FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

El Problema del Milenio PvsNP: Introducción a la NP-Completitud

Realizado por
Andrés Nicolás Uranga Limón

Dirigido por
Antonio Ramírez de Arellano Marrero

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Junio de 2023

Agradecimientos

Este trabajo no hubiera sido posible sin el cariño incondicional de mis padres y hermana, el apoyo de mis compañeros y la perseverancia de mi tutor. A todos ellos, mis más sinceros agradecimientos.

Resumen

La Teoría de la Complejidad Computacional es una rama de las ciencias de la computación que estudia la clasificación de problemas en función de los recursos requeridos para resolverlos (tiempo y espacio). Entre los problemas más estudiados en este campo se encuentra uno de los Siete Problemas del Milenio propuestos por la fundación Clay Mathematics Institute de Cambridge, el Problema "P vs NP", siendo uno de los más importantes no sólo en este ámbito, sino dentro de las matemáticas en general.

En este Trabajo Fin de Grado se introducirá al problema y uno de los grandes caminos para atacarlo: la Teoría de la NP-completitud, exponiendo sus problemas clásicos y métodos asociados. Para concluir, se hará una reflexión de cómo aplicar dicha teoría y la implicaciones en la sociedad de sus posibles respuestas.

Abstract

Computational Complexity Theory is a branch of computer science that studies the problems classification according to the resources required to solve them (time and space). Among the main problems studied in this field we find one of the Seven Millennium Problems proposed by the Clay Mathematics Institute of Cambridge, the Problem "P vs NP", being one of the most important not only in this field, but also in mathematics in general.

In this Final Degree Project we will introduce the problem and one of the main ways to attack it: the NP-completeness theory, exposing its classical problems and associated methods. Finally, a reflection will be made on how to apply this theory and the implications in society of its possible answers.

Índice general

Índice de tablas

Índice de figuras

Introducción

En el año 2000 la fundación Clay Mathematics Institute de Cambridge, Massachusetts, decide impulsar la investigación en el campo de las matemáticas señalando cuales serían los Siete Problemas del Milenio y estableciendo una remuneración de un millón de dólares para aquellos que consiguieran resolverlos. Los problemas pertenecían a diversas áreas de las matemáticas, desde Teoría Analítica de Números ("La Hipótesis de Riemman") hasta Geometría Algebraica ("Conjetura de Birch y Swinnerton-Dyer"), pasando por las ciencias de la computación y Teoría de la Complejidad Computacional("P vs NP"). Es en este último marco teórico de la matemáticas donde se encuentra este trabajo.

La Teoría de la Complejidad Computacional se encarga de analizar la eficiencia de los algoritmos y los problemas computacionales. Con eficiencia se hace referencia a qué cantidad de espacio necesitaría una computadora para resolver un problema o cuánto tiempo tardaría en resolverlo. En este ámbito aparece la complejidad.

La importancia de estudiar la complejidad reside en que si al atacar un problema, uno solo es capaz de encontrar un algoritmo exponencial para su resolución, entonces sabrá que su problema no es *tratable* en términos prácticos. Por ejemplo, en la tabla 1 se observa que para un dato de entrada de tamaño 10 la complejidad polinomial entre n^2 y 2^n se diferencian de forma notable, aunque al tratarse de números no muy "grandes" no habría problemas para computarlos. Sin embargo, para para un dato de tamaño 1000 la complejidad exponencial asciende a un número de 302 dígitos, lo cual es inimaginable. Para ponernos en contexto, el número de microsegundos desde el Big-Bang hasta hoy tiene 24 dígitos.

Complejidad	$n = 10$	$n = 50$	$n = 100$	$n = 300$	$n = 1000$
Polinomial n	10	50	100	300	1000
Polinomial n^2	100	2500	10000	90000	10^6
Polinomial n^5	100000	$3,13 \cdot 10^8$	10^8	$2,43 \cdot 10^9$	10^{12}
Exponencial 2^n	1024	$1,13 \cdot 10^{15}$	$1,27 \cdot 10^{30}$	$1,38 \cdot 10^{90}$	$1,07 \cdot 10^{301}$
Factorial $n!$	3628800	$3,04 \cdot 10^{64}$	$9,33 \cdot 10^{157}$	$3{,}06 \cdot 10^{642}$	$4,02 \cdot 10^{2568}$

Tabla 1: Recursos usados por un algoritmo de cierta complejidad. Mientras que los algoritmos polinómicos arrojan cifras que son fácilmente computables por los ordenadores actuales, los algoritmos exponenciales siguen siendo intratables.

"P vs NP" es uno de los problemas más importantes de la Teoría de la Complejidad tanto por sus implicaciones teóricas, da una clasificación de las distintas clases de complejidad, como prácticas pues influye directamente en los sistemas criptográficos, la optimización, la inteligencia artificial...

El problema "P vs NP" consiste en determinar si los problemas de los que sabiendo una solución, es posible verificar, en tiempo polinomial, si esta es o no factible (clase NP), se diferencian de aquellos cuya solución puede ser hallada en tiempo polinomial (clase P).

En este trabajo se definirá formalmente qué es la clase P y la clase NP. Posteriormente, se expondrá uno de los métodos y clases que se usan para probar que $P \neq NP$, la teoría de la NP-completitud y las reducciones. El objetivo, es que el lector, una vez concluido el texto, entienda el problema "P vs NP" y aprenda trabajar con algunas herramientas básicas que se emplean para atacar este problema.

La estructura del proyecto está dividida en cuatro capítulos. En el primero de ellos, se presentan las máquinas de Turing, el modelo computacional utilizado y se definen las clases P y NP. En el segundo, ya se definen la NP-completitud y las reducciones polinomiales. Además, aparece SAT como primer problema NP-completo, y como base del resto de las reducciones polinomiales ejecutadas para probar la NP-completitud de algunos problemas clásicos. En el tercero, se muestra cómo la NP-completitud se aplica para buscar una frontera entre P y los problemas NP-completos. Finalmente, en el capítulo cuatro, se resumirán las conclusiones del autor y se reflexionará sobre perspectivas futuras sobre el estudio del problema "P vs NP".

CAPÍTULO 1

Preliminares

El presente capítulo otorga las herramientas necesarias para comprender qué es la clase P y la clase NP. Para ello será conveniente familiarizarse con los lenguajes y los problemas de decisión. También, será preciso para formalizar la teoría un modelo de computación convencional: Las máquinas de Turing.

Para concluir el capítulo, debido a que la gran mayoría de los problemas expuestos en el segundo y tercer capítulos pertenecen a la teoría de grafos, se ha considerado oportuno añadir una sección donde se puedan consultar aspectos generales de los mismos. Seguidamente aparecerá una breve definición de Forma normal Conjuntiva , y se introducirá la notación $\mathcal{O}$ de Landau.

1.1– Introducción a la Teoría de la Complejidad Computacional

En esta sección, se procede a introducir el tema de la complejidad computacional. Se comienza con la definición de los problemas de decisión, problemas que plantean una pregunta que se responde con "sí" o "no", así como las herramientas necesarias para su tratamiento. Posteriormente, se realiza una revisión exhaustiva del modelo computacional que se empleará para abordar formalmente el estudio: las máquinas de Turing. Finalmente, se presentan las clases de complejidad P y NP. La nomenclatura de está sección está basada en los libros [1, 7], los cuales se recomiendan para más información.

1.1.1. Problemas de decisión, lenguajes y esquemas de codificación

Problemas de decisión

Por conveniencia la teoría de la NP-completitud está diseñada para ser aplicada a problemas de decisión. En teoría de la complejidad, un problema de decisión es un tipo de problema computacional que estudia si una instancia de un problema dado satisface o no una cierta propiedad o criterio específico.

Formalmente, un *problema de decisión* D_Π se puede definir como una función que toma como entrada una instancia o conjunto de datos de un problema y devuelve una respuesta

afirmativa o negativa (por ejemplo, "sí" o "no") según si la instancia satisface o no la propiedad en cuestión.

Por ejemplo, el problema de decisión "¿Existe un camino de longitud k entre los nodos u y v en un grafo G?" devuelve una respuesta afirmativa si existe un camino de longitud k entre u y v en G, y una respuesta negativa si no existe tal camino.

Otro problema de decisión relevante es el PROBLEMA DEL VENDEDOR AMBULANTE. Este consiste en dados un conjunto de ciudades, una distancia entre las ciudades y un entero B. ¿Existe un camino que pase por todas las ciudades y llegue a la ciudad de partida de manera que la longitud del trayecto es menor o igual que B? Formalmente, se escribe:

Problema del vendedor ambulante

INSTANCIA: Sean $C = \{c_1, c_2, \ldots, c_m\}$ de ciudades, d una distancia cumpliendo $d(c_i, c_j) \in \mathbb{N}$ para todo par de ciudades $c_i, c_j \in C$ y $B \in \mathbb{N}$.

PREGUNTA: ¿Existe un camino por todas las ciudades de C de longitud menor o igual que B, es decir, un ordenación $< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} >$ de C tal que

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B \ ?$$

La teoría de la complejidad se enfoca en los problemas de decisión porque pueden ser formulados de manera clara y precisa en términos de una pregunta que se puede responder con "sí" o "no". Además, muchos otros tipos de problemas pueden reducirse en problemas de decisión. Por ejemplo, si se quiere encontrar la ruta más corta entre dos puntos en un mapa, se puede reformular la pregunta como "¿existe una ruta de longitud menor o igual a X que conecte los dos puntos?".

El enfoque en problemas de decisión permite una mayor claridad y precisión en la definición de la complejidad computacional. En particular, se pueden establecer criterios objetivos para medir la dificultad de un problema, como el número de pasos computacionales necesarios para resolverlo en el peor de los casos. Esto permite clasificar los problemas en diferentes clases de complejidad, según su dificultad relativa y los recursos computacionales requeridos para resolverlos.

En resumen, la teoría de la complejidad se enfoca en problemas de decisión porque son una forma clara y precisa de definir problemas y porque permiten una medición objetiva de la dificultad de los mismos.

