo paralelo**, y su *costo* es el producto $H(c_n).T(c_n)$, es decir la cantidad de pasos ejecutados conjuntamente. Como todo algoritmo paralelo puede simularse por un algoritmo secuencial, el costo no puede ser menor que el mejor tiempo de una solución secuencial equivalente.

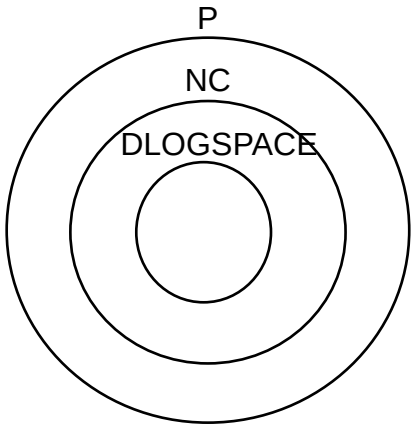
- Los problemas aceptados por este modelo, cuando $H(c_n) = poly(n)$ y $T(c_n) = \log^k(n)$, forman la clase **NC**, que caracteriza a los problemas con **solución paralela eficiente.** Problemas clásicos de esta clase son las operaciones aritméticas elementales.

- Se prueba que **NC** $\subseteq$ **P**. Se sospecha que la inclusión es estricta. Así, encontrar lenguajes P-completos (con respecto a una clase particular de reducciones) en la práctica implica probar que no están en NC. Un problema clásico de problema P-completo es justamente la evaluación de los circuitos booleanos.



Ejemplo de circuito booleano

Asumiendo NC $\subset$ P y DLOGSPACE $\subset$ P, un lenguaje P-completo entonces no admite una solución eficiente en tiempo paralelo ni en espacio.



En otras palabras, toda solución para un lenguaje P-completo requeriría mucha información simultánea en memoria.

Clase práctica 9

Misceláneas de Teoría de la Computación II

<u>Ejemplo 1</u>. Algoritmo probabilístico clásico para el problema CSAT, conocido como *random walk*.

1. Rechazar si el input no es una fórmula f.
2. Elegir una asignación A, y si satisface f, aceptar.
3. Repetir k veces (k se explica abajo):
    a. Elegir una cláusula C de f no satisfecha por A (todos los literales son falsos, por la sintaxis de CSAT).
    b. Invertir el valor de verdad de uno de los literales de C.
    c. Si la nueva A satisface f, aceptar.
4. Rechazar.

- Para 3-SAT, este algoritmo requiere un k exponencial (es decir, no funciona eficientemente).
- En cambio para 2-SAT, con $k = O(n^2)$, se demuestra que funciona con una probabilidad de error baja.
- Recordar que antes se vio el algoritmo determinístico polinomial para 2-SAT (tiempo $O(n^2)$).
- Claramente, la simplicidad del *random walk* puede ameritar su uso (confiando en la aleatoriedad).
- Hay muchas variantes del *random walk* que pueden mejorar la performance.

Ejemplo 2. Reducciones logarítmicas en espacio.

- Las reducciones logarítmicas en espacio (o log-space) son reducciones computadas en espacio logarítmico, y por lo tanto en tiempo polinomial. En el cálculo del espacio consumido no se consideran ni la cinta de entrada ni la cinta de salida.
- Son útiles por ejemplo para probar que un problema es de los más difíciles de una clase, cuando las reducciones polinomiales en tiempo (o poly-time) no son la herramienta adecuada. Un caso es el del problema CIRCUIT VALUE:

  CIRCUIT VALUE es el problema que consiste en evaluar un circuito booleano (el resultado es verdadero o falso). Está en P, y se sospecha que no está en LOGSPACE.

  Como todos los problemas de P se reducen entre sí en tiempo polinomial, no sirve en este caso demostrar que CIRCUIT VALUE es P-completo con respecto a las reducciones poly-time. Sí sirve utilizar reducciones log-space: se prueba que todos los problemas de LOGSPACE se reducen con reducciones log-space a CIRCUIT VALUE, entonces si CIRCUIT VALUE estuviera en LOGSPACE se estaría probando LOGSPACE = P. Ante la sospecha de que no se cumple esta igualdad, entonces la P-completitud de CIRCUIT VALUE con respecto a las reducciones log-space lo "condena" en la práctica a no estar en LOGSPACE.

- Hay una complicación técnica al componer una MT M_f que computa una reducción log-space con otra MT, por ejemplo una MT M_{cv} que resuelve CIRCUIT VALUE en tiempo poly(n): a partir de una cadena w, con |w| = n, la salida de M_f puede llegar a medir $O(n^k)$, con k constante, es decir un polinomio (espacio mayor que logarítmico). Esta dificultad se resuelve del siguiente modo:

  M_f no le envía la cadena completa f(w) a M_{cv}, sino un símbolo por vez, el que requiera M_{cv} en todo momento.
  M_{cv} se va ejecutando sobre la cadena f(w). Cuando requiere el 1er símbolo, M_f se ejecuta hasta generarlo y se lo envía. Cuando requiere el 2do sucede lo mismo. Y así siguiendo. Y toda vez que M_{cv} requiera un símbolo anterior del último recibido, M_f vuelve a ejecutarse desde el principio hasta alcanzar el símbolo requerido y enviarlo.