Facultad de Matemática, Astronomía, Física y Computación

Trabajo especial de la Licenciatura en Ciencias de la Computación

De PH a IP, un curso en Complejidad Computacional.

Ignacio Made Vollenweider

Director:
Dr. Miguel Campercholi

Noviembre de 2019

Se distribuye bajo la Licencia Creative Commons Attribución 4.0 International

Resumen

En este trabajo estudiamos algunas de las clases más importantes de la teoría de Complejidad Computacional. Nos basamos en el programa que propone el libro Computational Complexity a modern approach [8], del cual vemos la segunda mitad de la primera parte del programa (excluyendo Criptografía, Computación Cuántica y el Teorema PCP). En particular, estudiamos la clase de la Jerarquía Polinomial (**PH**), la clase de Circuitos Booleanos ($\mathbf{P}_{/\mathbf{poly}}$), la Computación Randomizada (**BPP**) y los Protocolos Interactivos (**IP**). Además vemos las principales técnicas de la teoría para obtener resultados las cuales son Diagonalización, Lower bounds y Arithmetization. Y estudiamos también sus respectivas limitaciones: Relativización, Natural proofs y Algebrization.

Palabras clave: Máquina de Turing - Clase de complejidad - Lenguaje - P vs NP - SAT - Polinomial.

Abstract

In this work we study some of the most important classes of the Computational Complexity Theory. We base on the program proposed by the book Computational Complexity a modern approach [8], of which we see the second half of the first part of the program (excluding Cryptography, Quantum Computing and the PCP Theorem). In particular, we study the class of the Polynomial Hierarchy (**PH**), the class of Boolean Circuits ($\mathbf{P}_{/\mathbf{poly}}$), the Randomized Computing (**BPP**) and the Interactive Protocols (**IP**). In addition we see the main techniques of the theory to obtain results which are Diagonalization, Lower bounds and Arithmetization. And we also study their respective limitations: Relativization, Natural proofs and Algebrization.

Key words: Turing machine - Complexity class - Lenguage - P vs NP - SAT - Polynomial.

Agradecimientos

A mis padres, por su sacrificio y esfuerzo,
a mi abuelo Juan, "cada problema tiene su solución",
a Camper, por su guía y amabilidad,
y a FaMAF, por no cortar alas y ayudar a volar.

Índice general

1 Preliminares **6**

2 Diagonalización **20**

3 Jerarquía Polinomial: PH **35**

4 Circuitos Booleanos: P/poly **45**

5 Computación Randomizada: BPP **72**

6 Protocolos Interactivos: IP **82**

Capítulo 1

Preliminares

> Los límites de mi lenguaje son los límites de mi mundo.
>
> Ludwig Wittgenstein

La teoría de Complejidad Computacional estudia la dificultad de los problemas que pueden ser resueltos por algoritmos. Para ello da un modelo matemático de algoritmo, de problema y de dificultad. Este modelo matemático llevado a la práctica son los programas, los problemas que ellos pueden resolver y el tiempo o el espacio que consumen en hacerlo.

En este capítulo repasaremos los conceptos fundamentales de la teoría de Complejidad Computacional y supondremos que el lector está familiarizado con los mismos. También veremos las elecciones y convenciones cruciales que seguiremos a lo largo del trabajo.

Modelo de Computación

> La idea detrás de las computadoras digitales puede ser explicada diciendo que las máquinas son diseñadas para poder realizar toda operación que pueda ser llevada a cabo por una computadora humana. Se supone que la computadora humana sigue unas reglas fijas; ella no tiene autoridad para poder desviarse de las mismas en el más mínimo detalle. Podemos suponer que tales reglas son brindadas en un libro, el cual es modificado cuando es utilizado para una nueva tarea. Ella también tiene un suministro ilimitado de papel donde hace sus cálculos.
>
> Alan Turing, 1950

> [Turing] ha tenido éxito en dar una definición absoluta de una interesante noción epistemológica, i.e., una que no depende del formalismo elegido
>
> Kurt Gödel, 1946

Definición 1 (Notación $\mathcal{O}$ y o). Sean $f : \mathbb{N} \longrightarrow \mathbb{N}$ y $g : \mathbb{N} \longrightarrow \mathbb{N}$ funciones.

- Decimos que $g \in \mathcal{O}(f)$ si $\exists c > 0$ y $\exists n_0 \in \mathbb{N}$ tal que $\forall n \in \mathbb{N}$ si $n > n_0$ entonces $g(n) \leq cf(n)$.
- Decimos que $g \in \mathrm{o}(f)$ si $\forall \epsilon > 0$ y $\exists n_0 \in \mathbb{N}$ tal que $\forall n \in \mathbb{N}$ si $n > n_0$ entonces $g(n) \leq \epsilon f(n)$.

Usualmente diremos que g es $\mathcal{O}(f)$ o que $g = \mathcal{O}(f)$ para denotar que $g \in \mathcal{O}(f)$. Análogamente con g y $\mathrm{o}(f)$.

Intuitivamente, un *problema de decisión* es un problema que solo admite "si" o "no" como respuesta. Como por ejemplo, determinar si un número es primo. Un número es primo o no lo es. Nos interesarán aquellos problemas de decisión que puedan ser resueltos por algoritmos. En nuestro ejemplo querríamos un algoritmo que dado cualquier número natural de entrada nos devuelva: sí, si es primo y no, en el caso contrario.

Definición 2 (Lenguajes y Problemas de Decisión). Sea Σ un alfabeto (i.e. un conjunto finito no vacío) de al menos dos elementos y sea Σ^* el conjunto de palabras finitas formadas a partir de Σ, incluyendo la palabra de largo 0 denotada por ε. Un *lenguaje* sobre Σ es un subconjunto L de Σ^*. Un *problema de decisión* será para nosotros un lenguaje sobre $\{0, 1\}$, es decir, un subconjunto de $\{0, 1\}^*$.

Como convención, en el trabajo nos referiremos con lenguaje a los problemas de decisión, es decir, un lenguaje será un subconjunto de $\{0, 1\}^*$.

Puede ser un poco difícil imaginar cómo se puede codificar un problema como un lenguaje, por ejemplo el problema de determinar si un número es primo. Hay muchas maneras de representar un problema como un lenguaje. Nosotros elegimos la siguiente: dado un problema a representar primero elegimos una codificación eficiente para las *instancias*, en nuestro ejemplo sería dar una manera de codificar números naturales tal que sea eficiente (i.e. polinomial) decidir si una palabra $\alpha \in \{0, 1\}^*$ es un número natural según nuestra codificación (tal eficiencia será definida a la brevedad cuando se vea el modelo de algoritmo, i.e, la *máquina de Turing*). Entonces, el lenguaje que representa al problema es el conjunto las *instancias positivas* de ese problema. En nuestro ejemplo de números primos si elegimos codificar a los números naturales como su codificación en binario entonces el problema de determinar si un número primo es representado por el lenguaje: $\{10, 11, 101, 111, 1011, \ldots\}$. Luego dado un número natural, para determinar si es primo, basta con ver si su codificación en binario pertenece al lenguaje. Vale observar que por ejemplo la palabra 010 no pertenece al lenguaje porque ni siquiera es una instancia (un número natural según nuestra codificación) pero es eficiente determinarlo. Puede haber muchas maneras de codificar números naturales, cualquiera que permita eficientemente determinar si una palabra sigue o no a tal codificación es válida. En particular, también es válido y en ocasiones útil directamente elegir una codificación tal que toda palabra en $\{0, 1\}^*$ sea una instancia. Simplemente se conviene que las palabras que originalmente no eran una instancia pasan a ser todas la misma instancia

canónica. En nuestro ejemplo se podría definir que si una palabra es de la forma 0α con $\alpha \in \{0,1\}^*$ y $\alpha \neq \epsilon$ entonces 0α representa al número natural 1.

A continuación daremos el modelo matemático de algoritmo determinista: la máquina de Turing determinista monocinta.

Definición 3 (Máquina de Turing determinista (monocinta))**.** Una *Máquina de Turing determinista* es una 5-upla $(Q, \Gamma, \delta, q_i, q_f)$ donde:

- Q es un conjunto finito no vacío cuyos elementos son llamados *estados*. Cuenta con dos estados especiales q_i, q_f que denotan al estado inicial y final respectivamente.

- Γ es un conjunto finito que contiene a $\{0, 1, \Box, \triangleright\}$ llamado *alfabeto* de la máquina, y son los símbolos que pueden ocurrir en su cinta (tiene una sola cinta).

- $\delta : (Q \setminus \{q_f\}) \times \Gamma \longrightarrow Q \times \Gamma \times MOV$ es una función total llamada *función de transición.*

Donde $MOV = \{I, D, S\}$ el conjunto de movimientos del cabezal de la máquina, (I izquierda, D derecha y S quedarse (stay)).

La máquina es *monocinta* (tiene una sola cinta) donde realiza sus computaciones. Ahí aparece escrita la entrada, realiza los cómputos a partir de ella y luego escribe la salida. A la cinta la podemos imaginar como una fila infinita de casilleros, en los cuales en cada uno ocurre un único símbolo del alfabeto de la máquina. Asumimos que la cinta antes de ser escrita por la máquina y antes de recibir una entrada para procesar tiene la siguiente forma:

$$\triangleright \Box \Box \Box \Box \Box \Box \Box \Box \Box \Box \Box \Box \Box \Box \Box \dots$$

Sea M una máquina de Turing determinista (abreviado MT) y $x \in \{0,1\}^*$. Decimos que *M parte de x, o que parte de la entrada x, o que recibe a x de entrada*, para indicar que la máquina comienza en el estado inicial con el cabezal apuntando al símbolo $\triangleright$ y a la derecha de este símbolo está escrita x. Por ejemplo si $x = \{10010\}$ entonces la cinta se vería:

$$\dot{\triangleright}10010 \Box \Box \Box \Box \Box \Box \Box \Box \Box \dots$$

En representaciones del estado de una MT como la de arriba, indicamos la posición del cabezal marcando con un punto al símbolo en el cual se encuentra (en el ejemplo el cabezal está sobre $\triangleright$). A la izquierda del símbolo $\triangleright$ la cinta finaliza (la máquina no puede moverse a la izquierda), y a la derecha de x hay infinito casilleros en blanco (representados por $\Box$).

Decimos que la máquina *se detiene* (o finaliza) cuando llega al estado final, desde donde no puede realizar ningún movimiento. (Notar que en nuestra definición de la función de transición de una MT excluimos los pares (q_f, r) del dominio.) Dado $x \in \{0,1\}^*$ y $t \in \mathbb{N}$, decimos que *M a partir de x corre en t*

pasos si recibiendo x de entrada M finaliza en a lo sumo t pasos. Contamos como un paso a una aplicación de δ. Dada una función $T : \mathbb{N} \longrightarrow \mathbb{N}$ decimos que M *corre en tiempo* T si para todo $x \in \{0,1\}^*$, M corre en $T(|x|)$ pasos. Con $M(x)$ denotamos a la *salida* de la máquina que es la palabra escrita en la cinta al llegar al estado final, (ignorando los infinitos símbolos $\square$) es decir la palabra más larga que no termina en $\square$. Asumimos que la cinta en el estado final tiene la siguiente forma:

$$\triangleright \alpha \square \square \square \square \square \square \square \square \square \ldots$$

con $\alpha \in \Gamma^*$ la salida de la máquina.

Dada una máquina y una entrada, en cualquier momento de le ejecución uno podría "pausar" la ejecución y "sacarle una foto" a la máquina. Es decir, "tomar una fotografía" del contenido de la cinta, con la posición del cabezal y el estado en el cual se encuentra la máquina. Tal imagen es el concepto de *configuración*, el cual es una herramienta útil para analizar el fenómeno de la computación. En particular, tiene una gran importancia en el teorema de Cook-Levin (Teorema 15), y en otros resultados que veremos en el trabajo.

Definición 4 (Configuración)**.** Sea M una MT, una *configuración* de M es una tripla (q, α, l) con $q \in Q$, $\alpha \in \Gamma^*$ y $l \in \mathbb{N}$, donde q es el estado en la cual se encuentra la máquina, α es el contenido de la cinta (excluyendo los infinitos símbolos $\square$) y l es la posición del cabezal en la cinta. Dado $x \in \{0,1\}^*$, la *configuración inicial* de M a partir de x, denotada por $c_{inicial}$, será la tripla $(q_i, \triangleright x, 1)$.

Una ejecución de M a partir de la entrada x puede representarse como una sucesión de configuraciones $c_1, c_2, \ldots$ donde $c_1 = c_{inicial}$ y cada otra configuración se obtiene de la anterior por una aplicación de la función de transición de M.

Definición 5 (Función Computable y Lenguaje Decidible)**.**

- Sea $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ una función. Decimos que la máquina de Turing M *computa* a f si $\forall x \in \{0,1\}^*$ $M(x) = f(x)$.
- Sea $L \subseteq \{0,1\}^*$ un lenguaje. Decimos que la máquina de Turing M *decide* a L si M computa a la función característica de L. Es decir, M decide a L si M finaliza para toda entrada, solo devuelve 1 o 0 y

$$\forall x \in \{0,1\}^* \; x \in L \text{ sii } M(x) = 1.$$

El hecho es que en nuestro campo de estudio nos movemos en el "mundo" de lo computable. Hay contadas excepciones como la clase $\mathbf{P_{/poly}}$ que veremos en el capítulo de circuitos y que contiene lenguajes no decidibles (ver teorema 56). Pero en general todo lenguaje será decidible y toda función computable. Como en general estudiaremos solo máquinas que deciden lenguajes, estás van a tener que finalizar para toda entrada. Luego, una manera válida de pensar una máquina es pensar en una función que transforma palabras de $\{0,1\}^*$ a palabras de $\{0,1\}^*$.

Por otro lado, hemos hecho varias elecciones o convenciones al definir la MT, como por ejemplo que es monocinta, que la cinta es infinita hacia la derecha, que la máquina cuenta con el movimiento S (de quedarse) y otras. Las razones en orden de prioridad son:

- La robustez de las máquinas de Turing hace que estos detalles de "bajo nivel" solo tengan un costo polinomial en tiempo cuando se pasa de una "convención" a otra y en general incluso este costo es cuadrático o cúbico. Usualmente al consultar la bibliografía uno sigue el modelo que más le gusta al autor sabiendo que "todos los caminos llevan a roma", toda variante de la máquina de Turing Determinista que se use lleva al mismo paradigma de computación.
- En mi opinión son más fáciles de imaginar y es más fácil razonar sobre ellas en los tópicos que este trabajo trata.
- Simplifican las demostraciones.

Vale agregar que desde los 50's se han descubierto otros modelos de computación, cada uno de ellos simulable por MT's. Por tal razón el modelo no importa, parece ser que las MT's son lo suficientemente universales para abarcarlos a todos. Estudiando este paradigma uno siente que más allá de las interesantes máquinas de Turing, uno está estudiando al fenómeno de la computación. Y la *tesis de Church-Turing* apunta a ello, la cual expresa: todo dispositivo computacional que pueda ser implementado físicamente, sin importar si está hecho de silicio, de neuronas o de cualquier tecnología alienígena; puede ser simulado por máquinas de Turing.

Clases de Complejidad Básicas

La clase P

A continuación tal vez la clase más importante de todas: la clase de los problemas eficientemente solucionables.

Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función, decimos que un lenguaje L (recordando: un subconjunto de $\{0,1\}^*$) pertenece a la clase de complejidad $\mathbf{DTIME}(T)$ si existe una MT M que decide a L y una constante $k \in \mathbb{N}$, tal que para toda entrada $x \in \{0,1\}^*$, M corre en $kT(|x|)$ pasos. Como es habitual, al emplear la notación $\mathcal{O}$ o **DTIME**, a veces denotaremos funciones de $\mathbb{N} \longrightarrow \mathbb{N}$ por expresiones algebraicas que describen su compartimiento. Por ejemplo, denotaremos la función $n \mapsto n^3$ simplemente por n^3, y escribiremos $\mathbf{DTIME}(n^3)$.

Si uno toma a un programador y le pide que defina computación eficiente, lo más probable es que conteste que computación eficiente es aquella lineal o cuadrática. Como en general los programadores hacen programas que llaman a otros programas como subrutinas, sería natural considerar que un programa es eficiente si las subrutinas también lo son. Luego, la noción de computación eficiente obtenida deriva exactamente en **P**.

Definición 6 (La clase **P**)**.** Definimos $\mathbf{P} = \bigcup_{c\geq 1}^{\infty} \mathbf{DTIME}(n^c)$.

Vale comentar que hay una versión *fuerte de la tesis de Church-Turing*, la cual afirma que todo dispositivo computacional realizable físicamente puede ser simulado por máquinas de Turing polinomialmente! De ser cierto, entonces la clase **P** definida por cualquier civilización alienígena sería idéntica a nuestra clase **P**! De todas maneras parece no ser el caso, no hace falta imaginar dispositivos extraños o viajar a Hogwarts para poner en serios aprietos a esta afirmación. Las computadoras cuánticas aparentan no poder ser eficientemente simulables por MT's y evidencia de ello es el famoso algoritmo de Shor [34].

La clase EXP

Con **EXP** modelamos la clase de los lenguajes decidibles en tiempo exponencial.

Definición 7. Definimos

$$\mathbf{EXP} = \bigcup_{c\geq 1}^{\infty} \mathbf{DTIME}(2^{n^c}).$$

Por citar algunos ejemplos, algunas generalizaciones del ajedrez y del juego Go son problemas en **EXP**.

La clase NP

> Uno obviamente puede fácilmente construir una máquina de Turing que para toda fórmula de primer orden F y para todo número n, le permita a uno determinar si hay o no una prueba de F de tamaño n (tamaño = cantidad de símbolos). Sea $\Psi(F, n)$ la cantidad de pasos que la máquina requiere para ello y sea $\varphi(n) = \max_F \Psi(F, n)$. La pregunta es qué tan rápido crece $\varphi(n)$ para una máquina óptima. Uno puede mostrar que $\varphi(n) \geq kn$. Si realmente hubiese una máquina con $\varphi(n) \sim kn$ (o incluso $\sim kn^2$)[1], esto tendría consecuencias de la mayor magnitud. Particularmente, indicaría claramente que más allá de la imposibilidad de resolver el Entscheidungsproblem [Hilbert], el esfuerzo mental de los matemáticos al tratar de responder preguntas del tipo "si o no" sería completamente reemplazado por máquinas. Después de todo, uno simplemente debería elegir un número n lo suficientemente grande para que cuando la máquina no devolviera una respuesta positiva, no tendría más sentido seguir pensando el problema. Sin embargo, me parece que ello no tiene posibilidad alguna. Ya que $\varphi(n) \geq kn$ es la única estimación que uno puede obtener mediante la generalización de la prueba de que el Entscheidungsproblem es indecidible y después de todo

[1] En terminología moderna: si `SAT` tiene un algoritmo cuadrático.

> $\varphi(n) \sim kn$ (o $\varphi(n) \sim kn^2$) solo significa que la cantidad de pasos, en oposición a prueba y error, puede ser reducido de N a $\log(N)$ o incluso a $\log(N)^2$.
>
> Carta[2] de Kurt Gödel a John Von Neumann, 1956.

Ahora veremos la otra clase más importante, la de los problemas cuyas potenciales soluciones se pueden verificar eficientemente. Se define usualmente de dos maneras, la siguiente es a partir de MT's y certificados.

Definición 8 (La clase **NP** a partir de certificados)**.** Sea L un lenguaje, decimos que $L \in \mathbf{NP}$ si existe una MT M que corre en tiempo polinomial y un polinomio p tal que para toda entrada $x \in \{0,1\}^*$ se cumple:

$$x \in L \text{ sii } \exists u \in \{0,1\}^{p(|x|)}\ M(x,u) = 1.$$

Cada u sería un *certificado* para acreditar que x está en el lenguaje. La clase **NP** también puede definirse a partir de una variante de las máquinas de Turing llamada máquina de Turing no determinista. No son máquinas que pueden ser construidas físicamente pero son útiles en distintos ámbitos de la teoría para simplificar demostraciones y otras cosas. En sí, originalmente **NP** fue definida a partir de ellas.

Definición 9 (Máquina de Turing no Determinista)**.** *Una máquina de Turing no determinista* (abreviada MTND) M es una 7-upla $(Q, \Gamma, \delta_0, \delta_1, q_i, q_{aceptacion}, q_{rechazo})$ donde:

- Q es el conjunto de *estados* de la máquina.
- Γ es el *alfabeto* de la máquina (con $\{0, 1, \square, \triangleright\} \subseteq \Gamma$).
- $\delta_0 : Q \setminus \{q_{aceptacion}, q_{rechazo}\} \times \Gamma \longrightarrow Q \times \Gamma \times MOV$ y $\delta_1 : Q \setminus \{q_{aceptacion}, q_{rechazo}\} \times \Gamma \longrightarrow Q \times \Gamma \times MOV$ son *funciones de transición.*
- $q_i \in Q$ es el *estado inicial.*

Cada uno de estos conceptos son los ya definidos para las ya definidas máquinas de Turing deterministas. Adicionalmente, se definen dos estados finales $q_{aceptacion}, q_{rechazo} \in Q$ que denotan respectivamente la aceptación o rechazo de una entrada. A $q_{aceptacion}$ lo llamaremos *estado de aceptación* y a $q_{rechazo}$ *estado de rechazo*, ambos son los únicos estados finales de la máquina, es decir, solo al llegar a uno de ellos la máquina *finaliza* o se *detiene.*

Las convenciones para la máquina de Turing determinista (Definición 3) aplican en la máquina de Turing no determinista (como que es monocinta, que la cinta es infinita hacia la derecha, etc). La radical diferencia respecto a su hermana determinista es su *salida* y cómo esta se computa. Sea M MTND y

[2] La carta completa puede leerse en [2].

$x \in \{0,1\}^*$ una entrada. La máquina en cada paso de manera no determinista aplica una de sus funciones de transición. Un *hilo* será para nosotros una sucesión de elecciones no deterministas que puede seguir la máquina a partir de x. Es decir, una sucesión $e_1, e_2, \ldots$ donde cada $e_i \in \{0,1\}$ representa la aplicación de una de las funciones de transición en el i-ésimo paso de le ejecución. (Si $e_i = 0$ significa que M a partir de la entrada x siguiendo tal hilo, en el i-ésimo paso aplicó la función de transición δ_0, si $e_i = 1$ significa que aplicó δ_1). Observemos que un hilo puede tener una longitud infinita (todo e_i tiene sucesor), si se da el caso significa que la máquina siguiendo ese hilo *no finaliza*, es decir, jamás llega a un estado final. Decimos que *M a partir de x finaliza* si todo hilo para tal entrada tiene longitud finita. Dada x de entrada, llamaremos *camino* a un hilo que lleva a la máquina del estado inicial a partir de x a un estado final. Decimos que M a partir de *x corre en $t \in \mathbb{N}$ pasos,* si la máquina finaliza a partir de x y todo camino tiene un largo menor o igual a t. Dada una función $T : \mathbb{N} \longrightarrow \mathbb{N}$ decimos que *M corre en tiempo T* si para toda entrada $x \in \{0,1\}^*$ la máquina corre en $T(|x|)$ pasos. Si M finaliza a partir de x entonces: decimos que *M acepta a x* si para M partiendo de x existe un camino que lleva al estado de aceptación, caso contrario (todos los caminos llevan al estado de rechazo) decimos que *M rechaza a x*. Denotamos que *M acepta a x* con $M(x) = 1$ y denotamos que *M rechaza a x* con $M(x) = 0$. Finalmente llamamos a $M(x)$ *salida* de la máquina.

Notar la gran diferencia del significado de $M(x) = 1$ para las MT's y las MTND's. Para las MT's con $M(x)$ simplemente denotamos a la palabra escrita en la cinta al finalizar M. Luego $M(x) = 1$ significa que M al finalizar dejó escrito en la cinta el símbolo 1. Totalmente distinto para las MTND's. Con $M(x)$ denotamos la *aceptación o rechazo de una entrada y no representa lo que está escrito en la cinta.* Notar que es físicamente imposible construir un dispositivo que implemente una MTND.

Definición 10 (La clase **NP** a partir de MTND)**.** Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función y L un lenguaje, decimos que $L \in \mathbf{NTIME}(T)$ si existe $k \in \mathbb{N}$ y una MTND M que corre en tiempo kT que para toda entrada $x \in \{0,1\}^*$ cumple:

$$x \in L \text{ sii } M(x) = 1.$$

Definimos

$$\mathbf{NP} = \bigcup_{c \geq 1}^{\infty} \mathbf{NTIME}(n^c).$$

Observemos que si $L \in \mathbf{NP}$ es decidido por M MTND entonces necesariamente M finaliza para toda entrada. Es decir dada $x \in \{0,1\}^*$ de entrada, todo hilo que pueda seguir M a partir de x tiene una longitud finita. Es otras palabras: la máquina aplique las función de transición que aplique, en una cantidad finita de pasos llegará a un estado final. Vale agregar que así como tenemos la clase **EXP**, tenemos la clase **NEXP** de lenguajes decididos en tiempo exponencial por MTND's.

Proposición 11. *Ambas definiciones de* **NP** *coinciden.*

Reducción, NP-completo y SAT

¿Cómo podemos probar que un lenguaje es más "difícil" que otro? Y de tener la solución para un problema: ¿Podemos transformarla eficientemente en la solución de otro? El concepto que necesitamos es el de reducción.

Definición 12 (Reducción de Karp)**.** Un *lenguaje* L *es polinomialmente reducible a un lenguaje* L' (denotado $L \leq_p L'$), si existe una función

$f : \{0,1\}^* \longrightarrow \{0,1\}^*$ computable por una MT polinomial que para todo $x \in \{0,1\}^*$ cumple:

$$x \in L \text{ sii } f(x) \in L'$$

Decimos que L' es *NP-hard* si $\forall L \in \mathbf{NP}$ $L \leq_p L'$. Decimos que L' es *NP-completo* si L' es NP-hard y $L' \in \mathbf{NP}$.

Teorema 13 (Propiedades de la reducción de Karp)**.**

1. (Transitividad). Si $L \leq_p L'$ y $L' \leq_p L''$ entonces $L \leq_p L''$.
2. Si existe un lenguaje L que es NP-hard tal que $L \in \mathbf{P}$ entonces $\mathbf{P} = \mathbf{NP}$.
3. Si un lenguaje L es NP-completo entonces: $L \in \mathbf{P}$ sii $\mathbf{P} = \mathbf{NP}$.

En sí, el concepto de reducción se puede formalizar de manera más general. Hay muchas variantes de reducciones, y en general para cada clase de complejidad distinta hay una reducción distinta. La de Karp recién vista es tal vez la primera que surgió y desde entonces ha sido siempre práctica y útil.

Uno puede preguntarse si es que existen problemas NP-completo, es decir, si es que **NP** tiene algún lenguaje tal que cualquier otro lenguaje en la clase es a lo sumo tan difícil como él. Veamos que sí.

Definición 14 (Fórmulas Booleanas y `SAT`)**.** Sea $X = \{x_1, x_2, \ldots\}$ un conjunto numerable, decimos que una *variable* es un elemento de X.

Una *fórmula Booleana* se define recursivamente de la siguiente manera:

1) Una variable es una fórmula Booleana.

2) Si φ es una fórmula Booleana entonces $\neg\varphi$ es una fórmula Booleana.

3) Si φ y ψ son fórmulas Booleanas entonces $(\varphi \vee \psi)$ y $(\varphi \wedge \psi)$ son fórmulas Booleanas.

Si la fórmula Booleana φ tiene de variables a $x_1, \ldots, x_n$ entonces una *asignación* para φ es cualquier palabra $u \in \{0,1\}^n$. Denotamos con $\varphi(u)$ al valor de computar la fórmula haciendo un reemplazo sintáctico de las variables por los valores de u, es decir, cada ocurrencia de la variable x_i en φ es reemplazada por el i-ésimo bit de u. Tal cómputo se realiza de manera natural, $\neg$ es la negación lógica y $\wedge, \vee$ son los restantes conectivos lógicos. Decimos que una fórmula φ es *satisfacible* si existe una asignación u, tal que $\varphi(u) = 1$.

Una fórmula Booleana está en CNF (Conjunctive Normal Form) si es de la forma

$$\bigwedge_i (\bigvee_j v_{ij})$$

donde v_{ij} es un *literal* (variable o la negación de una variable).

Finalmente definimos a `SAT` como el lenguaje de las fórmulas Booleanas en CNF satisfacibles (la codificación en ceros y unos de las fórmulas Booleanas en CNF satisfacibles). Siempre obviaremos aclarar que nos referimos a codificaciones al definir lenguajes. Y definimos a `3SAT` como el lenguaje de fórmulas Booleanas en CNF satisfacibles que tienen a lo sumo 3 literales por *cláusula* (términos de la forma $\bigvee_j v_{ij}$).

El lector interesado puede ver el célebre trabajo "The Complexity of Theorem-Proving Procedures" [15], considerado uno de los más importantes de la teoría, donde se define por primera vez la noción de NP-completo y donde se demuestra el siguiente teorema.

Teorema 15 (Teorema de Cook-Levin)**.**

1. `SAT` es NP-completo.

2. `3SAT` es NP-completo.

Hay muchos problemas NP-completos estudiados pero `SAT` es el principal. Más allá de `SAT`, las fórmulas Booleanas tienen el poder de expresar de manera "explícita" y simple la *localidad de la computación* y por ende son una herramienta primordial para tratar de encontrar problemas completos para otras clases. La prueba del teorema anterior se basa exclusivamente en ello: la computación es local y las fórmulas Booleanas lo pueden expresar eficientemente.

El lector interesado puede ver el célebre trabajo "Reducibility Among Combinatorial Problems" [21], donde se presentan 21 problemas NP-completo. Nombraremos algunos de ese trabajo y otros a modo de ejemplos (recordando, cuando definimos un lenguaje como un conjunto de elementos en realidad estamos haciendo alusión a alguna codificación en ceros y unos de tales elementos).

- Dada una teoría Λ, el lenguaje `THEOREMS` definido como el conjunto de pares $(\varphi, 1^n)$ tal que φ tiene una demostración en Λ de tamaño menor o igual a n.

- `HAMPATH` el cual es el conjunto de triplas (G, s, t) tal que G es un grafo dirigido con un camino hamiltoneano de s a t.

- `1/0 INTERGER PROGRAMMING` el cual consiste en dada una lista de m inecuaciones lineales con coeficientes racionales sobre n variables $u_1, \ldots, u_n$ (una inecuación lineal tiene la forma $a_1u_1 + \ldots + a_nu_n \leq b$ para algunos coeficientes $a_1, \ldots, a_n, b$), decidir si existe una asignación de las variables en unos y ceros que satisface todas las inecuaciones.

- `GRAPH COLORING` el cual es el conjunto de tuplas $(G, 1^k)$ tal que G es un grafo que tiene un coloreo de k colores.

- `3COLOR` el cual es el conjunto de grafos que tienen un coloreo de 3 colores.

Un hecho interesante es que `2SAT` (el análogo de `3SAT` solo que permite a lo sumo 2 literales por cláusula en vez de 3) y `2COLOR` (los grafos que tienen un coloreo de 2 colores) pertenecen a **P**. ¡Pasar de 3 a 2 nos hace pasar de NP-completo a **P**!

La clase coNP

Definimos esta clase también relacionada a **P** y **NP** que será de crucial importancia en el capítulo de La Jerarquía Polinomial. Ya que ahí la clase **PH** (Definición 39) se definirá a partir de "niveles de lenguajes" que serán generalizaciones de **NP** y de **coNP**.

Definición 16 (La clase **coNP**)**.** Sea $L \subseteq \{0,1\}^*$ un lenguaje, denotamos con $\overline{L}$ al *complemento* de L. Esto es $\overline{L} = \{0,1\}^* \setminus L$.

Definimos $\mathbf{coNP} = \{\overline{L} : L \in \mathbf{NP})$.

Vale observar que **coNP** no es el complemento de **NP**, de hecho es fácil ver que $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.

Es uno de los problemas abiertos más importantes de la teoría si ¿$\mathbf{NP} = \mathbf{coNP}$? Se cree que no.

Definición 17 (Definición alternativa de **coNP**)**.** Sea L un lenguaje, decimos que $L \in \mathbf{coNP}$ si existe una MT M polinomial y un polinomio p tal que para toda entrada x se cumple:

$$x \in L \text{ sii } \forall u \in \{0,1\}^{p(|x|)} \quad M(x,u) = 1.$$

Proposición 18. *Ambas definiciones de* **coNP** *coinciden.*

Las nociones de *coNP-hard* y *coNP-completo* son análogas a las respectivas de **NP**. Un problema coNP-completo que sería el `"SAT"` de **coNP** es `TAUTOLOGY`. Definido como el lenguaje de las fórmulas Booleanas en CNF satisfacibles por todas las asignaciones.

La clase PSPACE

Hemos definido la dificultad de los problemas en función al tiempo (como cantidad de pasos) que le lleva a las máquinas solucionarlos. En la práctica en muchas ocasiones los programadores no solo se preocupan por el tiempo sino también por el espacio que usan sus algoritmos (como la memoria que usa la computadora por ejemplo). La clase que veremos a continuación, **PSPACE**, consiste de los problemas solucionables por máquinas que usan el espacio eficientemente (polinomialmente). En la definición de máquina de Turing determinista (Definición 3) la máquina tiene una sola cinta, a continuación veremos otra variante en la que la máquina tiene muchas cintas (al menos tres). La razón es que es más fácil y natural en este contexto que las máquinas tengan más cintas. Otra vez, podríamos definir esta clase de lenguajes con máquinas de una sola cinta

pero ello iría en contra de la bibliografía y en sí haría ofuscadas muchas demostraciones y definiciones. El resto de las convenciones sobre las MT's de una sola cinta aplican a las MT's de muchas (como por ejemplo que las cintas son infinitas hacia la derecha, etc).

Definición 19 (Máquina de Turing determinista multicinta)**.** *Una Máquina de Turing determinista multicinta* es una 5-upla $(Q, \Gamma, \delta, q_i, q_f)$ donde:

- Q es un conjunto finito no vacío llamado *conjunto de estados* que tiene dos estados especiales q_i, q_f que denotan al estado inicial y final respectivamente.
- Γ es un conjunto finito que contiene a $\{0, 1, \square, \triangleright\}$ llamado *alfabeto* de la máquina, y son los símbolos que pueden ocurrir en las cintas.
- $\delta : Q \setminus \{q_f\} \times \Gamma^k \longrightarrow Q \times \Gamma^{k-1} \times MOV^k$ es una función total llamada *función de transición,* con $k \geq 3$ la cantidad de cintas.

Donde $MOV = \{I, D, S\}$ el conjunto de movimientos del cabezal de la máquina, (I izquierda, D derecha y S quedarse (stay)).