Lenguajes

El concepto de lenguaje surge de esa necesidad de un instrumento matemático que nos permita estudiar la complejidad de un problema de decisión.[7]

Un *lenguaje* es un conjunto de cadenas finitas formadas por símbolos de un alfabeto dado. Si Σ es un conjunto finito de símbolos, entonces Σ^* es el conjunto de todas las posibles cadenas finitas formadas por símbolos de Σ. Si L es un subconjunto de Σ^*, entonces L es un lenguaje sobre el alfabeto Σ. A la palabra que no tiene ningún símbolo se le conoce como palabra vacía y se denota por ϵ. Por ejemplo, si $\Sigma = \{0, 1\}$, entonces Σ^* está compuesto por la cadena vacía ϵ y todas las cadenas finitas que se pueden formar con los símbolos 0 y 1. Un ejemplo de un lenguaje sobre Σ sería $L = \{01, 111, 000, 10\}$.

Esquema de codificación de un lenguaje

La correspondencia entre los problemas de decisión y los lenguajes se lleva a cabo mediante los esquemas de codificación.

Un esquema e para un problema de decisión Π proporciona una descripción de cada instancia de Π mediante una palabra adecuada sobre cierto alfabeto Σ. Por lo tanto, el problema Π y su esquema de codificado e particionan Σ^* en tres clases: aquellas que no son codificaciones de instancias de Π, aquellas que codifica instancias con respuesta "no" y aquellas que codifican instancias con respuesta "si". Esta última clase constituye el lenguaje asociado a el problema de codificación Π por medio del esquema de codificación e:

$$L[\Pi, e] = \left\{ x \in \Sigma^* : \begin{array}{l} \Sigma \text{ es el alfabeto usado por } e, \text{ y } x \text{ es la codificación} \\ \text{por } e \text{ de una instancia con respuesta positiva} \end{array} \right\}$$

La teoría será aplicada a problemas de decisión utilizando que si un resultado se verifica para el lenguaje $L[\Pi, e]$, entonces también se tiene para el problema Π codificado por el esquema e.

Como el objetivo será estudiar la complejidad, resulta útil asociar a cada problema de decisión una función $Tamaño : D_\Pi \to \mathbb{N}$ que no dependa del esquema de codificación y que esté *polinomialmente relacionada* con las longitudes de los datos de entrada. La expresión *polinomialmente relacionada* se refiere a que para cualquier esquema de codificación e de Π, existan dos polinomios p y p' tal que si $I \in D_\Pi$ y x es una cadena codificando a I mediante e, entonces $Tamaño(I) \leq p(|x|)$ y $|x| \leq p'(Tamaño(I))$, donde $|x|$ es la longitud de x. Remarcar, también, que el hecho de que la función sea polinomial es importante, pues si fuera de otro tipo (exponencial, por ejemplo) habría un conflicto con la definición de complejidad.

Debido a la definición dada de esquemas de codificación, no hay unicidad en la codificación, es decir, para un mismo problema de decisión se pueden encontrar distintos esquemas de codificación válidos. Sin embargo durante el resto trabajo se usará un esquema concreto, el que se da en el próximo párrafo. La principal característica que debe tener uno de estos esquemas es que debe ser conciso y debe poder descodificarse.

El siguiente ejemplo de esquema de codificación forma parte de aquellos que llamamos razonables. La codificación consistirá en identificar instancias con cadenas del alfabeto $\Psi = \{0, 1, -, [,], (,), , \}$. Las cadenas se forman de la siguiente forma:

1. Si k es un número entero se le asocia su representación en binario (precedida de un signo "-" si k es negativo).

2. Si x es una cadena que representa a k, entonces la cadena $[x]$ se usa como nombre (por ejemplo, un vértice en un grafo o una ciudad en el problema del vendedor).

3. Si $x_1, x_2, \ldots, x_m$ es una cadena representado los objetos $X_1, X_2, \ldots, X_m$, entonces $(x_1, x_2, \ldots, x_m)$ es la cadena que codifica la sucesión $\langle X_1, X_2, \ldots, X_m \rangle$.

Debido a este método, la codificación de una instancia de un problema queda determinada por la codificación de sus objetos al usar la regla (3). Por lo tanto, únicamente es necesario especificar como se construye la representación de cada objeto. Los objetos que aparecerán en las instancias son: enteros, "elementos sin estructura", sucesiones, conjuntos, grafos, aplicaciones finitas y números racionales.

- Las reglas (1) y (3) dan la representación de los enteros y las sucesiones.
- A los elementos sin estructura se le asignarán un nombre usando la regla (2).
- Un conjunto de objetos se representará ordenando sus elementos en una sucesión $\langle X_1, X_2, \ldots, X_m \rangle$ y tomando la estructura de cadena correspondiente a esa sucesión.
- Un grafo de vértices V y aristas E se representará con la cadena (x, y), donde x es la cadena que representa V e y es la cadena que representa el conjunto E. Una aplicación finita $f : \{U_1, U_2, \ldots, U_m\} \to W$ se representará con la cadena $((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m))$ donde x_i codifica U_i y y_i, $f(U_i)$ para $1 \leq i \leq m$.
- Un número racional $q = \frac{a}{b}$ se representará con la cadena (x, y) donde x es la cadena asociada a a y y, a b.

Esto constituye un ejemplo de esquema de codificación. Aunque no será relevante a lo largo del trabajo, es conveniente que el lector tenga conocimiento de al menos uno de estos esquemas. Como ya se ha dicho, muchos esquemas de codificación son válidos para un mismo problema de decisión, siempre y cuando cumplan las dos condiciones de concisión y decodificación.

1.1.2. La máquina de Turing

En 1936, el matemático británico Alan Turing introdujo por primera vez el concepto de máquina de Turing [20]. Su objetivo era describir una máquina que pudiera realizar cualquier cálculo expresado por medio de un algoritmo. Aunque la máquina de Turing no sea un modelo práctico para la computación en máquinas reales, debido a su sencillez y su potencia, ha proporcionado una base teórica robusta para el desarrollo de las ciencias de la computación y de la Teoría de la Complejidad Computacional. La descripción de la máquina de Turing que aparece a continuación y en los siguientes apartados aparece de forma más detallada en [1]. También se inspira en [10].

La idea detrás de una máquina de Turing es la de una cinta infinita dividida en recuadros en los cuales hay escritos símbolos. Una cabeza lectora, que se puede mover a derecha o izquierda, se encarga de leer, borrar o escribir en los recuadros de acuerdo a una serie de reglas.

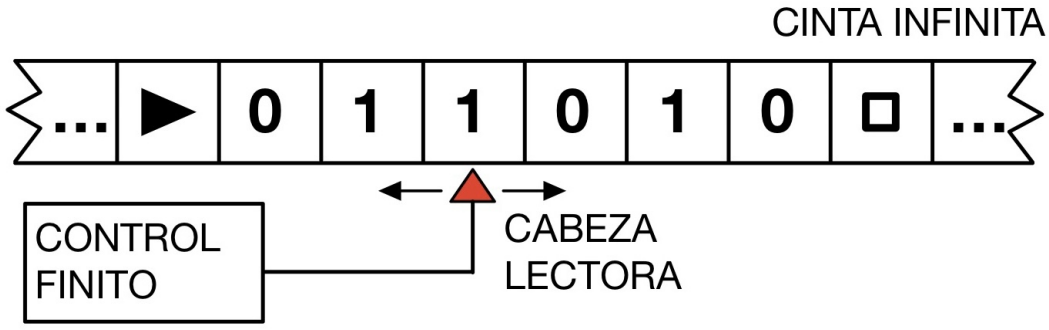


Figura 1.1: Ejemplo de Máquina de Turing sobre el alfabeto $\Sigma = \{0, 1\}$.

La cinta no es más que una línea infinita dividida en celdas en las que se escriben símbolos de un conjunto finito Γ llamado alfabeto de la máquina. Esta está equipada con una cabeza lectora que se encarga de leer, borrar o escribir símbolos en la cinta. La computación de la máquina se realiza en pasos de tiempo discretos, y la cabeza se mueve a izquierda o derecha en cada paso. Consultar figura 1.1.

La máquina tiene también un conjunto finito de estados denotado por $\mathcal{Q}$, en cada paso la máquina se encuentra en un único estado de $\mathcal{Q}$. Este estado determina cuál será la próxima computación de la máquina que consiste en:

1. Leer el símbolo que está en la celda de encima de la cabeza.
2. Cambiar el símbolo por otro símbolo (puede ser el mismo).
3. Pasar a otro estado de $\mathcal{Q}$.
4. Mover la cabeza a la casilla de la izquierda, la de la derecha o permanecer en la misma celda.

La máquina de Turing puede ser interpretada como una computadora moderna simplificada cuya cinta corresponde con la memoria de la computadora y su función de transición es la CPU (unidad central de procesamiento). Sin embargo, nuestro enfoque será el de describir algoritmos de manera simple y matemática.

En ocasiones, para facilitar la labor, conviene usar máquinas con más de una cinta. Sin embargo, se puede demostrar que son polinomialmente equivalentes a las de una cinta. Como la función de las máquinas de Turing en este trabajo es formalizar el modelo de computación, no se definirán este tipo de máquinas. El lector interesado puede encontrar más información en el primer capítulo de [1].

Máquina de Turing determinista

A continuación, se da una descripción formal de la máquina de Turing determinista:
Una *máquina de Turing determinista* (MTD) M es una tupla (Γ,$\mathcal{Q}$,δ) donde :

- Γ es un conjunto finito de símbolos que puedes aparecer en la cinta de M. Γ contiene un alfabeto de entrada con los símbolos que se le introducen a la máquina, llamado Σ. Asumimos que Γ contiene el símbolo $\square$ que designa el blanco, y $\triangleright$ que designa la celda por la que empezamos a leer. Se cumple que $\triangleright, \square \in \Gamma - \Sigma$
- $\mathcal{Q}$ son lo posibles estados en los que se puede encontrar M. $\mathcal{Q}$ contiene tres estados especiales, q_0 es el estado inicial, q_Y estado de aceptación y q_N estado de rechazo, cumpliendo $q_Y \neq q_N$.
- $\delta : \mathcal{Q} - \{q_Y, q_N\} \times \Gamma \to \mathcal{Q} \times \Gamma \times \{L, R, N\}$ es la función de transición (L indica que la cabeza se mueva a la izquierda; R, a la derecha; y N que se quede en la misma celda). A δ le exigimos que si $\delta(q, \triangleright) = (p, \sigma, D)$ entonces $\sigma = \triangleright$ y $D = R$.

La MTD M computa como sigue: Inicialmente, M recibe como entrada una palabra $w = w_1 w_2 \dots w_n \in \Sigma^*$ precedido de $\triangleright$, donde la cabeza empieza a leer, el resto de la cinta esta rellena con blancos. Una vez M empieza, la computación sigue las reglas descritas por la función de transición δ. La computación termina si se alcanza q_Y o q_N, de lo contrario la máquina no para.

Mientras M realiza su computación, ocurren ciertos cambios en los estados, el contenido de la cinta y la posición actual de la cabeza lectora. A una asignación de estos tres elementos la llamamos configuración de la MTD. Sean M una MTD dada por $(\Gamma,\mathcal{Q},\delta)$, un estado q y dos cadenas $u, v \in \Sigma^*$, se denota como $d = (q, u, v)$ a la configuración donde el estado actual es q, el contenido de la cinta es uv y la cabeza se encuentra en el primer símbolo de v (si $w = \epsilon$, la cabeza lee $\square$).

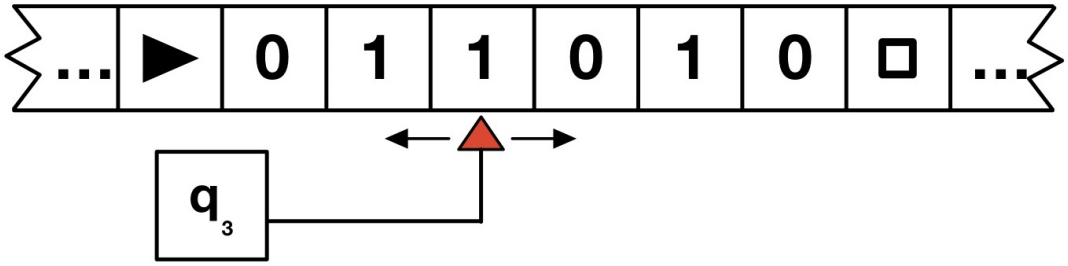


Figura 1.2: MTD con configuración $d = (q_3, 01, 1010)$.

Configuraciones

Ahora se formalizará la computación de una MTD. Se dice que la configuración d' es sucesora de d si M puede ir de d a d' en un único paso. Formalmente, d y d' configuraciones tales que $d = (q, v, sw)$ con $s \in \Gamma, q \neq q_Y$ y $q \neq q_N$. Se dice que $d' = (q', v', w')$ es la sucesora de d respecto de M, $d \vdash_M d'$, cuando:

- Si $\delta(q, s) = (q', t, N)$, entonces se indica que, estando en el estado q y leyendo el símbolo s, se realiza una transición hacia el estado q', donde se cambia el símbolo s por t y se mantiene en la misma posición en la cinta. En este caso, se establece que el contenido de la cinta no cambia antes ni después de la transición, es decir, $v' = v$ y $w' = tw$.

- Si $\delta(q, s) = (q', t, R)$, entonces se indica que, estando en el estado q y leyendo el símbolo s, se realiza una transición hacia el estado q', donde se cambia el símbolo s por t y se mueve hacia la derecha en la cinta. En este caso, se establece que el contenido de la cinta después de la transición es igual al contenido original de la cinta, excepto que el símbolo s ha sido reemplazado por t, es decir, $v' = vt$ y $w' = w$.

- Si $\delta(q, s) = (q', t, L)$, entonces se indica que, cuando se alcanza el estado q y se lee el símbolo s, se realiza una transición hacia el estado q', donde se cambia el símbolo s por t y se mueve hacia la izquierda en la cinta. En este caso, se establece que el contenido de la cinta después de la transición se obtiene al insertar el símbolo t en la posición actual. Es decir, si era $v = ur$ (donde r es cualquier símbolo de Γ), entonces el contenido de la cinta después de la transición se obtiene al poner t en la posición actual, seguido por w y precedido por ur, lo que se puede escribir como $v' = u$ y $w' = rtw$.

Si $d = (q, v, \epsilon)$, entonces $d \vdash_M d'$ si $(q, w, \square) \vdash_M d'$.

Una configuración de M, $d = (q, v, sw)$ es de parada si $q = q_Y$, en cuyo caso será una configuración de aceptación, o $q = q_N$, que será una configuración de rechazo.

Para acabar este apartado se muestra un ejemplo de MTD que dado un número escrito en binario determina si este es par en la tabla 1.1. En esta máquina de Turing, si el número binario es par, la máquina llegará al estado de aceptación q_Y, si no se llegará a un estado de rechazo q_N.

Estado actual	Símbolo actual	Nuevo símbolo	Movimiento	Nuevo estado
q_0	0	0	R	q_0
q_0	1	1	R	q_0
q_0	$\square$	$\square$	L	q_1
q_1	0	0	L	q_Y
q_1	1	1	L	q_N
q_1	$\square$	$\square$	R	q_N

Tabla 1.1: MTD que determina si un número binario es par.

Computación

Una vez explicado lo que es una configuración, se procede a definir que es una computación de M. Sea M una MTD, fijado $x \in \Sigma$ se define la computación de M sobre x como la sucesión (posiblemente infinita) de configuraciones de $M, d_0, d_1, d_2, \ldots$ tal que:

- $d_0 = (q_0, \triangleright, x)$
- Para cada i, si $d_i = (q, v, w)$ no es de parada, d_{i+1} está definida y $d_i \vdash_M d_{i+1}$

Lenguaje de una máquina de Turing

Se dice que una máquina de Turing M acepta una cadena x si la computación de M sobre x acaba en un estado de aceptación. El conjunto de cadenas que M acepta es el lenguaje de M o el lenguaje reconocido por M, denotado por L_M.

Hay que destacar que la definición de reconocimiento de lenguaje no requiere que M pare sobre todas las cadenas de Σ^*, solo para las de L_M. Si $x \in \Sigma^* - L_M$, entonces la computación de M sobre x puede que pare en el estado q_N o que sea infinita.

La resolución de problemas de reconocimiento de lenguajes y de problemas de decisión está ligada. Decimos que una MTD M resuelve el problema de decisión Π bajo el esquema de codificación e si M para sobre todas las cadenas del alfabeto de entrada y $L_M = L[\Pi, e]$.

1.1.3. La clase *P*

En esta sección se comienza a tratar el concepto de "complejidad en tiempo" definiendo la clase P. La clase de complejidad P es la clase de problemas de decisión que pueden ser resueltos por una máquina de Turing determinista en tiempo polinómico. Esto significa que existe un algoritmo que puede resolver el problema en un tiempo que crece como una función polinómica del tamaño de la entrada.

El tiempo empleado en una computación de una MTD M es el número de pasos que se ejecutan hasta que se alcanza un estado de parada. Para una MTD que para sobre todas

sus posibles entradas $x \in \Sigma$, su *función de complejidad de tiempo* $T_M : \mathbb{N} \to \mathbb{N}$ está dada por:

$$T_M(n) = max \left\{ m \in \mathbb{N} : \begin{array}{l} \Sigma \text{ existe } x \in \Sigma^*, \text{ con } |x| = n \text{ tal que la computación} \\ \text{de } M \text{ con entrada } x \text{ tarda tiempo } m \end{array} \right\}$$

Un programa M se conoce como MTD *que tarda tiempo polinomial* si existe un polinomio p tal que para todo $n \in \mathbb{N}$, $T_M(n) \leq p(n)$.

La clase de lenguajes P se define como sigue:

$$P = \{L \subseteq \Sigma^* : \text{ existe una } MTD \text{ que tarda tiempo polinomial } M \text{ con } L = L_M\}$$

Se dice que el problema de decisión Π pertenece a P con el esquema de codificación e si $L[\Pi, e] \in P$. Es decir, si hay una MTD que resuelve en tiempo polinomial el problema Π codificado por e. Sin embargo, para simplificar la notación debido a la existencia de varios esquemas de codificación válidos para el mismo problema de decisión Π, se dice simplemente que Π pertenece a P.

La clase de complejidad P es una de las clases más importantes en teoría de la computación y es considerada una medida de la eficiencia de los algoritmos. Muchos problemas prácticos pueden ser resueltos eficientemente en tiempo polinómico, lo que significa que son tratables en términos de complejidad computacional.

1.1.4. La clase *NP*

En esta sección se exponen las herramientas necesarias para entender la segunda clase importante de lenguajes que se estudiará en el trabajo, la clase NP. En el desarrollo de la sección tomarán un papel relevante los algoritmos no deterministas y las Máquinas de Turing no deterministas. Estos al poder tomar distintos valores para un mismo dato de entrada poseen más potencia y eficacia que los algoritmos deterministas, lo cual, en principio, permite resolver en tiempo polinomial nuevos problemas.

Algoritmos no deterministas

Intuitivamente se dice que NP está relacionado con los algoritmos no deterministas. Un algoritmo no determinista es aquel que puede producir resultados diferentes cada vez que se ejecuta, incluso con la misma entrada. A diferencia de los algoritmos deterministas, que producen un único resultado cada vez que se ejecutan con la misma entrada, los algoritmos no deterministas no tienen una computación preestableciada para un mismo dato de entrada sino que a medida que avanza el algoritmo se pueden dar distintas opciones.

Los algoritmos no deterministas suelen utilizar técnicas como la aleatoriedad o la búsqueda heurística para explorar diferentes caminos y posibilidades para resolver un problema. Por ejemplo, el algoritmo no determinista de búsqueda en profundidad puede explorar diferentes caminos en un grafo para encontrar un objetivo, pero es posible que elija diferentes caminos cada vez que se ejecuta. La figura 1.3 representa los distintos caminos que puede seguir un algoritmo no determinista.