La máquina tiene dos cintas especiales, una llamada *cinta de entrada* y otra *cinta de salida*. En la de entrada aparece escrita la entrada al comenzar la computación, la máquina solo puede leer en esta cinta, no puede escribir. La cinta de salida es donde se escribe la salida de la máquina valga la redundancia. Al resto de las cintas las llamaremos *cintas de trabajo*.

Sea M una máquina de Turing multicinta (abreviado MT multicinta) y $x \in \{0, 1\}^*$. Decimos que *M parte de x o que parte de la entrada x o que recibe a x de entrada*, para indicar que la máquina arranca en el estado inicial con todos los cabezales apuntando al símbolo $\triangleright$ de sus respectivas cintas. Y que en la cinta de entrada, a la derecha de $\triangleright$ ocurre x.

Decimos que la máquina *finaliza* o que se *detiene* cuando llega al estado final, a partir de ahí la máquina no puede realizar ningún movimiento. Ello lo aseguramos en la definición de δ ya que no existe $r \in \Gamma^k$ tal que (q_f, r) esté en el dominio de δ. Dado $x \in \{0, 1\}^*$ y $t \in \mathbb{N}$, decimos que *M a partir de x corre en t pasos*, si recibiendo a x de entrada M finaliza en a lo sumo t pasos. Con un paso una aplicación de δ. Dada una función $T : \mathbb{N} \longrightarrow \mathbb{N}$ decimos que *M corre en tiempo T* si para todo $x \in \{0, 1\}^*$, M a partir de x corre en $T(|x|)$ pasos. Con $M(x)$ denotamos a la *salida* de la máquina que es la palabra escrita en la cinta de salida al llegar al estado final, (ignorando los símbolos $\square$) es decir la palabra más larga que no finaliza en $\square$. Asumimos que la cinta de salida en el estado final tiene la siguiente forma:

$$\triangleright \alpha \square \square \square \square \square \square \square \square \square \ldots$$

con $\alpha \in \Gamma^*$ la salida de la máquina.

Decimos que la máquina en una ejecución *usa un casillero* de una cinta, si en la ejecución un cabezal lo "visita". Dado $x \in \{0, 1\}^*$ y $s \in \mathbb{N}$, decimos que *M a partir de x corre en espacio s*, si recibiendo x de entrada M finaliza usando

a lo sumo s casilleros de las cintas de trabajo (la sumatoria de la cantidad de casilleros usados en cada cinta de trabajo es menor o igual a s). Dada una función $S : \mathbb{N} \longrightarrow \mathbb{N}$ decimos que M *corre en espacio* S si para todo $x \in \{0, 1\}^*$, M corre en espacio $S(|x|)$.

Notar que sencillamente se puede adaptar la definición anterior para definir MTND multicinta.

Sea L un lenguaje y $S : \mathbb{N} \longrightarrow \mathbb{N}$ una función, decimos que $L \in \mathbf{SPACE}(S)$ si existe M MT multicinta y una constante $k \in \mathbb{N}$ tal que M decide a L corriendo en espacio kS.

Definición 20 (la clase **PSPACE**)**.**

$$\mathbf{PSPACE} = \bigcup_{c \geq 1}^{\infty} \mathbf{SPACE}(n^c).$$

Uno puede definir **NSPACE** como la clase análoga no determinista, definida a partir de las MTND's multicintas que usan espacio polinomial. Curiosamente **PSPACE** vs **NSPACE** tiene respuesta y es: **PSPACE** = **NSPACE**. Se puede demostrar también **PSPACE** $\subseteq$ **EXP** y para ello se usa el concepto de *grafo de configuración* de una máquina multicinta a partir de una entrada. A continuación definiremos el grafo de configuración porque es una herramienta útil que se puede generalizar a otras variantes de máquinas de Turing que veremos en capítulos ulteriores.

Definición 21 (Grafo de Configuración)**.** Sea M una máquina de Turing multicinta de $k \in \mathbb{N}$ cintas y $x \in \{0, 1\}^*$, podemos generalizar la noción de configuración vista (Definición 4) y adaptarla a esta máquina. Luego una configuración representaría el estado, el contenido de las cintas y la posición de los cabezales en algún paso de la ejecución de M a partir de x. Decimos que el *grafo de configuración de M para la entrada x* denotado como $G_{M,x}$ es un grafo dirigido que sus vértices corresponden a todas las configuraciones posibles que puede tener M a partir de x. Las aristas las definimos de la siguiente manera: dados dos vértices v, w del grafo, si la máquina puede llegar de la configuración representada por v a la configuración representada por w en un paso entonces (v, w) es una arista del grafo. Naturalmente, si la máquina es determinista entonces el grafo va a ser una "línea" donde de cada vértice "sale" a lo sumo una sola arista. En cambio, si la máquina es no determinista entonces de cada vértice salen a lo sumo dos aristas. Notar que M finaliza a partir de x si y solo si $G_{M,x}$ es finito y acíclico.

Para **PSPACE** tenemos la misma noción de hard y completo que vimos en **NP** porque usamos la misma reducción.

Definición 22 (PSPACE-hard y PSPACE-completo)**.** Decimos que un lenguaje L' es *PSPACE-hard* si para todo $L \in \mathbf{PSPACE}$ se cumple $L \leq_p L'$. Decimos que un lenguaje L' es *PSPACE-completo* si PSPACE-hard y $L' \in \mathbf{PSPACE}$.

¿Existe un lenguaje PSPACE-completo? Veremos que sí, y usa también a las fórmulas Booleanas.

Definición 23 (Fórmula Booleana Cuantificada y `TQBF`)**.** Una *fórmula Booleana Cuantificada (QBF)* es una fórmula de la forma

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \varphi$$

donde φ es una fórmula Booleana de variables $x_1, x_2, \dots, x_n$ y cada $Q_i \in \{\exists, \forall\}$, donde los cuantificadores se mueven en el universo de los valores que pueden tomar las variables, es decir, $\{0,1\}$. Por ejemplo con $\exists_1 x_1 \forall_3 x_3 (x_1 \wedge x_3)$ se expresa $\exists x_1 \in \{0,1\} \forall x_3 \in \{0,1\} (x_1 \wedge x_3)$. Observemos que dada una QBF Ψ, se tiene que Ψ es cierta o falsa (1 o 0). Definimos `TQBF` como el conjunto de las QBF ciertas.

Como otro ejemplo, `SAT` es un caso particular de `TQBF` ya que dada φ fórmula booleana de variables $x_1, x_2, \dots, x_n$, se tiene que

$$\varphi \in SAT \text{ sii } \exists_1 x_1 \exists_2 x_2 \dots \exists_n x_n \varphi \in \texttt{TQBF}$$

Teorema 24. `TQBF` *es PSPACE-completo*

Corolario 25. **NP** $\subseteq$ **PSPACE** *y* **coNP** $\subseteq$ **PSPACE**.

Capítulo 2

Diagonalización

> Este enunciado es falso.
>
> Paradoja de Epiménides

> La pregunta ¿$\mathbf{P} = \mathbf{NP}$? tiene una respuesta positiva para algunos oráculos y negativa para otros. Sentimos que esto evidencia la dificultad de la pregunta ¿$\mathbf{P} = \mathbf{NP}$?
>
> Baker, Gill, Solovay

En este capítulo estudiaremos la técnica llamada *diagonalización* la cual es una de las tres más importantes de la teoría de Complejidad Computacional. Veremos que se inspira en el argumento de la diagonal popularizado por G. Cantor en su genial teoría de los Números Transfinitos. Luego veremos su adaptación a la teoría de la Computabilidad para mostrar que existen funciones no computables y para mostrar que una función particular no es computable (el Halting Problem). Después veremos su refinamiento para obtener algunos resultados brillantes en Complejidad Computacional como los de jerarquía temporal, y el teorema de la existencia de lenguajes NP-intermedios. Por último veremos la *relativización* que puede ser interpretada como una limitación de la diagonalización y que sugiere que **P** vs **NP** no se puede resolver solo a partir de tal técnica.

Escapa a este trabajo un estudio de la *teoría de los números Transfinitos*, a nuestros fines nos alcanza con ver el siguiente teorema. Y para evitar entrar en tecnicismos de la teoría recién mencionada no haremos uso de la noción de cardinal y en lugar de ello nos concentraremos en la biyectividad de las funciones.

Teorema 26. *No existe una función biyectiva de* $\mathbb{N}$ *a* $\mathbb{R}$.

Demostración. Notemos que como hay una biyección entre $[0,1]$ y $\mathbb{R}$ con mostrar que no existe una función biyectiva entre $\mathbb{N}$ y $[0,1]$ alcanza. Para ello, utilizaremos el famoso *argumento de la diagonal* y razonaremos por el absurdo.

Supongamos que existe una función biyectiva entre $\mathbb{N}$ y $[0,1]$. Entonces podemos enumerar todos los números reales del intervalo $[0,1]$, sea $x_1, x_2, x_3, x_4, \ldots$ una enumeración. Sabemos que cada número en tal intervalo puede ser representado a partir de sus cifras decimales, luego para cada $i \in \mathbb{N}$ decimos que $0.c_{i1}c_{i2}c_{i3}\ldots$ es la representación de x_i a partir de sus cifras decimales[1] c_{ij}. Entonces la enumeración en una lista se vería como:

$$
\begin{aligned}
x_1 &= 0.c_{11}c_{12}c_{13}c_{14}\ldots\\
x_2 &= 0.c_{21}c_{22}c_{23}c_{24}\ldots\\
x_3 &= 0.c_{31}c_{32}c_{33}c_{34}\ldots\\
x_4 &= 0.c_{41}c_{42}c_{43}c_{44}\ldots\\
&\vdots
\end{aligned}
$$

Definamos el siguiente número real $y \in [0,1]$, a partir de su expresión decimal:

$$y = 0.(c_{11}\dot{+}1)(c_{22}\dot{+}1)(c_{33}\dot{+}1)\ldots$$

con $c_{jj}\dot{+}1 = \begin{cases} c_{jj}+1 & \text{si } c_{jj} \neq 9 \\ 0 & \text{cc} \end{cases}$.

Observemos que dado cualquier $i \in \mathbb{N}$ se tiene que la i-ésima cifra de x_i, c_{ii}, es distinta a la i-ésima cifra de y, por lo tanto $y \neq x_i$. En consecuencia, y no pertenece a la enumeración ¡Absurdo! Por lo tanto no existe tal enumeración y por ende, no existe una función biyectiva entre $\mathbb{N}$ y $[0,1]$. □

Notemos que y se forma a partir de la *diagonal* de cifras $c_{11}, c_{22}, c_{33}, c_{44}, \ldots$ haciendo que cada una de ellas sea diferente a la respectiva cifra de y. Para ver que hay funciones que no son computables usaremos la misma estrategia, haremos una enumeración inteligente y a partir de la diagonal de tal enumeración haremos una función que es imposible de computar.

Diagonalización

Es relativamente simple ver que podemos codificar toda MT como una palabra de ceros y unos. Simplemente se escribe la máquina en un papel y cada componente de la 5-upla se lo codifica en ceros y unos. Vamos a asumir que nuestra codificación de las MT's satisfacen las siguientes dos propiedades:

1. Cada palabra en $\{0,1\}^*$ representa alguna MT. Esto es fácil de asegurar, mapeando cada palabra que no siga nuestra codificación a una MT canónica, como por ejemplo, una que para toda entrada devuelva 0.

[1] Para que la representación por decimales sea biyectiva con los elementos de $[0,1]$, consideramos las representaciones que no terminan en un período de 9s.

2. Cada MT es representada por una cantidad infinita de palabras. Esto se asegura haciendo que nuestra codificación pueda finalizar con una cantidad arbitraria de 1's ignorados. Esto es algo similar al mecanismo utilizado para poner comentarios en los programas escritos en C o Python, por citar algunos lenguajes de programación populares.

Dada M MT denotamos con $\lfloor M \rfloor$ a nuestra codificación en ceros y unos que sigue las anteriores propiedades.

Teorema 27 (Máquina de Turing Universal)**.** *Existe una máquina de Turing U tal que para todas entradas $x, \alpha \in \{0,1\}^*$ $U(x, \alpha) = M_\alpha(x)$, donde M_α denota a la MT codificada en α. Además, si $M_\alpha(x)$ corre en $t \in \mathbb{N}$ pasos entonces $U(x, \alpha)$ corre en $kt \log(t)$ pasos, con $k \in \mathbb{N}$ una constante que no depende de x y que solo depende del tamaño del alfabeto de M_α y de la cantidad de estados de la máquina.*

Teorema 28. *Existe una función $UC : \{0,1\}^* \longrightarrow \{0,1\}^*$ que no es computable por ninguna máquina de Turing.*

Demostración. Definamos UC de la siguiente manera: dado $\alpha \in \{0,1\}^*$

$$UC(\alpha) = \begin{cases} 0 & \text{si } M_\alpha(\alpha) = 1 \\ 1 & \text{c.c} \end{cases}$$

Es decir, si la máquina codificada por α recibiendo de entrada a α devuelve 1 entonces $UC(\alpha) = 0$, si la máquina devuelve otra cosa o no finaliza entonces $UC(\alpha) = 1$. Supongamos que UC es computable y lleguemos a un absurdo. Si UC es computable entonces existe una MT M que la computa, luego $\forall \alpha \in \{0,1\}^*$ $M(\alpha) = UC(\alpha)$. En particular se debería cumplir $M(\lfloor M \rfloor) = UC(\lfloor M \rfloor)$ pero esto es absurdo porque por la definición de UC: si $M(\lfloor M \rfloor) = 1$ entonces $UC(\lfloor M \rfloor) = 0$ y en cambio si $M(\lfloor M \rfloor) \neq 1$ entonces $UC(\lfloor M \rfloor) = 1$. Por lo tanto UC no es computable. □

Para ver dónde hay una diagonal realicemos el siguiente análisis.
Supongamos que ordenamos las palabras de $\{0,1\}^*$ en el orden lexicográfico y escribimos en una tabla el valor de $M_\alpha(x)$ para todas las palabras $\alpha, x \in \{0,1\}^*$. Denotemos con $*$ en la tabla que la máquina para tal entrada no devuelve un bit o que no finaliza. Luego la tabla y la diagonal se podrían ver de la siguiente manera:

	0	1	00	01	10	11	...	α	...
0	1	*	*	0	1	1		$M_0(\alpha)$	...
1	0	1	0	0	0	0		*	...
00	*	0	*	1	*	1		1	...
01	*	0	1	1	1	*		*	...
10	0	*	*	*	0	*		0	...
11	*	0	*	0	0	*		0	...
$\vdots$									...
α	$M_\alpha(0)$	...				$M_\alpha(11)$	...	$M_\alpha(\alpha)$	...
$\vdots$									...

Figura 2.1:

Luego la función UC se forma "negando" los valores de la diagonal. Como toda MT ocurre en alguna fila concluimos que UC no puede ser computada por una MT.

Con lo anterior hemos visto que existen funciones no computables pero no nos muestra de manera explícita alguna. A continuación daremos una.

Teorema 29. *La función $Halt$ no es computable*

Demostración. Sea $Halt : \{0,1\}^* \longrightarrow \{0,1\}^*$ definida como

$$Halt(\alpha, x) = \begin{cases} 1 & \text{si } M_\alpha(x) \text{ finaliza} \\ 0 & \text{c.c} \end{cases}$$

Intuitivamente, $Halt$ determina si una máquina de Turing finaliza o no para una entrada particular. Este célebre problema lleva el nombre del *problema de la parada* o en inglés el *Halting Problem.* Supongamos que $Halt$ es computable y lleguemos a un absurdo. Sea M_{halt} la MT que computa a la función $Halt$ luego dadas $\alpha, x \in \{0,1\}^*$ de entrada, la máquina finaliza y devuelve $Halt(\alpha, x)$.

Definamos la siguiente MT M_{UC}: dada $\alpha \in \{0,1\}^*$ de entrada, M_{UC} realiza lo siguiente:

if $M_{halt}(\alpha, \alpha) = 1$:
  if $M_\alpha(\alpha) = 1$:
    return 0
return 1

Observemos que M_{UC} computa a la función UC que no es computable! Llegamos a un absurdo, por ende necesariamente M_{halt} no existe y en consecuencia $Halt$ no es computable. □

Una pregunta que uno puede hacerse es: ¿A mayor tiempo que se le da a las MT más lenguajes pueden decidir? El siguiente teorema muestra que sí. Antes, una función $f : \mathbb{N} \longrightarrow \mathbb{N}$ es *tiempo construible* si $\forall n \in \mathbb{N}$ $f(n) \geq n$ y existe M MT que corre en tiempo f y computa a $1^n \longmapsto \lfloor f(n) \rfloor$ (con $\lfloor\ \rfloor$ la codificación en binario). Vale observar que todas las funciones del trabajo exceptuando a las no computables son tiempo construible.

Teorema 30 (Jerarquía Temporal para **DTIME**). *Si f, g son funciones tiempo construible que cumplen $f \log(f) = o(g)$ entonces* $\mathbf{DTIME}(f) \subsetneq \mathbf{DTIME}(g)$.

Demostración. Para ser más simples y didácticos demostraremos $\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n^{1,5})$ ya que la prueba del teorema es esencialmente la misma pero con algunos detalles técnicos que no nos aportan mucho. Consideremos la siguiente MT D : dada una entrada x la máquina ejecuta $U(x,x)$ (U la MT Universal) por $|x|^{1,4}$ pasos. Luego si en tal ejecución $U(x,x)$ devolvió un $b \in \{0,1\}$ entonces D devuelve $1-b$, caso contrario (si $U(x,x)$ no finalizó o devolvió otra palabra) D devuelve 0. Sea L_D el lenguaje decidido por D, necesariamente $L_D \in \mathbf{DTIME}(n^{1,5})$. Supongamos que $L_D \in \mathbf{DTIME}(n)$ y lleguemos a un absurdo. Supongamos que existe MT M y $c \in \mathbb{N}$ tal que M decide a L_D en tiempo cn, es decir: $\forall x \in \{0,1\}^*$ $x \in L_D$ sii $M(x) = 1$ y M corre en tiempo $k|x|$. En consecuencia para toda entrada $x \in \{0,1\}^*$ se cumple

$$x \in L_D \text{ sii } M(x) = 1 \text{ sii } D(x) = 1.$$

Observemos que para toda entrada x se tiene $M(x) = D(x)$. Por el teorema de la MT Universal tenemos que existe una constante k (que no depende del tamaño de la entrada) tal que para toda entrada y, la máquina U simula a $M(y)$ en a lo sumo $kc|y|\log(c|y|)$ pasos. Luego, existe algún número n_0 tal que si $n \geq n_0$ entonces $n^{1,4} > kcn\log(cn)$. Sea α una codificación de M tal que $|\alpha| > n_0$ (tal α existe ya que en nuestra codificación M tiene infinitas representaciones) luego $U(\alpha, \alpha)$ finaliza en a lo sumo $kc|\alpha|\log(c|\alpha|)$ pasos que es menor a $|\alpha|^{1,4}$ y por ende $D(\alpha)$ devuelve $1 - M(\alpha)$. Por lo tanto para este α se tiene que $D(\alpha) \neq M(\alpha)$ ¡Absurdo!. □

Como corolario importante, $\mathbf{P} \subsetneq \mathbf{EXP}$. En particular se sabe que problemas como algunas generalizaciones del ajedrez, y del juego Go son problemas que están en **EXP** y no en **P**.

Teorema 31 (Jerarquía Temporal para **NTIME**)**.** *Si f, g son funciones tiempo construibles que satisfacen $f = o(g)$ entonces* $\mathbf{NTIME}(f) \subsetneq \mathbf{NTIME}(g)$.

Demostración. La demostración usa diagonalización y es parecida a la del teorema anterior. □

Veamos un teorema muy interesante que no es trivial cómo utiliza la diagonalización.

Teorema 32 (Teorema de Ladner)**.** *Si* $\mathbf{P} \neq \mathbf{NP}$*, entonces existe un lenguaje* $L \in \mathbf{NP} \setminus \mathbf{P}$ *que no es NP-completo.*

Demostración. En el trabajo original se lo demuestra de una manera más artificial y difícil, ver [24]. A continuación veremos una prueba moderna y más didáctica. La estrategia de la demostración es la siguiente. Si suponemos que $\mathbf{P} \neq \mathbf{NP}$ entonces $\texttt{SAT} \notin \mathbf{P}$, con ello en mente construiremos un lenguaje L que será una "versión de `SAT` más fácil" para que L no sea NP-completo. Pero que será lo suficientemente "difícil" para que L no pertenezca a **P**. Intuitivamente para hacer "más fácil" a `SAT`, le sacaremos instancias. Es decir, le sacaremos las suficientes fórmulas Booleanas para que deje de ser NP-completo, pero no tantas para que lo haga pertenecer a **P**.

Dada una MT M y un polinomio p se pueden construir efectivamente MTs M_p y M'_p tal que para todo $x \in \{0,1\}^*$:

$$M_p(x) = \begin{cases} M(x) & \text{si } M \text{ termina en } p(|x|) \text{ pasos a partir de } x \\ 0 & \text{c.c.} \end{cases}$$

$$M'_p(x) = \begin{cases} \text{primer bit de } M(x) & \text{si } M \text{ termina en } p(|x|) \text{ pasos a partir de } x \\ 0 & \text{c.c.} \end{cases}$$

Por lo tanto, los conjuntos

$$\mathcal{R} := \{\langle M_p \rangle : M \text{ MT y } p \text{ polinomio}\}$$

y

$$\mathcal{D} := \{\langle M'_p \rangle : M \text{ MT y } p \text{ polinomio}\}$$

son efectivamente enumerables. (Ya que el conjunto de polinomios es efectivamente enumerable y el conjunto de las codificaciones de MT's también lo es). Sean $D, R : \mathbb{N} \to \{0,1\}^*$ funciones computables en tiempo polinomial tales que enumeran a $\mathcal{D}$ y $\mathcal{R}$ respectivamente, i.e.,
$\mathrm{Im}(D) = \mathcal{D}$ e $\mathrm{Im}(R) = \mathcal{R}$. (Todo conjunto efectivamente enumerable, es enumerable por una función computable en tiempo polinomial.)

Definimos

$$L := \{x \in \mathtt{SAT} : f(|x|) \text{ es par}\},$$

donde $f : \{1\}^* \to \mathbb{N}$ es la función definida por el siguiente algoritmo.

Algoritmo

Casos base. Si $n = 0, 1$ devolver 2

Si $n > 1$ realizar las siguientes etapas:

Etapa 1.

Por exactamente n pasos computar $f(0), f(1), ...$

Sea k el último valor de la f que llegó a calcularse en el tiempo permitido.

El valor de $f(n)$ será k o $k+1$, en función de lo que se determine en la siguiente etapa.

Etapa 2.

Caso $k = 2i - 1$

Computar R_i.

Por exactamente n pasos realizar las siguientes tareas. Recorrer lexicográficamente $\{0,1\}^*$ en busca de un z tal que o bien

$z \in \mathtt{SAT}$ y $[R_i(z) \notin \mathtt{SAT}$ o $f(|R_i(z)|)$ es impar]; o

$z \notin \mathtt{SAT}$ y $[R_i(z) \in \mathtt{SAT}$, $f(|R_i(z)|)$ es par].

Si se encuentra un tal z en el tiempo permitido, devolver $k+1$. Sino, devolver k.

Caso $k = 2i$

Computar D_i.

Por exactamente n pasos realizar las siguientes tareas. Recorrer lexicográficamente $\{0,1\}^*$ en busca de un z tal que o bien

$D_i(z) = 1$ y $[z \notin \mathtt{SAT}$ o $f(|z|)$ es impar]; o

$D_i(z) = 0$ y $[z \in \mathtt{SAT}$ y $f(|z|)$ es par].

Si se encuentra un tal z en el tiempo permitido, devolver $k+1$. Sino, devolver k.

Notar que f es computable en tiempo polinomial por definición, y por lo tanto $L \in \mathbf{NP}$. Notar además que para todo n vale que $f(n+1) \geq f(n)$.

Afirmación. f es no acotada.

Demostración. Supongamos, por el contrario, que hay $k \in \mathbb{N}$ tal que k es el máximo valor alcanzado por f. Consideramos primero el caso en que $k = 2i$. Luego, en el caso par de la Etapa 2 del algoritmo nunca encontramos un z con las características deseadas. Se sigue que para todo $z \in \{0,1\}^*$ tenemos

$$D_i(z) = 1 \Longleftrightarrow z \in \texttt{SAT} \text{ y } f(|z|) \text{ es par.}$$

Notar que $L = \{z \in \texttt{SAT} : f(|z|) \text{ es par}\}$ difiere de SAT en una cantidad finita de palabras, y D_i es un algoritmo polinomial que decide a L. Esto contradice la hipótesis de que $\mathbf{P} \neq \mathbf{NP}$.

Si, por otra parte, el máximo valor alcanzado por f es $k = 2i-1$, significa que en el caso impar de la Etapa 2 no se encontró nunca un z. Es decir que R_i es una reducción de Karp de SAT a L. Pero en este caso L es finito, y por lo tanto $\texttt{SAT} \in \mathbf{P}$, lo cual es una contradicción. Esto concluye la prueba de la afirmación. □

Afirmación. $L \notin \mathbf{P}$

Demostración. Supongamos en pos de una contradicción que $L \in \mathbf{P}$. Luego hay un algoritmo polinomial que decide a L, y por lo tanto hay i tal que D_i decide a L. Ahora, como f es no acotada, hay n_0 tal que $f(n_0) = 2i$; pero inspeccionando el algoritmo vemos que, como nunca es hallado un z en la Etapa 2, se sigue que $f(n) = 2i$ para todo $n \geq n_0$. Esto no es posible ya que f es no acotada. □

Afirmación. L no es NP-completo

Demostración. Una vez más razonamos por el absurdo. Si hay una reducción de Karp de SAT a L, entonces hay un i tal que R_i es una MT que computa dicha reducción. Haciendo un razonamiento similar al del párrafo anterior concluimos que f está acotada por $2i-1$, obteniendo la contradicción deseada. □

Esto concluye la prueba. □

¿Será la factorización de enteros un lenguaje intermedio?

Relativización

¿Puede diagonalización resolver **P** vs **NP**? Primero y principal: ¿Qué entendemos por diagonalización? Con intencionalidad hemos evitado plantear esta reflexión (por más polémico que le pueda parecer al lector) y las razones se verán a continuación. Invito al lector a tomarse un tiempo y a preguntarse qué fue el hilo conductor de las demostraciones recién vistas. En particular, cuál es la esencia de la diagonalización.

En los resultados recién vistos que atañen a la complejidad computacional o a la computabilidad hemos utilizado solo las siguientes dos propiedades de las máquinas de Turing:

1. Las MT's tienen una representación efectiva como palabras.

2. Hay una MT universal U que, dada una codificación de una MT M y una entrada x, computa $M(x)$ eficientemente.

En la comunidad toda demostración que utiliza dichas propiedades de las máquinas se dice que utiliza la técnica de diagonalización. En sí, pienso que la esencia de la idea de la diagonal es poder enumerar y poder autoreferenciar. ¿Es cierto que la única manera de poder enumerar y autoreferenciar en la teoría de Complejidad Computacional es a partir de las propiedades mencionadas? La respuesta afirmativa a esta pregunta me parece muy fuerte y tal vez imprudente. A mi criterio, la comunidad inconscientemente supone que sí y por eso la técnica lleva el nombre de diagonalización. Lo que opino es que más justo y preciso es que lleve el nombre de *simulación*. A fin de cuentas, nos basamos solamente en la característica fundamental de las máquinas de Turing de que son simulables por ellas mismas de manera eficiente. No trato de ir contra la corriente sino dar un aire de optimismo e ideas distintas al lector. La bibliografía utiliza el nombre de "diagonalización" y por ello en este trabajo hacemos uso de él, pero creo que más preciso es el de "simulación".

En síntesis y para que quede claro: seguiremos la concepción de la comunidad y convendremos que toda demostración que solo se base en las citadas propiedades de las MT's es una prueba por diagonalización. Y bajo este concepto de diagonalización demostraremos su limitación conocida como *relativización*. Estos resultados de independencia solo aplicarán a las propiedades mencionadas, puede haber otras maneras de utilizar la enumeración y la autorreferencia en la teoría de la Complejidad Computacional que no se vean afectadas por estas barreras. El lector interesado puede consultar [12] que es el trabajo original en el cual estos conceptos fueron presentados por primera vez.

Como una "entrada en calor" pensemos en los familiares reticulados y en la propiedad distributiva, a partir de ahora PD. Se da un conjunto finito de propiedades (ninguna de las cuales es la PD) y se dice que todo modelo que satisfaga tales propiedades es un reticulado, llámese tal conjunto como teoría de reticulados. Uno quiere analizar si PD puede ser deducida a partir de la teoría de reticulados. Por el teorema de Correctitud de Gödel, si PD puede demostrarse a partir de la teoría de reticulados entonces todos sus modelos (i.e. reticulados)

deben satisfacerla. Uno muestra un reticulado (un objeto que satisface todas las propiedades de la teoría) que satisface PD, y luego muestra otro reticulado que no satisface PD. Entonces concluye que PD no puede ser deducible a partir de la teoría de reticulados. Es decir, la propiedad distributiva es independiente a la teoría de reticulados, y por lo tanto no se la puede demostrar ni refutar en dicha teoría.

Con eso en mente, trataremos de adaptar la relación entre diagonalización y **P** vs **NP** a la relación entre la teoría de reticulados y la propiedad distributiva. Definimos como teoría de los algoritmos a las propiedades 1 y 2, y diremos que un algoritmo es todo objeto que satisfaga tales propiedades. Siguiendo tal línea de pensamiento las máquinas de Turing son un modelo de tal teoría. Uno quiere analizar si la propiedad del milenio es deducible a partir de la teoría, la cual es que los lenguajes decidibles polinomialmente por los algoritmos son los lenguajes verificables polinomialmente por los algoritmos. Construiremos un modelo de la teoría donde valdrá la propiedad del milenio y construiremos otro modelo de la teoría donde no. Concluyendo que la propiedad del milenio es independiente de la teoría.

A continuación veremos la *Máquina de Turing Oráculo*. Uno puede entenderla como un algoritmo que llama a otro algoritmo de *subrutina* y que lo usa como "*caja negra*", es decir, no sabe cómo funciona pero sabe que funciona correctamente. El lector puede detenerse y decir: "Nada nuevo, las Máquinas de Turing estudiadas también pueden tener otras Máquinas de Turing de subrutinas, (algoritmos que llaman a otros algoritmos)". Tiene razón el lector, la crucial diferencia es que esta nueva máquina oráculo, como el nombre sugiere, puede tener de subrutina a algoritmos que las máquinas previamente estudiadas no. Por ejemplo, va a poder tener subrutinas que decidan a lenguajes indecidibles o que decidan un lenguaje de **EXP** en un solo paso.

Definición 33 (Máquina de Turing Oráculo)**.** Una *Máquina de Turing Oráculo* (MTO) es una máquina de Turing determinista M que tiene una cinta adicional especial de lectura/escritura (llamada *cinta oráculo*) y tres estados especiales: $q_{pregunta}$, q_{si} y q_{no}. Para ejecutar M especificamos un lenguaje $O \subseteq \{0,1\}^*$ (llamado *lenguaje oráculo*) que es utilizado como un oráculo por la máquina. En cualquier momento de la ejecución, si M entra al estado $q_{pregunta}$ entonces la máquina en el siguiente paso va al estado q_{si} si $y \in O$ o al estado q_{no} si $y \notin O$, donde y denota el contenido de la cinta oráculo (ignorando los infinitos $\square$). Notar que independientemente a la elección de O la máquina determina si $y \in O$ en un solo paso. Dada una entrada $x \in \{0,1\}^*$ denotamos con $M^O(x)$ *a la salida de* M *con acceso al oráculo* O *a partir de la entrada* x.

Las máquinas no deterministas Oráculos (MTNDO) son definidas de manera análoga.

La observación clave es que independientemente del lenguaje oráculo, todas las máquinas que lo tengan de oráculo satisfarán las propiedades y por ende serán modelos de la teoría. La razones son que las podemos representar mediante palabras de la misma manera que lo hacemos con las MT's y que podemos simular eficientemente una máquina con acceso oráculo O usando una MT universal

con acceso oráculo O. Por lo tanto, todos los resultados sobre MT's o sobre clases de complejidad que solo se basen en dichas propiedades también valdrán para todas las máquinas Oráculos. Diremos que tales resultados *relativizan* o que son *relativizantes*, es decir, que la técnica utilizada en su demostración se puede extender a las MTO's.

Definición 34. Sea $O \subseteq \{0,1\}^*$ y $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función, decimos que un lenguaje L pertenece a $\mathbf{DTIME}(T)^O$ (resp. $\mathbf{NTIME}(T)^O$) si existe una MTO M (resp. MTNDO) con acceso oráculo O que decide a L corriendo en tiempo cT, con c alguna constante.
Definimos $\mathbf{P}^O = \bigcup_{c \geq 1}^{\infty} \mathbf{DTIME}(n^c)^O$ y $\mathbf{NP}^O = \bigcup_{c \geq 1}^{\infty} \mathbf{NTIME}(n^c)^O$.

($\mathbf{EXP}^O$, $\mathbf{PSPACE}^O$, etcétera se definen análogamente).

Proposición 35. *Sea L un lenguaje, L pertenece a $\mathbf{NP}^O$ si existe una MTO M polinomial con acceso oráculo O y existe p polinomio tal que para todo $x \in \{0,1\}^*$ se cumple:*

$$x \in L \text{ sii } \exists u \in \{0,1\}^{p(|x|)} M^O(x,u) = 1.$$

Recapitulando: dado un lenguaje A, las MTO's con acceso oráculo A son un modelo de la teoría (las propiedades de ser simulable eficientemente). Por Gödel, una propiedad es deducible de una teoría si y solo si todos sus modelos la satisfacen. Como sabemos, un resultado es relativizante si es válido para todas las MTO's por consiguiente, un resultado es relativizante si y solo si es deducible solo a partir de las propiedades. Por ejemplo la siguiente propiedad es relativizante: existe un lenguaje decidido en tiempo exponencial que no es decidido en tiempo polinomial. Ya que para todo lenguaje L vale $\mathbf{P}^L \subsetneq \mathbf{EXP}^L$, y la demostración es esencialmente la misma que la del teorema de jerarquía para **DTIME** (Teorema 30). A continuación veremos que la propiedad del milenio no es relativizante y que por ende no es deducible solo a partir de la teoría. En el siguiente teorema construiremos dos lenguajes A, B; y mostraremos que en las máquinas con oráculo en A vale la propiedad del milenio (nuestra noción de $\mathbf{P} = \mathbf{NP}$ para tal modelo) y que en las máquinas con oráculo en B no. Mostrando que hay un modelo de la teoría donde vale la propiedad del milenio y otro en el que no.