Para entender un algoritmo no determinista se establecen dos fases, la primera etapa es la *fase aleatoria o fase no determinista* y la segunda la *fase de comprobación o fase determinista*. Dada I una instancia de un problema, la primera etapa genera una posible estructura S. Posteriormente, I y S se utilizan como entradas para la fase de comprobación,

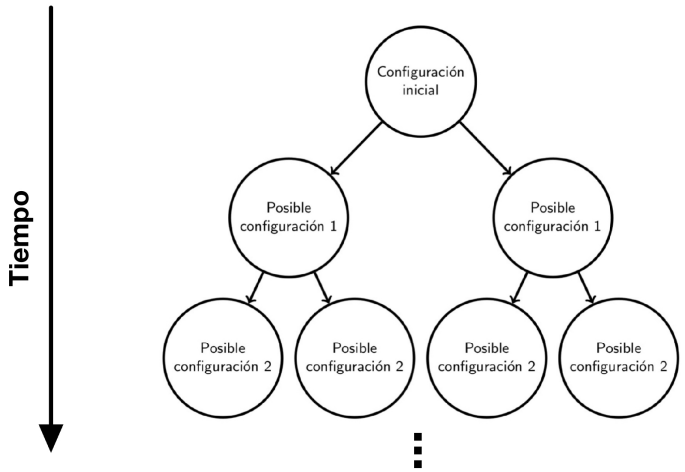


Figura 1.3: Computación no determinista. A medida que la computación avanza se pueden seguir distintos caminos.

que procede de manera determinista. En la primera fase se propone una posible solución al problema y en la segunda se comprueba de forma determinista si esta es, efectivamente, una solución. Un algoritmo no determinista resuelve un problema de decisión Π si se tienen las siguientes propiedades para todas las instancias $I \in D_\Pi$:

1. Si I tiene respuesta "sí", entonces existe una estructura S que, una vez generada I, te lleva a una respuesta positiva para I y S en el estado de comprobación.

2. Si I no tiene respuesta "sí", entonces no existe una estructura S que, una vez generada I, te lleva a una respuesta positiva para I y S en el estado de comprobación.

Se dice que un algoritmo no determinista resuelve el problema de decisión Π en "tiempo polinomial" si existe un polinomio p tal que para toda instancia con respuesta positiva hay alguna estructura S que lleve la comprobación determinista a la respuesta "sí" para I y S en tiempo $p(Tamaño(I))$. Destacar que esto impone una restricción a S pues solo se puede usar tiempo polinomial en comprobar la posible respuesta generada.

De manera informal la clase NP se define como la clase de los problemas de decisión Π que, bajo un esquema de codificación adecuado, pueden ser resueltos por un algoritmo no determinista en tiempo polinomial. El término "resuelto" debe ser entendido con precaución, un"algoritmo no determinista en tiempo polinomial" es básicamente un instrumento para definir la noción de tiempo polinomial de verificación, en lugar de un método real para resolver problemas de decisión. Ahora, en vez de una única posible computación, puede tener varias diferentes.

Hay otro aspecto en el que la solución de un problema de decisión por un algoritmo no determinista se diferencia de la de un algoritmo determinista: la ausencia de simetría entre el "sí" y el "no". Si el problema "¿Dado I, X es verdadero para I?" puede ser resuelto en tiempo polinomial por un algoritmo determinista, entonces el problema complementario, "¿Dado I, X es falso para I?",también. Esto se debe a que en un algoritmo determinista

el problema para sobre toda sus entradas, luego lo único que tendremos que hacer es intercambiar las respuestas "sí" y "no" (cambiar q_Y y q_N en la MTD). Lo mismo no es obvio para problemas que se puedan resolver en tiempo polinomial por algoritmos no deterministas. Es en esta reflexión donde aparece el concepto de $coNP$, el complementario de la clase NP. Una definición formal y propiedades de este concepto pueden ser consultados en el capítulo 10 de [12].

Máquinas de Turing no deterministas

Este apartado concluye formalizando la definición de NP en términos de lenguajes y máquinas de Turing. El homólogo a los algoritmos no determinista son las *máquinas de Turing no deterministas de una cinta*($MTND$).

El modelo de $MTND$ en el desarrollo del trabajo tendrá una estructura muy similar al de la MTD, salvo que se amplía añadiéndole un *módulo de suposición* que poseerá su propia cabeza de escritura que solo realiza la función de escribir, como se indica en la figura 1.2. El módulo de suposición proporcionará un medio para escribir.

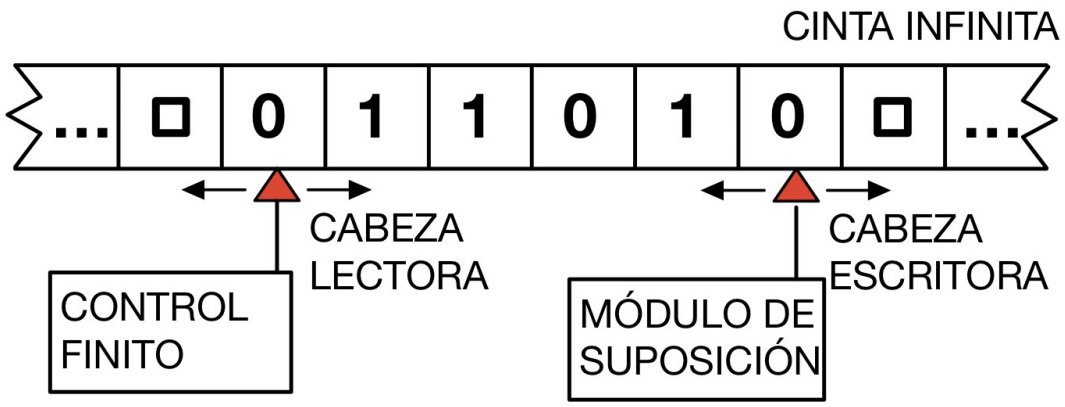


Figura 1.4: Esquema de una $MTND$.

Una $MTND$ es un programa determinado por los mismos elementos que una MTD, es decir, un alfabeto Γ, un alfabeto de entrada $\Sigma \subset \Gamma$, el símbolo de blanco $\square$, el conjunto de estados $\mathcal{Q}$, el estado inicial q_0, los estados de parada q_Y y q_N, y la función de transición $\delta : \mathcal{Q} - \{q_Y, q_N\} \times \Gamma \to \mathcal{Q} \times \Gamma \times \{L, R, N\}$. La diferencia entre una MTD y una $MTND$ reside en la computación, ya que la $MTND$ pasa por dos escenarios distintos:

1. La primera fase que afronta la máquina es el "escenario de suposición". Inicialmente, la cadena de entrada x se escribe en la primera casilla donde se encuentra la cabeza escritora-lectora, mientras que la cabeza escritora comienza escaneando dos celdas a la izquierda, y el control finito esta inactivo. El módulo de suposición dirige paso a paso la cabeza de escritura ya sea para escribir, moverse o pararse, momento en el cual el módulo de suposición se detiene y se activa el estado de control en el estado q_0. La elección de mantenerse activo y, en ese caso, qué símbolo de Σ escribir, se lleva a cabo mediante el módulo de suposición. De este modo, el módulo de suposición puede obtener cualquier cadena de Σ hasta que se para o, incluso, no parar.

2. La segunda fase, el "escenario de comprobación", comienza cuando el control finito se activa en q_0. Desde este punto, la computación ocurre al igual que en una MTD. El módulo de suposición y la cabeza de escritura no se volverán a involucrar en el proceso. La computación acaba si se llega a alguno de los estados de parada q_Y y q_N, y se acepta si llega a q_Y. Las demás computaciones, paren o no, no serán de aceptación.

Se observa que cualquier $MTND$ podrá tener infinitas computaciones para la misma cadena x. Decimos que una $MTND$ M acepta x si al menos una de esas computaciones son de aceptación. El lenguaje reconocido por M es:

$$L_M = \{x \in \Sigma^* : M \text{ acepta } x\}$$

La clase NP

El tiempo requerido por una $MTND$ M para aceptar la palabra $x \in L_M$ se define como el mínimo, de todas las posibles computaciones de M para x, de número de pasos necesarios que se necesitan para alcanzar el estado q_Y. La función de la complejidad temporal $T_M : \mathbb{N} \to \mathbb{N}$ de M es:

$$T_M = max\left\{\{1\} \bigcup \left\{m \in \mathbb{N} : \begin{array}{l} \text{existe } x \in L_M, \text{ con } |x| = n \\ M \text{ acepta } x \text{ en tiempo } m \end{array}\right\}\right\}$$

El 1 se añade por si M no acepta ninguna entrada de longitud n.

Un programa M es una *$MTND$ de tiempo polinomial* si existe un polinomio p tal que para todo $n \in \mathbb{N}$, $T_M(n) \leq p(n)$. Finalmente, la clase NP queda definida formalmente como:

$$NP = \{L \subseteq \Sigma^* : \text{ existe una } MTND \text{ en tiempo polinomial } M \text{ para la cual } L = L_M\}$$

Se establece que el problema de decisión Π pertenece a NP con el esquema de codificación e si $L[\Pi, e] \in NP$. Como ocurre en P, se olvidará e suponiendo que e es un esquema de codificación adecuado y únicamente se dirá que Π está en NP.

1.2– Definiciones de teoría de grafos

La mayoría de los principales problemas NP-completos surgen en el ámbito de la teoría de grafos. Esta sección se establecerá la nomenclatura que se usará durante el trabajo y se dará la definición de grafo. Para una descripción más completa se puede consultar [3], el libro donde se basa esta sección.

Intuitivamente, un grafo es un conjunto de puntos y una relación entre ellos. Si dos puntos está relacionados entonces se dibuja una arista entre ambos. Los puntos podría representar ciudades y las aristas, si existe una carretera que conecte ambas ciudades.