Teorema 36. *Existen oráculos A, B tales que $\mathbf{P}^A = \mathbf{NP}^A$ y $\mathbf{P}^B \neq \mathbf{NP}^B$.*

Demostración. Primero construyamos el lenguaje A. Sea $A = \{(M, x, 1^n) :$ $M(x)$ retorna 1 en a lo sumo 2^n pasos $\}$. Veamos que $\mathbf{P}^A = \mathbf{NP}^A = \mathbf{EXP}$. Por un lado tenemos que $\mathbf{EXP} \subseteq \mathbf{P}^A$ ya que tener el oráculo A permite realizar en un solo paso una computación de una cantidad exponencial de pasos en MT's sin oráculo. Por otro lado, si M es una MTND con acceso oráculo A, con una MT podemos en tiempo exponencial simular cada posible camino que puede seguir M a partir de una entrada dada, y en cada uno de tales caminos podemos en tiempo exponencial responder cada consulta al oráculo A. Luego

$\mathbf{NP}^A \subseteq \mathbf{EXP}$ y como $\mathbf{P}^A \subseteq \mathbf{NP}^A$ tenemos $\mathbf{EXP} \subseteq \mathbf{P}^A \subseteq \mathbf{NP}^A \subseteq \mathbf{EXP}$ y por lo tanto $\mathbf{P}^A = \mathbf{NP}^A$.

Ahora construiremos el lenguaje B, basándonos en las ideas de [22]. Vale agregar que en el último capítulo de [18] también se construye este B con mayor atención a los detalles.
Para cada lenguaje L definamos

$$U_L = \{1^n : n \in \mathbb{N} \text{ y } \exists x \in \{0,1\}^n \text{ tal que } x \in L\}.$$

Observemos que para cada oráculo L, el lenguaje U_L está en NP^L. Ya que una MTND polinomial dada 1^n de entrada, puede elegir un $x \in \{0,1\}^n$ de manera no determinista y preguntarle al oráculo: ¿$x \in L$? Luego si $1^n \in U_L$ entonces algún x va a pertenecer a L y por ende la máquina va a aceptar, caso contrario no va a existir un x así en L por lo que va a rechazar. La intuición es que la MTND polinomial va a poder "realizar una cantidad exponencial de consultas al oráculo" en tiempo polinomial, mientras que la MT polinomial no. Con todo esto en mente construiremos un oráculo B tal que $U_B \notin \mathbf{P}^B$, implicando $\mathbf{P}^B \neq \mathbf{NP}^B$.

Primero veremos la siguiente proposición como para "entrar en calor", luego la utilizaremos para demostrar lo que queremos.

Proposición 37. *Para toda máquina de Turing Oráculo polinomial M existe un lenguaje L tal que U_L no es decidido por M^L.*

Demostración. Sea M MTO que corre en tiempo acotado por el polinomio p. Sea n el menor natural tal que $2^n > p(n)$. Notemos que dado de entrada 1^n, habrá palabras de tamaño n que la máquina no podrá preguntarle al oráculo (sea cual sea este) es decir, habrá un $\alpha \in \{0,1\}^n$ tal que M no podrá realizar una consulta del tipo ¿Pertenece α al oráculo? Explotamos este hecho para construir L de la siguiente manera:
Primero, notar que podemos suponer sin perdida de generalidad que M se detiene para toda entrada y devuelve un bit, ya que de lo contrario no puede decidir ningún lenguaje. Sea C el conjunto de todas las consultas realizadas por $M^\emptyset$ corriendo a partir de la entrada 1^n. Sea $\alpha \in \{0,1\}^n \setminus C$ (sabemos que existe por lo discutido arriba). Si $M^\emptyset(1^n) = 0$, entonces definimos $L = \{\alpha\}$, y si $M^\emptyset(1^n) = 1$, entonces definimos L como el conjunto vacío. Observemos que por nuestra definición de L, se cumple $M^L(1^n) = M^\emptyset(1^n)$. Ya que todas las consultas que pueda hacerle M a L en la entrada 1^n las definimos que no pertenecen a L, por ende, tanto con oráculo L como con oráculo $\emptyset$ la máquina en la entrada 1^n va a recibir 0 en todas sus consultas. Observemos que necesariamente que $M^L(1^n)$ es errado. Si $M^L(1^n) = 0$, entonces no debería existir un valor $y \in \{0,1\}^n$ tal que $y \in L$ pero eso no se cumple ya que en tal caso $\alpha \in L$. Si por el contrario $M^L(1^n) = 1$, entonces debería existir un $y \in \{0,1\}^n$ tal que $y \in L$ pero eso no ocurre porque por nuestra definición de L, en este caso $L = \emptyset$. Por lo tanto M^L erra al decidir la entrada 1^n y por ende M^L no decide a U_L. ¡Como queríamos! □

La anterior proposición no nos alcanza, tenemos que cambiar el orden de los cuantificadores! Acabamos de demostrar que para toda máquina polinomial [...] existe un lenguaje [...]. Lo que ahora debemos demostrar es que existe un lenguaje [...] tal que para toda máquina polinomial[...]. El lenguaje requerido B lo construiremos "por niveles" usando la proposición anterior.

Enumeremos todas las MT que corren en tiempo polinomial $M_1, M_2, M_3, \ldots$ con sus respectivos polinomios $p_1, p_2, p_3, \ldots$
Si esta acción le parece dudosa al lector notar que no damos una MT que enumera a las máquinas de Turing polinomiales, no existe tal máquina porque como sabemos: el conjunto de las MT's que finalizan para toda entrada no es un efectivamente enumerable (porque sino podríamos computar la función $HALT$, ver Teorema 29). Pero aún así, podemos enumerarlas (no de manera automática a partir de una MT) sino simplemente notando que existe una enumeración. Y tal enumeración existe porque del conjunto de las MT's es numerable (existe una función biyectiva entre $\mathbb{N}$ y el conjunto de las MT's).

A continuación definimos recursivamente un secuencia de pares (B_i, n_i), con $n_i \in \mathbb{N}$ y $B_i \subseteq \{0,1\}^{\leq n_i}$. Definimos $B_0 = \emptyset$ y $n_0 = 1$, y para cada entero $i > 0$ definimos B_i de la siguiente forma:
Sea M_i la i-ésima MT de la enumeración y p_i el polinomio tal que dicha máquina corre en tiempo p_i. Sea n_i el menor entero tal que $2^{n_i} > p_i(n_i)$ y que $n_i > p_j(n_j)$ para $1 \leq j < i$. Se ejecuta $M_i^{B_{i-1}}$ a partir de la entrada 1^{n_i} y sea z lo que devuelve. Sea C el conjunto de consultas de largo n_i que realizó M_i en tal ejecución, y sea $\alpha \in \{0,1\}^{n_i} \setminus C$ (sabemos que tal α existe por lo analizado anteriormente). Si $z = 0$ definimos $B_i = B_{i-1} \cup \{\alpha\}$, caso contrario definimos $B_i = B_{i-1}$ (notar que en el último caso B_i no tiene palabras de tamaño n_i).

Sea $k > 0$ un número natural veamos que $M_k^{B_k}$ no decide a U_{B_k}; para ello nos alcanza con ver que $M_k^{B_k}(1^{n_k})$ es errado. En pos de una contradicción, supongamos que $M_k^{B_k}$ decide a U_{B_k}. Supongamos primero que $M_k^{B_k}(1^{n_k}) = 0$. Entonces no existe $y \in \{0,1\}^{n_k}$ tal que $y \in B_k$, pero entonces (por construcción) tenemos que $B_{k-1} = B_k$, y en consecuencia $M_k^{B_k}(1^{n_k}) = M_k^{B_{k-1}}(1^{n_k}) = 0$. Pero si $M_k^{B_{k-1}}(1^{n_k}) = 0$, revisando la construcción de B_k, vemos que existe un $\alpha \in \{0,1\}^{n_k}$ tal que $\alpha \in B_k$, ¡Absurdo! Por otra parte, si $M_k^{B_k}(1^{n_k}) = 1$, entonces existe un $\alpha \in \{0,1\}^{n_k}$ tal que $\alpha \in B_k$ luego necesariamente $M_k^{B_{k-1}}(1^{n_k}) = 0$ (porque si fuera 1 entonces en la construcción tendríamos $B_{k-1} = B_k$). Observemos que en la entrada 1^{n_k} la máquina con oráculo B_{k-1} devuelve 0 y con oráculo B_k devuelve 1. Luego necesariamente debe haber una consulta que hace la máquina en la cual las respuestas del oráculo difieren, y esa consulta únicamente puede ser α. Lo que es absurdo porque en la construcción de B_k, si $M_k^{B_{k-1}}(1^{n_k}) = 0$ entonces se agrega a B_k un $\alpha \in \{0,1\}^{n_k}$ tal que α no fue una consulta de la ejecución de $M_k^{B_{k-1}}$ en la entrada 1^{n_k}.

Definimos ahora $B = \bigcup B_i$; veremos que para toda MTO polinomial M tenemos que M^B no decide a U_B. Sea M un MTO polinomial, y sea $i \in \mathbb{N}$ tal que $M = M_i$. Por lo anterior tenemos que $M_i^{B_i}$ no decide a U_{B_i}, por lo tanto tampoco decide a U_B. Notar que $M_i^{B_i}(1^{n_i}) = M_i^B(1^{n_i})$. Ya que en la entrada

1^{n_i} la máquina corre en tiempo $p_i(n_i)$ (finaliza en a lo sumo $p_i(n_i)$ pasos) y como las palabras que puede haber en $B \setminus B_i$ son de tamaño mayor a $p_i(n_i)$ se tiene que toda consulta que realiza M_i en entrada 1^{n_i} puede ser contestada por B_i. En consecuencia, M_i^B no decide a U_B. De esto podemos concluir que $U_B \notin \mathbf{P}^B$, y por lo tanto $\mathbf{P}^B \neq \mathbf{NP}^B$. □

Vale observar que el lenguaje B construido no es decidible pero modificando un poco su construcción utilizando el lenguaje
$\mathcal{R} := \{\langle M_p \rangle : M \text{ MT y } p \text{ polinomio}\}$, que es efectivamente enumerable, definido en el teorema de Ladner (Teorema 32) se lo puede "transformar" en decidible. Es decir, con algunos cambios que al lector no le aportan mucho se puede hacer un lenguaje C decidible tal que $\mathbf{P}^C \neq \mathbf{NP}^C$ y la manera de construirlo es esencialmente la manera de construir B.

Sea Υ un reticulado, a partir de la definición de distributividad, a partir de los axiomas de la teoría de reticulados y a partir del mismo Υ (sus características) se puede demostrar o refutar que Υ es distributivo. Cuando fijamos el lenguaje oráculo como A (o cualquier otro lenguaje) estamos fijando el "reticulado", luego a partir de las características de él y haciendo uso de la diagonalización pudimos demostrar que $\mathbf{P}^A = \mathbf{NP}^A$. Naturalmente, el "reticulado" fijado por el oráculo A tiene características que lo hacen diferente al "reticulado" fijado por B ya que $\mathbf{P}^B \neq \mathbf{NP}^B$. ¿Cuáles serán tales características?

El lector se puede preguntar ¿Por qué $\mathbf{P} = \mathbf{NP}$ no implica $\forall A\ \mathbf{P}^A = \mathbf{NP}^A$? o análogamente ¿Por qué $\mathbf{P}^B \neq \mathbf{NP}^B$ no implica $\mathbf{P} \neq \mathbf{NP}$? La notación estándar de los lenguajes decididos por oráculos como por ejemplo $\mathbf{P}^A$ es engañosa. La noción de $\mathbf{P}^A$ es *parecida* a la de $\mathbf{P}$ pero es *distinta*, las máquinas oráculos son *sintácticamente* casi idénticas a las MT's pero *semánticamente* son radicalmente distintas. En sí, $\mathbf{P}$ es un caso particular de $\mathbf{P}^A$ con $\mathbf{P} = \mathbf{P}^\emptyset$. A priori: ¿Por qué un caso particular como $\mathbf{P}^\emptyset = \mathbf{NP}^\emptyset$ debería implicar $\forall A\ \mathbf{P}^A = \mathbf{NP}^A$? Con la misma intuición, dados x, y reales ¿Por qué un caso particular como $x^0 = y^0$ debería implicar $\forall a$ real $x^a = y^a$? Recordando: un modelo de la teoría son las MTO's con acceso oráculo A, otro modelo distinto son las MTO's con acceso oráculo B, cada lenguaje distinto L hace un modelo distinto (las MTO's con acceso oráculo L). ¿A priori por qué una propiedad debería valer para todos los modelos?

A la pregunta del principio de la sección (¿Puede diagonalización resolver $\mathbf{P}$ vs $\mathbf{NP}$?) la respuesta es: sí, pero utilizando algún hecho o característica de las MT's que sea independiente a las propiedades mencionadas. Si bien muchos descubrimientos en la teoría son relativizantes, hay algunos que no, como el teorema de Cook-Levin (Teorema 15) de que `SAT` es NP-completo. Y en el último capítulo veremos la clase $\mathbf{IP}$ (Definición 76) y que $\mathbf{PSPACE} = \mathbf{IP}$ (Teorema 79 y Teorema 77), y para estas clases se sabe que existe un oráculo B tal que $\mathbf{PSPACE}^B \neq \mathbf{IP}^B$. Por ende $\mathbf{PSPACE} = \mathbf{IP}$ es otro resultado no relativizante.

Relativización vs Independencia

La noción de *relativización* es parecida a los resultados de independencia en la lógica matemática que son resultados que prueban que una afirmación no puede ser demostrada ni refutada en una teoría en particular. Como ejemplos clásicos podemos citar la independencia del quinto postulado en la geometría Euclidiana y la Hipótesis del Continuo de la teoría de Conjuntos de Zermelo-Fraenkel. Pero en nuestro ámbito estos resultados en lugar de mostrarnos la independencia de **P** vs **NP**, nos sugieren la independencia de una generalización de **P** vs **NP** de la diagonalización (que podría ser una técnica pera resolver **P** vs **NP**). En un intento de formalizar estos conceptos en el trabajo [9] se presenta un sistema axiomático que se sabe que implica exactamente aquellos resultados sobre **P** que relativizan. Por lo tanto, un resultado que no relativiza no es demostrable en tal sistema. Entonces: ¿Cómo se lo puede extender para que también pueda demostrar aquellos resultados que no relativizan? Una idea no muy astuta sería agregar como axioma cada resultado que vaya surgiendo que no relativice. Una idea más inteligente sería encontrar un hecho o unos pocos que agregados al sistema impliquen los resultados conocidos que no relativizan. Sorprendentemente esa propiedad es conocida. Aparentemente el único hecho que hace falta agregar es la localidad de la computación. El teorema de Cook-Levin de que `SAT` es NP-completo parece ser la clave para desentrañar el problema del milenio. Hay varias maneras satisfactorias y equivalentes de formalizar la localidad de la computación. Lamentablemente por ahora no hemos podido sacarle todo el "jugo" a esa fundamental característica de la computación. Y es natural que sea así, ¡Conocer los axiomas de la aritmética no le brinda a uno demasiadas pistas para demostrar el último teorema de Fermat!

Capítulo 3

Jerarquía Polinomial: PH

Hay muchos problemas que de manera natural "entran" en alguna clase de complejidad, en este capítulo veremos una que capturará de manera natural muchos problemas que otras clases no. Por lo general todos los problemas de "optimización" caerán naturalmente en esta clase. Ella es **PH**, de *Polinomial Hierarchy* (Jerarquía Polinomial), y consiste de infinitas subclases llamadas *niveles* que son generalizaciones de **P**, **NP** y **coNP** . Se conjetura que los niveles son distintos siendo esta conjetura una forma más fuerte de $\mathbf{P} \neq \mathbf{NP}$. Esta conjetura aparece incluso a veces de manera inesperada y sorprendente en muchas y diversas áreas del campo. Por todo ello el estudio de esta clase es fundamental para seguir adentrándose en el mundo de complejidad computacional.

Definición 38. La clase $\Sigma_\mathbf{2}^\mathbf{p}$ se define como el conjunto de lenguajes L para los cuales existe una Máquina de Turing determinista polinomial M y un polinomio q tal que $\forall x \in \{0,1\}^*$, $x \in L$ si y solo si

$$\exists u \in \{0,1\}^{q(|x|)} \forall v \in \{0,1\}^{q(|x|)} M(x,u,v) = 1.$$

Observación. Note que **NP** y **coNP** están incluidas en $\Sigma_\mathbf{2}^\mathbf{p}$.

Ejemplo. Sea `MAXCLIQUE` el lenguaje de los pares (G,k) tal que G es un grafo que tiene una *clique* máxima de tamaño k. Se tiene que `MAXCLIQUE` $\in \Sigma_\mathbf{2}^\mathbf{p}$ ya que $(G,k) \in$ `MAXCLIQUE` sii G tiene una clique de tamaño k (el certificado u) y para cualquier otra clique v se tiene $|v| \leq k$.

Como se mencionó en la introducción, **PH** consiste de infinitos "niveles", $\Sigma_\mathbf{2}^\mathbf{p}$ vendría a ser el nivel "dos" y **NP** el uno. En general el $(i+1)$-ésimo nivel tendrá un cuantificador existencial más que el i-ésimo nivel. A continuación, la definiremos formalmente.

Definición 39 (La clase **PH**). Para cada $i \geq 1$, decimos que un lenguaje L pertenece a $\Sigma_\mathbf{i}^\mathbf{p}$ si existe una MT M polinomial y un polinomio q tal que

$\forall x \in \{0,1\}^*$, $x \in L$ si y solo si

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \dots Q_i u_i \in \{0,1\}^{q(|x|)} M(x, u_1, \dots, u_i) = 1.$$

Donde Q_i denota $\forall$ o $\exists$ dependiendo de si i es par o impar respectivamente.

La Jerarquía Polinomial se define como $\mathbf{PH} = \cup_{i \geq 1} \sum_i^{\mathbf{P}}$. Y para cada $i \geq 1$ análogamente se define $\mathbf{\Pi_i^P} = \mathbf{co}\mathit{\Sigma}_\mathbf{i}^\mathbf{P} = \{\overline{L} : L \in \mathbf{\Sigma_i^P}\}$.

Observación. Note que $\mathbf{NP} = \mathit{\Sigma}_\mathbf{1}^\mathbf{P}$ y $\mathbf{coNP} = \mathbf{\Pi_1^P}$. Y note también que $\mathit{\Sigma}_\mathbf{i}^\mathbf{P} \subseteq \mathbf{\Pi_{i+1}^P} \subseteq \mathit{\Sigma}_\mathbf{i+2}^\mathbf{P}$ ya que si por ejemplo L es tal que $x \in L$ si y solo si

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \dots Q_i u_i \in \{0,1\}^{q(|x|)} M(x, u_1, \dots, u_i) = 1$$

para una MT M polinomial y q polinomio, entonces $x \in L$ sii

$$\forall u_0 \in \{0,1\}^{q(|x|)} \exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \dots$$

$$\dots Q_i u_i \in \{0,1\}^{q(|x|)} M(x, u_1, \dots, u_i) = 1$$

Por ende también se tiene $\mathbf{PH} = \bigcup_{i \geq 1}^{\infty} \mathbf{\Pi_i^P}$.

Ahora veamos una forma más fuerte de $\mathbf{NP} \neq \mathbf{coNP}$.

Teorema 40 (**PH** colapsa)**.** *Para cada $i \geq 1$, si $\mathit{\Sigma}_\mathbf{i}^\mathbf{P} = \mathbf{\Pi_i^P}$ entonces $\mathbf{PH} = \mathit{\Sigma}_\mathbf{i}^\mathbf{P}$; léase, la Jerarquía Polinomial colapsa al nivel i.*

Demostración. Hagamos inducción sobre i:

1) $i = 1$:

Supongamos que $\mathit{\Sigma}_\mathbf{1}^\mathbf{P} = \mathbf{\Pi_1^P}$ (equivalentemente $\mathbf{NP} = \mathbf{coNP}$) queremos ver que $\mathbf{PH} = \mathit{\Sigma}_\mathbf{1}^\mathbf{P}$. Es decir que

$$\forall j : \mathit{\Sigma}_\mathbf{j}^\mathbf{P} = \mathbf{NP}$$

hagamos inducción sobre j. El caso base ($j = 1$) es trivial, vayamos al inductivo. Supongamos que $\mathit{\Sigma}_\mathbf{j}^\mathbf{P} = \mathbf{NP}$ queremos ver que $\mathit{\Sigma}_\mathbf{j+1}^\mathbf{P} = \mathbf{NP}$.
Sea $L \in \mathit{\Sigma}_\mathbf{j+1}^\mathbf{P}$ queremos ver que $L \in \mathbf{NP}$. Por def hay una MT M que corre en tiempo polinomial y un polinomio q que cumple:
$x \in L$ si y solo si

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \dots Q u_{j+1} \in \{0,1\}^{q(|x|)} M(x, u_1, \dots, u_{j+1}) = 1.$$

Sea

$$L_2 = \{(x, u_1) : \forall u_2 \in \{0,1\}^{q(|x|)} \dots Q u_{j+1} \in \{0,1\}^{q(|x|)} : M(x, u_1, \dots, u_{j+1}) = 1\}$$

notemos que $L_2 \in \mathbf{\Pi_j^P}$. Luego $\overline{L}_2 \in \mathit{\Sigma}_\mathbf{j}^\mathbf{P}$ y por hipótesis inductiva $\overline{L}_2 \in \mathbf{NP}$, por lo tanto $L_2 \in \mathbf{coNP}$. Y como por hipótesis del caso base ($i = 1$ equivalente a $\mathbf{NP} = \mathbf{coNP}$) tenemos que $L_2 \in \mathbf{NP}$. Luego existe una MT M_2 polinomial y un polinomio p tal que $x \in L_2 \iff \exists v \in \{0,1\}^{p(|x|)} M_2(x, v) = 1$.

Entonces $x \in L \iff \exists u \in \{0,1\}^{q(|x|)} \exists v \in \{0,1\}^{p(|x|)} M_2((x,u),v) = 1$. Luego definiendo h polinomio como $h = q+p$ y definiendo M_3 MT que realice lo mismo que M_2 pero leyendo la entrada convenientemente se tiene que:

$x \in L \iff \exists z \in \{0,1\}^{h(|x|)} M_3(x,z) = 1$ y por lo tanto $L \in \mathbf{NP}$.

2) $i \Rightarrow i+1$:

La hipótesis inductiva expresa que para i se cumple: si $\Sigma_\mathbf{i}^\mathbf{p} = \mathbf{\Pi}_\mathbf{i}^\mathbf{p}$ entonces $\mathbf{PH} = \Sigma_\mathbf{i}^\mathbf{p}$. Lo aclaro porque puede ser fácil confundirse. Entonces queremos ver que bajo tal hipótesis: si $\Sigma_\mathbf{i+1}^\mathbf{p} = \mathbf{\Pi}_\mathbf{i+1}^\mathbf{p}$ entonces $\mathbf{PH} = \Sigma_\mathbf{i+1}^\mathbf{p}$.
Supongamos que $\Sigma_\mathbf{i+1}^\mathbf{p} = \mathbf{\Pi}_\mathbf{i+1}^\mathbf{p}$ queremos ver que $\mathbf{PH} = \Sigma_\mathbf{i+1}^\mathbf{p}$. Que equivale a probar $\forall j : \Sigma_\mathbf{j}^\mathbf{p} \subseteq \Sigma_\mathbf{i+1}^\mathbf{p}$. Hagamos inducción sobre j.

a) $j = 1$:

Como $i + 1 > 1$ se tiene que $\Sigma_\mathbf{1}^\mathbf{p} \subseteq \Sigma_\mathbf{i+1}^\mathbf{p}$.

b) $j \Rightarrow j + 1$:

Supongamos que $\Sigma_\mathbf{j}^\mathbf{p} \subseteq \Sigma_\mathbf{i+1}^\mathbf{p}$ queremos probar que $\Sigma_\mathbf{j+1}^\mathbf{p} \subseteq \Sigma_\mathbf{i+1}^\mathbf{p}$. Sea $L \in \Sigma_\mathbf{j+1}^\mathbf{p}$ queremos ver que $L \in \Sigma_\mathbf{i+1}^\mathbf{p}$. Por definición se cumple que existe M MT polinomial y polinomio q tal que $x \in L$ si y solo si

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q_{j+1} u_{j+1} \in \{0,1\}^{q(|x|)} M(x, u_1, \ldots, u_{j+1}) = 1$$

sea

$$L_2 = \{(x, u_1) : \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q_{j+1} u_{j+1} \in \{0,1\}^{q(|x|)} M(x, u_1, \ldots, u_{j+1}) = 1\}$$

entonces $L_2 \in \mathbf{\Pi}_j^\mathbf{p}$ y por ende $\overline{L}_2 \in \Sigma_\mathbf{j}^\mathbf{p}$, luego por hipótesis como $\Sigma_\mathbf{j}^\mathbf{p} \subseteq \Sigma_\mathbf{i+1}^\mathbf{p}$ se tiene que $\overline{L}_2 \in \Sigma_\mathbf{i+1}^\mathbf{p}$. En consecuencia $L_2 \in \mathbf{\Pi}_\mathbf{i+1}^\mathbf{p}$ y como por hipótesis $\Sigma_\mathbf{i+1}^\mathbf{p} = \mathbf{\Pi}_\mathbf{i+1}^\mathbf{p}$ se tiene que $L_2 \in \Sigma_\mathbf{i+1}^\mathbf{p}$. Por lo tanto hay una MT M_2 polinomial y un polinomio r tal que $x \in L_2$ si y solo si

$$\exists u_1 \in \{0,1\}^{r(|x|)} \forall u_2 \in \{0,1\}^{r(|x|)} \ldots Q u_{i+1} \in \{0,1\}^{r(|x|)} M_2(x, u_1, \ldots, u_{i+1}) = 1.$$

Luego $x \in L$ sii

$$\exists u_0 \in \{0,1\}^{q(|x|)} \exists u_1 \in \{0,1\}^{r(|x|)} \forall u_2 \in \{0,1\}^{r(|x|)} \ldots$$
$$\ldots Q_{i+1} u_{i+1} \in \{0,1\}^{r(|x|)} M_2((x, u_0), u_1, \ldots, u_{i+1}) = 1.$$

Sea h un polinomio definido como $h = q+r$ y sea M_3 MT polinomial que realiza lo mismo que M_2 interpretando la entrada convenientemente. Entonces $x \in L$ si y solo si

$$\exists u_1 \in \{0,1\}^{h(|x|)} \forall u_2 \in \{0,1\}^{h(|x|)} \ldots Q u_{i+1} \in \{0,1\}^{h(|x|)} M_3(x, u_1, \ldots, u_{i+1}) = 1$$

y en conclusión $L \in \Sigma_\mathbf{i+1}^\mathbf{p}$. □

Corolario. *Si* $\mathbf{P} = \mathbf{NP}$ *entonces* $\mathbf{PH} = \mathbf{P}$.

Demostración. Supongamos que $\mathbf{P} = \mathbf{NP}$ entonces $\mathbf{P} = \mathbf{coNP}$ y por ende $\mathbf{NP} = \mathbf{coNP}$, luego por el teorema anterior se tiene que $\mathbf{PH} = \mathbf{NP}$ en consecuencia $\mathbf{PH} = \mathbf{P}$. □

El "sentido común" (a veces errado) parece indicar que el anterior corolario es una pista muy fuerte o casi una evidencia de que $\mathbf{P} \neq \mathbf{NP}$. En lo personal, al iniciar el estudio de complejidad computacional tenía mis dudas sobre qué creer sobre **P** vs **NP**, luego de entender el anterior corolario me paré en la vereda de los que creen que $\mathbf{P} \neq \mathbf{NP}$. Si seguimos esta creencia como punto de partida podríamos entonces tratar de demostrar $\mathbf{P} \neq \mathbf{NP}$ mostrando que $\mathbf{P} \neq \mathbf{PH}$. Y esto último debería en principio parecer más fácil ya que hay problemas arbitrariamente más difíciles que **P** (bajo nuestra creencia). En ese cometido uno podría tratar de definir alguna noción de problema completo para **PH** o para los niveles, análoga a dicha noción para los problemas completos de **NP** (ver Definición 12). Y esperar que demostrar que alguno de esos problemas no está en **P**, sea más fácil que demostrar que un problema NP-completo no está en **P**.

Problemas completos

Definición 41. Sea $i \in \mathbb{N}$ y sea L' un lenguaje, decimos que L' es *completo para* $\Sigma_{\mathbf{i}}^{\mathbf{P}}$ si:

$L' \in \Sigma_{\mathbf{i}}^{\mathbf{P}}$ y si $L \in \Sigma_{\mathbf{i}}^{\mathbf{P}}$ entonces $L \leq_p L'$. (Recordando $\leq_p$ es la reducción de Karp, [ver Definición 12])

De manera análoga se define lenguaje *completo* para $\mathbf{\Pi_i^P}$ y para **PH**.

Podría parecer difícil imaginar si cada nivel tiene un problema completo, por suerte `SAT` viene a nuestro rescate. Una generalización de éste nos permite tener un problema completo para cualquier nivel.

Definición 42. Para cada $i \geq 1$ definimos $\Sigma_{\mathtt{i}}\mathtt{SAT}$ como el conjunto de φ fórmulas boolenas tal que

$$\exists u_1 \in \{0,1\}^{q(|\varphi|)} \forall u_2 \in \{0,1\}^{q(|\varphi|)} \dots Q_i u_i \in \{0,1\}^{q(|\varphi|)} \varphi(u_1, \dots, u_i) = 1$$

con q un polinomio adecuado y φ adecuada para que $u_1 \dots u_i$ sea una asignación de φ. Se puede demostrar de manera sencilla por inducción que para cada i, $\Sigma_{\mathtt{i}}\mathtt{SAT}$ es completo para $\Sigma_{\mathtt{i}}^{\mathtt{P}}$. Recíprocamente se puede definir $\Pi_{\mathtt{i}}\mathtt{SAT}$ como el conjunto de fórmulas Booleanas satisfacibles φ que cumplen

$$\forall u_1 \in \{0,1\}^{q(|\varphi|)} \exists u_2 \in \{0,1\}^{q(|\varphi|)} \dots Q_i u_i \in \{0,1\}^{q(|\varphi|)} \varphi(u_1, \dots, u_i) = 1$$

para q un polinomio adecuado también. Análogamente, $\Pi_{\mathtt{i}}\mathtt{SAT}$ es completo para $\Pi_{\mathtt{i}}^{\mathtt{P}}$.

Observemos que $\mathtt{SAT} = \Sigma_1\mathtt{SAT}$ y que para todo i se cumple $\Sigma_{\mathtt{i}}\mathtt{SAT} \leq_p \mathtt{TQBF}$. (Para recordar `TQBF` ver Definición 23).

Teorema 43. *Si existe un lenguaje completo para* **PH** *entonces* $\exists i : \mathbf{PH} = \Sigma_{\mathbf{i}}^{\mathbf{P}}$.

Demostración. Supongamos que existe un lenguaje L' completo para **PH**, luego $\exists i : L' \in \Sigma_{\mathbf{i}}^{\mathbf{p}}$. Sea $L \in \mathbf{PH}$ luego $L \leq_p L'$ y por ende,
$L \in \Sigma_{\mathbf{i}}^{\mathbf{p}}$ por lo tanto $\mathbf{PH} \subseteq \Sigma_{\mathbf{i}}^{\mathbf{p}}$. □

Ahora que hemos ido adentrándonos en **PH**, comenzamos a evidenciar su dimensión y es natural preguntarse qué tan grande es. Tiene infinitos niveles aparentemente distintos, cada uno con problemas más "difíciles" que los problemas del nivel anterior. Parece enorme y lo es! Pero aun así, el próximo teorema muestra que **PH** está incluido en **PSPACE** mostrándonos lo gigantesco que es **PSPACE**.

Teorema 44. $\mathbf{PH} \subseteq \mathbf{PSPACE}$.

Demostración. Queremos ver que $\forall j \geq 1$ se cumple $\Sigma_{\mathbf{j}}^{\mathbf{p}} \subseteq \mathbf{PSPACE}$, lo veamos haciendo inducción en j.

El caso base se cumple ya que $\mathbf{NP} \subseteq \mathbf{PSPACE}$. Veamos el otro caso:

$j \Rightarrow j+1$:

Supongamos que $\Sigma_{\mathbf{j}}^{\mathbf{p}} \subseteq \mathbf{PSPACE}$ queremos probar que $\Sigma_{\mathbf{j+1}}^{\mathbf{p}} \subseteq \mathbf{PSPACE}$. Sea $L \in \Sigma_{\mathbf{j+1}}^{\mathbf{p}}$ queremos ver que $L \in \mathbf{PSPACE}$. Por definición existe MT M polinomial y un polinomio q tal que para toda entrada x se cumple $x \in L$ si y solo si

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q_{j+1} u_{j+1} \in \{0,1\}^{q(|x|)} M(x, u_1, \ldots, u_{j+1}) = 1.$$

Como es usual definamos

$$L_2 = \{(x, u_1) : \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q_{j+1} u_{j+1} \in \{0,1\}^{q(|x|)} M(x, u_1, \ldots, u_{j+1}) = 1\}$$

luego $L_2 \in \mathbf{\Pi}_{\mathbf{j}}^{\mathbf{p}}$, y pertenece a **PSPACE** ya que $\overline{L}_2 \in \Sigma_{\mathbf{j}}^{\mathbf{p}}$ y por hipótesis $\Sigma_{\mathbf{j}}^{\mathbf{p}} \subseteq \mathbf{PSPACE}$. Por ende hay una MT multicinta M_2 que usa espacio polinomial que decide L_2, en consecuencia $(x, u_1) \in L_2 \Longleftrightarrow M_2(x, u_1) = 1$, entonces

$$x \in L \Longleftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} : M_2(x, u) = 1.$$

Podemos hacer una MT multicinta M_3 que dado un x de entrada, simule $M_2(x, u)$ para cada $u \in \{0,1\}^{q(|x|)}$ hasta encontrar uno que haga que M_2 retorne 1 o hasta probar con todos los u. La observación necesaria es que M_3 solo necesita tener almacenado u en una cinta hasta que se termina de ejecutar $M_2(x, u)$ y luego sobrescribe el próximo u a probar en tal cinta. Y además, M_3 puede reusar el espacio usado en la computación de $M_2(x, u)$ para ejecutar M_2 con el próximo u. Por lo tanto, M_3 decide a L usando espacio polinomial, finalmente $L \in \mathbf{PSPACE}$. □

Máquinas Alternantes

Así como tenemos una noción de **NP** a partir de certificados (ver Definición 8) y otra a partir de máquinas no deterministas (ver Definición 10), podemos tener

una noción de **PH** a partir de *máquinas alternantes*. Por supuesto, la noción de **PH** a partir de certificados es lo que hemos estudiado hasta ahora. Una *máquina de Turing Alternante* es una generalización de la MTND, en la cual los estados (exceptuando $q_{aceptacion}$ y $q_{rechazo}$) tienen una *etiqueta* $e \in \{\exists, \forall\}$. Con estas nuevas máquinas tampoco se pretende representar un modelo realista de la computación (no se puede construir físicamente) sino que nos ayudarán a estudiar y comprender algunos fenómenos naturales de la computación general. En particular, la diferencia entre *buscar una solución y verificarla.*

Definición 45. Una *máquina de Turing Alternante* M (abreviada MTA) es un par (M_2, l) donde M_2 es una máquina de Turing no determinista y

$l : Q/\{q_{aceptacion}, q_{rechazo}\} \longrightarrow \{\exists, \forall\}$ es una función. Sea $T : \mathbb{N} \to \mathbb{N}$; decimos que M *corre en tiempo* T si M_2 corre en tiempo T. Dada una entrada $x \in \{0,1\}^*$ decimos que M *finaliza a partir de* x si M_2 finaliza a partir de x. Las nociones de hilo y de camino para MTND aplica a la MTA (ver Definición 9). Dada $x \in \{0,1\}^*$ si M finaliza a partir de x entonces definimos que M *acepta* a x de la siguiente manera:

EL *grafo de configuración de* M *a partir de la entrada* x denotado $G_{M,x}$, será para nosotros $G_{M_2,x}$, el grafo de configuración de M_2 a partir de x (recordar que es un grafo finito, dirigido y acíclico, en el cual cada vértice es una configuración y en el cual si hay una arista de v a s vértices, entonces M_2 puede llegar de la configuración v a la configuración s en un paso a partir de alguna de sus funciones de transición, ver Definición 21). Etiquetemos los vértices de recursivamente de la siguiente manera:

Sea v un vértice del grafo.

1) Si v tiene el estado $q_{aceptacion}$ etiquetamos a v con ACCEPT.

2) Si v tiene un estado etiquetado con $\exists$ y hay una arista de v a un vértice s tal que s esta etiquetado con ACCEPT entonces etiquetamos a v con ACCEPT.

3) Si v tiene un estado etiquetado con $\forall$ y los vértices s, t tales que vs, vt son aristas, están etiquetados con ACCEPT entonces etiquetamos a v con ACCEPT.

Decimos que M acepta a x si al finalizar este proceso el vértice que representa a la configuración inicial está etiquetado con ACCEPT. Y lo denotamos con $M(x) = 1$, si M *rechaza* a x (i.e, no acepta a x) lo denotamos con $M(x) = 0$. Decimos que M *decide a un lenguaje* L si para toda entrada $x \in \{0,1\}^*$ M finaliza a partir de x y $x \in L$ sii $M(x) = 1$.

Cantidad de alternaciones limitada

A continuación veremos una caracterización de **PH** a partir de MTA's que tendrán una cantidad máxima de alternaciones entre etiquetas ($\{\exists, \forall\}$) en la ejecución.

Definición 46. Sea $i \in \mathbb{N}$ y $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función. Decimos que un lenguaje L pertenece a $\Sigma_{\mathbf{i}}^{\mathbf{P}}\mathbf{TIME}(T)$ si existe una MTA M que decide a L que corre en tiempo T cuyo estado inicial (q_i) esta etiquetado con $\exists$ y que dada una entrada x se tiene que M a partir de x siga el camino que siga, alterna entre estados

de diferentes etiquetas a lo sumo $i-1$ veces. La clase $\mathbf{\Pi_i^p TIME}(T)$ se define análogamente, con la diferencia que la respectiva M en el estado inicial tiene la etiqueta $\forall$.

Teorema 47. *Para cada $i \in \mathbb{N}$ se tiene* : $\Sigma_{\mathbf{i}}^{\mathbf{p}} = \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{i}}^{\mathbf{p}}\mathbf{TIME}(n^c)$

y $\mathbf{\Pi_i^p} = \bigcup_{c\geq 1}^{\infty} \mathbf{\Pi_i^p TIME}(n^c)$.

Demostración. Demostraremos solo que $\Sigma_{\mathbf{i}}^{\mathbf{p}} = \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{i}}^{\mathbf{p}}\mathbf{TIME}(n^c)$ ya que la otra es análoga. Lo haremos por inducción en i.

Caso base $i=1$:

Queremos ver que $\mathbf{NP} = \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{1}}^{\mathbf{p}}\mathbf{TIME}(n^c)$. Veamos que

$\mathbf{NP} \subseteq \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{1}}^{\mathbf{p}}\mathbf{TIME}(n^c)$. Sea $L \in \mathbf{NP}$ y M una MTND que corre en tiempo polinomial que decide a L.

Notemos que si definimos MTA $M_a = (M, l)$ con l una función que etiqueta cada estado de M con $\exists$ (exceptuando $q_{aceptacion}$, $q_{rechazo}$) entonces M_a es una MTA polinomial que decide a L haciendo 0 alternaciones. Luego $L \in \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{1}}^{\mathbf{p}}\mathbf{TIME}(n^c)$. Con un razonamiento análogo podemos demostrar trivialmente la inclusión restante. Demostremos el caso $i=2$ ya que el caso inductivo es un poco engorroso de escribir pero se demuestra con la misma idea.

1) $\Sigma_{\mathbf{2}}^{\mathbf{p}} \subseteq \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{2}}^{\mathbf{p}}\mathbf{TIME}(n^c)$:

Sea $L \in \Sigma_{\mathbf{2}}^{\mathbf{p}}$ entonces hay una MT M polinomial y q polinomio tal que $x \in L \iff \exists u \in \{0,1\}^{q(|x|)} \forall v \in \{0,1\}^{q(|x|)} : M(x,u,v) = 1$.

Hagamos M_3 MTA polinomial que dada x de entrada realiza:

$r := q(|x|)$
$u := \{0,1\}^r$ no determinista
$v := \{0,1\}^r$ no determinista
computa $M(x,u,v)$ y termina.

Establezcamos que los estados que utiliza M_3 para hacer u son distintos a los estados utilizados para hacer v.También, que M_3 no simula M sino que tiene su código dentro del suyo, con la salvedad de que los estados de M no coinciden con los estados de M_3. Etiquetamos con $\exists$ a q_i y a los estados utilizados para hacer r y u, y con $\forall$ a los estados para hacer v y para ejecutar M. Entonces por esta definición conveniente de M_3, se tiene que para cualquier x se cumple

$$x \in L \iff M_3(x) = 1$$

y como corre en tiempo polinomial y solo hace una alternación, se tiene que $L \in \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{2}}^{\mathbf{p}}\mathbf{TIME}(n^c)$.

2) $\bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{2}}^{\mathbf{p}}\mathbf{TIME}(n^c) \subseteq \Sigma_{\mathbf{2}}^{\mathbf{p}}$:

Sea $L \in \bigcup_{c\geq 1}^{\infty} \Sigma_{\mathbf{2}}^{\mathbf{p}}\mathbf{TIME}(n^c)$ y sea M MTA que corre en tiempo q para q un polinomio que decide a L haciendo una alternación empezando en la etiqueta $\exists$ (l de la máquina etiqueta con $\exists$ a q_i). Y sean δ_0, δ_1 sus funciones de transición. Construyamos la siguiente MT M_2: dada de entrada $(x, z) \in \{0,1\}^*$, M_2 simula $M(x)$ siguiendo el hilo dado por z, esto es, si el i-ésimo bit de z es 0 entonces M_2 simula que en el i-ésimo paso M aplica δ_0, y que aplica δ_1 en el caso contrario (si el i-ésimo bit de z es 1). Notemos que M_2 corre en tiempo polinomial. Hagamos que si $M_2(x, z)$ lleva a $M(x)$ a una configuración con $q_{aceptacion}$ entonces $M_2(x, z)$ retorna 1, caso contrario, retorna 0. Por nuestra def de M_2, se puede ver que

$$x \in L \Longleftrightarrow \exists u \in \{0,1\}^{\leq q(|x|)} \forall v \in \{0,1\}^{\leq q(|x|)} M_2(x, uv) = 1$$

Por lo tanto, se puede hacer una MT M_3 polinomial prácticamente igual a M_2 pero con los cambios triviales y convenientes para que cumpla:

$$x \in L \Longleftrightarrow \exists u \in \{0,1\}^{q(|x|)} \forall v \in \{0,1\}^{q(|x|)} M_3(x, u, v) = 1.$$

En consecuencia $L \in \Sigma_{\mathbf{2}}^{\mathbf{p}}$. □

Cantidad de Alternaciones ilimitada

Uno naturalmente puede reparar en que si una cantidad de alternaciones limitada nos alcanza para caracterizar **PH** qué podremos caracterizar con una cantidad de alternaciones ilimitada! Con esta mayor capacidad veremos a continuación que podremos caracterizar a **PSPACE**.

Definición 48. Dada una función $T : \mathbb{N} \longrightarrow \mathbb{N}$ decimos que un lenguaje L está en $\mathbf{ATIME}(T)$ si existe una constante $c > 0$ y una MTA M que corre en tiempo cT tal que $\forall x \in \{0,1\}^*$ se tiene $x \in L$ sii $M(x) = 1$.

Teorema 49. **AP** = **PSPACE**.

Demostración. Para ver que **PSPACE** ⊆ **AP** basta con mostrar que **TQBF** ∈ **AP**, y ello se desprende de que simplemente podemos "adivinar" valores para cada variable ligada al cuantificador existencial mediante un estado etiquetado con $\exists$ y para aquellas ligadas al universal mediante el uso de estados de etiqueta $\forall$. Para ver que **AP** ⊆ **PSPACE** se puede utilizar un procedimiento recursivo similar al utilizado en la prueba de que `TQBF` ∈ **PSPACE**. □

Comentario al margen, si definimos **APSPACE** como el conjunto de lenguajes decididos por MTA's multicinta (si se generaliza la Definición de MTA) que

usan espacio polinomial, entonces se puede demostrar que **APSPACE** = **EXP**. Similarmente, el conjunto de lenguajes decididos por máquinas alternantes multicintas que usan espacio logarítmico es igual a **P**. Estos resultados se obtienen con la mismas técnicas usadas en este capítulo.

Otro resultado interesante que nos regala el estudio de **PH** y que vale la pena mencionar es que SAT no se puede decidir por una MT multicinta que use espacio logarítmico y tiempo lineal simultáneamente. Sin entrar en muchos detalles, para todas funciones S,$T : \mathbb{N} \to \mathbb{N}$, se define $\mathbf{TISP}(T,S)$ como el conjunto de lenguajes para los cuales existe una MT multicinta que los decide que corre en tiempo T y en espacio S. Luego, con las técnicas vistas en el capítulo como el uso del grafo de configuración y demás, se puede probar $\mathtt{SAT} \notin \mathbf{TISP}(n^{1,1}, n^{0,1})$. El lector interesado puede estudiar en detalle tal resultado en la subsección 5.4 de [8].

Jerarquía Polinomial vía Oráculos

A continuación veremos una caracterización de **PH** a través de máquinas de Turing Oráculos (ver Definición 33).

Recordando, una MTO M es una máquina de Turing que tiene tres estados adicionales $q_{pregunta}$, q_{si} y q_{no} y una cinta adicional llamada cinta oráculo. Para correr M a partir de una entrada x, fijamos un lenguaje $O \subseteq \{0,1\}^*$ utilizado como oráculo por la máquina. Si en algún momento de la ejecución M entra al estado $q_{pregunta}$ entonces M en un solo paso va al estado q_{si} si $y \in O$ y a q_{no} en caso contrario, donde $y \in \{0,1\}^*$ es lo que está escrito en la cinta oráculo al momento de entrar al estado $q_{pregunta}$. Dada $x \in \{0,1\}^*$, con $M^O(x)$ denotamos a la salida de la MTO M a partir de la entrada x con acceso al lenguaje O.

Para cualquier lenguaje $O \subseteq \{0,1\}^*$, con $\mathbf{P}^O$ denotamos al conjunto de lenguajes decididos por MTO's polinomiales con acceso oráculo O, y con $\mathbf{NP}^O$ al conjunto de lenguajes decididos por MTNDO's polinomiales con acceso oráculo O.

Teorema 50. *Para cada $i \geq 2$, $\Sigma_{\mathbf{i}}^{\mathbf{P}} = \mathbf{NP}^{\Sigma_{\mathrm{i}-1}\mathtt{SAT}}$.*

Demostración. Se demuestra por inducción, veamos el caso base ya que el caso inductivo usa la misma idea. Mostremos entonces que $\Sigma_{\mathbf{2}}^{\mathbf{P}} = \mathbf{NP}^{\mathbf{SAT}}$. (Recordando $\Sigma_{\mathbf{1}}\mathtt{SAT} = \mathtt{SAT}$ ver Definición 42).

Sea $L \in \Sigma_{\mathbf{2}}^{\mathbf{P}}$ luego hay una MT M polinomial y q polinomio tal que:
$x \in L \iff \exists u \in \{0,1\}^{q(|x|)} \forall v \in \{0,1\}^{q(|x|)} M(x,u,v) = 1$. Definamos
$L_2 = \{(x,u) : \forall v \in \{0,1\}^{q(|x|)} M(x,u,v) = 1\}$. Notemos que dado (x,u) se cumple $(x,u) \notin L_2$ sii $\exists v \in \{0,1\}^{q(|x|)} M(x,u,v) = 0$. Por ende
$\overline{L_2} = \{(x,u) : \exists v \in \{0,1\}^{q(|x|)} M(x,u,v) = 0\}$. Observemos que $\overline{L_2} \in \mathbf{NP}$ luego $\overline{L_2}$ puede ser decidido polinomialmente por una MTO con acceso oráculo a SAT, y por ende L_2 también puede ser decidido por una MTO polinomial con acceso oráculo a SAT, sea M_2 tal MTO (la que decide a L_2). En consecuencia $x \in L \iff \exists u \in \{0,1\}^{q(|x|)} M_2^{\mathtt{SAT}}(x,u) = 1$ y por ende $L \in \mathbf{NP}^{\mathtt{SAT}}$.

Ahora la otra inclusión, sea $L \in \mathbf{NP}^{\mathtt{SAT}}$ y sea N la MTNDO con oráculo en SAT que decide a L. Sea q polinomio y $c \in \mathbb{N}$ tal que N corre en cq pasos. Notemos que dada una entrada x, N acepta a x sii hay un camino $c_1, c_2, \ldots, c_m$ que lleva a N al estado de aceptación a partir de x, con $m = cq(|x|)$. Entonces podría haber $k \leq m$ preguntas al oráculo y por ende k respuestas a tales consultas. Sean $a_1, \ldots, a_k$ las respuestas a tales consultas. Luego podemos decir que N acepta a x si y solo si existe un camino que de seguirlo, haciendo preguntas al oráculo (tal vez cero) y obteniendo las respuestas (todas siendo respuestas correctas) entonces llega al estado de aceptación. Expresemos esto con cuantificadores, la parte no trivial es expresar que todas las respuestas son correctas y eso lo podemos hacer así.
Sea φ_i la i-ésima pregunta al oráculo y a_i su respuesta. Luego si $a_i = 1$ entonces $\exists u_i$ asignación tal que $\varphi_i(u_i) = 1$ y en cambio si $a_i = 0$ entonces $\forall v_i$ asignación $\varphi_i(v_i) = 0$. Por lo tanto podemos afirmar que $x \in L$ sii $\exists c_1, \ldots, c_m, \exists a_1, \ldots, a_k, \exists u_1, \ldots, u_k \forall v_1, \ldots, v_k$ tal que N acepta a x siguiendo el camino $c_1, \ldots, c_m$ obteniendo las respuestas a_i y que todas las a_i cumplen la condición antes descrita. Luego podemos construir una MT polinomial que dada $(x, c_1, \ldots, c_m, a_1, \ldots, a_k, u_1, \ldots, u_k, v_1, \ldots, v_k)$ de entrada simule a $N(x)$ siguiendo el camino $c_1, \ldots, c_m$ con tales respuestas a sus preguntas y comprobando la correctitud de éstas (de manera polinomial computando $\varphi_i(u_i)$ o $\varphi_i(v_i)$ dependiendo del caso) a partir de los respectivos u_i y v_i. La máquina para corroborar que la respuesta a_i a la consulta φ_i es correcta realiza lo siguiente: si $a_i = 1$ entonces computa $\varphi_i(u_i)$, si da 1 entonces asume que a_i es correcta sino asume lo contrario. Si en cambio $a_i = 0$ entonces computa $\varphi_i(v_i)$, si da 0 entonces asume que es correcta sino asume lo contrario. Con todo esto se ve que $L \in \Sigma_2^p$. □

Notación: algunos autores denotan la clase
$\Sigma_\mathbf{2}^\mathbf{p} = \mathbf{NP}^{\mathbf{SAT}}$ por $\mathbf{NP}^{\mathbf{NP}}$, $\Sigma_\mathbf{3}^\mathbf{p}$ por $\mathbf{NP}^{\mathbf{NP}^{\mathbf{NP}}}$ y así, ya que por el teorema anterior tener acceso oráculo a un lenguaje completo para una clase permite decidir todos los lenguajes en esa clase.

Capítulo 4

Circuitos Booleanos: P/poly

En este capítulo estudiaremos un nuevo modelo de computación llamado *Circuitos Booleanos*. Como se verá más adelante, estos circuitos son mucho más simples que las máquinas de Turing por lo que a priori razonar sobre ellos o descubrir cosas aún no descubiertas sobre las MT's parece más fácil. En particular, encontrar *lower bounds* de circuitos (cotas inferiores) parece mucho más fácil que encontrarlas en MT's (demostrar que un problema no puede ser resuelto por un circuito de determinado tamaño parece ser más fácil que demostrar que determinado problema no puede ser resuelto por una MT en determinado tiempo). Ya solo esto justifica su estudio ya que es un camino que ha seguido la comunidad para tratar de demostrar que $\mathbf{P} \neq \mathbf{NP}$ y otras separaciones de clases. Además, por más que el gran problema siga irresuelto por mucho tiempo, encontrar un circuito chico que resuelva `SAT` u otros problemas difíciles, en digamos, instancias de hasta 100000 variables, tendría gran utilidad ya que en la práctica problemas "intratables" se volverían tratables.

Sea (V, E) un grafo dirigido; un vértice $v \in V$ se dirá *inicial* (resp. *final*) si para todo $x \in V$ vale que $(x, v) \notin E$ (resp. $(v, x) \notin E$).

Definición 51 (Circuito)**.** Un *circuito booleano* es una 4-upla $((V, E), e, <_i, <_o)$ donde (V, E) es un grafo dirigido acíclico finito. Además, si $I, F \subseteq V$ son los conjuntos de vértices iniciales y finales de (V, E), respectivamente, tenemos:

- $e : V \setminus I \to \{\neg, \vee, \wedge\}$ es una función.
- $<_i$ es un orden total sobre I, y
- $<_o$ es un orden total sobre O.

Adicionalmente se cumple para cada $v \in V$ que: si $e(v) \in \{\vee, \wedge\}$, entonces el grado de entrada de v es 2, y si $e(v) = \neg$, entonces el grado de entrada de v es 1. (Observar que no hay ninguna restricción al grado de salida de los vértices, y que I,O no pueden ser vacíos ya que el grafo es en particular finito y acíclico.)

Dado un vértice $v \in V \setminus I$, diremos que $e(v)$ es su *etiqueta.* Un circuito con n nodos iniciales y m nodos finales "transforma" palabras de n bits en palabras de m bits. A continuación formalizamos esta idea. Sea $C = ((V, E), e, <_i, <_o)$ un circuito con n nodos iniciales, definimos recursivamente la función
$val : V \times \{0,1\}^n \longrightarrow \{0,1\}$ de la siguiente manera:

$$\text{val}(v, x) = \begin{cases} x_k & \text{si } v \text{ es el } k\text{-ésimo nodo inicial (resp. de } <_i \text{)} \\ \neg val(w, x) & \text{si } e(v) = \neg \text{ y } (w, v) \in E \\ val(w, x) \vee val(u, x) & \text{si } e(v) = \vee \text{ y } (w, v), (u, v) \in E \\ val(w, x) \wedge val(u, x) & \text{si } e(v) = \wedge \text{ y } (w, v), (u, v) \in E \end{cases}$$

La *salida de C a partir del input x*, denotada $C(x)$, es la palabra
$val(s_1, x) \dots val(s_m, x)$ donde $s_1 <_o \cdots <_o s_m$ son los nodos finales de C.

Usualmente representaremos circuitos mediante diagramas, para lo cual utilizamos la convención de que el orden de los nodos iniciales y finales es de izquierda a derecha en el dibujo. En algunos diagramas se verán rectángulos de etiqueta con el nombre de un circuito, esto es para representar que un circuito puede tener dentro otros circuitos como "subrutinas". Esto hace más legible, simple y didáctica la explicación de algunos circuitos complejos. Veamos un ejemplo simple.

Ejemplo. Circuito Xor que computa la función $XOR : \{0,1\}^2 \longrightarrow \{0,1\}$

$$XOR(a, b) = 1 \iff a \neq b$$

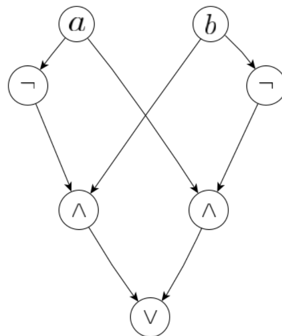

Figura 4.1:

Por lo que se ve, un circuito tiene una cantidad fija de nodos iniciales y solo procesa entradas de dicha cantidad de bits. Entonces: ¿Cómo un circuito puede decidir un lenguaje? No puede, el lenguaje va a ser decidido por una sucesión infinita de circuitos. Cada uno de esos circuitos va a decidir una "porción" del lenguaje, en general el i-ésimo circuito va a tener i nodos iniciales y va a decidir a las entradas de i bits.

Definición 52. Sea $\{C_n\}_{n\in\mathbb{N}}$ una sucesión de circuitos booleanos tal que para todo $n \in \mathbb{N}$, el circuito C_n tiene n nodos iniciales y un único nodo final. Dado $L \subseteq \{0,1\}^*$ diremos que $\{C_n\}_{n\in\mathbb{N}}$ *decide* a L, si

$$\forall n \in \mathbb{N}\ \forall x \in \{0,1\}^n : x \in L \Longleftrightarrow C_n(x) = 1$$

Definición 53. Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ una función. Un lenguaje $L \subseteq \{0,1\}^*$ pertenece a la clase de complejidad $\mathbf{SIZE}(T(n))$ si existe un sucesión $\{C_n\}_{n\in\mathbb{N}}$ de circuitos booleanos que decide a L, y que cumple $|C_n| \leq T(n)$ para todo $n \in \mathbb{N}$. Donde $|C_n|$ es $|V|$, llámese tamaño del circuito.

A continuación definiremos la clase más importante del capítulo, la que modela a los lenguajes decididos eficientemente por circuitos.

Definición 54 (La clase $\mathbf{P}_{/\mathbf{poly}}$)**.** Decimos que $\mathbf{P}_{/\mathbf{poly}}$ es la clase de lenguajes decididos por sucesiones de circuitos booleanos de tamaño polinomial, esto es

$$\mathbf{P}_{/\mathbf{poly}} = \bigcup_{c\geq 1}^{\infty} \mathbf{SIZE}(n^c).$$

El teorema que se demostrará a continuación es lo que explica de manera profunda la razón por la cual un chip puede simular máquinas de Turing deterministas, y por ende, computar. Este teorema podría haberse demostrado de una manera mucho más simple y corta adaptando el teorema de Cook-Levin (Teorema 15). En la bibliografía las demostraciones encontradas suelen ser esquemas o vagas explicaciones y no se encontró alguna con un nivel de detalle comparado a la prueba que a continuación se presenta. Se decide darla con tanto detalle porque:

- Me pareció muy importante lo que se mencionó al principio de este párrafo.
- Luego de seguirla se entiende en profundidad a los circuitos.
- Se puede adaptar para dar otra demostración del teorema de Cook-Levin.

Como se adelantó, la demostración es larga y es lo que más espacio ocupa del capítulo. El campo es grande, se eligió presentar los conceptos principales de manera profunda y omitir otros menos importantes pero muy interesantes también. Estos últimos son mencionados y se deja al lector sus respectivas referencias. Finalmente sin más preámbulos:

Teorema 55. $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

Demostración. Dada una MT polinomial y un número natural n, la demostración consiste en hacer un circuito de n nodos iniciales y tamaño polinomial en n que simule a la máquina cuando recibe una entrada de largo n. Esta demostración es larga y está divida en tres partes para que los lectores más experimentados en el área puedan convencerse de este teorema sin que haga falta seguir toda la prueba. La primera parte es un esquema de la demostración. Quienes

estén acostumbrados a trabajar con circuitos pueden ignorar las siguientes partes. La segunda parte de la prueba es la construcción del circuito en "mediano nivel" es decir, se describe el circuito y los circuitos auxiliares que lo implementan explicando lo que hacen y cómo lo hacen. En la parte tres y final se da una codificación en ceros y unos de los símbolos utilizados en la segunda parte, y se codifica de manera precisa los circuitos auxiliares que implementan a los circuitos de la parte dos.

(1) Parte uno: Esquema de la demostración

Sea $L \in \mathbf{P}$ luego existe M MT que lo decide que corre en tiempo p, para p un polinomio. Luego por la robustez de las Máquinas de Turing podemos exigirle sin pérdida de generalidad a M las siguientes propiedades:

- Decide a L.
- Es monocinta y su alfabeto es exactamente $\{0, 1, \square, \triangleright\}$. (Toda MT de alfabeto Γ se puede simular por una de alfabeto $\{0, 1, \square, \triangleright\}$ con costo polinomial en tiempo).
- Para todo $x \in \{0, 1\}^*$ se cumple que M finaliza en a lo sumo $p(|x|)$ pasos.

Luego para todo n natural y para toda entrada $x \in \{0, 1\}^n$, la computación de $M(x)$ es una sucesión finita de configuraciones (ver Definición 4), de a lo sumo $p(|x|) + 1$ configuraciones (a lo sumo la máquina realiza $p(|x|)$ pasos pero la configuración inicial que no cuenta como paso sí cuenta como configuración y por eso el $+1$). Podemos codificar tales configuraciones como palabras de un alfabeto S que definiremos a continuación, tales palabras las llamaremos *descripciones instantáneas.* La i-ésima descripción instantánea de una ejecución la denotaremos con d_i. Sea $m = p(|x|) + 1$, se puede convenir:

- El largo de cada descripción instantánea es $m + 2$, donde el primer y el último símbolo es el símbolo especial # que llamaremos borde (y que no está dentro del alfabeto de la máquina, solo sirve para simplificar la prueba).
- Sea Q el conjunto de estados de la máquina, definamos:

$CABEZAL = \{0^q, 1^q, \square^q : q \in Q\}$ y $S = CABEZAL \cup \{0, 1, \square, \triangleright\}$, es decir S es el conjunto de todos los símbolos que pueden ocurrir en una descripción instantánea excluyendo al símbolo del borde, #. Los elementos de $CABEZAL$ representan el símbolo que está leyendo la máquina, es decir, la posición del cabezal y el estado en el que está la máquina. Por ejemplo si $d_5[3] = 1^{q_6}$ significa que en el quinto paso de la ejecución, la máquina está leyendo un 1, se encuentra en el estado q_6 y el cabezal está en la segunda posición de la cinta.

- Dada como entrada $x = x_1 x_2 \dots x_n$ definimos que d_1 tiene la siguiente forma:

  $$\# \quad x_1^{q_i} \quad x_2 \quad \dots \quad x_n \quad \square \quad \square \quad \dots \quad \square \quad \#$$

  donde q_i es el estado inicial.

- Se tiene que la ejecución tiene m descripciones instantáneas (si la máquina finaliza en $k < m$ pasos entonces definimos $d_i = d_k$ para todo $i \in [k+1, m]$.
- Se tiene que d_m (la última descripción instantánea) es de la siguiente forma:

$$\# \quad l^{q_f} \quad \square \quad \square \quad \square \quad \square \quad \square \quad \ldots \quad \square \quad \#$$

  donde q_f es el estado final, y l es 1 o 0 dependiendo de si la máquina acepta o no a x.

El resto de la demostración es construir un circuito C_n de tamaño polinomial a n, que dada $x \in \{0,1\}^n$ de entrada construya las descripciones instantáneas mencionadas y devuelva l. Es decir, C_n simula a M en las entradas de largo n. Antes de dar una descripción de alto nivel de este circuito, es necesario hacer una observación crucial: *la computación es local*. Esto significa que $\forall i < m$ se tiene que d_i difiere de d_{i+1} en a lo sumo dos símbolos, y que estas diferencias se dan en los dos símbolos contiguos al cabezal. Por esta razón vamos a poder hacer un circuito polinomial que "simule" a la función de transición de M en entradas de largo n, es decir un circuito que dada una descripción como entrada devuelva la siguiente descripción. Más adelante se profundizará esta idea.

Todavía no tenemos una codificación en ceros y unos, se la tendrá en la parte tres. No obstante, como un acto de fe diremos que $\|\| : H \longrightarrow \{0,1\}^*$ será una codificación razonable en ceros y unos. El alfabeto H está formado por los símbolos necesarios para escribir descripciones instantáneas, que son símbolos de S, y de otros símbolos no definidos aun. Esto es para dejar en claro en los diagramas de los circuitos, que estos tienen de entrada y salida ceros y unos. Vale decir que se hará "abuso" de notación en los diagramas y en la definición de algunos circuitos para mayor simpleza, por ejemplo si $a \in H$ se verá tanto $\|a\|$ como $\|aaaa\|$. Pero en la codificación real, solo tiene sentido $\|a\|$ y por lo tanto no se verá jamás en la construcción real $\|aaaa\|$. La siguiente imagen es un diagrama en alto nivel de C_n.

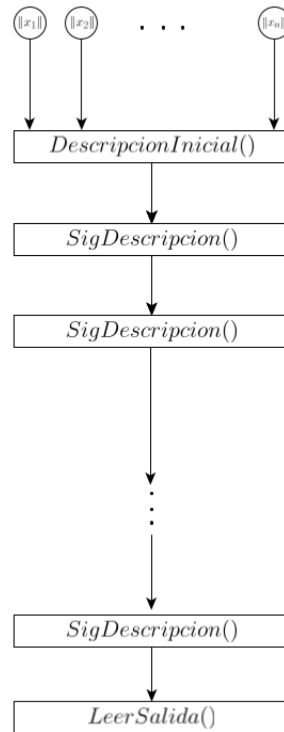

Figura 4.2:

$DescripcionInicial$ es un circuito que recibe como entrada a x y devuelve d_1, primera descripción instantánea. El circuito $SigDescripción$ recibe una descripción y devuelve (construyendo) la descripción sucesora, en el diagrama hay m de estos circuitos. El circuito $LeerSalida$ recibe la última descripción y devuelve el valor de l. Como veremos, el circuito $SigDescripción$ es el que realiza la tarea fundamental de simular la función de transición, y será el que nos demandará mayor trabajo. Para hacerlo, ahora sí, nos basaremos en la localidad de la computación antes mencionada.

Explicación de $SigDescripción$: dado que la computación es local, $\forall i < m$ se tiene que d_i difiere de d_{i+1} en a lo sumo dos símbolos, y que estas diferencias se dan en los dos símbolos contiguos al cabezal. Entonces si analizamos d_i de a *"ventanitas"* de tres símbolos contiguos, encontraremos necesariamente una que tendrá el cabezal en el medio de la ventanita. Por ejemplo, si una parte de la descripción es $\#\ 1\ 0\ 1^{q_2}\ \square$ entonces, una ventanita va a ser $\#\ 1\ 0$, otra $1\ 0\ 1^{q_2}$, otra $0\ 1^{q_2}\ \square$ y así. Llamemos como *ventanita_cabezal* a la ventanita que tiene el cabezal en el medio. Dada un ventanita de una descripción instantánea, existe una y solo una ventanita que la reemplazará en la descripción sucesora. Definiremos una función δ_v que dada una ventana devuelve la ventana que corresponde según la δ de la máquina. Implementaremos

dicha función con otro circuito auxiliar llamado $TransformarVentana$, que dada como entrada una ventanita, si esta es una ventanita_cabezal devolverá la ventanita correspondiente según δ, caso contrario, devolverá la ventanita que recibió de entrada. La última observación crucial es que la cantidad posible de ventanitas que ocurren en una computación es una constante que no depende de x. Esta constante depende solo de M, y en nuestra prueba, del cardinal de S. Por lo tanto, el circuito $TransformarVentana$ va a tener un tamaño constante, que no depende de x. Como la cantidad de ventanitas que hay en una descripción instantánea es a lo sumo $m-2$ se tiene que $SigDescripción$ tiene un tamaño polinomial a m. ¡Como queríamos! Un detalle a refinar en las siguientes partes es que algunas ventanitas van a tener que sobrescribir a otras al armar la descripción en cuestión. La idea es darle precedencia a la δ_v(ventanita_cabezal) para que pueda sobrescribir y para que no pueda ser sobrescrita, y que todas las demás si puedan ser sobrescritas y que no puedan sobrescribir. La idea es la de una cascada: se obtiene la ventanita1, luego cae la ventanita2 sobre ella corrida un símbolo a la derecha, sobrescribiéndola (si es que puede), luego la ventanita3 cae sobre la ventanita2 repitiendo el mismo procedimiento, y así.

(2) Parte dos: Construcción de los circuitos Ahora se describirán los circuitos en mayor detalle que en la parte anterior, pero todavía sin construir los circuitos de manera explícita, ya que aún no se cuenta con una codificación en ceros y unos de S. Concentraremos nuestros esfuerzos en $SigDescripción$. Definamos el conjunto de ventanitas $V = \{abc : a, b, c \in S \cup \{\#\}\}$ y su subconjunto de ventanitas cabezal

$$VC = \{abc : a, c \in \{0, 1, \square, \#\} \text{ y } b \in CABEZAL\}$$

Naturalmente, habrá ventanitas que no ocurrirán en ninguna descripción instantánea, por ejemplo # # # . Por lo que se explicó en la parte anterior, el cardinal de V es una constante que no depende de x, y que solo depende de M, es por ello que podremos hacer $SigDescripción$ de tamaño polinomial a n. Ahora definiremos la precedencia de la sobrescritura. Para ello, definiremos el siguiente conjunto de ventanitas con precedencia, la idea es que una ventanita que tiene precedencia no puede ser sobrescrita y sí puede sobrescribir. La codificación de esta precedencia se verá en detalle en la parte tres pero adelantando, la idea es que el primer bit de la codificación represente tal precedencia es decir, si ese bit es igual a 1 ese símbolo codificado tiene precedencia, caso contrario no. Utilizaremos el símbolo especial $\bigstar$ para denotar la precedencia de un símbolo sobre otro. Definamos

$$S^\bigstar = \{a^\bigstar : a \in S\}$$

$$V^\bigstar = \{a^\bigstar b^\bigstar c^\bigstar : abc \in V\}$$

$$VC^\bigstar = \{a^\bigstar b^\bigstar c^\bigstar : abc \in VC\}$$

Ahora sí, definamos la función δ_v que dada una ventana devuelva la ventana correspondiente según la δ, con la precedencia. Sea $\delta_V : V \longrightarrow V \cup V^\bigstar$ dada

por

$$\delta_v(a_1a_2a_3) = \begin{cases} a_1a_2a_3 & \text{si } a_1a_2a_3 \notin VC \\ b_1^\star b_2^\star b_3^\star & \text{c.c} \end{cases}$$

donde $b_1b_2b_3$ es la ventana que se obtiene de "aplicar" δ a $a_1a_2a_3$. Por ejemplo, si tenemos que $\delta(1, q_2) = (0, R, q_1)$ entonces $\delta_v(\ \#\quad 1^{q_2}\quad 1\ ) = \ \#^\star\quad 0^\star\quad 1^{q_1^\star}$, $\delta_v(\ 0\quad 1^{q_2}\quad \square\ ) = \ 0^\star\quad 0^\star\quad \square^{q_1^\star}$, etc. Recordemos que δ es total y por ello la definición anterior tiene sentido.