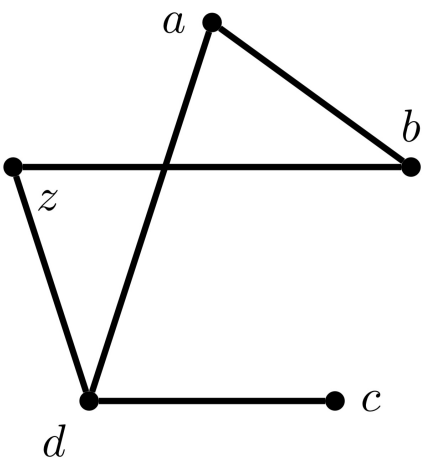
Un *grafo* G consiste en un conjunto finito V de *vértices*, y un conjunto E de subconjuntos de dos elementos de V, cuyos elementos se conocen como *aristas*. Al número de aristas que contienen a un vértice $v \in V$ de le conoce como *grado de* v y se denota por $gr(v)$. A lo largo del trabajo los grafos se representarán por $G = (V, E)$ donde V es el conjunto de vértices y E, el conjunto de aristas.

Por otra parte, el *grafo complemento* o *complementario de un grafo*, G^c, será otro grafo, con el mismo conjunto de vértices del original, y tal que dos vértices están conectados por

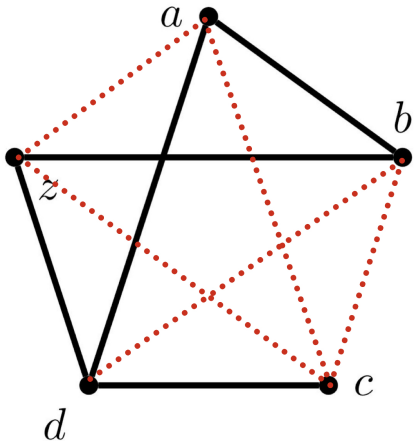
una arista si y solo si esa arista no existe en el primero. En la figura 1.5b se muestra el grafo complementario del ejemplo anterior.

En la figura 1.5a se representa el grafo $G = (V, E)$ donde:

$$V = \{a, b, c, d, z\} \qquad E = \{\{a, b\}, \{a, d\}, \{b, z\}, \{c, d\}, \{d, z\}\}$$



(a) Representación de un grafo.



(b) Grafo complementario en línea de puntos.

En otras ocasiones también se trabaja con grafos dirigidos (o digrafos). Un *grafo dirigido* consta de un conjunto finito V, cuyos elementos se conocen vértices, y un subconjunto A de $V \times V$, cuyos elementos son los arcos. La única diferencia con los grafos es que ahora (a, b) y (b, a) son elementos distintos en el conjunto de los arcos. En la representación cada vez que aparezca (a, b) en el conjunto de arcos se pintará una flecha desde el vértice a hasta el b. En la figura 1.6 aparece un ejemplo de un grafo dirigido.

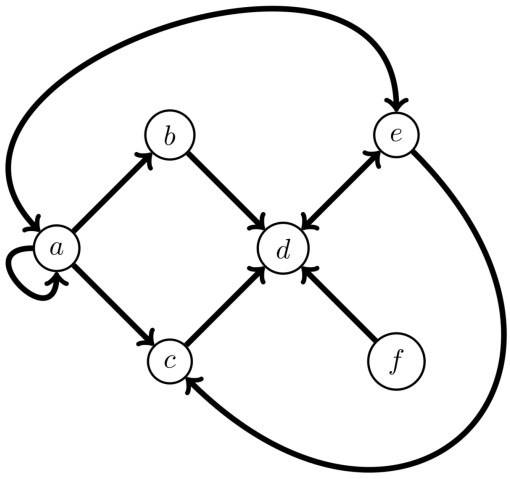


Figura 1.6: Ejemplo de grafo dirigido.

1.3– Forma Normal Conjuntiva

Los ejemplo más simples de problemas *NP*-completos provienen de la lógica proposicional. Una *fórmula Booleana* sobre las variable $u_1, u_2, \ldots, u_n$ está formada por las variables y los operados lógicos AND ($\wedge$), OR ($\vee$) y NOT ($\neg$). Por ejemplo, $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_1 \wedge u_3)$ es una fórmula Booleana. Si φ es una fórmula sobre las variable $u_1, u_2, \ldots, u_n$, y $z \in \{0,1\}^n$, entonces $\varphi(z)$ denota una valoración de φ cuando φ (donde se identifica 1 con $Verdadero$ y 0 con $Falso$). Una fórmula φ es satisfactible si existe alguna asignación z tal que $\varphi(z)$ sea $Verdadera$. La fórmula de arriba es satisfactible pues la asignación $z = (1, 0, 1)$ la satisface.

Se conocen como *literales* a las variables y sus negaciones. Una *cláusula* es una disyunción de literales. Una fórmula Booleana está en *Forma Normal Conjuntiva (FNC)* si es una conjunción de cláusulas. La fórmula anterior está en FNC.

Para facilitar la notación, a lo largo del trabajo las cláusulas se denotaran por conjuntos de literales y la FNC será un conjunto de cláusulas. Por ejemplo, $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_1 \wedge u_3) = \{\{u_1, u_2\}, \{u_2, u_3\}, \{u_1, u_3\}\}$.

1.4– Notación $\mathcal{O}$

A la hora de medir la complejidad de un algoritmo, no se suele contar el número de pasos del algoritmo en concreto si no que se intenta estudiar que ocurriría en el peor caso. Por ello, al analizar la complejidad de un algoritmo lo que realmente se busca es una cota superior para el número de operaciones que se necesitan. La notación $\mathcal{O}$ *grande* de Landau es la que se suele usar en estas ocasiones:

Definición 1.1. Sean f y g funciones de $\mathbb{N} \to \mathbb{R}_{\geq 0}$. Se dice que f es del orden de g, o de orden g, notado por $f = \mathcal{O}(g)$, si existe una constante positiva tal que $f(n) < Cg(n)$ para todo $n \in \mathbb{N}$ con $n \geq n_0$. Es decir:

$$f = \mathcal{O}(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} \in \mathbb{R}_{\geq 0}$$

Esta definición nos permitirá definir la clase de complejidad polinomial más adelante.

La sección queda concluida, en el próximo capítulo se verá como se puede usar esta teoría para atacar el problema "P vs NP" y construir el concepto de NP-completitud.

CAPÍTULO 2

Introducción a la NP-completitud

En este capítulo se concentra el groso del trabajo. Su misión es introducir los conceptos básicos de la NP-completitud. Las reducciones polinomiales tendrán un papel fundamental pues, una vez hallando el primer problema NP-completo, otorgan una forma directa de probar la NP-completitud de un problema. A lo largo del capítulo, se expondrán algunos ejemplos clásicos de este tipo de problemas. En su gran mayoría, el capítulo está basado en [7].

2.1– La relación entre P y NP

La relación entre las clase P y NP es fundamental en la teoría de la NP-completitud. Se tiene que $P \subseteq NP$, es decir, todo problema de decisión resuelto por un algoritmo determinista en tiempo polinomial puede ser resuelto también por un algoritmo no determinista en tiempo polinomial. Esto ocurre porque cualquier algoritmo determinista puede ser usado como la fase de comprobación en un algoritmo no determinista. Si $\Pi \in P$, y A es un algoritmo determinista en tiempo polinomial para Π, se puede obtener un algoritmo no determinista en tiempo polinomial para Π únicamente usando A como fase de comprobación e ignorando la fase de suposición. Por lo tanto, $\Pi \in P$ implica $\Pi \in NP$.

En un sentido más coloquial, el párrafo anterior equivale a decir que si es posible encontrar una solución en tiempo polinomial, también será posible comprobar en tiempo polinomial si es solución. De hecho, se ha conseguido hallar la solución.

Aunque los algoritmos no deterministas en tiempo polinomial parecen tener un mayor potencial a la hora de resolver problemas de decisión, no hay ninguna razón para afirmar que $P \neq NP$. El siguiente resultado muestra cual es el coste en términos de tiempo de convertir un problema de NP en uno de P:

Teorema 2.1. *Sea Π un problema de decisión, si $\Pi \in NP$, entonces existe un polinomio p tal que Π puede ser resuelto por un algoritmo de complejidad temporal $\mathcal{O}(2^{p(n)})$.*

Demostración. Sea A un algoritmo no determinista en tiempo polinomial para Π, y sea $q(n)$ un polinomio que acota la complejidad de tiempo de A (Sin pérdida de generalidad, se asume que q puede evaluarse en tiempo polinomial tomando, por ejemplo, $q(n) = c_1 n^{c_2}$ para $c_1, c_2 \in \mathbb{N}$ suficientemente grandes). Para una cadena aceptada de longitud n debe existir alguna cadena supuesta (de Γ) de tamaño al menos $q(n)$ que lleva al estado de

comprobación de A a la respuesta "sí" para esa entrada en no más de $q(n)$ pasos. Por lo tanto, el número de posibles suposiciones que necesitamos considerar es a lo más $k^{q(n)}$, donde $k = |\Gamma|$, y como las suposiciones son mas cortas que $q(n)$ se pueden considerar como suposiciones de longitud exactamente $q(n)$ rellenando con blancos si es necesario. Ahora se puede comprobar de forma determinista si A tiene una computación aceptación dada una entrada de longitud n aplicando la fase de comprobación de A, hasta que pare o haga $q(n)$ pasos, en cada una de las $k^{q(n)}$ suposiciones. Este programa devolverá la respuesta "sí" si llega a una computación de aceptación sin llegar al límite de tiempo y "no"en caso contrario. Esto da un algoritmo determinista para resolver Π. Además, su complejidad es exactamente $q(n)k^{q(n)}$, como se quería ver.

□

La capacidad de un algoritmo no determinista de comprobar un número exponencial de posibilidades en un tiempo polinomial lleva a pensar que este tipo de algoritmos son más potentes que los deterministas. De hecho, para algunos problemas concreto de NP, como el del Vendedor ambulante o el del Isomorfismo de subgrafos entre otros, no se han podido encontrar algoritmos en tiempo polinomial para su resolución.

Por estas razones, la finalidad de este trabajo será intentar dar las técnicas que se utilizan actualmente para demostrar que $P \neq NP$ buscando los problemas más "difíciles" de NP.