En el resto de esta parte, describiremos un circuito $TransformarVentana$ que implementa a δ_v, y describiremos la implementación de *SigDescripción* a partir de muchos circuitos $TransformarVentana$.

Notar que el número $k = |V|$ (i.e., la cantidad de ventanitas) sólo depende de M, y por lo tanto es constante. Para cada $z \in V$ haremos el circuito C_z, que recibe de input una ventana y realiza lo siguiente:

$$C_z(\|a_1\|\,\|a_2\|\,\|a_3\|) = \begin{cases} \|\delta_v(z)\| & \text{si } a_1a_2a_3 = z \\ \|0\|\,\|0\|\,\|0\| & \text{c.c.} \end{cases}$$

En la parte tres se hará en detalle este circuito, cuando hagamos explícita la codificación $\|\|$. Se define un circuito $VentanaNoNula$ que recibe como input ventanas $w_1, w_2, \ldots, w_k$ y devuelve la ventana que no es 000 (de haber una así), o w_1 en caso contrario. Este circuito tendrá la precondición de que todas las ventanas que recibe de input son 000, o que todas ellas son 000 salvo una. El cumplimiento de la precondición se explicará a la brevedad. Describamos $TransformarVentana$:

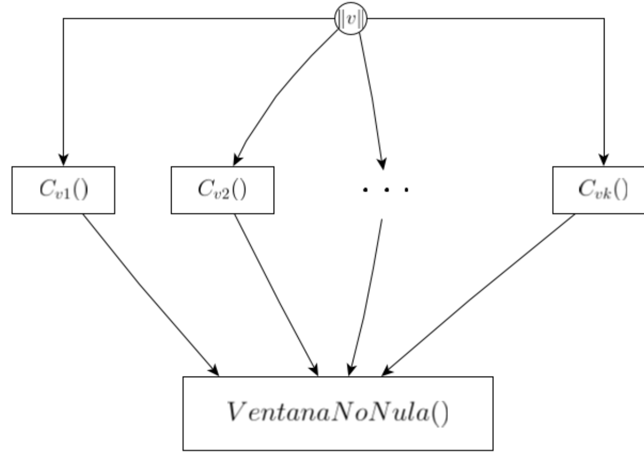

Figura 4.3:

Como $v \in V$ se tiene que existe un único i tal que $v_i = v$ entonces $\forall j \neq i$ se tiene $C_{v_j}(v)$ devuelve 000 en consecuencia se cumple la mencionada precondi-

ción de $VentanaNoNula$. Por ende este circuito devuelve $C_v(v)$ que es lo que queríamos! Por otro lado, como hay una cantidad constate de circuitos auxiliares, cada uno de tamaño constante, se tiene que $TransformarVentana$ tiene un tamaño constante que no depende de x y sí de la máquina ¡Como queríamos!

Vale agregar que no se explicó cómo implementar $VentanaNoNula$ la idea es que en la codificación que se dará en la parte tres, 000 se codifica como una palabra de ceros, luego el circuito simplemente hace un gran "or lógico" de cada símbolo de cada ventanita que recibe de input. Osea devuelve intuitivamente la palabra $Or(w_{1_1}, \ldots, w_{k_1})Or(w_{1_2}, \ldots, w_{k_2})Or(w_{1_3}, \ldots, w_{k_3})$ donde w_{i_j} es el $j-$ésimo símbolo de la $i-$ésima entrada..

A continuación describiremos los últimos circuitos auxiliares que necesitaremos en *SigDescripción*. Describamos al circuito Or_2, que recibe de input $a_1, a_2 \in S \cup S^\bigstar$:

$$Or_2(a_1, a_2) = \begin{cases} a_2 & \text{si } a_2 \in S^\bigstar \\ a_1 & \text{c.c} \end{cases}$$

Y al circuito Or_3, que recibe de input $a_1, a_2, a_3 \in S \cup S^\bigstar$:

$$Or_3(a_1, a_2, a_3) = Or_2(Or_2(a_1, a_2), a_3)$$

Lo que hay que tener en mente es que el significado que tienen es que son los encargados de sobrescribir los símbolos según la precedencia. La idea es que se parte de que a_1 es el símbolo escrito en la descripción, cae a_2, si tiene precedencia se escribe a_2 en el lugar de a_1 sino lo deja a a_1. Luego cae a_3, si tiene precedencia sobrescribe y sino deja el símbolo que está. La precondición de estos circuitos es que a lo sumo un solo a_i pertenece a $S^\bigstar$. Por ende los $a_i \in S^\bigstar$ son los únicos que pueden sobrescribir y nunca pueden ser sobrescritos. Este circuito se describirá en detalle en la tercera parte. Resta decir que el tamaño de ellos es una constante que no depende de x y que sí, de la máquina. Por último definamos un circuito $SacarPrecedenciaDescripcion$ que recibe de input una descripción que tiene una ventanita con símbolos de precedencia, y retorna esa misma descripción sacándole la precedencia a esa ventanita. El circuito simplemente pone en 0 al primer bit de cada codificación de cada símbolo de la descripción que recibe de input, evidentemente tiene un tamaño polinomial a n. Ahora sí, finalmente describamos a *SigDescripción*:

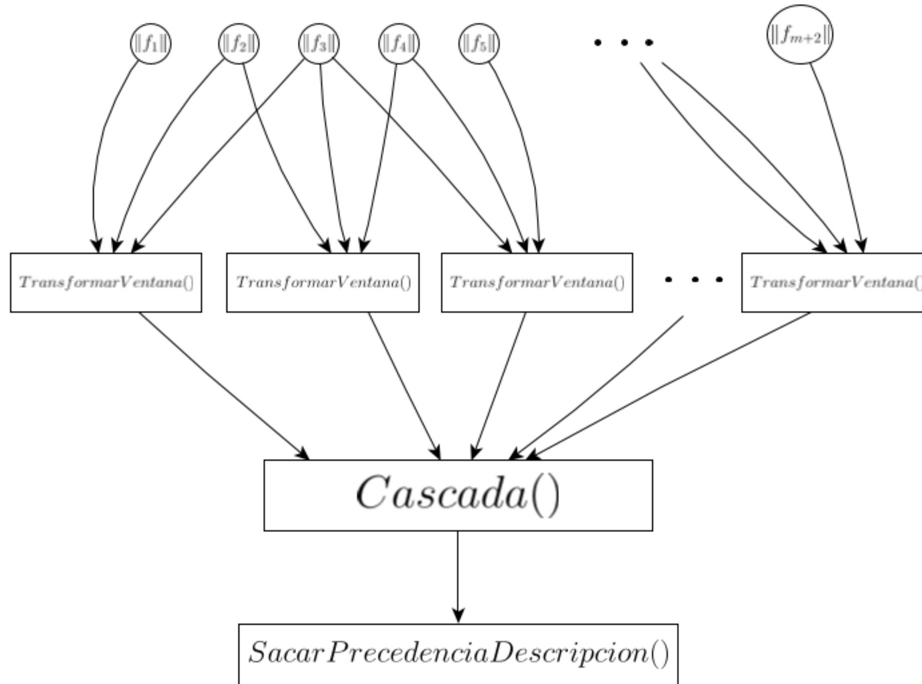

Figura 4.4:

Donde $f_1 f_2 f_3 \ldots f_{m+2}$ es una descripción, es decir $f_1, f_{m+2} = \#$ y el resto de los $f_i \in S$. El circuito envía cada ventanita de la descripción, $f_i f_{i+1} f_{i+2}$ al circuito $TransformarVentana$, luego cada uno de estos envía la ventanita transformada al circuito $Cascada$ encargado de armar la siguiente descripción. La manera más didáctica de describir este circuito es con un ejemplo simple, si $a_1a_2a_3$, $b_1b_2b_3$, $c_1c_2c_3$ y $d_1d_2d_3$ son las ventanas dadas como entradas a $Cascada$ en ese orden, entonces realiza lo siguiente:

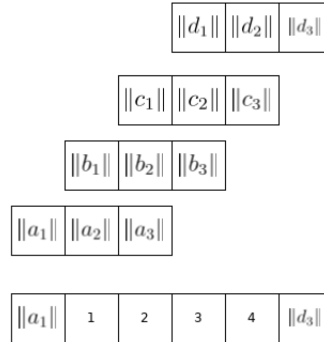

Figura 4.5:

Donde 1 es $Or_2(\|a_2\|, \|b_1\|)$, 2 es $Or_3(\|a_3\|, \|b_2\|, \|c_1\|)$, 3 es $Or_3(\|b_3\|, \|c_2\|, \|d_1\|)$ y 4 es $Or_2(\|c_3\|, \|d_2\|)$. Más en detalle, *Cascada* tendría la forma:

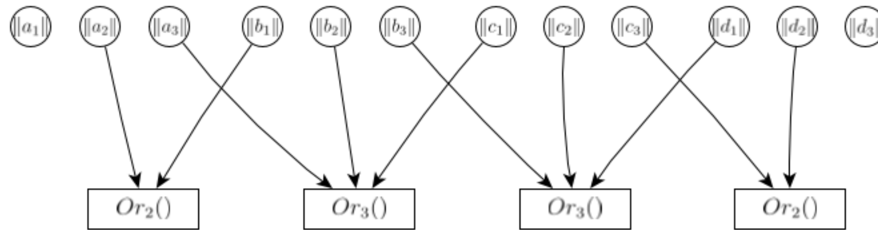

Figura 4.6:

Se observa que por ejemplo a_1, d_3 quedan inalterados es decir, son nodos iniciales y finales, y está bien ya que son los bordes de la descripción! Por otro lado, se ve que *Cascada* devuelve menos ventanas de las que recibe de entrada, y está bien, recordemos que se recibieron m ventanas. Luego evidentemente no se las puede devolver a todas ya que sino se devolvería una palabra de $3m$ símbolos cuando sabemos que la descripción recibida tiene $m+2$ símbolos. Por esa razón es que se deben sobrescribir símbolos utilizando los circuitos Or_2 y Or_3.

Por todo lo explicado anteriormente, cada circuito auxiliar utilizado, Or_2 y Or_3, tiene un tamaño constante que no depende de x y la cantidad de estos circuitos es menor a $m + 2$ donde m es polinomial a n. Por lo tanto el tamaño de *Cascada* es polinomial a m y por ende a n. Además como *SacarPrecedenciaDescripcion* también tiene un tamaño polinomial a m se tiene que *SigDescripción* tiene un tamaño polinomial a n.

Por otro lado, existe una sola ventanita_cabezal que es la que tendrá precedencia, sea $f_z f_{z+1} f_{z+2}$ tal ventanita. Luego si $h = h_1 h_2 \dots h_{m+2}$ es lo que

devuelve $Cascada$ entonces $h_z h_{z+1} h_{z+2}$ es la única ventanita que tiene símbolos de precedencia y que ocasiona sobrescrituras. Por ende para el resto de los i se tiene $h_i = f_i$ esto significa que en la descripción que se obtiene, h, los símbolos que no estaban cerca del cabezal quedaron igual a los símbolos de f y que solo cambiaron los símbolos de la ventana cabezal, como queríamos! Lo único malo es que los símbolos en $h_z h_{z+1} h_{z+2}$ tienen precedencia, es decir que pertenecen a $S^\bigstar$ pero como h es dada de entrada a $SacarPrecedenciaDescripcion$ se elimina tal precedencia y por ende se devuelve la siguiente descripción, ¡Como queríamos!

(3) Parte tres: Codificación.

Hasta ahora hemos descripto circuitos en alto o mediano nivel. No podíamos darlos en mayor detalle porque no teníamos la codificación de los distintos símbolos que aparecen en las descripciones. Ahora se dará una manera de codificar cada uno de estos símbolos usados anteriormente como palabras de ceros y unos, y también la construcción explícita de los circuitos más importantes como los C_z y Or_2 ya son como los ladrillos usados para construir al gran edificio: $SigDescripción$.

Sabemos que con n bits podemos codificar alfabetos de hasta 2^n símbolos. Como se mencionó, se agrega un bit adicional para codificar el símbolo especial $\bigstar$. Entonces utilizaremos $1 + \lceil \log(|S|) \rceil$ bits donde $\lceil\rceil$ es la función techo, sea t ese número.

Denotamos con $\|\| : S \cup S^\bigstar \longrightarrow \{0,1\}^t$ a la codificación. Definimos $\|0\| = 0^t$, $\|1\| = 0^{t-1}1$. Y para todo $s \in S$ el primer símbolo de $\|s\|$ es 0. El código $\|s^\bigstar\|$ es igual a $\|s\|$ solo que su primer símbolo es 1. Por ejemplo

$\|0^\bigstar\| = 10\ldots0$ y $\|1^\bigstar\| = 10\ldots1$. Las codificaciones de los restantes símbolos pueden ser cualesquiera que quiera el lector mientras mantengan lo antes mencionado. Dos codificaciones distintas necesariamente tendrán circuitos distintos pero estas últimas distinciones serán detalles despreciables.

Vale comentar que en las codificaciones estándar de las máquinas de Turing en ceros y unos, se suele utilizar la técnica de duplicado de bits. En nuestra codificación no hace falta, el trabajo de "reconstruir" los símbolos de S a partir de ceros y unos no existe, en los circuitos ya todo viene "hardcodeado" y nos ahorran ese trabajo. Más adelante quedará claro.

Es importante notar también que r es una constante que solo depende de M y no de x, y por ende tampoco de su tamaño. Veamos un ejemplo de una codificación:

Sea $Q = \{q_i, q_f, q_1\}$ entonces $S = \{0^{q_i}, 0^{q_f}, 0^{q_1}, 1^{q_i}, \ldots, \square^{q_1}, 0, 1, \rhd, \square, \#\}$, $|S| = 12 + 5 = 17$ y por ende $t = 1 + \lceil \log(|S|) \rceil = 1 + 5 = 6$. Luego una codificación válida sería por ejemplo:

$\|0\| = 000000$
$\|1\| = 000001$
$\|\square\| = 000010$
$\|\#\| = 000011$
$\|\rhd\| = 011111$
$\|0^{q_i}\| = 000100$

$\|0^{q_f}\| = 000101$
$\|0^{q_1}\| = 000110$
$\|1^{q_i}\| = 000111$
$\|1^{q_f}\| = 001000$
$\|1^{q_1}\| = 001001$
$\|\square^{q_i}\| = 001010$
$\|\square^{q_f}\| = 001011$
$\|\square^{q_1}\| = 001100$
$\|\triangleright^{q_i}\| = 001101$
$\|\triangleright^{q_f}\| = 001111$
$\|\triangleright^{q_1}\| = 010000$
y
$\left\|0^{\bigstar}\right\| = 100000$
$\left\|1^{\bigstar}\right\| = 100001$
$\left\|\square^{\bigstar}\right\| = 100010$
$\left\|\#^{\bigstar}\right\| = 100011$
$\left\|\triangleright^{\bigstar}\right\| = 111111$
$\left\|0^{q_i^{\bigstar}}\right\| = 100100$
$\vdots$
$\left\|\triangleright^{q_f^{\bigstar}}\right\| = 101111$
$\left\|\triangleright^{q_1^{\bigstar}}\right\| = 110000$

Entonces por ejemplo si $x = 0110$ entonces la descripción instantánea inicial de $M(x)$, d_1 es:

$$\# \quad \triangleright^{q_i} \quad 0 \quad 1 \quad 1 \quad 0 \quad \square \quad \square \quad \ldots \quad \square \quad \square \quad \#$$

que es nuestra codificación sería:

000011 001101 000000 000001 000001 000000 000010 000010 ... 000010 000010 000011

Ya dada la codificación, ahora estamos en condiciones de comenzar a construir los circuitos. Vale adelantar que algunos de estos circuitos se definirán para $t \in \mathbb{N}$ pero no hay que confundirse, t no depende de x ya que es el largo de la codificación, la cual depende de S. Lo que se quiere expresar es que dado un lenguaje en P hay una máquina que lo decide y hay un t para codificar la familia de circuitos booleanos que lo decide, construida a partir de tal máquina. Por consiguiente, dos lenguajes en P distintos pueden tener t distintos y por eso es que algunos de estos circuitos se definen para t en general.

1) Dados $i, j \in \{0, 1\}$ definimos:

$$BitsIguales(i,j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{c.c} \end{cases}$$

Hagamos la tabla de verdad:

$$\begin{bmatrix} i & j & BitsIguales(i,j) & \text{minterminos} \\ 0 & 0 & 1 & \neg i \wedge \neg j \\ 0 & 1 & 0 & \\ 1 & 0 & 0 & \\ 1 & 1 & 1 & i \wedge j \end{bmatrix}$$

Como sabemos, la función equivalente al circuito es el $\vee$ de los mintérminos. Entonces el circuito debe tener la semántica de $(\neg i \wedge \neg j) \vee (i \wedge j)$. Que quedaría como:

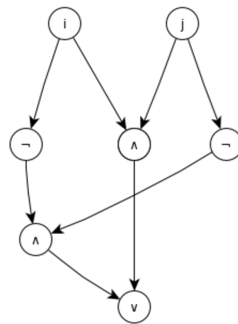

Figura 4.7:

2) Para cada $t \in \mathbb{N}$ dados $\alpha, \beta \in \{0,1\}^t$ definimos:

$$StringsIguales(\alpha, \beta) = \begin{cases} 1 & \text{si } \alpha = \beta \\ 0 & \text{c.c} \end{cases}$$

Es fácil ver que si $\alpha = a_1 a_2 \ldots a_t$ y $\beta = b_1 b_2 \ldots b_t$ entonces

$$StringsIguales(\alpha, \beta) = BitsIguales(a_1, b_1) \wedge BitsIguales(a_2, b_2) \wedge \ldots \wedge BitsIguales(a_t, b_t)$$

Como ejemplo se deja el caso $t = 3$. Para otros t la construcción de $StringsIguales(\alpha, \beta)$ es análoga.

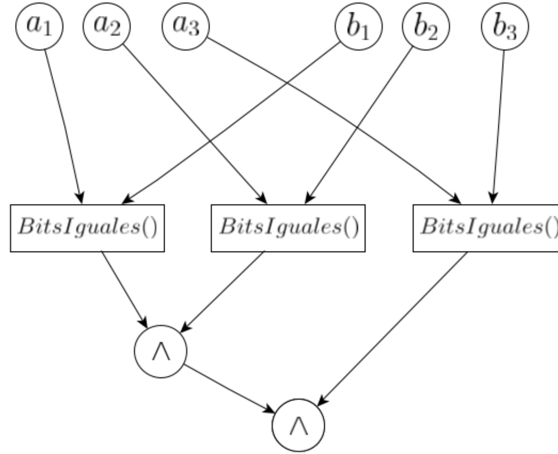

Figura 4.8:

3) Para cada $t \in \mathbb{N}$ dados $\alpha, \beta, \gamma \in \{0,1\}^t$ definimos:

$$If_{\alpha,\beta}(\gamma) = \begin{cases} \beta & \text{si } StringsIguales(\alpha, \gamma) = 1 \\ 00\ldots0 & \text{c.c} \end{cases}$$

donde $00\ldots0$ tiene t bits también. El objetivo de estos circuitos es el de implementar los circuitos C_z utilizados en $TransformarVentana$. Vale recordar que la cantidad de C_z en $TransformarVentana$ es una constante que no depende de x y sí de M, por ende la cantidad de $If_{\alpha,\beta}$ utilizados es una constante que no depende de x y sí de M. En particular dada una ventana $w \in V$, C_w se implementa con $If_{\|w\|,\|\delta_v(w)\|}$. Veamos cómo construirlo para el caso $t = 3$ y para dos palabras particulares de largo t. Para el resto de los t y para el resto de las palabras de largo t utilizadas, la construcción es análoga. Veamos la construcción de por ejemplo $If_{001,110}(a_1a_2a_3)$ con $a_i \in \{0,1\}$.

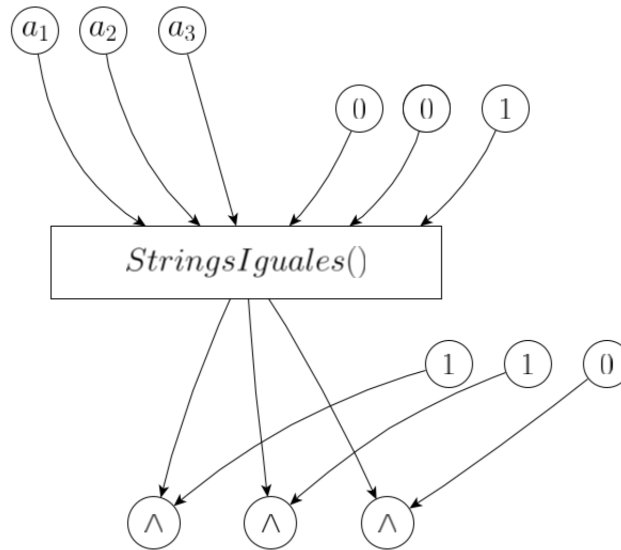

Figura 4.9:

Vale agregar que se observan nodos de etiqueta 1 y 0, ello no supone ningún problema ya que es trivial hacer un nodo que tenga esos valores a partir de los nodos iniciales agregando una cantidad constante de nodos.

4) Para cada $t \in \mathbb{N}$ dados $\alpha, \beta \in \{0,1\}^t$ definimos:

$$Or_2(\alpha, \beta) = \begin{cases} \beta & \text{si } b_1 = 1 \\ \alpha & \text{c.c} \end{cases}$$

con $\alpha = a_1 a_2 \dots a_t$ y $\beta = b_1 b_2 \dots b_t$. Construiremos el caso $t = 3$, para el resto de los t la construcción es análoga. En nuestra construcción haremos:

$$Or_2(a_1 a_2 a_3, b_1 b_2 b_3) = \begin{cases} b_1 b_2 b_3 & \text{si } b_1 = 1 \\ a_1 a_2 a_3 & \text{c.c} \end{cases}$$

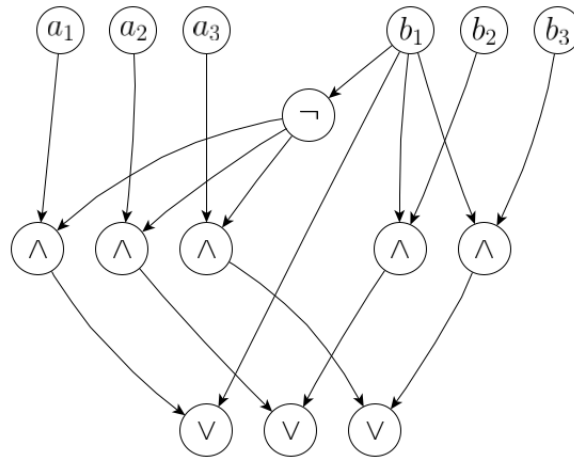

Figura 4.10:

Con esto termina la parte tres y la prueba. Resta decir que algunos circuitos no fueron descritos ni construidos como por ejemplo *DescripcionInicial*, luego de haber seguido toda la construcción de *SigDescripción* todos esos circuitos deberían parecerle triviales al lector. □

Como se mencionó, una adaptación de esta prueba se puede utilizar para demostrar el teorema de Cook-Levin "traducido" a circuitos. En lugar de utilizar `SAT` se emplea `CSAT` para probar que hay un lenguaje NP-completo utilizando la localidad de la computación. Este lenguaje `CSAT` se define como el conjunto de circuitos de un solo nodo final para los cuales existe una entrada que los hace retornar 1.

Teorema 56. *Sea L un lenguaje unario (i.e. $L \subseteq \{1\}^*$), entonces $L \in \mathbf{P}_{/\mathbf{poly}}$.*

Demostración. Sea $L \subseteq \{1\}^*$ luego $L \subseteq \{1^n : n \in \mathbb{N}\}$. La sucesión de circuitos booleanos, $\{C_n\}_{n\in\mathbb{N}}$, definida a continuación claramente cumple que decide a L y que tiene tamaño polinomial en n.

Dado $n \in \mathbb{N}$ definimos

$$C_n(1^n) = \begin{cases} 1 & \text{si } 1^n \in L \\ 0 & \text{c.c} \end{cases}$$

□

Observación. Hay lenguajes indecidibles que están en $\mathbf{P}_{/\mathbf{poly}}$. Por el teorema anterior se ve que por ejemplo el lenguaje indecidible `UHALT` pertenece a $\mathbf{P}_{/\mathbf{poly}}$. Donde `UHALT` es el conjunto de 1^n tal que la codificación en binario de $n \in \mathbb{N}$ es la codificación de una tupla (M, x) con M una MT que finaliza al recibir x de entrada. Dónde está el error puede preguntarse el lector, en ningún lugar, la

observación clave es que los circuitos que deciden lenguajes como el anterior no pueden ser construidos por MT's.

Por otro lado, incluso se puede probar que dada una función $f : \{0,1\}^l \longrightarrow \{0,1\}$ siendo l algún natural, existe un circuito que la computa de tamaño $10l2^l$. Por lo tanto, para todo lenguaje $L \subseteq \{0,1\}^*$ existe una sucesión de circuitos $\{C_n\}$ de tamaño $\mathcal{O}(n2^n)$ que lo decide.

P-uniform y codificación de circuitos

Vimos que $\mathbf{P_{/poly}}$ tiene lenguajes no decidibles y circuitos "no construibles". Es natural entonces modificar nuestros circuitos para que puedan ser construidos y para que puedan decidir solo lenguajes decidibles en el sentido de las MT's. Para ello nos concentraremos en circuitos que puedan ser construidos eficientemente por MT's y para ello necesitaremos una codificación eficiente de los mismos. En particular, necesitaremos una codificación que cumpla que para todo $n \geq 10$, todo circuito de n nodos puede ser codificado en a lo sumo $9n\log(n)$ bits. En el resto del capítulo la codificación a continuación presentada será la utilizada, y por ende cuando se mencione una codificación de un circuito se pensará en esta codificación. La razón es que distintos resultados sobre $\mathbf{P_{/poly}}$ como teoremas de jerarquía y otros, utilizarán en sus demostraciones esta codificación eficiente.

Hay varias maneras de codificar a los circuitos de manera eficiente, nosotros lo haremos por medio de una lista de adyacencia. La codificación que presentaremos es un poco difícil y tal vez haya otras más elegantes, pero en la bibliografía no encontramos ninguna tan precisa y explícita como la nuestra. La idea clave para codificar el circuito tan eficientemente es notar que todo nodo a lo sumo recibe dos aristas (el grado de entrada es menor o igual a dos). Entonces a priori podemos representar el circuito como una lista de triplas (e_1u, e_2v, e_3w) donde (v,u) y (w,u) son aristas, y donde e_1,e_2 y e_3 son las etiquetas de los nodos (teniendo una etiqueta especial para los nodos iniciales) y teniendo un valor especial z (que no es un nodo del circuito) para representar la ausencia de una arista, es decir de ocurrir (e_1u, e_2v, e_3z) significa que solo (v,u) es una arista. Pero a posteriori tal representación no nos alcanza para representar eficientemente el orden de los nodos iniciales y el orden de los nodos finales. Uno ingenuamente podría simplemente definir que el orden en el que ocurren los nodos iniciales en tal lista (de izquierda a derecha) es el orden de los nodos iniciales en el circuito y recíprocamente con los nodos finales, pero ello tiene el problema de que puede haber nodos iniciales que también son nodos finales. La solución será representar al circuito como una lista de 4-uplas (e_1u, e_2v, e_3w, i) donde en el caso de que u sea un nodo final entonces i representará el orden del nodo en el orden de los nodos finales. El orden de los nodos iniciales será el orden en el que ocurran en la lista (leyéndose de izquierda a derecha).

Sea C un circuito de $N \in \mathbb{N}$ nodos. Representamos tales nodos por los números naturales $1, 2, \ldots, N$, y los codificamos con la codificación en binario de t bits con $t = \log(N)$. La palabra para representar la ausencia de arista, es decir z, es la palabra 0^t. El cuarto elemento de la 4-upla es decir i, es la codificación

en binario de t bits de un número natural menor o igual a N incluyendo a 0. Todo nodo que no sea final tiene a $i = 0^t$. Las etiquetas son palabras de dos bits definidas como: $00 =$ nodo inicial, $01 = \wedge$, $10 = \wedge$, $11 = \neg$. La codificación del circuito comienza (izquierda) con una palabra de la forma 1^t0, y a continuación ocurren la lista de las 4-uplas. Por ende la cantidad de 1's hasta llegar al primer 0 es el valor de t.

Por ejemplo codifiquemos que $(5, 3)$ y $(4, 3)$ son aristas de un circuito de $t = 4$, con 5 un nodo de etiqueta $\neg$, 4 un nodo inicial y 3 un nodo de etiqueta $\wedge$ que no es un nodo final. La codificación en binario de 3 en cuatro bits es 0011, la de 4 es 0100 y la de 5 es 0101, como 3 no es nodo final $i = 0^4$ luego la 4-upla tiene la siguiente forma:

$$\wedge 0011 \neg 0101 \text{nodoinicial} 0100 0^4$$

por lo tanto tras codificar las etiquetas, la codificación resulta en:

$$10001111010100010000000$$

Por si no quedo clara la codificación de la ausencia de alguna arista mostraremos el siguiente ejemplo. Supongamos que $(1, 2)$ es una arista de un circuito de $t = 4$, con 1 un nodo inicial y 2 un nodo de etiqueta $\neg$ que no es nodo final (por lo tanto $i = 0^t$). Luego, la 4-upla resulta en (obviando codificar las etiquetas así se le hace más legible al lector):

$$\neg 0010 \text{nodoinicial} 0001 \text{nodoinicial} 0000 0^t$$

Acá vemos que hay un nodo 0 de etiqueta nodo inicial tal que $(0, 2)$ es arista pero como asumimos que los nodos van de 1 a N luego 0 no es un nodo y por ende $(0, 2)$ no es una arista. No elegimos etiquetar al nodo 0 con etiqueta nodo inicial por alguna razón particular, cualquier etiqueta sería igualmente válida porque 0 no es un nodo del circuito.

Ya dada la codificación resta chequear el tamaño de la misma. Hay N 4-uplas de la forma (e_1u, e_2v, e_3w, i), cada una de ellas ocupa $3(2 + t) + t$ bits, por lo tanto la codificación de la lista de 4-uplas tiene un tamaño de $N(3(2 + t) + t)$ bits. Recordando, la codificación del circuito comienza con la palabra 1^t0 por lo tanto el tamaño de la codificación del circuito es de $t + 1 + N(3(2 + t) + t)$ bits o equivalentemente $\log(N) + 1 + N(3(2 + \log(N)) + \log(N))$ bits. Finalmente es sencillo ver que para $N \geq 4$ (y por ende para $N \geq 10$):

$$\log(N) + 1 + N(3(2 + \log(N)) + \log(N)) \leq 9N\log(N).$$

¡Como queríamos!

Definición 57. Decimos que una sucesión de circuitos $\{C_n\}_{n\in\mathbb{N}}$ es **P**-*uniform* si existe una MT polinomial que para todo n natural, dada como entrada 1^n devuelve la codificación del circuito C_n.

Teorema 58. *Un lenguaje $L \subseteq \{0, 1\}^*$ es decidido por una sucesión de circuitos* **P**-*uniform si y solo si $L \in$* **P**.

Demostración. Sea L un lenguaje decidido por una sucesión de circuitos **P**-uniform, $\{C_n\}_{n\in\mathbb{N}}$, y sea M la MT poly de la definición anterior. Hagamos la siguiente MT M_2, dado un $x \in \{0,1\}^*$ crea el circuito $C := M(1^{|x|})$ y luego computa y devuelve $C(x)$. Por la definición de C éste tiene tamaño polinomial a x, por lo tanto se le puede pedir a M_2 que ejecute C a partir de la entrada x en tiempo polinomial y como C decide la pertenencia de x, se tiene que M_2 decide a L y por ende $L \in \mathbf{P}$. La otra implicación es prácticamente la demostración del teorema $\mathbf{P} \subseteq \mathbf{P_{/poly}}$. □

Por el teorema anterior se ve que limitando los circuitos a ser eficientemente construibles se llega a **P** ¡No ganamos nada! Por otro lado $\mathbf{P_{/poly}}$ contiene lenguajes indecidibles! ¿Es este poder asombroso suficiente para contener por ejemplo a **NP**? Si la respuesta es afirmativa entonces **PH** colapsa (ver Teorema 40).

Teorema 59. *Si* $\mathbf{NP} \subseteq \mathbf{P_{/poly}}$ *entonces* $\mathbf{PH} = \Sigma_2^{\mathbf{p}}$.

Demostración. Recordemos por el capítulo anterior que para demostrar $\mathbf{PH} = \Sigma_2^{\mathbf{p}}$ alcanza con demostrar que $\mathbf{\Pi_2^p} \subseteq \Sigma_2^{\mathbf{p}}$ y en particular es suficiente con mostrar que $\Pi_2 - \mathtt{SAT} \in \Sigma_2^{\mathbf{p}}$ (ver Definición 42). Recordando, $\varphi \in \Pi_2 - \mathtt{SAT}$ sii $\forall u \in \{0,1\}^{p(|\varphi|)} \exists v \in \{0,1\}^{p(|\varphi|)} \varphi(u,v) = 1$ con p un polinomio adecuado para que uv sea una asignación para φ, donde el i-ésimo bit de uv es el valor que se le asigna a la i-ésima variable de φ.

Primero definamos una nueva variante de SAT, SATCons$t$ compuesto por las fórmulas booleanas con constantes que son satisfacibles. Estas fórmulas son una generalización de las fórmulas booleanas en las que pueden ocurrir 0,1 en reemplazo sintáctico de variables. Por ejemplo $(x_1 \vee 0 \wedge \neg 1) \vee (\neg x_1 \vee x_2) \vee 0$, 1, $x_2 \vee x_4$ son fórmulas booleanas con constantes. En general si x_i es una variable de φ, fórmula booleana, entonces $\varphi[x_i : 0], \varphi[x_i : 1]$ son fórmulas booleanas con constantes, donde $\varphi[x_i : 0]$ denota al reemplazo sintáctico de cada ocurrencia de x_i por 0, respectivamente con $\varphi[x_i : 1]$. Naturalmente, SATConst es NP-completo. Observemos también que es posible dar una codificación en ceros y unos de las fórmulas booleanas con constantes, $\|\|$, que cumpla que si x_i es una variable de φ, fórmula booleana con constantes, entonces el tamaño de $\|\varphi\|$ es igual al tamaño de $\|\varphi[x_i : 0]\|$ y de $\|\varphi[x_i : 1]\|$. El truco es simplemente hacer que el tamaño de la codificación de 1 sea igual al tamaño de la codificación de 0 y éste, igual al tamaño de la codificación de las variables que ocurren en la fórmula.

Supongamos que $\mathbf{NP} \subseteq \mathbf{P_{/poly}}$ entonces existe una sucesión de circuitos booleanos de tamaño polinomial, $\{C_n\}_{n\in\mathbb{N}}$ que decide a SATConst. Queremos ver que $\Pi_2 - \mathtt{SAT} \in \Sigma_2^{\mathbf{p}}$ y para ello que existe una MT M polinomial y un polinomio q que cumple que para toda φ, fórmula booleana se tiene:

$$\forall u \in \{0,1\}^{p(|\|\varphi\||)} \exists v \in \{0,1\}^{p(|\|\varphi\||)} \varphi(u,v) = 1 \text{ sii}$$

$$\exists D \in \{0,1\}^{q(|\|\varphi\||)} \forall z \in \{0,1\}^{q(|\|\varphi\||)} M(\|\varphi\|, D, z) = 1$$

con p el polinomio adecuado para φ, anteriormente explicado. Vimos que nuestra codificación de un circuito C_n tiene un tamaño $9n\log(n)$ que es menor a $10n^2$.

Definamos $q(n) = p(n) + 10n^2$ y describamos a M, la máquina recibe de entrada $\|\varphi\|, D, z \in \{0,1\}^*$ y realiza lo siguiente:

// elimina los bits "basura" de D
// donde ϵ es el string vacío
$C := \epsilon$
// sea $n = |\,\|\varphi\|\,|$
for $i := 1$ **to** $9n\log(n)$:
    // concatena
    $C := C(i$-ésimo bit de $D)$
// luego chequea si C es un circuito adecuado, es decir, si C es un circuito según nuestra codificación de n nodos iniciales y de $p(|\,\|\varphi\|\,|)$ nodos finales. Si no es adecuado M finaliza devolviendo 0.
// elimina los bits "basura" de z
$u := \epsilon$
for $i = 1$ **to** $p(|\,\|\varphi\|\,|)$:
    // concatena
    $u := u(i$-ésimo bit de $z)$
// hace un remplazo sintáctico, cada ocurrencia de la i-ésima variable de φ es reemplazada por el i-ésimo bit de u
$\varphi := \varphi[u]$
// se crea un string vació v que será el certificado a construir
$v := \epsilon$
// sea m la cantidad de variables que tiene φ sin asignar
for $i = 1$ **to** m :
    if $C(\|\varphi[v0]\|) = 1$:
        $v := v0$
    else:
        $v := v1$
// si φ es satisfacible y C justo es el circuito que decide a $SATConst$ entonces el v construido
// es un certificado que la hace 1
return $\varphi(v)$

Como $\mathbf{NP} \subseteq \mathbf{P}/_{\mathbf{poly}}$ tenemos que `SATConst` $\in \mathbf{P}/_{\mathbf{poly}}$ y por ende existe C circuito booleano polinomial de $|\,\|\varphi\|\,|$ nodos iniciales cuya codificación tiene menos de $q(|\,\|\varphi\|\,|)$ bits que decide si φ es satisfacible. Supongamos que $\forall u \in \{0,1\}^{p(|\|\varphi\||)} \exists v \in \{0,1\}^{p(|\|\varphi\||)} \varphi(u,v) = 1$ luego podemos usar C como lo usa M para que dado un u fijo se construya un v para que $\varphi(u,v) = 1$ y por ende para que M devuelva 1. Y análogamente si se cumple

$$\exists D \in \{0,1\}^{q(|\|\varphi\||)} \forall z \in \{0,1\}^{q(|\|\varphi\||)} M(\|\varphi\|, D, z) = 1$$

entonces para cualquier z y por ende para cualquier u, se va a poder construir mediante el circuito C dentro de D, un certificado v que constate $\varphi(u,v) = 1$. □

Si uno asume la intuitiva idea de que **PH** no colapsa entonces necesariamente **NP** no está en $\mathbf{P_{/poly}}$ y por ende $\mathbf{P} \neq \mathbf{NP}$. Por esa razón un camino que siguió la comunidad durante décadas para tratar de demostrar $\mathbf{P} \neq \mathbf{NP}$ ha sido justamente tratar de demostrar que $\mathbf{NP} \nsubseteq \mathbf{P_{/poly}}$. Además uno podría preguntarse si $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$ y análogamente, se tiene que si $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$ entonces $\mathbf{EXP} = \Sigma_{\mathbf{2}}^{\mathbf{p}}$. La demostración de lo anterior es muy parecida a la que vimos recién por lo que no se la mostrará, se puede ver en la subsección 6.4 de [8] y en el trabajo original [29]. Por otro lado, vale decir que modificando un poco la definición de circuitos se pueden dar nuevas caracterizaciones de las clases **PH** y **EXP** e incluso, se puede definir una clase **NC** que modela a los lenguajes que pueden ser "eficientemente paralelizables" (ver subsecciones 6.7 y 6.8 de [8]). Vale agregar que es un problema abierto muy importante: **P** vs **NC**.

Jerarquía en P/poly

¿A mayor tamaño de circuitos más lenguajes que se deciden? Si. Primero veamos que hay funciones "hards". Es decir que no pueden ser computadas por circuitos de tamaño polinomial.

Dada una función $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ y una sucesión de circuitos booleanos, $\{C_n\}_{n\in\mathbb{N}}$, decimos que $\{C_n\}_{n\in\mathbb{N}}$ *computa* a f si $\forall x \in \{0,1\}$ se cumple que $C_{|x|}(x) = f(x)$.

Teorema 60 (Existen funciones *hard*)**.** *Para todo $n \in \mathbb{N}$ existe una función $f : \{0,1\}^n \longrightarrow \{0,1\}$ que no puede ser computada por un circuito C_n tal que $|C_n| \leq \lfloor 2^n/10n \rfloor$.*

Demostración. Podemos asumir sin pérdida de generalidad que $n > 9$ ya que para $n \leq 9$ se tiene que $n > \lfloor 2^n/10n \rfloor$, es decir hay más nodos iniciales que nodos en el circuito lo que es absurdo. Vamos a dar un argumento de conteo, mostrando que para todo n hay más funciones de $\{0,1\}^n$ a $\{0,1\}$ de las que se puede computar por circuitos de n nodos iniciales de tamaño menor o igual a $\lfloor 2^n/10n \rfloor$. Dado $n \geq 10$ se tiene que la cantidad de funciones de $\{0,1\}^n$ a $\{0,1\}$ es 2^{2^n}. Y como vimos en nuestra codificación, todo circuito de tamaño a lo sumo S (con $S \geq 4$) se puede codificar usando a lo sumo $9Slog(S)$ bits. Por lo tanto, seteando $S = \lfloor 2^n/10n \rfloor$ (y por ende $S \geq 10$) tenemos que la cantidad de circuitos de a lo sumo ese tamaño es de a lo sumo $2^{9Slog(S)}$ y se tiene que

$$2^{9Slog(S)} = 2^{9(\frac{2^n}{10n})log(\frac{2^n}{10n})} = 2^{9(\frac{2^n}{10n})(log(2^n)-log(10n))} \leq 2^{9(\frac{2^n}{10n})log(2^n)} = 2^{9(\frac{2^n}{10n})n}$$

que equivale a $2^{9(\frac{2^n}{10})}$ que es estrictamente menor a 2^{2^n} que es la cantidad de funciones. □

Y ahora sí, veamos el teorema de jerarquía.

Teorema 61 (Jerarquía para $\mathbf{P_{/poly}}$)**.** *Para todas funciones* $T : \mathbb{N} \longrightarrow \mathbb{N}$, $T_2 : \mathbb{N} \longrightarrow \mathbb{N}$ *tal que para todo* n *suficientemente grande se cumple* $2^n/n > T_2(n) > 10T(n) > n$*, se tiene*

$$\mathbf{SIZE}(T) \subsetneq \mathbf{SIZE}(T_2).$$

Demostración. Demostremos $\mathbf{SIZE}(n) \subsetneq \mathbf{SIZE}(n^2)$, para el caso general se usa la misma idea. Por teorema anterior tenemos que para todo l existe una función $f : \{0,1\}^l \longrightarrow \{0,1\}$ no computada por circuitos de tamaño $2^l/10l$. Por otro lado, toda función de $\{0,1\}^l$ a $\{0,1\}$ es computada por un circuito de tamaño $2^l 10l$. Entonces seteando $l = \frac{11}{10}\log(n)$ y definiendo $g : \{0,1\}^n \longrightarrow \{0,1\}$ como la aplicación de f a los primeros l bits de la entrada, tenemos que:

$$g \in \mathbf{SIZE}(2^l 10l) = \mathbf{SIZE}(11n^{\frac{11}{10}}\log(n)) \subseteq \mathbf{SIZE}(n^2)$$

$$g \notin \mathbf{SIZE}(2^l/10l) = \mathbf{SIZE}(n^{\frac{11}{10}}/11\log(n)) \supseteq \mathbf{SIZE}(n).$$

□

Caracterización de P/poly a partir de Máquinas de Turing

Como es usual en el campo, al estudiar un nuevo modelo de computación se lo trata de "llevar" al modelo más familiar, las máquinas de Turing. Caracterizaremos **P/poly** a partir de una variante de **DTIME**, definida a continuación, en la cual las MT's contarán con "manuales" que llamaremos *advices* que los ayudarán en la decisión. La idea es que dada una MT y una función $a : \mathbb{N} \longrightarrow \mathbb{N}$, para todo n natural existirá un advice α_n de tamaño menor o igual a $a(n)$ para ayudar a la máquina a decidir entradas de tamaño n.

Definición 62 (Máquina de Turing con advices)**.** Sean $T, a : \mathbb{N} \longrightarrow \mathbb{N}$ funciones. Un lenguaje L pertenece a la clase $\mathbf{DTIME}(T)/a$ si existe una sucesión $\{\alpha_n\}_{n\in\mathbb{N}}$ de palabras con $\alpha_n \in \{0,1\}^{a(n)}$ y una MT M que corre en tiempo cT (para alguna constante c) que cumple que $\forall n \in \mathbb{N}\ \forall x \in \{0,1\}^n$ se tiene:

$$x \in L \Leftrightarrow M(x, \alpha_n) = 1.$$

Estos α_n son los *advices* mencionados y a las MT que los usan las llamaremos MT *con advices.*

Teorema 63. $\mathbf{P_{/poly}} = \bigcup_{c,d\geq 1}^{\infty} \mathbf{DTIME}(n^c)/n^d$.

Demostración. Sea $L \in \mathbf{P_{/poly}}$, luego existe una sucesión de circuitos $\{C_n\}_{n\in\mathbb{N}}$ que lo decide acotados en tamaño por un polinomio p. Podemos hacer una MT M que corre en tiempo polinomial q que dada como entradas $C_{|x|}$ y x, devuelve $C_{|x|}(x)$. Luego la codificación en binario de C_n es el advice α_n, y por ende:

$L \in \mathbf{DTIME}(q)/p$ entonces $L \in \bigcup_{c,d\geq 1}^{\infty} \mathbf{DTIME}(n^c)/n^d$.

Por otro lado, sea $L \in \bigcup_{c,d \geq 1}^{\infty} \mathbf{DTIME}(n^c)/n^d$ entonces existe una MT M polinomial, un polinomio a y $\{\alpha_n\}_{n\in\mathbb{N}}$ advices de tamaño menor o igual $a(n)$, tal que para todo n se cumple $\forall x \in \{0,1\}^n \; x \in L \Leftrightarrow M(x, \alpha_n) = 1$. Sea $L_2 = \{(x,y) | y \leq a(|x|) \text{ y } M(x,y) = 1\}$, luego $L_2 \in \mathbf{P}$ y por ende $L_2 \in \mathbf{P_{/poly}}$, sea $\{D_m\}_{m\in\mathbb{N}}$ una sucesión de circuitos de tamaño polinomial que lo decide. Dado un n, queremos construir un circuito C_n de tamaño polinomial a n tal que $\forall x \in \{0,1\}^n \; x \in L \Leftrightarrow C_n(x) = 1$. Definamos $m = n + |\alpha_n|$ entonces tenemos que $\forall x \in \{0,1\}^n \; x \in L \Leftrightarrow D_m(x, \alpha_n) = 1$. Pero α_n es el mismo para todo $x \in \{0,1\}^n$, por lo tanto podríamos construir α_n en el circuito a partir de los nodos iniciales que reciben a x para que α_n no sea dada como entrada, tarea que nos demandaría solo una cantidad polinomial en n de nodos. En consecuencia, si C_n es ese nuevo circuito entonces se tiene que C_n es de tamaño polinomial a n y cumple $\forall x \in \{0,1\}^n \; x \in L \Leftrightarrow C_n(x) = 1$. Por lo tanto, $L \in \mathbf{P_{/poly}}$. □

Métodos para probar Lower Bounds de Circuitos

La presente sección se basa en el trabajo [39], el cual expone un resumen didáctico del tópico. Un estudio profundo del mismo queda fuera del alcance de este trabajo, al lector que le interese profundizar más puede ver la segunda parte del libro [8]. O también, el trabajo [14].

Dada una función, una *lower bound* para tal función es una cota inferior del tamaño de los circuitos que la computan. Por ejemplo un resultado de la forma: "La función $f : \{0,1\}^n \longrightarrow \{0,1\}^*$ no puede se computada por circuitos de tamaño menor a $10n^4$". Como un lenguaje de decisión se puede pensar también como su función característica, entonces las lowers bound sirven también para hacer separaciones de clases. Por ejemplo de demostrar que la función característica de `SAT` no tiene circuitos de tamaño polinomial que la computen entonces se demostraría que $\mathbf{NP} \not\subseteq \mathbf{P_{/poly}}$.

A grandes rasgos, hay tres grande formas de probar lower bounds. Queda fuera de este trabajo ejemplificarlas matemáticamente, en lugar de ello se las va a explicar.

1. Métodos de Restricción

   Es un método inductivo que dada una función $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ que pueda ser "partible en trozos" (esto es que exista una sucesión de funciones $\{g_n : \{0,1\}^n \longrightarrow \{0,1\}^*\}$ tal que $\forall n \in \mathbb{N} \; \forall x \in \{0,1\}^n \; f(x) = g_n(x)$ y que cumpla que computar g_n esté estrechamente relacionado a computar g_k para $k < n$). Se setea algunos de los bits de entradas a un pequeño circuito de n nodos iniciales y se lo simplifica, entonces el circuito se hace demasiado chico para poder computar a g_k.

2. Métodos de Polinomios

   La idea es representar, aproximadamente o exactamente, un circuito como un polinomio. Luego probar que la función en cuestión no puede ser representada por tal polinomio.

3. Fuerza Bruta

   Se sigue el "camino inverso", se trata de diseñar funciones que no puedan ser computadas por circuitos pequeños. Si pudiéramos hacer esto en **PSPACE** o **NP**, separaríamos **P** de **PSPACE** y respectivamente de **NP**.

Estrategias

Los anteriores tipos de métodos pueden ser pensados como tácticas. Entonces, ¿En qué estrategias o enfoques suelen utilizarse tales métodos? Podemos destacar tres grandes estrategias o enfoques para probar resultados en este campo de estudio.

1. Bottom-up

   Se agregan restricciones al modelo de circuitos y se encuentran "fuertes" lower bounds para tales modelos. Luego gradualmente se va eliminando las restricciones volviendo al modelo más poderoso y tratando de adaptar tales lower bounds. En resumen, se hacen modelos de circuitos tan "débiles" que no es "tan" difícil encontrar funciones que no puedan ser computadas por ellos.

2. Top-down

   Se arranca con funciones que sabe que son difíciles para circuitos en algún sentido, y luego se trata de "transformar" estas funciones en funciones más fáciles. Se parte de funciones tan difíciles que demostrar que no hay circuitos que las computan no es demasiado difícil.

3. Middle

   Se comienza con funciones fáciles de computar y se trata de encontrarle lower bounds indiscriminadamente.

Los métodos Polinomiales son principalmente utilizados en el enfoque Bottom-up: en algún punto el circuito debe tener alguna restricción para que pueda ser bien representado por un polinomio. Los métodos de Fuerza Bruta caen dentro del enfoque To-down y los métodos de Restricción suelen aparecer en enfoques Top-Down y Middles.

Limitación de los métodos, Natural Proofs

¿Por qué son tan difíciles las lower bounds? ¿Por qué más allá de los grandes esfuerzos de los investigadores en estas últimas décadas, no ha sido posible encontrar fuertes lower bounds para circuitos generales?

Como se comentó en la introducción del capítulo, así como la técnica de diagonalización tiene su "limitación" con la relativización, los métodos para encontrar lower bounds tienen su "limitación" llamada *Natural Proofs*. Estas Natural Proofs vendrían a constatarnos que los esfuerzos por resolver los grandes problemas del campo solo utilizando lower bounds son una pérdida de tiempo. Y nos

sirven de consuelo en el sentido de que si no se los ha podido resolver no es por una incapacidad de los investigadores sino por una limitación de la técnica per se. Pero más importante, nos enseñan que para resolver los grandes problemas se deberá tener nuevas ideas y técnicas. En 1994 se definió una noción de *"Natural mathematical proof"* para lower bounds. Se mostró que los métodos para probar lower bounds caían dentro de esa noción, y que encontrar fuertes lower bounds a partir de tales técnicas violaría una forma más fuerte de la conjetura $\mathbf{P} \neq \mathbf{NP}$. Particularmente, violaría la conjetura de que las *funciones one-way* existen (se verán en el siguiente capítulo (Definición 69), son funciones computables polinomialmente pero que su inversa no es computable sub exponencialmente), se hipotetiza que factorización de enteros, logaritmo discreto y la función RSA [31], son ejemplos de funciones one-way. Como la actual evidencia sugiere que sí existen, se concluye que tales argumentos no son capaces de llevarnos a buen puerto. Otra manera de verlo también es que si existe una función "difícil" va a ser difícil probar que lo es.

Como en la sección anterior, un estudio profundo del tópico que a continuación se presenta queda fuera del alcance de este trabajo. Nos basaremos en las principales nociones del capítulo 23 de [8] y en [36]. Por si le interesa ver al lector, el trabajo original en el cual se presentan por primera vez estos conceptos es [30].

Con la siguiente "definición en alto nivel" alcanza para que nos llevemos una noción aceptable del concepto de *Natural Proof*. Un estudio más profundo pero también didáctico se realiza en la sección 3.9 de [36]. Dada una función booleana $f : \{0,1\}^n \longrightarrow \{0,1\}$ y $c \geq 1$, toda prueba de que f no tiene circuitos de tamaño n^c que la computen puede entenderse como exhibir alguna propiedad P tal que f la satisface pero que todas las funciones booleanas que tienen circuitos de tamaño n^c no. Es decir, tal prueba puede verse como exhibir un predicado P sobre funciones booleanas tal que $\mathsf{P}(f) = 1$ pero que para toda función $g : \{0,1\}^n \longrightarrow \{0,1\}$ computable por algún circuito en $\mathbf{SIZE}(n^c)$ se cumple $\mathsf{P}(g) = 0$.

Decimos que una propiedad P de funciones Booleanas de $\{0,1\}^n$ a $\{0,1\}$ es *natural* si satisface los siguientes requerimientos:

1. Extensión: "muchas" funciones cumplen P.
2. Constructividad: es "fácil" verificar si una función cumple P observando su tabla de verdad.
3. Utilidad: todo circuito booleano que compute alguna función que satisfaga la propiedad P tiene que ser de "gran tamaño".

Donde "muchas", "fácil" y "gran tamaño" dependen del modelo del circuito donde se esté probando la lower bound.

Es loable que el lector se pregunte por qué la noción de natural proof atrapa algunas de tales propiedades, para analizarlo en detalle ver subsección 23.2 de [8]. ¿Por qué Extensión? ¿Por qué una lower bound utilizada para una función particular, por ejemplo una que compute `3SAT` debería utilizar una propiedad

compartida por muchas otras funciones? Se puede demostrar que toda prueba de que una función $f_o : \{0,1\}^n \longrightarrow \{0,1\}$ no es computable por circuitos de tamaño S implica que al menos la mitad de las funciones de $\{0,1\}^n$ a $\{0,1\}$ no es computable por circuitos de tamaño menor a $\frac{S}{2} - 10$.

¿Por qué Constructividad? Es una antiguo debate filosófico sobre las demostraciones "no constructivas", donde la existencia de un objeto es demostrada sin dar una construcción explícita del mismo o incluso, sin dar un una manera de construirlo. En este contexto utilizamos una noción más fuerte de constructividad! En general, "fácil" va a significar determinista y polinomial! No solo pedimos que el objeto sea construible sino que también sea determinista y eficientemente construible! ¿Por qué? El núcleo de las lowers bounds se erige sobre técnicas del campo de la Combinatoria y en general, las técnicas en Combinatoria suelen ser constructivas en nuestro sentido. Si bien existen técnicas no constructivas en Combinatoria, no se las ha podido utilizar para probar lower bounds.

¿Puede obtenerse lower bounds usando demostraciones que no son natural proofs? Sí. Un ejemplo es el siguiente teorema:

$$\forall c \in \mathbb{N} \ \mathbf{PromiseMA} \nsubseteq \mathbf{ProSIZE}(n^c)$$

donde $\mathbf{ProSIZE}(n^c)$ denota a los *promise problems* de circuitos de tamaño n^c y **PromiseMA** la generalización de la clase **MA** a promise problems (ver página 504 de [8]).
Esta prueba solo recae en la antigua y simple técnica de diagonalización, que es inherentemente no natural. E incluso el trabajo [3] muestra que este resultado no relativiza (que no vale para todas las MTO's, para repasar relativización ver 33). Así hay contados ejemplos que de manera ingeniosa con otras técnicas como *aritmetización* (se verá en el último capítulo) se logra sortear los obstáculos de las naturals proofs. Pero son escasos y complicados, la inmensa mayoría no logra superarlos. En general la propiedad de constructividad parece ser la menos difícil de esquivar.

Finalmente, podría parecer un paisaje desolador luego que irrumpió en él las naturals proofs. Pero es muy valioso su aporte, cuando uno está atascado en un problema es muy útil saber que es muy difícil de resolver con tal planteo. Uno reconoce un obstáculo, lo esquiva y avanza.

Capítulo 5

Computación Randomizada: BPP

Albert Einstein – El azar no existe, Dios no juega a los dados.
Neils Bohr – Einstein, deje de decirle a Dios lo que tiene que hacer!

Cualquiera que considere un método aritmético para producir dígitos aleatorios está, por supuesto, en un estado de demencia.

John von Neumann

Dejaremos para físicos y filósofos la existencia del azar en el universo, a fines prácticos, el supuesto de que existe ha permitido solucionar problemas de manera eficiente. Desde el método de Monte Carlo a algoritmos de inteligencia artificial, se requiere en menor o mayor medida, el empleo de algoritmos probabilísticos. Con ese fin, casi todos los lenguajes de programación vienen equipados con algún método para generar números pseudoaleatorios. Esta aleatoriedad parece ser una cuestión que no es contemplada por las máquinas de Turing que hemos considerado hasta aquí. En este capítulo estudiaremos una versión de ellas que incorpora aleatoriedad en sus transiciones. Por lo recién mencionado, entender el poder de estas nuevas máquinas ya puede parecer sumamente interesante y puede traer grandes beneficios prácticos. Pero también puede resultar seductor que en este campo haya tantos problemas abiertos importantes. Estudiaremos la clase **BPP**, que es la colección de lenguajes decididos eficientemente por estas máquinas y veremos que muchas cosas que conocemos de las clases que hemos estudiado no las conocemos para ésta. Por ejemplo, no tenemos un teorema de jerarquía para **BPP** ni tampoco conocemos algún lenguaje completo para ella, o si quiera si existe. Tampoco sabemos si $\mathbf{BPP} \subsetneq \mathbf{NEXP}$, lo cual sospechamos cierto. Tampoco conocemos si $\mathbf{BPP} = \mathbf{P}$, cuestión de por sí interesante, pero que como se verá más adelante vale que $\mathbf{P} \neq \mathbf{BPP} \implies \mathbf{P} \neq \mathbf{NP}$. Por lo tanto, la pregunta ¿$\mathbf{BPP} = \mathbf{P}$? parece crucial. Pero como estudiosos de complejidad computacional, la importancia de estudiar este campo es aún mayor. Más allá de lo estudiado en este capítulo en relación a **BPP**, áreas enteras

como Criptografía, Pruebas Interactivas, y el teorema **PCP** por citar algunas, requieren del azar en un nivel fundamental. Por lo tanto, entender este campo es una obligación para continuar adentrándonos en la teoría de complejidad computacional.

Un algoritmo probabilista es un algoritmo que utiliza elecciones aleatorias, puede ser por ejemplo que el algoritmo en algún paso asigne un número elegido aleatoriamente a una variable. Así como a los algoritmos deterministas los modelamos con máquinas de Turing deterministas, a los algoritmos probabilísticos los modelamos con máquinas de Turing Probabilística*s*. Una *máquina de Turing Probabilística* es una variante de la Máquina de Turing monocinta (ver Definición 3). Esta variante cuenta con dos funciones de transición que las utiliza de manera aleatoria en su ejecución. En cada paso de la ejecución elige de manera aleatoria aplicar una de las funciones de transición. Luego, lo que devuelve la máquina a partir de una entrada (el contenido de la cinta al llegar al estado final) es una variable aleatoria.

Definición 64 (Máquina de Turing Probabilística)**.** Una *Máquina de Turing Probabilística* (abreviada MTP) es una 6-upla $(Q, \Gamma, \delta_0, \delta_1, q_i, q_f)$ donde Q es el conjunto de *estados*, Γ es el *alfabeto* de la máquina donde $\{0, 1, \square, \triangleright\} \in \Gamma$, $q_i, q_f \in Q$ son los *estados inicial y final* respectivamente, y δ_0, δ_1 son las *funciones de transición*. Sea M MTP y $x \in \{0,1\}^*$ una entrada, luego en la ejecución la máquina elige en cada paso con probabilidad $\frac{1}{2}$ aplicar δ_0 o aplicar δ_1. Cada una de tales elecciones se realiza de manera independiente a las elecciones tomadas anteriormente. El concepto de *hilo* y *camino* para la MTND aplica a la MTP (ver Definición 9). Recordando, un hilo es una sucesión de elecciones de función de transición aplicadas, y un camino es un hilo que comienza en q_i y finaliza en q_f. Decimos que M *a partir de la entrada* x *finaliza* si todo hilo de la máquina a partir de tal entrada tiene una longitud finita. En tal caso, con $M(x)$ denotamos a la *salida* de la máquina la cual es el contenido de la cinta al llegar al estado final (sin los símbolos $\square$ como es usual). Observemos que $M(x)$ *es una variable aleatoria.* Decimos que M *a partir de* x *corre en tiempo* $t \in \mathbb{N}$ si M finaliza a partir de x y todo camino tiene una longitud menor o igual a t. Dada $T : \mathbb{N} \longrightarrow \mathbb{N}$ función, d*ecimos que* M *corre en tiempo* T si para toda entrada x, M a partir de x corre en tiempo $T(|x|)$. Vale agregar que en ocasiones diremos corre en T pasos en vez de decir corre en tiempo T, el significado es el mismo.

Dado un lenguaje L con χ_L denotamos a su función característica, es decir $\chi_L : \{0,1\}^* \longrightarrow \{0,1\}^*$ cumple

$$\chi_L(x) = \begin{cases} 1 & \text{si } x \in L \\ 0 & \text{si } x \notin L \end{cases}$$

Definición 65 (**BPP**)**.** Sea $L \subseteq \{0,1\}^*$ un lenguaje. Decimos que MTP M decide a L si M finaliza para toda entrada y cumple:

$$Pr[M(x) = \chi_L(x)] \geq \frac{2}{3}.$$

Donde $Pr[A] = B$ denota que la probabilidad de que ocurra el evento A es B.

Sea $T : \mathbb{N} \to \mathbb{N}$, decimos que $L \in \mathbf{BPTIME}(T)$ si existe MTP M y $c \in \mathbb{N}$ tal que M corre en tiempo cT y decide a L. Definimos

$$\mathbf{BPP} = \bigcup_{c \geq 1}^{\infty} \mathbf{BPTIME}(n^c).$$

Vale observar que la manera "explícita" (y equivalente) de ver que M MTP decide a L o al menos la forma que más se encuentra en la bibliografía es la siguiente, sea $x \in \{0,1\}^*$ entonces:

$$\text{si } x \in L \Longrightarrow Pr[M(x) = 1] \geq \frac{2}{3}$$

$$\text{si } x \notin L \Longrightarrow Pr[M(x) = 0] \geq \frac{2}{3}$$

A veces directamente nos referiremos a esta última forma al expresar que una MTP decide a un lenguaje.

Notemos que este modelo tiene una diferencia crucial con los modelos que veníamos estudiando: a estas nuevas máquinas les damos la posibilidad de *errar*. Por ejemplo sea $L \in \mathbf{BPP}$ y M una MTP que lo decide, dada una entrada x, podemos obtener valores diferentes de $M(x)$ en diferentes ejecuciones. Como $x \in L$ o $x \notin L$, alguno de esos valores obtenidos necesariamente no es correcto y no sabemos cuál. ¡Pero no nos importa! Tenemos un procedimiento que puede aceptar una palabra (que la máquina devuelva 1) que no pertenece al lenguaje o rechazar (que la máquina devuelva 0) a una que sí pertenece. Pero en la mayoría de las casos el procedimiento devuelve la respuesta correcta y eso es lo que cuenta. Además como se verá a continuación, la probabilidad de errar se puede reducir arbitrariamente de manera sencilla y solo con un costo polinomial. Vale mencionar que hay algoritmos que solo "erran de un *solo lado*", por ejemplo puede haber un algoritmo y un lenguaje tal que si una entrada está en el lenguaje entonces el algoritmo no se equivoca pero que si una entrada no está en el lenguaje entonces el algoritmo puede equivocarse. Esta clase de lenguajes lleva el nombre de **RP**, más formalmente si x pertenece al lenguaje entonces $Pr[M(x) = 1] = 1$ y si x no pertenece al lenguaje entonces $Pr[M(x) = 0] \geq \frac{2}{3}$. Como es natural $\mathbf{RP} \subseteq \mathbf{BPP}$, también sencillamente se puede demostrar que $\mathbf{RP} \subseteq \mathbf{NP}$.

Observación. Notar que $\mathbf{P} \subseteq \mathbf{BPP}$, pues una MT con función de transición δ es también una MTP con $\delta_0 = \delta_1 = \delta$.

Sobre la robustez de la definición se puede decir que la constante $\frac{2}{3}$ es arbitraria ya que con una constante mayor a $\frac{1}{2}$ y menor a 1 se define la misma clase (**BPP**). El siguiente lema lo muestra, e incluso algo más fuerte, podemos reducir el error de predicción de manera arbitraria polinomialmente. Mostrando que en la práctica los algoritmos probabilísticos son tan precisos como los determinísticos.

Teorema 66 (Reducción de error)**.** *Sea $c > 0$ y sea L un lenguaje. Supongamos que existe una MTP M polinomial tal que para toda entrada x vale que*

$$Pr[M(x) = \chi_L(x)] \geq \frac{1}{2} + |x|^{-c}.$$

Entonces, $\forall d > 0$ existe una MTP polinomial M' tal que para toda entrada x cumple

$$Pr[M'(x) = \chi_L(x)] \geq 1 - 2^{-|x|^d}.$$

Demostración. La máquina M' realiza simplemente lo siguiente: dada una entrada x, corre M a partir de x unas $k = 8|x|^{2c+d}$ veces, obteniendo k salidas $y_{1,...,}y_k \in \{0, 1\}$. Si la mayoría de dichas salidas es 1 entonces M' devuelve 1, caso contrario devuelve 0. Analicemos esta máquina: para cada i entre 1 y k definimos la variable aleatoria X_i igual a 1 si $y_i = \chi_L(x)$ e igual a 0 en caso contrario. Nótese que $X_1, ..., X_k$ son variables aleatorias Booleanas independientes que cumplen $E[X_i] = Pr[X_i = 1] \geq p$ para $p = \frac{1}{2} + |x|^{-c}$. Se puede ver que para un z suficientemente chico:

$$Pr[|\sum_{i=1}^{k} X_i - pk| > zpk] < e^{-\frac{z^2}{4}pk}.$$

Como $p = \frac{1}{2} + |x|^{-c}$ y tomando $z = |x|^2$, garantizamos que si $\sum_{i=1}^{k} X_i \geq pk - zpk$ entonces retornamos la respuesta correcta. Por lo tanto, la probabilidad de que retornemos la respuesta errada esta acotada por

$$e^{-\frac{1}{4|x|^{2c}}\frac{1}{2}8|x|^{2c+d}} \leq 2^{-|x|^d}.$$

□

Por lo anterior, en la práctica podemos pensar que la computación eficiente es modelada por **BPP** (incluso de llegar a darse **P** $\neq$ **BPP**) ya que podemos reducir el error de nuestros algoritmos probabilísticos todo lo que queramos solo con un costo polinomial en tiempo. Por lo tanto **BPP** vs **NP** se nos presenta como un problema de la mayor importancia, casi tan importante como **P** vs **NP** en mi opinión. Como se cree que **P** = **BPP** se cree que **BPP** $\subseteq$ **NP** pero lejos se está de demostrarlo.

Un ejemplo para mostrar el poder que el azar puede darle a la computación.

Ejemplo. Testeo probabilístico de Primalidad.

El *testeo de primalidad* consiste en: dado un número natural N, determinar si es primo. Algoritmos para tal tarea han sido descubiertos milenios antes de la llegada de las computadoras porque los matemáticos los han necesitado para testear diversas conjeturas. Dicen que Gauss por más que en sí era una "computadora muy rápida" le pedía ayuda a una persona autista que tenía una capacidad extraordinaria para el cálculo numérico para que le testeara la

primalidad de números. Idealmente querrían algoritmos eficientes que corrieran en un tiempo polinomial a la representación de N i.e. $poly(\log(N))$. Durante siglos los matemáticos no conocieron algoritmos así, hasta que en la década del 70 del siglo pasado algoritmos probabilísticos eficientes fueron descubiertos demostrando el gran poder de estos algoritmos. A continuación veremos uno simple.