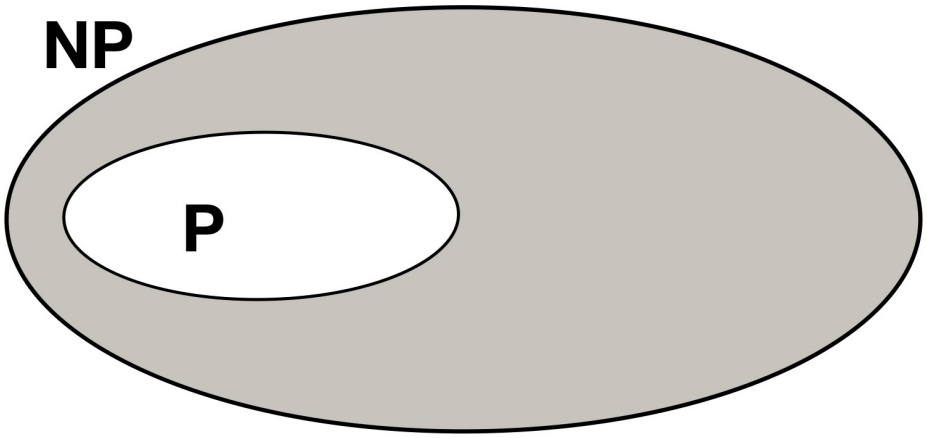


Figura 2.1: Posible imagen de NP.

2.2– Reducciones polinomiales y *NP*-Completitud

Si P fuera distinto de NP, la distinción entre P y $NP - P$ es un importante objeto de estudio. Todos los problemas de P pueden ser resueltos por un algoritmo en tiempo polinomial, mientras que los de $NP - P$ son irresolubles físicamente con la potencia computacional actual. Por lo tanto, dado un problema de decisión $\Pi \in NP$, el objetivo es saber que si pertenece o no a P.

Naturalmente, hasta que no se pruebe que $P \neq NP$, no hay esperanzas de mostrar que ningún problema particular pertenece a $NP - P$. Por esta razón, la teoría de la NP-completitud se centra en probar resultados más débiles del tipo "si $P \neq NP$, entonces $\Pi \in NP - P$". Veremos que, aunque este método puede parecer igual de complicado, existen técnicas que nos permiten dar resultados de forma directa.

La clave para llevar a cabo esta tarea son las reducciones polinomiales. Una *reducción polinomial* de un lenguaje $L_1 \subseteq \Sigma_1^*$ a un lenguaje $L_2 \subseteq \Sigma_2^*$ es una función $f : \Sigma_1^* \to \Sigma_2^*$ satisfaciendo la dos siguientes condiciones:

1. Existe un programa de una MTD que computa f en tiempo polinomial.
2. Para todo $x \in \Sigma_1^*$, $x \in L_1$ si y solo si $f(x) \in L_2$.

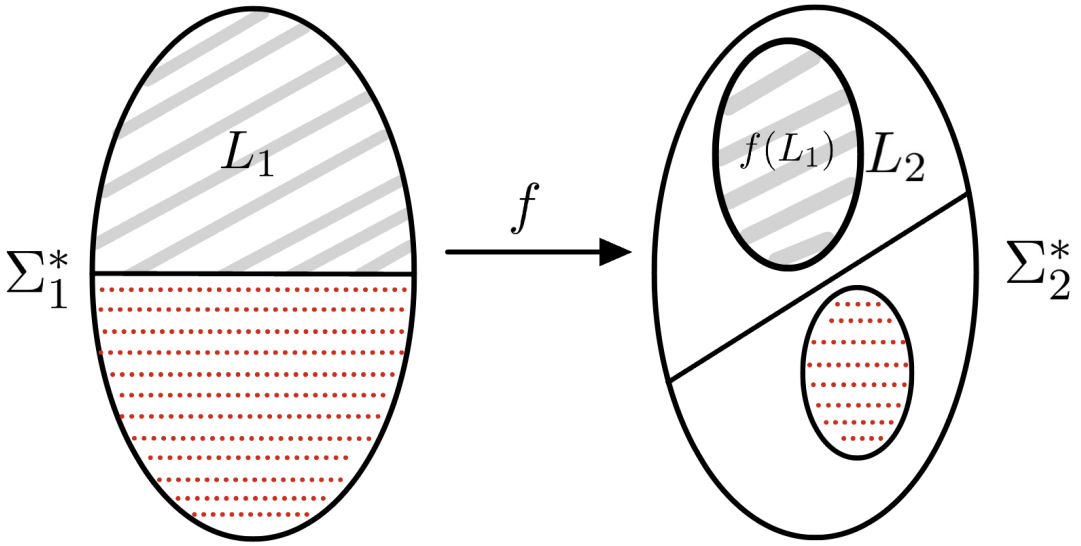


Figura 2.2: Esquema de una reducción polinomial f, $x \in L_1$ si y solo si $f(x) \in L_2$.

Si existe tal reducción, "L_1 se reduce a L_2" y se denotará $L_1 \leq_p L_2$, de aquí en adelante. La importancia de las reducciones se verá en el siguiente lema:

Lema 2.2. *Si $L_1 \leq_p L_2$ y $L_2 \in P$ entonces $L_1 \in P$.*

Demostración. Sean Σ_1 y Σ_2 los alfabetos de L_1 y L_2 respectivamente, $f : \Sigma_1^* \to \Sigma_2^*$ una reducción polinomial de L_1 en L_2, M_f la MTD que computa f, y M_2 la MTD que reconoce L_2. Es posible construir una MTD para L_1 componiendo M_f con M_2. Para $x \in \Sigma_1^*$, se aplica en primer lugar M_f para obtener $f(x) \in \Sigma_2^*$. Seguidamente se aplica M_2 para comprobar si $f(x) \in L_2$. Como $L_1 \leq_p L_2$, $x \in L_1$ si y solo si $f(x) \in L_2$, hemos construido una MTD que reconoce L_1. Que este programa opere en tiempo polinomial se sigue de que M_f y M_2 también lo hacen. □

Volviendo a los problemas de decisión, si Π_1 y Π_2 son dos problemas de decisión con esquemas de codificación asociados e_1 y e_2, diremos que $\Pi_1 \leq_p \Pi_2$ si existe una reducción polinomial de $L[\Pi_1, e_1]$ en $L[\Pi_2, e_2]$.

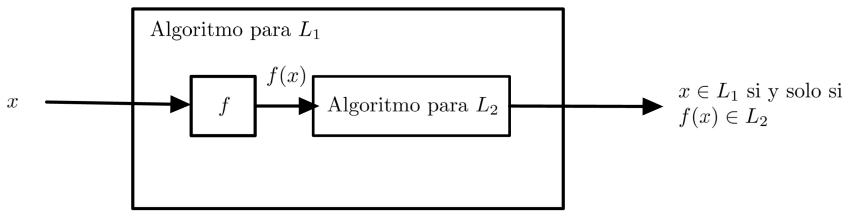


Figura 2.3: Obtención de un algoritmo para L_1 a partir de una reducción y un algoritmo polinomial para L_2.

Uno de los hechos por lo que la "reducibilidad polinomial" es importante es debido a que cumple la propiedad transitiva, la cual es demostrada en el siguiente lema.

Lema 2.3. *Si $L_1 \leq_p L_2$ y $L_2 \leq_p L_3$, entonces $L_1 \leq_p L_3$.*

Demostración. Sean Σ_1, Σ_2 y Σ_3 los alfabetos de L_1, L_2 y L_3 respectivamente, sea $f_1 : \Sigma_1^* \to \Sigma_2^*$ una reducción polinomial de L_1 en L_2, y $f_2 : \Sigma_2^* \to \Sigma_3^*$ una reducción polinomial de L_2 en L_3. Entonces la función $f : \Sigma_1^* \to \Sigma_3^*$ definida como $f(x) = f_2(f_1(x))$ para todo $x \in \Sigma_1^*$ es la transformación de L_1 en L_3 buscada. Esto es así pues la composición de dos funciones que pueden ser computadas en tiempo polinomial también puede ser computada en tiempo polinomial; y porque

$$(x \in L_1 \iff f_1(x) \in L_2) \ \wedge \ (f_1(x) \in L_2 \iff f_2(f_1(x)) = f(x) \in L_3)$$

Luego $x \in L_1$ si y solo si $f(x) \in L_3$ □

Se dirá que L_1 y L_2 son *polinomialmente equivalentes* si se cumple $L_1 \leq_p L_2$ y $L_2 \leq_p L_1$. El lema 2.3 nos prueba que $\leq_p$ nos da una relación de orden legítima en la clase de los lenguajes. De hecho, la clase P forma la menor clase de equivalencia bajo este orden parcial, pudiendo ser vista como los lenguaje más "fáciles". La clase de los problemas NP-completos forma otra clase de equivalencia, destacada por contener los problemas más difíciles de NP.

Formalmente, un lenguaje L se define como *NP-duro* si, para todo lenguaje $L' \in NP$, se tiene que $L' \leq_p L$. Si además, $L \in NP$, entonces se dirá que L es NP-completo. Un problema de decisión Π es NP-completo si $\Pi \in NP$ y, para todos los problemas de decisión Π' se tiene que $\Pi' \leq_p \Pi$.

Es importante que el problema cumpla ambas condiciones, de hecho existen problemas que son NP-duros y no pertenecen a NP como el problema de la Parada, que no es computable por ninguna máquina de Turing, pero sí es NP-duro[1].

El lema 2.3 sugiere que los problemas NP-completos se identifican con los más difíciles de NP. Sí algún problema NP-completo puede ser resuelto en tiempo polinomial, entonces todos los demás problemas de NP, también. Si algún problema de NP es intratable, este tendrá que ser un problema NP-completo. Un problema Π NP-completo cumple la propiedad que buscábamos: "si $P \neq NP$, entonces $\Pi \in NP - P$". Exactamente, $\Pi \in P$ si y solo si $P = NP$.

Asumiendo que $P \neq NP$, la figura 2.4 esquematiza el "mundo NP". Como se observa, en $NP - P$ hay tanto problemas NP-completos como otro que no están en P.