Formalmente queremos ver que el lenguaje

$$\texttt{PRIMOS} = \{[\![N]\!] : \ N \ \ es \ \ un \ \ numero \ \ primo\} \in \mathbf{BPP}$$

Para lo cual exhibimos un algoritmo probabilístico polinomial. Para todo numero N y $A \in [1, ..., N-1]$, definimos:

$$QR_N(A) = \begin{cases} 0 & \text{MCD}(A, N) \neq 1 \\ +1 & \exists B : \text{MCD}(B, N) = 1 \wedge A = B^2(\text{mod}(N)) \\ -1 & c.c \end{cases}$$

Con mod módulo y MCD máximo común divisor. Usamos las siguientes propiedades que pueden probarse con teoría de números básica (por ejemplo ver [35]):

- Para todo primo impar N y $A \in [1, ..., N-1]$ se tiene $QR_N(A) = A^{(N-1)/2}(\text{mod}(N))$.
- Para todos impares N, A definimos el *símbolo Jacobi*

$$\left(\frac{N}{A}\right) := \prod_{i=1}^{k} QR_{P_i}(A)$$

  donde $P_1, ..., P_k$ son los factores primos de N (no necesariamente distintos). Luego $(\frac{N}{A})$ es computable en tiempo $\mathcal{O}(\log(A)\log(N))$.
- Para todo impar compuesto N, entre todos los $A \in [1, ..., N-1]$ que cumplen $\text{MCD}(A, N) = 1$ se tiene que a lo sumo la mitad de tales A cumplen $(\frac{N}{A}) = A^{(N-1)/2}(\text{mod}(N))$.

Estas propiedades implican un algoritmo simple para testear si un número N es primo. Dado un entero N (asumiremos sin perder generalidad que no es par) elegimos aleatoriamente un entero A tal que $1 \leq A < N$. Si $\text{MCD}(A, N) > 1$ o $(\frac{N}{A}) \neq A^{(N-1)/2}(\text{mod}(N))$ entonces retornamos "compuesto", caso contrario retornamos "primo". Notar que este algoritmo siempre devolverá primo si el número es primo, y si es compuesto devolverá compuesto con una probabilidad de al menos $\frac{1}{2}$. Naturalmente, esta última probabilidad se puede mejorar a $\frac{2}{3}$ (y así satisfacer los requerimientos de la Definición 65), repitiendo este algoritmo una cantidad constante de veces.

Vale mencionar que hace relativamente poco, se descubrió un algoritmo polinomial determinista para testear primalidad (ver [7]). Curiosamente el "search

problem" del testeo de primalidad (encontrar la factorización de un número), parece ser mucho más difícil. La conjeturada dificultad de este problema subyace muchos sistemas criptográficos, y recientemente se ha mostrado que este problema puede ser resuelto eficientemente por máquinas cuánticas [34].

Por último, mencionamos el problema *Polynomial Identity Testing,* `PIT`, otro problema en **BPP** muy importante (en mi opinión el más importante) que tiene muchas y variadas aplicaciones, por citar algunas: se utiliza en general en la técnica de aritmetización y en la demostración de **PSPACE** $\subseteq$ **IP** (los veremos en el próximo capítulo, Teorema 79), en el testeo de primalidad, en determinar si existe un matching perfecto en un grafo bipartito, y otras. Encontrar un algoritmo polinomial determinista que lo resuelva sería un logro superlativo para las ciencias de la computación. Una presentación didáctica del problema se puede encontrar en [35, 23]. Dados dos polinomios p, q decimos que son idénticos, $p \equiv q$, si los coeficientes de los monomios de p son iguales a los coeficientes de los monomios de q. Luego, dada la descripción de dos polinomios, `PIT` consiste en determinar si son idénticos. Vale aclarar que dado un dominio no preguntamos si dos polinomios son iguales en tal dominio, sino que preguntamos si son iguales en cualquier dominio. Por ejemplo $x^2 - x$, 0 son equivalentes en $\mathbb{Z}_2$ pero no son idénticos. Como $p \equiv q$ sii $p - q \equiv 0$ a veces se define al problema como determinar si un polinomio es nulo. Si la representación de p tiene sus coeficientes de manera explícita entonces determinar si el polinomio nulo es trivial, en general nunca se utiliza tal codificación, la representación de polinomios estándar se da a partir de circuitos algebraicos, los cuales son una generalización de los circuitos booleanos que reciben de entrada no solo ceros y unos sino valores de un cuerpo arbitrario como $\mathbb{R}$ y que tienen de etiquetas $\cdot, +, -$ que se computan naturalmente como el producto, la suma y la resta, el lector interesado en estos circuitos puede ver [36] y el capítulo 16 de [8]. Vale agregar que el núcleo de este problema el Lema de DeMillo-Lipton-Schwartz–Zippel [16] el cual afirma:

Lema. *Si un polinomio no nulo $p(x_1, x_2, \dots, x_m)$ sobre $F = GF(q)$ es de grado menor o igual a d, entonces*

$$Pr[p(a_1 \dots a_m) \neq 0] \geq 1 - \frac{d}{q}$$

donde la probabilidad es sobre todas las posibles elecciones de $a_1 \dots a_m \in F$.

Así como **NP** (ver Definición 8) se puede definir mediante máquinas deterministas y certificados, **BPP** también. El certificado que se le da a la máquina es la codificación de un camino.

Definición 67 (**BPP** definición alternativa por certificados)**.** Un lenguaje L pertenece a **BPP** si existe una MT M polinomial y un polinomio p, tal que para toda entrada x se tiene $Pr_{r \in \{0,1\}^{p(|x|)}} [M(x, r) = \chi_L(x)] \geq \frac{2}{3}$.

Notar que **BPP** $\subseteq$ **EXP** ya que una MT en tiempo exponencial dada x de entrada puede simular y computar $M(x, r)$ para todo $r \in \{0,1\}^{p(|x|)}$ calculando la proporción que devuelve 1.

Lema 68. *Ambas definiciones de* **BPP** *coinciden.*

Demostración. Si $L \in \mathbf{BPP}$ en el sentido de la Definición 65 entonces existe una MTP M que corre en tiempo polinomial p que decide a L. Dada una entrada x hay a lo sumo $2^{p(|x|)}$ caminos que puede seguir la máquina, cada uno puede servirle de certificado r a una MT M_2 polinomial que reciba de entrada (x, r) y simule a M en entrada x siguiendo el camino dado por r. Luego $L \in \mathbf{BPP}$ en el sentido de los certificados (Definición 67). No importa que haya caminos de distintos tamaños, sabemos que la mayoría (tengan el tamaño que tengan) lleva a la salida correcta. Y la cantidad total de caminos, tengan el tamaño que tengan, es menor o igual a $2^{p(|x|)}$. Por el otro lado, si $L \in \mathbf{BPP}$ en el sentido de los certificados (Definición 67) entonces existe M MT polinomial y polinomio p que cumplen tal definición. Podemos construir una M_2 MTP que realiza lo siguiente: dada x de entrada, aleatoriamente construye un certificado $r \in \{0,1\}^{p(|x|)}$ y de manera determinista computa M a partir de la entrada (x, r). Luego M_2 corre polinomialmente y decide a L luego $L \in \mathbf{BPP}$ en el sentido de la Definición 65. □

Como se mencionó en la introducción, lo único que se sabe hasta el momento es que $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{EXP}$. También, que la pregunta $\mathbf{P} = \mathbf{BPP}$ es muy importante. ¿Qué cree el lector $\mathbf{P} = \mathbf{BPP}$ o $\mathbf{P} \neq \mathbf{BPP}$? La primera impresión suele ser que deben ser clases distintas, y la comunidad durante mucho tiempo también la sostuvo. Pero hoy en día la impresión es otra, la mayoría de los investigadores en el área cree que $\mathbf{P} = \mathbf{BPP}$! Las razones de esa creencia quedan fuera de este capítulo, pero a grandes rasgos, se descubrió que asumiendo algunas *lower bounds* razonables, se puede "transformar" cualquier algoritmo probabilístico en determinista solo con un costo polinomial. Para buscar más sobre el tema véase el capítulo Derandomization de [8].

Funciones One-way

Veamos una aplicación interesante e importante de los algoritmos probabilísticos que hemos venido estudiando. Veremos unas funciones que tienen una importancia crucial en la criptografía. Se hipotetiza que existen y es una forma más fuerte de la conjetura $\mathbf{P} \neq \mathbf{NP}$. Vale mencionar que gran parte de la seguridad de internet y de otros sistemas informáticos se sostienen en que esta hipótesis es cierta, ya que se sostienen en que la función RSA es one-way, ver [31].

Intuitivamente, una función *one-way* es una función computable polinomialmente que cumple que toda MTP polinomial tiene una baja probabilidad de computar su inversa.

Definición 69 (One-way function)**.** Diremos que $\varepsilon : \mathbb{N} \to \mathbb{R}$ es una función *negligible* si

$\forall c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}$: si $n \geq n_o$ entonces $\varepsilon(n) < n^{-c}$.

Diremos que una función $f : \{0,1\}^* \to \{0,1\}^*$ es *one-way* si:

1) f es computable polinomialmente.

2) $\forall A$ MTP polinomial $\exists \varepsilon$ función negligible tal que $\forall n \in \mathbb{N}$ se cumple:

$$\forall x \in \{0,1\}^n \; Pr[A(f(x), 1^n) \in f^{-1}(f(x))] \leq \varepsilon(n).$$

Teorema 70. *Si existe una función One-way entonces* $\mathbf{P} \neq \mathbf{NP}$.

Demostración. Supongamos que $\mathbf{P} = \mathbf{NP}$, supongamos que existe una función one-way $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ y lleguemos a un absurdo. Sea L_f el lenguaje definido como $L_f = \{(x_0, y, 1^n) : \exists x : \; |x| = \; n \wedge \; x_0 \leq x \; \wedge \; f(x) = y\}$ para $n \in \mathbb{N}$ y con $x_0 \leq x$ significando que la palabra x_0 es prefijo de la palabra x. Sea M_f MT polinomial que computa a f (esta máquina existe porque f es one-way). Notemos que $L_f \in \mathbf{NP}$ ya que x es un certificado de que $(x_0, y, 1^n) \in L_f$ para M_f como verificadora, y como $|x| = n$ se tiene que esta verificación es polinomial. Por hipótesis $\mathbf{P} = \mathbf{NP}$ en consecuencia $L_f \in \mathbf{P}$, sea M_2 MT polinomial que decide a L_f. A continuación usaremos M_2 para crear A MT polinomial (y por ende una MTP polinomial) que dada como entrada $(y, 1^n)$ devuelve, de existir, un x tal que $f(x) = y$ y $|x| = n$. La idea es usar x_0 para armar x.
Descripción de A, sea $y \in \{0,1\}^*$ y $n \in \mathbb{N}$, dada $(y, 1^n)$ de entrada:

$x_0 := \varepsilon$ // primero x_0 es la palabra vacía
for $i = 1$ **to** n **do**:
  if $M_2(x_0 0, y, 1^n) = 1$: // si existe x tal que $|\, x \,| = n \wedge x_0 0 \leq x \wedge f(x) = y$
    $x_0 := x_0 0$
  else:
    $x_0 := x_0 1$
return x_0

Este x_0 construido cumple que $(x_0, y, 1^n) \in L_f$ si y solo si $f(x_0) = y$. Y lo hemos construido polinomialmente, en consecuencia f no puede ser one-way. ¡Absurdo! □

Relación entre BPP y otras clases

Ya vimos que $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{EXP}$, a continuación veremos que $\mathbf{BPP} \subseteq \mathbf{P_{/poly}}$ [6] y que $\mathbf{BPP} \subseteq \mathbf{PH}$ [38], pero más fuerte aún, que $\mathbf{BPP} \subseteq \mathbf{\Sigma_2^p} \cap \mathbf{\Pi_2^p}$ [26].

Teorema 71. $\mathbf{BPP} \subseteq \mathbf{P_{/poly}}$.

Demostración. Sea $L \in \mathbf{BPP}$, luego por la definición alternativa de $\mathbf{BPP}$ y el procedimiento de reducción de error (Teorema 66) existe una MT M que corre en tiempo polinomial q y un polinomio p tal que para todo $n \in \mathbb{N}$ y para toda entrada $x \in \{0,1\}^n$

$$Pr_{r \in \{0,1\}^m}[M(x, r) \neq \chi_L(x)] < \frac{1}{2^{n+1}}$$

con $m = p(|x|)$. Digamos que un *certificado* $r_0 \in \{0,1\}^m$ es malo para una entrada $x \in \{0,1\}^n$ si $M(x,r) \neq \chi_L(x)$, y en caso contrario digamos que r_0 es *bueno* para x. Observemos que a lo sumo $\frac{2^m}{2^{n+1}}$ certificados son malos para x. Entonces la cantidad de certificados que son malos para alguna entrada de largo n es a lo sumo $2^n.\frac{2^m}{2^{n+1}} = \frac{2^m}{2}$. En consecuencia debe existir un certificado $r_0 \in \{0,1\}^m$ bueno para todas las entradas $x \in \{0,1\}^n$. Luego $\forall x \in \{0,1\}^n M(x,r_0) = \chi_L(x)$, por ende r_0 es un *advice* para M (ver Definición 62) y por lo tanto $L \in \mathbf{DTIME}(q)/p$ y entonces $\mathbf{BPP} \subseteq \mathbf{P_{/poly}}$. □

Hemos visto que los lenguajes en **BPP** tienen circuitos de tamaño polinomial, junto al teorema 59 demostrado en el capítulo anterior
($\mathbf{NP} \subseteq \mathbf{P_{/poly}} \Rightarrow \mathbf{PH} = \Sigma_\mathbf{2}^\mathbf{p}$) se deduce el interesante hecho de que `3SAT` no tiene un algoritmo probabilístico eficiente a menos que la jerarquía polinomial colapse.

Teorema 72. $\mathbf{BPP} \subseteq \Sigma_\mathbf{2}^\mathbf{p} \cap \mathbf{\Pi_2^p}$.

La demostración es interesante y difícil, y no nos aporta demasiado a estas alturas por lo que no merece la pena verla. El lector interesado puede consultarla en la subsección 7.5 de [8].

Observemos que si $\mathbf{P} = \mathbf{NP}$ entonces $\mathbf{PH} = \mathbf{P} = \mathbf{NP} = \mathbf{BPP} = \Sigma_\mathbf{2}^\mathbf{p} \cap \mathbf{\Pi_2^p}$, en particular $\mathbf{P} = \mathbf{BPP}$, en consecuencia si $\mathbf{P} \neq \mathbf{BPP}$ entonces $\mathbf{P} \neq \mathbf{NP}$.

Problemas completos y Jerarquía para BPP?

Como hemos definido algoritmos probabilísticos tiene sentido definir alguna noción de reducción aleatoria entre dos lenguajes, esta noción se prueba útil en algunas áreas de complejidad.

Definición 73. Un lenguaje B se reduce a un lenguaje C bajo reducción aleatoria polinomial, $B \leq_r C$, si existe una MTP polinomial M tal que para cada $x \in \{0,1\}^*$ se cumple $Pr[\chi_B(x) = \chi_C(M(x))] \geq \frac{2}{3}$.

Observación. Observemos que esta relación no es transitiva, de todas maneras es útil en el sentido de que si $C \in \mathbf{BPP}$ y $B \leq_r C$ entonces $B \in \mathbf{BPP}$. Ya que si M_c MTP decide a C entonces dado un x, si $x \in B$ entonces
$Pr[M_c(M(x)) = 1] \geq \frac{5}{9} > \frac{1}{2}$. Y lo mismo si $x \notin B$. Luego con el procedimiento de reducción de error (Teorema 66) se puede llevar la constante de $\frac{5}{9}$ a $\frac{2}{3}$.

Si bien **BPP** es una clase natural, difiere en muchos sentidos a otras clases que hemos visto. Las máquinas que deciden sus lenguajes pueden errar y no se conoce un lenguaje completo ni un resultado de jerarquía. Sin embargo, como se cree que $\mathbf{P} = \mathbf{BPP}$ se cree que **BPP** tiene un lenguaje completo (ya que **P** lo tiene) y que hay un teorema de jerarquía para **BPTIME** (ya que **DTIME** lo tiene). Está de más decir que encontrar un lenguaje completo o un resultado de jerarquía sería un logro extraordinario para la teoría. Un candidato natural de un lenguaje así es el conjunto de las triplas $(M, x, 1^n)$, tal que la probabilidad de que la MT M a partir de la entrada x devuelva 1 en a lo sumo n pasos es al

menos $\frac{2}{3}$. Tal lenguaje es BPP-hard pero no se sabe si pertenece a **BPP**, y de hecho se cree que no ya que sino la jerarquía polinomial colapsaría. Preguntas como: ¿$\mathbf{BTIME}(n) \subsetneq \mathbf{BTIME}(n^2)$? No se han dejado contestar usando diagonalización que sí funcionó para sus primas **DTIME** y **NTIME**. Pero bueno, más allá de que estos problemas sigan hoy sin solución, recientes esfuerzos en encontrar teoremas de jerarquía han dado frutos para una clase muy cercana, la clase **BPP/1** (ver [13, 17, 20]). Que a grandes rasgos es **BPP** con 1 bit de *advice* (la misma noción de advice que vimos en el capítulo de circuitos, ver Definición 62). Y además, aparentemente las dificultades que tenemos para resolver estos problemas en **BPP**, desaparecen en la generalización de **BPP** a *promise problem* (ver página 504 de [8]). Por lo que no hay que bajar los brazos todavía, queda mucho por descubrir.

Capítulo 6

Protocolos Interactivos: IP

> ¿Qué es intuitivamente requerido en un procedimiento demostrador de teoremas? Primero, que le sea posible "probar" un teorema verdadero. Segundo, que le sea imposible "probar" un falso teorema. Tercero, que la comunicación de la prueba sea eficiente, en el siguiente sentido. No importa cuánto tiempo le lleva al "probador" encontrar la prueba, pero es esencial que su verificación sea fácil.
> GoldWasser, Micali, y Rackoff, 1985

La noción de demostración matemática estándar está relacionada íntimamente a la noción de certificados que tenemos de la definición alternativa de **NP**. Y lejos está de ser una cuestión filosófica, como vimos en Preliminares, dada una teoría Λ (por ejemplo la aritmética de Peano o la teoría de conjuntos ZF) el lenguaje
$\{(\varphi, 1^n) : \varphi$ tiene una demostración formal en Λ de largo menor o igual a $n\}$ es NP-completo. Ahora entenderemos el certificado de una manera totalmente distinta porque entenderemos de manera distinta la computación en general. Supongamos que Kurt quiere convencer a John de que la fórmula booleana φ es satisfacible. Kurt emplea una máquina de poder exponencial que le da una asignación u que hace verdadera a la fórmula. Luego le envía u a John y este verifica con su modesta máquina polinomial que $\varphi(u) = 1$. Entonces el algoritmo no determinista para decidir `SAT` puede ser pensado como un *protocolo,* un procedimiento interactivo en el cual actúan actores que intercambian mensajes. En el ejemplo recién mostrado tenemos un actor Kurt que quiere probar algo, que llamaremos *prover*; un actor John que lo quiere verificar, que lo llamaremos *verificador*; y el *mensaje u* que le envía el primero al segundo.

En este capítulo estudiaremos el poder computacional de los *protocolos interactivos,* sus aplicaciones son muy importantes en la criptografía pero también aparecen en muchos otros campos de la teoría de complejidad computacional. Este tópico tiene una gran importancia en los límites del poder de la aproximación de algoritmos, en el campo de program checking, da evidencia de que problemas famosos como el isomorfismo de grafos no es NP-completo, y se obtiene el teo-

rema *PCP* que es muy conocido e importante en toda la teoría. Queda fuera de este trabajo tal teorema porque merecería una tesis entera. Como si fuera poco, veremos una nueva técnica de demostración conocida como *aritmetización* que parece escapar de la relativización de los argumentos de diagonalización, y que le dio a la teoría nuevos resultados que solo con la otra técnica jamás se hubieran descubierto. El famoso teorema *PCP* utiliza esta técnica pero la veremos en el teorema **PSPACE** $\subseteq$ **IP** (Teorema 79) también importante y relativamente reciente, que en su momento fue muy sorprendente que da una nueva caracterización de **PSPACE** a partir de protocolos interactivos.

Antes de ir a las definiciones daremos otro ejemplo simple de un protocolo interactivo para que el lector vaya intuyendo los conceptos que van a tener las definiciones. Y para cuando la vea empleada en distintos problemas pueda identificar cada uno de los actores sin confundirse, sobre todo al principio. El ejemplo es una adaptación de un problema muy conocido en el campo entre dos personajes ficticios, Arthur y Merlín. El siguiente problema se presenta a modo de "acertijo" para que el lector si quiere, pueda entretenerse tratando de resolverlo.

Ejemplo. Medias de colores

Arthur tiene un problema en la vista que hace que vea todos los colores como gris. Hoy como tiene una fiesta se vistió elegante pero se equivocó porque se puso una media de color rojo y otra de color azul, muy ridículo! Su amigo Merlín lo vio y lo alertó pero no le creyó! ¿Cómo puede hacer Merlín para convencerlo de que tiene medias de colores diferentes? Se invita al lector a tratar de hacer un procedimiento interactivo para que Merlín convenza a Arthur de que tiene medias de colores distintos. Hint: Arthur se puede sacar las medias y puede usar una moneda.

Solución: Arthur se saca las medias, toma una con la mano izquierda y la otra con la derecha. Le pregunta de qué color es la media que tiene en cada mano y Merlín le responde. Luego se realiza el siguiente experimento: Merlín cierra los ojos y Arthur tira la moneda, si sale cruz deja las medias como están de lo contrario toma con su mano derecha la media que tiene en la mano izquierda y viceversa, luego le indica que abra los ojos y que le diga el color de la media que tiene en cada mano. Si se equivoca entonces se convence de que Merlín miente y que en realidad las medias son del mismo color, pero si acierta entonces repite el experimento. Arthur puede determinar si su amigo se equivoca o no, porque conoce el movimiento que hizo y porque conoce, por hipótesis, el color de cada media al inicio del procedimiento. Si las medias fueran del mismo color, de repetir el experimento n veces, la probabilidad de que Merlín no se equivoque es $(\frac{1}{2})^n$. Por lo tanto repitiendo el experimento, digamos, unas 10 veces Arthur puede convencerse de si su amigo miente o no.

En este ejemplo el verificador es Arthur porque verifica que las respuestas de Merlín son correctas. Éste último es el prover ya que trata de demostrarle algo, y los mensajes entre ellos son los mensajes de la interacción. Se observa

que Merlín tiene "más poder" que él, ya que puede ver muchos colores y él solo gris. No limitaremos el poder computacional del prover, no nos interesará cómo realiza ni cuánto le lleva realizar una computación para dar un mensaje. Pero sí limitaremos el poder computacional del verificador, esté solo correrá en tiempo polinomial. En la introducción del capítulo el verificador era determinista y en este ejemplo es probabilístico, recordando una MT es un caso especial de una MTP, en la definición el verificador va a poder ser probabilístico.

Hay muchas maneras distintas y equivalentes de definir a los protocolos interactivos, en el sentido que todas modelan a la misma clase **IP**. A partir del estudio del capítulo Interactive Protocols de [8], del capítulo Probabilistic Proof Systems de [19], de la sección 10.4 de [37] y del trabajo [28] formamos la definición que usaremos de protocolo interactivo; "entre todas hicimos la nuestra rescatando lo mejor de cada una". La definición que veremos modela la misma clase que las definiciones de los citados trabajos pero creemos que la nuestra es más didáctica.

A continuación definiremos los mensajes, el prover y el verificador.

Definición 74. Sea $k \geq 0$, *un* $k-$ *historial de mensajes es una sucesión* $m_1, m_2, ..., m_k$ tal que $m_i \in \{0,1\}^*$ (si $k = 0$ entonces el 0-historial de mensajes es la palabra vacía ϵ). Decimos que un *prover* P es una función $P : \{0,1\}^* \longrightarrow \{0,1\}^*$ tal que existe un polinomio q tal que para todo $\alpha \in \{0,1\}^*$ $|P(\alpha)| \leq q(|\alpha|)$. Un *verificador* V es una MTP polinomial. Una *entrada* $\alpha \in \{0,1\}^*$ para el verificador es la codificación de una tupla $(x, \text{mensajes})$ donde mensajes es un k-historial de mensajes.

A continuación se define el *protocolo interactivo*, que lo llamaremos *interacción.*

Definición 75 (Interacción)**.** Dado $k \geq 0$, P prover, un verificador V y un polinomio q, diremos que una $k-$*interacción entre* V *y* P *para una entrada* $x \in \{0,1\}^*$ denotada $< V, P >_k (x)$, es un $k-$historial de mensajes $m_1, ..., m_k$ que cumple:

1. Para toda $\alpha \in \{0,1\}^*$ que recibe V en la interacción (entradas de la forma $(x, \text{mensajes})$) se cumple que V corre en $q(|x|)$ pasos.

2. $V(x, m_1, ..., m_k)$ es un 0 o 1.

3. Los mensajes son de la siguiente forma

$$m_1 = V(x)$$

$$m_i = \begin{cases} V(x, m_1, ..., m_{i-1}) & \text{si } i \leq k \text{ y } i \text{ es impar} \\ P(x, m_1, ..., m_{i-1}) & \text{si } i \leq k \text{ y } i \text{ es par} \end{cases}$$

Llamamos a $V(x, m_1, ..., m_k)$ la *salida de la interacción* y la denotamos con $out(< V, P >_k (x))$.

A continuación se define la clase más importante del capítulo, **IP**, de "interactive protocols". Que representa a los lenguajes para los cuales hay un protocolo interactivo de tamaño polinomial que lo decide.

Definición 76 (La clase **IP**)**.** Dada $T : \mathbb{N} \longrightarrow \mathbb{N}$ y un lenguaje $L \subseteq \{0,1\}^*$, decimos que

$L \in \mathbf{IP}[T]$ si existe un verificador V y una constante k tal que para todo $x \in \{0,1\}^*$ se cumple:

$$\text{(Completeness) } \exists P \text{ prover } : \text{ si } x \in L \Rightarrow Pr[output(< V, P >_{kT(|x|)} (x)) = 1] \geq \frac{2}{3}$$

$$\text{(Soundness) } \forall P \text{ prover } : \text{ si } x \notin L \Rightarrow Pr[output(< V, P >_{kT(|x|)} (x)) = 1] \leq \frac{1}{3}$$

Definimos $\mathbf{IP} = \bigcup_{c \geq 1}^{\infty} \mathbf{IP}[n^c]$.

Observación. Sobre la robustez de la definición, se puede ver con ideas similares al Teorema de reducción de error en **BPP** (ver Teorema 66) que **IP** sería la misma clase de utilizar $1 - 2^{-n^s}$ como la constante de la primera propiedad y 2^{-n^s} como la constante de la segunda, para cualquier $s \in \mathbb{N}$. Curiosamente cambiando $\frac{2}{3}$ por 1 en la propiedad *Completeness* también se define la misma clase. Y cambiando $\frac{2}{3}$ por 0 en la propiedad *Soundness* se define **NP**.

Vale agregar la definición de la famosa clase **AM** conocida como *interactive proofs with public coins.* Para toda $T : \mathbb{N} \longrightarrow \mathbb{N}$ definimos $\mathbf{AM}[T]$ como el subconjunto de $\mathbf{IP}[T]$ resultado de restringir al verificador para que solo pueda enviar mensajes aleatorios y para que solo pueda obtener "aleatoriedad" a partir de ellos. **AM** se define como $\mathbf{AM}[2]$.

Ahora veamos un protocolo de IP

Ejemplo. Isomorfismo de Grafos

En esta ocasión Arthur quiere convencer a Merlín de que dos grafos no son isomorfos. Recordando: intuitivamente, dos grafos son isomorfos si "estructuralmente" son iguales. Formalmente: sean $G = (V, E)$, $G_2 = (V_2, E_2)$ grafos, decimos que son isomorfos $G \cong G_2$ si existe una función biyectiva $f : V \longrightarrow V_2$ que cumple:

$$\forall u, v \in V : (u, v) \in E \iff (f(u), f(v)) \in E_2.$$

Por otro lado, vale también que dos grafos son isomorfos si y solo si existe una permutación de los "nombres" de los vértices que aplicándola a uno de los grafos hace que sean iguales. En el protocolo decidiremos el isomorfismo de grafos a partir de esa idea. Dado un grafo $G = (V, E)$, una permutación de G, $\Phi(G)$, es un grafo (V, E_2) para el cual existe una función biyectiva $h : V \longrightarrow V$ que cumple:

$$\forall u, v \in V : (u, v) \in E \iff (h(u), h(v)) \in E_2.$$

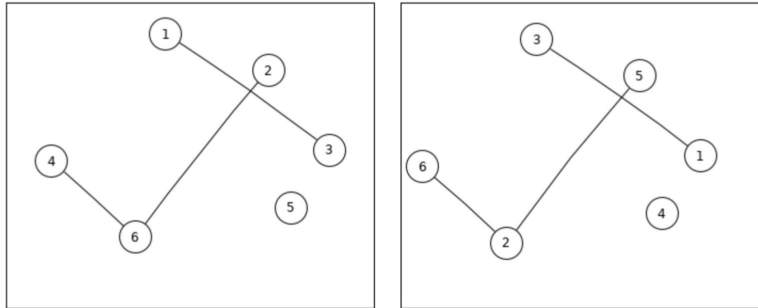

Figura 6.1: dos grafos isomorfos

Sean $G_1 = (V, E)$, $G_2 = (V_2, E_2)$ grafos. Como precondición asumiremos que $|V| = |V_2|$ y $|E| = |E_2|$ (si no se cumpliera alguna entonces sería trivial y polinomial determinar que no son isomorfos). De cumplirse ambas, también asumiremos que $V = V_2$ ya que si no se cumpliera polinomialmente se puede renombrar G_2 a partir de V. Es decir si $V = v_1, \ldots, v_n$ y $V_2 = u_1, \ldots, u_n$ entonces se puede realizar polinomialmente el siguiente reemplazo sintáctico. Cada ocurrencia de u_1 en G_2 se reemplaza por v_1 y en general cada ocurrencia de u_i en G_2 se reemplaza por v_i. La interacción tiene la siguiente forma:

V: Le asigna 0 a una variable llamada *contador*. Toma $i \in \{0, 1\}$ aleatoriamente. Aleatoriamente permuta G_i obteniendo un nuevo grafo $H = \Phi(G_i)$. Le envía H a P.

P: Identifica qué grafo de G_1, G_2 fue utilizado para crear a H. Sea G_j tal grafo, le envía j a V.

V: Si $i = j$ asigna 1 a la variable *contador* sino deja *contador* como esta. Luego repite lo que hizo anteriormente tomando un nuevo i y enviándole un nuevo grafo H a P. (Pero sin asignarle 0 a la variable *contador*).

P: Repite lo que hizo enviándole un nuevo j a V.

V: Si $i = j$ suma 1 a la variable *contador*. Si *contador*$= 2$ entonces acepta, caso contrario rechaza.

Observemos que este protocolo satisface la definición, si $G_1 \ncong G_2$ entonces existe un prover tal que $Pr[V \text{ acepta}] = 1$ ya que si los grafos no son isomorfos, un "todo poderoso" prover siempre determina el G_i a partir del cual se hizo H. Por otro lado, si $G_1 \cong G_2$ entonces lo mejor que puede hacer un prover para determinar el j es hacerlo aleatoriamente. Ya que una permutación aleatoria de G_1 luce exactamente igual a una permutación aleatoria de G_2. Por lo tanto en este caso, para cualquier prover se tiene que $Pr[V \text{ acepta}] \leq \frac{1}{4}$.

Teniendo en cuanta que un prover podría no ser computable en tiempo polinomial, o incluso no ser computable, no queda claro que "tan grande" es **IP** y por ende que clases lo "acotan". El siguiente teorema lo esclarece.

Teorema 77. $\mathbf{NP} \subseteq \mathbf{IP} \subseteq \mathbf{PSPACE}$.

Para ver $\mathbf{IP} \subseteq \mathbf{PSPACE}$ nos basaremos en la didáctica prueba de $\mathbf{PSPACE} \subseteq \mathbf{IP}$ de [28]. La principal idea es que dado un lenguaje $L \in \mathbf{IP}$, para decidir si $x \in \{0,1\}^*$ pertenece o no a L, va a existir un verificador V y por ende un polinomio q tal que sea cual sea el prover, sean cuales sean los mensajes que pueda enviar un prover y sean cuales sean las respuestas del verificador, éste va a correr en $q(|x|)$ pasos para cualquier α que reciba de las interacciones que puedan ocurrir. Por lo tanto hay un espacio de posibles de interacciones para V acotado por $2^{q(|x|)}$. Cada una de tales interacciones explorable en espacio $\mathcal{O}(q(|x|))$. Con ello en mente una máquina usando espacio $\mathcal{O}(q(|x|))$ puede llevar la cuenta de los caminos que dan como salida 1 ($output(< V, P >_{kT(|x|)} (x)) = 1$) y también llevar la cuenta de la cantidad de caminos posibles que es $\mathcal{O}(2^{q(|x|)})$. Luego el cociente entre ambas cantidades determina la proporción de interacciones que llevan al verificador a aceptar x decidiendo la pertenencia de x en el lenguaje. Vale observar que como esas cantidades que se calculan son $\mathcal{O}(2^{q(|x|)})$, se las puede representar en base 2 en espacio $\mathcal{O}(q(|x|))$. Ahora sí, veamos la prueba.

Demostración. Primero veamos que $\mathbf{NP} \subseteq \mathbf{IP}$, y para ello que $\texttt{SAT} \in \mathbf{IP}[2]$. La demostración es la formalización de la interacción entre Kurt y John vista en la introducción. Sea M una MT que decide SAT que corre en tiempo p para algún polinomio, y sea q el polinomio que acota a los certificados de M. Es decir:

$$\forall \varphi \in \{0,1\}^* \ \varphi \in \texttt{SAT} \text{ sii } \exists u \in \{0,1\}^{q(|\varphi|)} M(\varphi, u) = 1.$$

Exijamos sin pérdida de generalidad que $\forall n \ q(n) \leq p(n)$. Queremos ver que existe un verificador V que cumple la definición. Definamos V de la siguiente manera: si V recibe una sola entrada entonces devuelve 0, en cambio si recibe tres entradas (φ, m_1, m_2) entonces si $m_2 \in \{0,1\}^{q(|\varphi|)}$ devuelve $M(\varphi, m_2)$ caso contrario devuelve 0. Entonces dado un prover P y una entrada φ, la 2-interacción tiene la siguiente forma:

V: Recibe φ de entrada y envía el bit 0.

P: Envía un mensaje m_2.

V: Si $m_2 \in \{0,1\}^{q(|\varphi|)}$ computa y devuelve $M(\varphi, m_2)$, caso contrario devuelve 0.

Definamos un prover P de la siguiente manera: dada una entrada $(\varphi, 0)$, si $\varphi \in \texttt{SAT}$ entonces $P(\varphi, 0)$ devuelve un certificado u para φ, es decir un u que cumple $M(\varphi, u) = 1$. Por otro lado, si $\varphi \notin \texttt{SAT}$ entonces devuelve 0.
Sea $\varphi \in \{0,1\}^*$ supongamos que $\varphi \in \texttt{SAT}$ luego existe u certificado para φ es decir, $\exists u \in \{0,1\}^{q(|x|)} M(\varphi, u) = 1$ por lo tanto m_2 es un certificado para φ (con $P(\varphi, 0) = m_2$) y por ende $out(< V, P >_2 (\varphi)) = 1$. En consecuencia, $Pr[output(< V, P >_2 (\varphi)) = 1] = 1 \geq \frac{2}{3}$. Ahora supongamos que $\varphi \notin \texttt{SAT}$ luego para todo m_2 que pueda devolver cualquier prover se tiene que $V(\varphi, 0, m_2) = 0$ por lo tanto $out(< V, P >_2 (\varphi)) = 0$. En consecuencia,

$$\forall P \text{ prover } : \text{ si } \varphi \notin \texttt{SAT} \Rightarrow Pr[output(< V, P >_2 (\varphi)) = 1] = 0 \leq \frac{1}{3}.$$

Ahora veamos que $\mathbf{IP} \subseteq \mathbf{PSPACE}$. Sea $L \in \mathbf{IP}$ luego existe un polinomio T tal que $L \in \mathbf{IP}[T]$. Sea V el verificador que lo decide, q el polinomio que acota las interacciones en función de x y k la constante de la definición. Queremos ver que podemos decidir L usando espacio $\mathcal{O}(q(|x|))$. La idea es que dada una entrada x usando espacio polinomial vamos a maximizar z con
$z = Pr[output(< V, P >_{kT(|x|)} (x)) = 1]$ y este z va a cumplir $x \in L$ sii $z \geq \frac{2}{3}$. Esto último se da porque como L está en $\mathbf{IP}$ nunca dará el caso $\frac{1}{3} < z < \frac{2}{3}$.
Sea x una entrada luego se cumple:

$$\exists P \text{ prover } : \text{ si } x \in L \Rightarrow Pr[output(< V, P >_{kT(|x|)} (x)) = 1] \geq \frac{2}{3}$$

$$\forall P \text{ prover } : \text{ si } x \notin L \Rightarrow Pr[output(< V, P >_{kT(|x|)} (x)) = 1] \leq \frac{1}{3}$$

Ahora necesitamos mostrar que es posible computar z en espacio polinomial. Observemos que $output(< V, P >_{kT(|x|)} (x))$ es equivalente a $V(x, m_1, \ldots, m_{kT(|x|)})$ y sabemos que V para tal entrada corre en $q(|x|)$ pasos. Podemos simular recursivamente cada posible mensaje de V (m_i con i impar menor o igual a $kT(|x|)$) y cada posible m_i con i par que sería cada posible mensaje de algún prover. Vale

aclarar que no computamos prover alguno, directamente simulamos cada posible mensaje que un prover cualquiera puede dar ya que cada uno está acotado por $q(|x|)$. Observemos que cada camino tiene un largo en espacio acotado por $q(|x|)$, por lo podemos recorrer cada camino en espacio $q(|x|)$. Definamos un contador c_1 que cuente la cantidad de caminos que llevan a que $V(x, m_1, \ldots, m_{kT(|x|)}) = 1$. Notemos que $c_1 \leq 2^{q(|x|)}$ y que por ende su representación puede codificarse en base 2 en espacio $q(|x|)$. Y definamos otro contador c_2 que cuente la cantidad total de caminos posibles en la interacción, análogamente su representación puede efectuarse en espacio $q(|x|)$. Luego $z = \frac{c_1}{c_2}$ y lo hemos calculado usando espacio polinomial, por lo tanto $L \in \mathbf{PSPACE}$. □

Aritmetización y PSPACE $\subseteq$ IP.

Como hemos ido mencionando, en este capítulo introducimos una nueva técnica que tiene la teoría de complejidad computacional llamada *aritmetización.* Ésta permitió demostrar el teorema nombrado en el título de esta sección, el teorema *PCP* y otros muy importantes que fueron muy sorprendentes en los noventas. Esta técnica parece escapar de la limitación llamada relativización y es de una naturaleza distinta a la limitación vista en el capítulo de Circuitos llamada *Natural proofs.* Pero lamentablemente no permitió resolver los grandes problemas del campo y con el tiempo la comunidad se volvió a estancar. Finalmente se "justificó" o "formalizó" tal estancamiento a partir de su respectiva limitación conocida como *algebrización.* Esta limitación se analizará en la próxima sección. La presente sección se basa principalmente en la subsección 8.3 de [8]. Vale agregar que el enunciado y la demostración del teorema *PCP* queda fuera del alcance del trabajo, el lector interesado puede ver los capítulos 11 y 22 de [8] y los trabajos [10, 11].

Yendo al meollo de la cuestión, la *aritmetización* consiste en transformar una fórmula booleana φ de n variables en un polinomio multivariable de n variables equivalente, p_φ, tal que toda asignación $u \in \{0,1\}^n$ cumpla $\varphi(u) = p_\varphi(u)$. Una vez que tenemos p_φ nada nos impide evaluarlo en otros valores distintos del cero y del uno, digamos, evaluarlo en otro cuerpo como $\mathbb{R}$ o en cuerpos finitos. Esta acción le dará un sorprendente poder a los verificadores.

Cuando las variables toman $0, 1$ podemos transformar los $\wedge$ en . (producto) y $\neg$ en la resta ($\neg x$ en $1 - x$). Por dar algunos ejemplos veamos las siguiente correspondencias entre fórmulas y polinomios cuando las variables toman cero o uno.

$$x \wedge y \quad \longleftrightarrow \quad x.y$$
$$\neg x \quad \longleftrightarrow \quad 1 - x$$
$$x \vee y \quad \longleftrightarrow \quad 1 - (1-x)(1-y)$$
$$x \vee y \vee \neg z \quad \longleftrightarrow \quad 1 - (1-x)(1-y)z$$

La transformación es directa, dada φ una 3CNF de n variables y m cláusulas definimos $p_\varphi = \prod_{i\geq 1}^{m} p_i$ con p_i un polinomio de n variables equivalente a la i-ésima cláusula de m. Esta equivalencia nace a partir de la correspondencia recién vista. Se observa que cada p_i tiene al menos $n-3$ variables que no "usa" ya que como φ está en 3CNF cada cláusula tiene a lo sumo tres variables. Por ejemplo si la j-ésima cláusula de φ es $x_2 \vee \neg x_3 \vee \neg x_9$ entonces $p_j(x_1, x_2, \ldots, x_9, \ldots, x_n) = 1 - (1 - x_2)x_3x_9$. Otra observación importante es que el grado del polinomio es menor o igual a $3m$. Representaremos p_φ sin "abrir los paréntesis", por lo tanto p_φ tiene una representación de tamaño $\mathcal{O}(m)$.

A continuación utilizaremos la aritmetización para probar un teorema también importante cuya demostración nos servirá como "entrada en calor" para probar **PSPACE** $\subseteq$ **IP**.

Definamos el lenguaje #`SAT` como el conjunto de pares (φ, K) tal que φ es una 3CNF con exactamente K asignaciones que la satisfacen. Por ejemplo $\overline{\texttt{3SAT}} \times \{0\} \subseteq \#\texttt{SAT}$ ya que $\varphi \in \overline{\texttt{3SAT}}$ sii φ tiene exactamente cero asignaciones que la satisfacen. Como un dato curioso, #`SAT` es un problema completo (en el sentido de reducción de Karp, Definición 12) para una clase muy "poderosa" llamada #**P** en la cual está incluida por ejemplo **PH**. A continuación daremos un protocolo interactivo polinomial para #`SAT` mostrando que pertenece a **IP** y por lo tanto mostrando que #**P** $\subseteq$ **IP**. El lector interesado puede ver el trabajo original [27].

Teorema 78. #`SAT` $\in$ **IP**

Demostración. Sea φ una 3CNF de n variables, m cláusulas y K un entero. Sea p_φ el polinomio multivariable construido por aritmetización a partir de φ. Como vimos se cumple que $\forall u \in \{0,1\}^n\ \varphi(u) = p_\varphi(u)$, luego si $\#\varphi$ es la cantidad de asignaciones que la satisfacen entonces se cumple también:

$$\#\varphi = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} p_\varphi(b_1, \ldots, b_n). \tag{6.1}$$

El prover va tratar de convencer al verificador de que dicha suma es exactamente K, y a partir de ahora, nos podemos olvidar de φ y pensar en términos de p_φ. En el primer paso el prover le envía al verificador un número primo q en el intervalo $(2^n, 2^{2n}]$. El verificador chequea si q es primo con algún test de primalidad eficiente. Notemos que necesariamente $0 \leq \#\varphi \leq 2^n$, entonces como $q > 2^n$ se tiene que la ecuación (6.1) se cumple sobre $\mathbb{Z}$ si y solo si se cumple para $\mathbb{Z}_q$, el cuerpo de enteros módulo q. Por lo tanto pensaremos todas las operaciones sobre $\mathbb{Z}_q$. Probaremos el teorema dando un protocolo para verificar ecuaciones como (6.1).

Dado un polinomio multivariable $g(x_1, \ldots, x_n)$ de grado d, un entero K y un primo q, mostremos un protocolo interactivo para decidir

$$K = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \ldots \sum_{b_n \in \{0,1\}} g(b_1, \ldots, b_n). \tag{6.2}$$

donde todas las operaciones son módulo q. Los únicos requerimientos para g es que sea representable en tamaño polinomial, y que pueda ser evaluado en tiempo polinomial para cualquier valor de las variables en el cuerpo, el cual es en este caso $\mathbb{Z}_q$. Naturalmente, estos requerimientos son satisfechos por p_φ. Notemos que para cualquier asignación de valores $b_2, \ldots, b_n$ a las variables $x_2, \ldots, x_n$ se tiene que $g(x_1, b_2, \ldots, b_n)$ es un polinomio de una sola variable de grado d. Por lo tanto el siguiente polinomio también tiene grado d y es de una sola variable:

$$h(x_1) = \sum_{b_2 \in \{0,1\}} \ldots \sum_{b_n \in \{0,1\}} g(x_1, b_2, \ldots, b_n). \tag{6.3}$$

Si (6.2) es cierto entonces necesariamente $h(0) + h(1) = K$. A continuación mostraremos el protocolo *Sumcheck* para chequear (6.2), sea V el verificador y P el prover.

V : Si $n = 1$ chequea que $g(0) + g(1) = K$, si se cumple acepta sino rechaza. Si $n \geq 2$ le dice a P que le envíe el respectivo $h(x_1)$ como el definido en (6.3).

P : Le envía algún polinomio $s(x_1)$ (si P no está mintiendo entonces efectivamente $s(x_1) = h(x_1)$).

V : Va a tratar de verificar si efectivamente $s(x_1) = h(x_1)$. Lo primero que hace es chequear que $s(0)+s(1) = K$ (si no lo cumple automáticamente rechaza). Luego toma aleatoriamente un $a \in \mathbb{Z}_q$ y recursivamente usa este protocolo para chequear

$$s(a) = \sum_{b_2 \in \{0,1\}} \ldots \sum_{b_n \in \{0,1\}} g(a, b_2, \ldots, b_n).$$

Observemos que definiendo $l(x_2, \ldots, x_n) = g(a, x_2, \ldots, x_n)$ tenemos que l es un polinomio multivariable de $n-1$ variables que cumple todos los requerimientos. Por ende es válido dar de entrada $s(a)$ y l al protocolo recursivamente.

Analicemos: si efectivamente se cumple

$$K = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \ldots \sum_{b_n \in \{0,1\}} g(b_1, \ldots, b_n).$$

Entonces existe un prover P que siempre brinda el correspondiente h en cada iteración del protocolo. Por lo tanto la probabilidad de aceptación es 1 que es mayor a $\frac{2}{3}$.

Por otro lado, si no se cumple entonces V rechaza con probabilidad mayor o igual a $(1-\frac{d}{p})^n$. Lo demostremos por inducción.

Caso $n = 1$. Supongamos que $K \neq \sum_{b_1\in\{0,1\}} g(b_1)$ luego el verificador simplemente chequea que $K \neq g(0) + g(1)$ y rechaza.

Caso inductivo. Supongamos que

$$K \neq \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} \sum_{b_{n+1}\in\{0,1\}} g(b_1, \ldots, b_n, b_{n+1}).$$

Luego P le envía un polinomio $s(x_1)$. El verificador chequea $s(0)+s(1) = K$ (si no se cumple lo rechaza) supongamos que lo cumple. Luego $s(x_1)$ es diferente a $h(x_1)$ en consecuencia el polinomio $s(x_1) - h(x_1)$ no es el polinomio nulo y tiene a lo sumo d raíces. Osea, hay a lo sumo d valores a tal que $s(a) = h(a)$. Entonces cuando V elije $a \in \mathbb{Z}_q$ aleatorio se cumple

$$Pr[s(a) \neq h(a)] \geq 1 - \frac{d}{q}.$$

Por lo tanto si el a elegido es uno tal que $s(a) \neq h(a)$ entonces en el paso recursivo se tratará de probar que $s(a) = g(a, x_2, \ldots, x_n, x_{n+1})$, hecho falso. Como este nuevo polinomio tiene n variables, por la hipótesis inductiva la probabilidad de que V rechace es de al menos $(1-\frac{d}{p})^n$. Por lo tanto

$$Pr[V \text{ rechaza}] \geq (1-\frac{d}{q}).(1-\frac{d}{p})^n = (1-\frac{d}{p})^{n+1} \geq \frac{2}{3}.$$

□

A continuación adaptaremos la anterior prueba para dar un protocolo interactivo a `TQBF`. Recordando: `TQBF` es el lenguaje de fórmulas booleanas cuantificadas (ver Definición 23) y cumple que es PSPACE-completo (ver Definición 22), por lo tanto demostrando que `TQBF` $\in$ **IP** demostramos **PSPACE** $\subseteq$ **IP**. El lector interesado puede ver el trabajo original [32] y también [33] donde se lo demuestra con el operador lineal que veremos.

Teorema 79. PSPACE $\subseteq$ IP

Demostración. Dada una fórmula Booleana cuantificada $\Psi = \forall x_1 \exists x_2 \forall x_3 \ldots \exists x_n \varphi(x_1, \ldots, x_n)$, creamos su respectivo polinomio multivariable mediante aritmetización, p_φ. Luego $\Psi \in$ `TQBF` si y solo si

$$\prod_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \prod_{b_3\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} p_\varphi(b_1, \ldots, b_n) \neq 0. \qquad (6.4)$$

La primera impresión es que podríamos haber usado exactamente el mismo protocolo usado en el teorema anterior, la diferencia es que en este caso usamos el productorio para las variables cuantificadas con $\forall$. Vamos a chequear $s(0).s(1) = K$, y vamos a hacer algo análogo con las demás variables cuantificadas con $\exists$. Se verá que no hay nada malo con esto, solo el tiempo de ejecución,

ya que el producto a diferencia de la suma aumenta el grado del polinomio. Si definiéramos $h(x_1)$ análogamente a (6.3) dejando como variable libre a x_1 en (6.4) entonces su grado aumentaría tal vez hasta 2^n. Tal polinomio podría tener 2^n coeficientes y por ende el prover no se lo podría transmitir al verificador. La solución a este problema es notar que (6.4) se evalúa en $\{0,1\}$ entonces para cada $x \in \{0,1\}$ se tiene $x^k = x$ para todo $k \geq 1$. Luego en principio podemos convertir todo polinomio $p(x_1,\ldots,x_n)$ en un polinomio multilineal $l(x_1,\ldots,x_n)$ (i.e, el grado de l es a lo sumo 1 en cada variable) que coincide con p en toda asignación $x_1,\ldots,x_n \in \{0,1\}$. En ese cometido definimos el operador L_i sobre polinomios de la siguiente manera:

Sea p un polinomio de n variables definimos:

$$L_i(p)(x_1,\ldots,x_n) = x_i.p(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$$
$$+\ (1-x_i).p(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n).$$

Luego $L_i(p)$ es lineal en x_i y coincide con p en cualquier valor de las variables mientras x_i esté en $\{0,1\}$. Por ende, $L_1(L_2(\ldots(L_n(p)\ldots)$ es un polinomio multilineal que coincide con p en todas las posibles asignaciones de las variables en $\{0,1\}$.

Por lo tanto podemos pensar en $\forall x_i$ y $\exists x_i$ como operadores sobre polinomios donde:

$$\forall x_i p(x_1,\ldots,x_n) = p(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n).p(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n).$$

$$\exists x_i p(x_1,\ldots,x_n) = p(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)+p(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n).$$

Finalmente, podemos repensar (6.4) de la siguiente manera: si aplicamos la secuencia de operadores $\forall x_1 \exists x_2 \forall x_3 \ldots \exists x_n$ sobre el polinomio $p_\varphi(x_1,\ldots,x_n)$ (donde primero aplicamos $\exists x_n$ y por último $\forall x_1$) entonces obtenemos el valor no nulo. Como se ha observado, solo nos interesa el resultado al que se llega cuando las variables toman 0 o 1 por lo tanto podemos utilizar una cantidad arbitraria de operadores L_i y el resultado no va a variar. En particular vamos a utilizar los operadores para que todo polinomio resultante de algún paso intermedio del protocolo Sumcheck tenga un grado bajo. Específicamente, utilizaremos la siguiente expresión:

$$\forall x_1 L_1 \exists x_2 L_1 L_2 \forall x_3 L_1 L_2 L_3 \ldots \exists x_n L_1 L_2 L_3 \ldots L_n\ p_\varphi(x_1,\ldots,x_n).$$

Observemos que el tamaño de la expresión es $\mathcal{O}(1+2+3+\ldots+n) = \mathcal{O}(n^2)$.

En la adaptación del protocolo *Sumcheck* así como tenemos una ronda para cada operador $\sum$ también tendremos una para cada $\prod$ y otra para cada L_i (donde en cada una el prover le enviará un polinomio de grado menor o igual a 2). A continuación damos una descripción inductiva del protocolo. Supongamos que para un polinomio $g(x_1,\ldots,x_n)$ el prover tiene la capacidad de convencer al verificador que $g(a_1,\ldots,a_n) = K$ con probabilidad 1 para cualquier $a_1,\ldots,a_n,K$ que lo cumpla. Y con probabilidad de a lo sumo ϵ para aquellos que no. Sea $u(x_1,\ldots,x_l)$ cualquier polinomio de l variables obteniendo como

$$u(x_1,\ldots,x_l) = \varDelta g(x_1,\ldots,x_n)$$

donde Δ es $\exists x_i$ o $\forall x_i$ o L_i para algún $i \in \{1, \ldots, n\}$. Por ende, l es $n-1$ en los primeros dos casos y n en el último. Sea d una cota superior del grado de u sobre la variable x_i conocida por el verificador (en nuestro caso $d \leq 3m$). Mostraremos que el prover puede convencer al verificador de que $u(a_1, \ldots, a_l) = K'$ con probabilidad 1 para cualesquiera valores $a_1, \ldots, a_l, K'$ que sea cierto y con probabilidad menor o igual a $\epsilon + \frac{d}{q}$ cuando no lo sea (con q el primo definido como en el protocolo *Sumcheck*). Para simplicidad supongamos que $i = 1$. El verificador realiza lo siguiente:

Caso 1: $\Delta = \exists x_1$. El prover le brinda un polinomio $s(x_1)$ de grado d que supuestamente es $g(x_1, a_2, \ldots, a_n)$. El verificador chequea que $s(0) + s(1) = K'$ (si no lo cumple automáticamente rechaza). Caso contrario, toma un $a \in \mathbb{Z}_q$ aleatorio y le pide al prover que demuestre que $s(a) = g(a, a_2, \ldots, a_n)$.

Caso 2: $\Delta = \forall x_1$. Igual que el caso anterior solo que el verificador chequea $s(0).s(1) = K'$ en lugar de $s(0) + s(1)$.

Caso 3: $\Delta = L_1$. Análogamente a los casos anteriores el prover envía un polinomio $s(x_1)$. El verificador chequea que

$$a_1 s(0) + (1 - a_1) s(1) = K'$$

si no lo cumple entonces lo rechaza. Caso contrario, toma un $a \in \mathbb{Z}_q$ aleatorio y le pide al prover que demuestre que $s(a) = g(a, a_2, \ldots, a_n)$.

La demostración de correctitud sigue la misma vista en $\#\texttt{SAT} \in \mathbf{IP}$. Otra vez la observación clave es que si $s(x_1)$ no es el polinomio correcto entonces con probabilidad $1 - \frac{d}{q}$ el prover falla en probar la falsa proposición de la próxima ronda. □

Algebrización e Independencia de P vs NP

Ya hemos mencionado que la limitación de la aritmetización es conocida como *algebrización*. El objetivo de esta sección es similar al de la sección de *Natural Proof*, y un estudio profundo del mismo queda fuera del alcance de este trabajo. La intención es que el lector se lleve una buena noción de la algebrización. Nos basaremos principalmente en [5].

Recordando, el núcleo de la aritmetización es dada una fórmula booleana, φ, reinterpretarla como un polinomio multivariable de grado bajo, p_φ, sobre algún cuerpo. Dado un lenguaje, por ejemplo SAT (aplica para cualquier lenguaje pero con este se nos hace más familiar), podemos reescribirlo por "capas". Para todo n sea $\texttt{SAT} - \texttt{n}$ el conjunto de las fórmulas booleanas satisfacibles de n variables. Luego podemos definir $\widetilde{\texttt{SAT} - \texttt{n}}$ como el conjunto de p_φ tal que $\varphi \in \texttt{SAT} - \texttt{n}$. Entonces la extensión de SAT denotada $\widetilde{\texttt{SAT}}$ es

$$\bigcup_{n \geq 0}^{\infty} \widetilde{\texttt{SAT} - \texttt{n}}.$$

El nombre de algebrización surge de *algebraica relativización*, la idea es que cuando se relativiza una inclusión de alguna clase de complejidad, a la máquina que simula no solo se le debe dar acceso a un oráculo A sino a una extensión A sobre un cuerpo finito. Para modelar eso, consideramos *oráculos algebraicos:* oráculos que pueden evaluar no solo una función Booleana φ sino también p_φ sobre algún cuerpo. Es decir que dado un lenguaje A la máquina tienen acceso oráculo a $\widetilde{A}$. Dadas C, D clases de complejidad, decimos que la inclusión $C \subseteq D$ algebriza sii $C^A \subseteq D^{\widetilde{A}}$ para todo oráculo A y para toda $\widetilde{A}$, extensión de A. Análogamente decimos que $C \not\subseteq D$ algebriza sii $C^{\widetilde{A}} \not\subseteq D^A$ para todos $A, \widetilde{A}$. El lector puede preguntarse por qué en tales definiciones solo una clase tiene el oráculo algebraico, y la respuesta lisa y llanamente es: porque funciona. A partir de este concepto de algebrización se prueba que todos los resultados conocidos que usan aritmetización que no relativizan, efectivamente algebrizan. Por ejemplo se puede ver que para todo oráculo A y para toda extensión de $\widetilde{A}$ se cumple:

$$\mathbf{PSPACE^A} \subseteq \mathbf{IP^{\widetilde{A}}}.$$

Por otro lado, bajo este concepto de algebrización, se ha mostrado con otros resultados que resolver los grandes problemas como **P** vs **NP** va a requerir técnicas que no algebrizan. Por dar algunos ejemplos, para todo oráculo A y para toda extensión de $\widetilde{A}$ se cumple:

- $\mathbf{NP}^{\widetilde{A}} \subseteq \mathbf{P}^A$
- $\mathbf{NP}^A \not\subseteq \mathbf{P}^{\widetilde{A}}$
- $\mathbf{PSPACE}^{\widetilde{A}} \subseteq \mathbf{P}^A$
- $\mathbf{NP}^A \not\subseteq \mathbf{BPP}^{\widetilde{A}}$
- $\mathbf{NP}^{\widetilde{A}} \subset \mathbf{SIZE}(n)^A$
- $\mathbf{NEXP}^{\widetilde{A}} \subset \mathbf{P}^A_{\mathbf{/poly}}$

En resumen: la algebrización nos provee una limitación de las técnicas que se basan en aritmetización. Entonces: ¿En qué falla la aritmetización que no puede resolver tales problemas? La opinión de los autores del trabajo en el cual se basa esta sección es la siguiente: "La aritmetización falla en no abrir la caja negra lo suficiente. En una típica demostración que usa aritmetización uno parte de una fórmula booleana φ de tamaño polinomial y crea a partir de ella el respectivo polinomio p_φ. A partir de ahí uno se olvida de φ y trata a p_φ como una caja negra que se puede computar eficientemente. De φ solo se aprovecha que tiene tamaño polinomial asegurando que p_φ se puede evaluar eficientemente. Lo que la algebrización sugiere es que se va a tener que utilizar φ de alguna manera más profunda."

¿Es P vs NP Independiente?

> ¿Cómo podría semejante clara pregunta matemática; una pregunta sobre máquinas finitas y problemas, una pregunta de enorme importancia para la ciencia y la industria; quedar por siempre sin respuesta?
> Scott Aaronson

> Voy a declarar, como una de las pocas conclusiones de este trabajo, que $\mathbf{P} \neq \mathbf{NP}$ es verdadero ó falso. Es una o la otra. Pero tal vez no sea posible demostrar cuál de las dos es, y tal vez tampoco sea posible probar que no lo podemos demostrar.
> Scott Aaronson

Hemos visto que a cada técnica le hemos encontrado "meta-teoremas" que le propician su propia limitación. En algún punto, nos recuerda alguna noción de resultados de independencia. No solo nos hemos atascado al tratar de resolver **P** vs **NP** sino que formalizamos las razones de tal atascamiento. Por ello, ha sido natural investigar si no estamos tratando de resolver un problema que es independiente a la teoría, digamos, la teoría de conjuntos ZF. Luego de varias décadas de esfuerzo sostenido en ese camino la "sensación" en los investigadores es que **P** vs **NP** no tiene más "carácter" de independiente que otros grandes problemas de las matemáticas, como la conjetura de Goldbach o la Hipótesis de Riemann. La presente sección se basa en [4], como con la sección anterior, un estudio profundo queda fuera del alcance de este trabajo. Y ya que hemos pensado tanto en **P** vs **NP** el lector interesado por curiosidad o diversión puede consultar el enunciado preciso que brinda el Instituto Clay de Matemáticas del problema, [1].

Antes de seguir, es menester aclarar que este análisis sobre la independencia de **P** vs **NP** aplica a otros grandes problemas del campo como: **NP** vs **coNP**, **NP** vs $\mathbf{P_{/poly}}$, **P** vs **PSPACE**, **BPP** vs **QBP** y otros. Se elige **P** vs **NP** porque es el más familiar y famoso. De ser independiente a digamos, ZF entonces habría dos posibilidades: o no existiría un algoritmo polinomial para `SAT` pero no sería posible probarlo en ZF, o existiría tal algoritmo pero sería imposible probar en ZF que funciona. Para demostrar su independencia se la ha tratado de probar en teorías débiles para luego ir "escalando" el poder de las mismas. Por ejemplo, se probó que $\mathbf{P} \neq \mathbf{NP}$ es independiente de la teoría ET que es un "fragmento" de la teoría de números. Donde los objetos de dicha teoría son números enteros, y el lenguaje consiste en las funciones: $x + y$, $x.y$, $\max(x, y)$, $\min(x, y)$, c^x donde c es una constante, más todos los predicados computables polinomialmente (pero no funciones). Luego se le ha ido agregando otras funciones a tal teoría para hacerla más "fuerte" tratando de demostrar en ella los mismos resultados de independencia. Sin embargo, no se ha estado ni cerca de reproducir tales resultados en teorías de un poder aceptable, como ZF o la Aritmética de Peano.

Definamos como $\prod_1$-sentencia a una sentencia de la forma "para todo x, $P(x)$" con P una función demostrada como recursiva en la aritmética de Peano,

PA. (Esto es que existe una MT M que computa P y que se tiene una demostración en PA que prueba que M finaliza en toda entrada). Análogamente una $\prod_2$-sentencia es una sentencia de la forma $\forall x \exists y \; P(x,y)$ donde P se puede probar recursiva, una $\prod_3$-sentencia es una sentencia de la forma $\forall x \exists y \forall z \; P(x,y,z)$ y así sucesivamente. Es divertido clasificar los problemas matemáticos más famosos en algún tipo de estas sentencias. Por ejemplo, la conjetura de Goldbach es una sentencia $\prod_1$ ya que el simple predicado recursivo en PA:
$P(x) =$ si $x > 2$ es par $\Rightarrow \exists i, j \leq x : i + j = x$ y i, j son primos, verifica que el entero x, de ser par mayor a 2, es la suma de dos números primos. Clasificar por ejemplo a la Hipótesis de Riemann es un poco más difícil, la idea es que hay un resultado [25] que muestra que tal hipótesis es equivalente a la siguiente afirmación: para todo natural n, la suma de los divisores de n es a lo sumo

$$\alpha(n) = H_n + e^{H_n} \ln(H_n)$$

donde $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$ es el n-ésimo número armónico. Y esta afirmación es $\prod_1$ ya que hay muy buenas cotas superiores de $\alpha(n)$. Volviendo a nuestro asunto, **P** vs **NP** es $\prod_2$ ya que $\mathbf{P} \neq \mathbf{NP}$ es equivalente a: para toda máquina de Turing determinista M y para todo polinomio p, existe una fórmula booleana φ de tamaño n tal que $M(\varphi)$ no finaliza con la respuesta correcta a $¿\varphi \in \texttt{SAT}?$ en a lo sumo $p(n)$ pasos. Análogamente se puede ver que **NP** vs $\mathbf{P}_{/\mathbf{poly}}$ es $\prod_2$. Supongamos que PA es consistente, y sea PA+$\prod_1$ la aritmética de Peano con todas las sentencias verdaderas de $\prod_1$ agregadas como axiomas. Naturalmente, como las sentencias verdaderas de $\prod_1$ no son recursivamente enumerables, PA+$\prod_1$ tiene un "poder asombroso". Aún así: ¡Hay sentencias simples en $\prod_2$ que se han demostrado independientes de PA+$\prod_1$! Tranquilamente una de ellas podría ser la que nos atormenta.

Para finalizar con más optimismo, como observamos en la subsección de Relativización vs Independencia del capítulo Diagonalización, la simple pero fundamental propiedad de que la computación es local alcanza para darle poder a una teoría para superar la barrera de la relativización. Tal vez dicha propiedad u otras propiedades fundamentales de la computación que no logramos ver o comprender, en el futuro nos permitan de una vez definitiva, entender con profundidad a la computación y develar sus misterios que hoy nos son inalcanzables.

Bibliografía

[1] Enunciado p vs np. [http://www.claymath.org/sites/default/files/pvsnp.pdf].

[2] Godel letter. [https://www.cs.cmu.edu/ aada/courses/15251s15/www/notes/godel-letter.pdf].

[3] S. Aaronson. Oracles are subtle but not malicious. volume 2006, pages 15 pp.–354, 01 2005.

[4] Scott Aaronson. Is p versus np formally independent. *Bulletin of the EATCS*, 81:109–136, 01 2003.

[5] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *TOCT*, 1, 01 2009.

[6] Leonard Adleman. Two theorems on random polynomial time. pages 75–83, 11 1978.

[7] Agrawal, Kayal, and Saxena. Errata: Primes is in p. *Annals of Mathematics*, 189:317, 01 2019.

[8] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[9] Sanjeev Arora, Russell Impagliazzo, U San, and Diego Vazirani. Relativizing versus nonrelativizing techniques: The role of local checkability. 10 2019.

[10] Sanjeev Arora, Carsten Lund, Rajeev Motwani, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45:501–555, 09 2001.

[11] Shilpkumar Arora and S. Safra. Probabilistic checking of proofs; a new characterization of np. volume 45, pages 2–13, 11 1992.

[12] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =?\mathcal{NP}$ question. *Siam Journal on Computing - SIAMCOMP*, 4, 12 1975.

[13] Boaz Barak. A probabilistic-time hierarchy theorem for "slightly non-uniform.ªlgorithms. volume 2483, 08 2002.

[14] Ravi Boppana and Michael Sipser. The complexity of finite functions. 01 1989.

[15] Stephen Cook. The complexity of theorem-proving procedures. pages 151–158, 01 1971.

[16] Richard Demillo and Richard Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7:193–195, 06 1978.

[17] Lance Fortnow and Rahul Santhanam. Hierarchy theorems for probabilistic polynomial time. pages 316–324, 11 2004.

[18] O. Goldreich. Introduction to complexity theory [lectures notes]. July 1999.

[19] O. Goldreich. *Computational Complexity A Conceptual Perspective*. Cambridge University Press, 2008.

[20] Oded Goldreich and Luca Trevisan. From logarithmic advice to single-bit advice. *Electronic Colloquium on Computational Complexity (ECCC)*, 6650, 01 2004.

[21] Richard Karp. Reducibility among combinatorial problems. volume 40, pages 85–103, 01 1972.

[22] J. Katz. Lecture on relativization [lecture notes]. October 2008.

[23] M. Kim. Polynomial identity testing [lecture notes]. April 2017.

[24] Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, January 1975.

[25] Jeffrey Lagarias. An elementary problem equivalent to the riemann hypothesis. *The American Mathematical Monthly*, 109:534–543, 06 2002.

[26] Clemens Lautemann. Bpp and the polynomial hierarchy. *Information Processing Letters*, 17:215–217, 11 1983.

[27] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, October 1992.

[28] B. Mares. A detailed proof that ip=pspace.

[29] R. Lipton. R. Karp. Turing machines that take advice. *L' Ensignement Mathématique*, (28):191–210, 1982.

[30] Alexander Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55, 10 1999.

[31] Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, 01 1978.

[32] Adi Shamir. Ip = pspace. *J. ACM*, 39(4):869–877, October 1992.

[33] A. Shen. Ip = space: Simplified proof. *J. ACM*, 39(4):878–880, October 1992.

[34] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 10 1997.

[35] Victor Shoup. A computational introduction to number theory and algebra. *A Computational Introduction to Number Theory and Algebra*, 01 2005.

[36] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5:207–388, 01 2010.

[37] M. Sipser. *Introduction to the theory of computation.* Cengage Learning, third edition, 2013.

[38] Michael Sipser. A complexity theoretic approach to randomness. pages 330–335, 01 1983.

[39] R. Williams. An overview of circuit complexity [lecture notes].