[1]Una prueba de esto se puede consultar en [1].

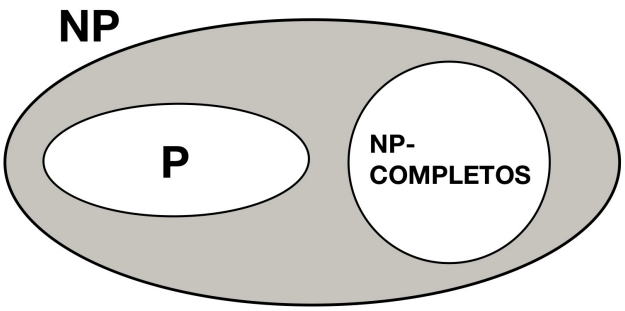


Figura 2.4: Posible imagen de NP, añadiendo los problemas NP-completos.

El interés del trabajo es el estudio de los problemas NP-completos. Probar que un problema es NP-completo puede ser un poco exigente: uno debe probar que todo problema de NP se reduce a el posible problema NP-completo con el que se está trabajando. El siguiente lema simplificará la tarea proporcionando una técnica directas para ello.

Lema 2.4. *Si L_1 y L_2 pertenecen a NP, L_1 es NP-completo, y $L_1 \leq_p L_2$, entonces L_2 es NP-completo.*

Demostración. Si L_1 y L_2 pertenecen a NP, L_1 es NP-completo, y $L_1 \leq_p L_2$, entonces L_2 es NP-completo. Como $L_2 \in NP$, falta ver que, para todo $L' \in NP$, $L' \leq_p L_2$. Sea $L' \in NP$. Como L_1 es NP-completo, $L' \leq_p L_1$. Aplicando la transitividad se tiene el resultado. □

Volviendo al mundo de los problemas de decisión, este lema da esa manera directa de probar que un problema es NP-completo, una vez que se haya encontrado un primer problema NP-completo. Para probar que Π es NP-completo, solo es necesario mostrar que:

1. $\Pi \in NP$, y

2. un problema NP-completo conocido Π' se reduce a Π.

Antes de poder usar ese enfoque, será preciso encontrar ese primer problema NP-completo. Dicho problema será tratado en la próxima sección.

2.3– Teorema de Cook

El problema de decisión que tiene el honor de ser el primer NP-completo es el problema de lógica boolena "Satisfiability" (SAT abreviado). En esta sección se desarrollarán los términos necesarios para describirlo, así como la prueba de que este problema es, efectivamente, NP-completo.

El problema SAT se plantea como sigue:

Satisfiability

INSTANCIA: Sean U un conjunto de variables booleanas y C un conjunto de cláusulas sobre U.

PREGUNTA: ¿Existe una asignación de verdad satisfactible para C?

Una vez definido el problema SAT, vamos a enunciar y probar el teorema de Cook. La versión que aquí se expone fue la que Cook dio en su artículo "The Complexity of Theorem Proving Procedures" en 1971 y se extrae del libro [7]. Este teorema fue probado independientemente por Leonid Levin aproximadamente en la misma fecha, por lo que también se conoce como "Teorema de Cook-Levin".

Teorema 2.5. *Satisfiability es NP-completo.*

Demostración. Sean $U = \{u_1, u_2, \ldots, u_m\}$ un conjunto finito de variables booleanas y C una colección de cláusulas.

En primer lugar, ha de probarse que SAT pertenece a la clase *NP*. Para ello damos un algoritmo 1 no determinista que resuelve el problema:

Algoritmo 1 SAT

Entrada: C una colección de cláusulas sobre $U = \{u_1, \ldots, u_n\}$
$\sigma \leftarrow \emptyset$
FASE NO DETERMINISTA
para $i \leftarrow 1, n$ **hacer**
    Elegir e_1, e_2
    $e_1 : \sigma \leftarrow \sigma \cup \{(u_i, T)\}$
    $e_2 : \sigma \leftarrow \sigma \cup \{(u_i, F)\}$
FASE DETERMINISTA
Comprobar si σ es una asignación de verdad satisfactoria para C
si σ es una asignación de verdad satisfactoria para C **entonces**
    devolver **sí**
si no
    devolver **no**

En segundo lugar, volviendo a nivel de los lenguajes, SAT está representado por el lenguaje $L_{SAT} = L[SAT, e]$ para algún esquema de codificación e. Debido a que hay infinitos lenguajes en NP no es posible dar una reducción para cada uno de ellos. Sin embargo, cada uno de estos lenguajes están descritos por una *MTND* polinomial. Este hecho permite trabajar con una *MTND* genérica M y dar una reducción general entre L_M y L_{SAT}. Por lo tanto, se probará también que $L_M \leq_p L_{SAT}$ para todo $L \in NP$.

Sea M una *MTND* polinomial, dada por Γ, Σ, $\square$, $\mathcal{Q}$, q_0, q_Y, q_N y δ que reconoce el lenguaje $L = L_M$. Con el motivo de facilitar el acceso a las celdas de la cinta, estas aparecerán enumeradas durante la prueba, siendo el cero el número de la casilla por el que empieza la cabeza lectora, y situando a la derecha los enteros positivos y a la izquierda los negativos en orden creciente. Además, sea $p(n)$ un polinomio sobre los naturales que acota la función complejidad en tiempo $T_M(n)$.

Para simplificar la prueba, f_L será descrita como una aplicación entre cadenas sobre Σ e instancias de SAT, en lugar de cadenas para el alfabeto del esquema de codificación de SAT. Por lo tanto, f_L cumplirá que para toda $x \in \Sigma^*$, $x \in L$ si y solo si $f_L(x)$ tiene una asignación de verdad satisfactoria. La clave de la construcción de f_L es mostrar como los conjuntos de cláusulas se usan para comprobar si una entrada x es aceptada por una $MTND$ M, es decir, si $x \in L$.

Si la entrada $x \in \Sigma^*$ es aceptada por M, entonces existe una computación de aceptación para M sobre x tal que el número de pasos en la computación de aceptación y el número de

símbolos en la cadena supuesta están acotados por $p(n)$, donde $n = |x|$. Dicha computación no puede involucrar ningún cuadrado de la cinta que no esté entre el $-p(n)$ y el $p(n)+1$, debido a que la cabeza lectora comienza en el cuadrado 1 y se mueve como máximo un cuadrado en cada paso. El estado de la computación de comprobación se especifica dando los contenidos de esos cuadrados, el estado en que se encuentra M y la posición de la cabeza lectora. Asimismo, como no hay más de $p(n)$ pasos en la computación de aceptación, a lo más se deben considerar $p(n)+1$ tiempos distintos. Esto nos permite describir dicha computación usando únicamente un número finito de variables y una asignación de verdad asociadas a ellas.

Se procede, a continuación, a construir el conjunto de variables U que resulta al reducir el problema de decisión. Identificamos los elementos de $\mathcal{Q}$ como $q_0, q_1 = q_Y, q_2 = q_N, q_3, \ldots, q_r$, con $r = |\mathcal{Q}| - 1$; y los elementos de Γ, como $s_0 = \square, s_1, \ldots, s_v$, y $v = |\Gamma| - 1$. Se describen tres tipos de variables cada una de las cuales tendrá una función específica detallada en la tabla 2.1

Variable	Rango	Significado
Q[i,k]	$0 \leq i \leq p(n)$ $0 \leq k \leq r$	M está en q_k en tiempo i
H[i,j]	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$	La cabeza lectora escanea la celda j en el instante i
S[i,j,k]	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$ $0 \leq k \leq v$	El símbolo s_k está en el cuadrado j en el instante i

Tabla 2.1: Variables de f_L y sus significados.

Una computación de M induce una asignación de verdad de una manera directa, bajo la asunción de que si el programa para antes del tiempo $p(n)$, la configuración permanece estática en los siguiente pasos, manteniendo el mismo estado de parada, la posición de la cabeza y el contenido de la cinta. En el tiempo 0 el contenido de la cinta consiste en la entrada x escrita en las celdas de la 1 a la n, y de la suposición w, escrita en las celdas de la -1 a la $-|w|$, con los demás cuadrados blancos.

Por otra parte, una asignación de verdad arbitraria para estas variables no tiene por qué corresponderse con una computación, mucho menos con una computación de aceptación que contenga los símbolos en un instante concreto, la máquina podría estar simultáneamente en varios estados y la cabeza lectora podría estar en cualquier subconjunto de posiciones entre $-p(n)$ y $p(n)+1$. La transformación f_L consiste en construir,usando las variables anteriores, una colección de cláusulas cumpliendo que una asignación de verdad es satisfactible si y solo si es la asignación de verdad inducida por computación de aceptación para x cuya fase de comprobación tome $p(n)$ o menos pasos y cuya cadena supuesta tenga longitud al menos $p(n)$. En este caso, se tiene:

$$
\begin{aligned}
x \in L \quad &\Leftrightarrow \quad \text{hay una computación de aceptación de } M \text{ sobre } x \\
&\Leftrightarrow \quad \text{hay una computación de aceptación de } M \text{ sobre } x \\
&\qquad\quad \text{con } p(n) \text{ o menos pasos en su fase de comprobación} \\
&\qquad\quad \text{y su cadena supuesta } w \text{ tiene longitud exactamente } p(n) \\
&\Leftrightarrow \quad \text{existe una asignación de verdad satisfactible para la} \\
&\qquad\quad \text{colección de cláusulas de } f_L(x).
\end{aligned}
$$

Gracias a esto f_L satisfará una de las dos condiciones necesarias para ser una reducción polinomial (existe una MTD que computa f_L). Falta que f_L puede ser computada en tiempo polinomial.

Las cláusulas en $f_L(x)$ se pueden dividir en seis grupos, a cada uno de los cuales se le impondrá un tipo de restricción distinta en su asignación de verdad satisfactible como se ve en la tabla 2.2.

Grupo de cláusulas	Restricción impuesta
G_1	En cada tiempo i, M está en exactamente un estado
G_2	En cada tiempo i, la cabeza lectora está en exactamente una celda
G_3	En cada tiempo i, cada celda contiene exactamente un símbolo de Γ
G_4	En el tiempo 0, la computación está en la configuración inicial de la fase de comprobación para la entrada x
G_5	En tiempo $p(n)$, M ha alcanzado q_Y y aceptado x
G_6	Para cada tiempo i, $0 \leq i \leq p(n)$, la configuración de M en tiempo $i+1$ se sigue de una única función de transición δ desde la configuración i

Tabla 2.2: Grupos de cláusulas y su restricción correspondiente.

Por la forma en la que se han construido los grupos de cláusulas si se cumplen, entonces existe una asignación de verdad satisfactible para x. Por lo tanto, todo lo que hay que mostrar es cómo pueden se construidos dichos grupos de cláusulas.

El grupo G_1 está formado por:

$$\{Q[i,0], Q[i,1], \ldots, Q[i,r]\} \qquad 0 \leq i \leq p(n)$$
$$\{\overline{Q[i,j]}, \overline{Q[i,j']}\} \qquad 0 \leq i \leq p(n), \quad 0 \leq j < j' \leq p(n)$$

Las primeras $p(n)+1$ de estas cláusulas pueden ser simultáneamente satisfechas si y solo si para cada tiempo i, M está en al menos un estado. Las demás $(p(n)+1)(r+1)(r/2)$ cláusulas pueden ser simultáneamente satisfechas si y solo si no existe un tiempo i en el que M esté en más de un estado. Es decir, G_1 cumple la misión de asegurar que en el tiempo i, M esté en un único estado.

Análogamente, los grupos G_2 y G_3 se encargan, respectivamente, de que la cabeza lectora escanee una única celda y de que cada cuadrado contenga un único símbolo en cada tiempo i. Los grupos G_4 y G_5 son más simples y están formados por cláusulas de un literal. En la tabla 2.3. Notar que el número de cláusulas en estos grupos y el número de literales en cada cláusulas están acotados por una función polinomial que depende de n (ya que r y v son constantes determinadas por M y, por lo tanto, por L).

El último grupo de cláusulas G_6 garantiza que cada configuración tenga una única configuración sucesora. A su vez, este grupo está dividido en dos subgrupos de cláusulas.

El primer subgrupo asegura que si la cabeza lectora no está escaneando la celda j en el tiempo i, entonces el símbolo en el cuadrado j no cambia entre los tiempos i y $i+1$. Las cláusulas de este subgrupo son como siguen:

$$\{\overline{S[i,j,l]}, H[i,j], S[i+1,j,l]\}, \quad 0 \leq i < p(n), -p(n) \leq j \leq p(n)+1, 0 \leq l \leq v$$

Grupo de cláusulas	Cláusulas en el grupo
G_1	$\{Q[i,0], Q[i,1], \ldots, Q[i,r]\} \quad 0 \leq i \leq p(n)$ $\{\overline{Q[i,j]}, \overline{Q[i,j]'}\} \quad 0 \leq i \leq p(n), \quad 0 \leq j < j' \leq p(n)$
G_2	$\{H[i,-p(n)], H[i,-p(n)+1], \ldots, H[i,p(n)+1]\} \quad 0 \leq i \leq p(n)$ $\{\overline{H[i,j]}, \overline{H[i,j]'}\} \quad 0 \leq i \leq p(n), \quad -p(n) \leq j < j' \leq p(n)$
G_3	$\{S[i,j,0], S[i,j,1], \ldots, S[i,j,v] \quad 0 \leq i \leq p(n),$ $-p(n) \leq j \leq p(n)$ $\{\overline{S[i,j,k]}, \overline{S[i,j,k]'}\} \quad 0 \leq i \leq p(n), -p(n) \leq j \leq p(n),$ $0 \leq k < k' \leq v$
G_4	$\{Q[0,0]\}, \{H[0,1]\}, \{S[0,0,0]\},$ $\{S[0,1,k_1]\}, \{S[0,2,k_2]\}, \ldots, \{S[0,n,k_n]\}$ $\{S[0,n+1,0]\}, \{S[0,n+2,0]\}, \ldots, \{S[0,p(n)+1,0]\}$ donde $x = s_{k_1}, s_{k_2}, \ldots s_{k_n}$
G_5	$\{Q[p(n),1]\}$

Tabla 2.3: Primeros 5 grupos de Cláusulas.

Para cualquier tiempo i, celda j y símbolo s_l, ocurre que si la cabeza lectora no está leyendo la celda j en el tiempo i, y el cuadrado j contienen el símbolo s_l en tiempo i pero no en tiempo $i+1$, entonces la cláusula de arriba que depende de i, j y l no será satisfecha. Por lo tanto, las $2(p(n)+1)^2(v+1)$ de este grupo cumplen su función.

El subgrupo restante de G_6 garantiza que el cambio de una configuración a su sucesora se haga de acuerdo a la función de transición δ de M. Para cada cuádrupla (i,j,k,l), $0 \leq i < p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq r$ y $0 \leq l \leq v$, el subgrupo contiene los siguientes tres grupo de cláusulas:

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1, j+\Delta]\}$$
$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, Q[i+1,k']\}$$
$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, S[i+1,j,l']\}$$

donde si $q_k \in Q - \{q_Y, q_N\}$, los valores de Δ, k' y l son tales que $\delta(q_k, s_l) = (q_{k'}, s_{l'}, \Delta)$ y si $q_k \in \{q_Y, q_N\}$, entonces, $\Delta = 0$, $k' = k$ y $l' = l$.

Estas $6p(n)(p(n)+1)(r+1)(v+1)$ cláusulas imponen la restricciones buscadas a la asignación de verdad.

Por lo tanto, se ha dado una forma general de construir los grupos de cláusulas buscados. Si $x \in L$, existe una computación de aceptación de M sobre x de tamaño a los más $p(n)$, y dicha computación, dada la interpretación de las variables, impone una asignación de verdad que satisface todas las cláusulas en $C = \bigcup_{i=1}^{6} G_i$.

A la inversa, la construcción de C es tal que cualquier asignación de verdad satisfactible para C debe corresponder a una computación de aceptación de M sobre x. Es decir, $f_L(x)$ si tiene una asignación de verdad satisfactible si y solo si $x \in L$.

Falta ver que, para un lenguaje L dado, $f_L(x)$ puede ser construida a partir de x en tiempo polinomial en función de $n = |x|$. Dado L, se toma una $MTND$ M que reconozca L en tiempo polinomial acotado por un polinomio p. Una vez que se tienen M y un polinomio p concretos, es posible construir el conjunto de variables U y la colección de cláusulas C. La acotación polinomial de esta computación se tiene inmediatamente una vez se haya probado que $Tamaño(f_L(x))$ está acotada superiormente por un polinomio en

función de n, donde $Tamaño(I)$ refleja el tamaño de la instancia I bajo un esquema de codificación razonable. Dicha función *Tamaño* para el problema SAT puede ser: $|U||C|$. Ninguna cláusula contendrá mas de $2|U|$ literales, y el número de símbolos requeridos para describir un literal necesita añadir el factor $\log(|U|)$, que esta acotado polinomialmente. Como r y v están fijos los añaden factores constantes a $|U|$ y a $|C|$. Se tiene que $|U| = \mathcal{O}(p(n)^2)$ y $|C| = \mathcal{O}(p(n)^2)$. De ahí que la función $Tamaño(I) = |U||C| = \mathcal{O}(p(n)^4)$, es decir, está acotada polinomialmente en función de n como se buscaba.

Por lo tanto, la reducción f_L se puede computar con un algoritmo en tiempo polinomial que depende de L y de las elecciones de M y p. Se concluye que para todo $L \in NP$, f_l es una reducción polinomial de L a L_{SAT}, y, por tanto, SAT es NP-completo.

□

2.4– Problemas *NP*-completos clásicos

La NP-completitud es una de las herramientas fundamentales que se utilizan hoy en día para estudiar la frontera entre P y NP. En la presente sección, se abordarán algunos de los problemas NP-completos más relevantes como pueden ser 3-SATISFIABILITY, 3-DIMENSIONAL MATCHING, VERTEX COVER, HOMILTONIAN CIRCUIT Y PARTITION. En cada apartado se expondrá el problema y se dará una prueba de su NP-completitud siguiendo el esquema de reducciones que se muestra en la figura 2.5. Estas pruebas generalmente se basan en dar una reducción adecuada partiendo del Teorema de Cook 2.5 que nos dice que SAT es NP-completo. Gracias a esto se entenderá la importancia de dicho teorema y el por qué se ha invertido tiempo y esfuerzo en su demostración. En su mayoría, las pruebas están tomadas de [7], aunque también se han extraido detalles de [12].

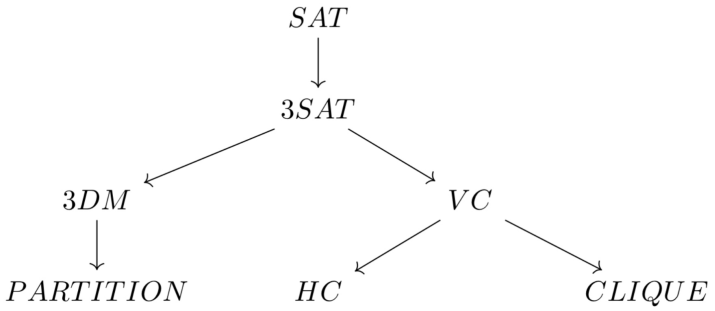


Figura 2.5: Secuencia de reducciones de los problemas NP-completos que se trabajarán.

2.4.1. 3-SATISFIABILITY

El problema 3-SATISFIABILITY es una versión de SATISFIABILITY donde se le impone a las cláusulas la restricción de que estén formadas por exactamente 3 literales. Su estructura hace que sea uno de los problemas más usados a la hora de probar resultados de NP-completitud.

3-SATISFIABILITY (3SAT)

INSTANCIA: Sean U un conjunto de variables booleanas y $C = \{c_1, c_2, \ldots, c_m\}$ un conjunto finito de cláusulas sobre U cumpliendo $|c_i| = 3$ para $1 \leq i \leq m$.

PREGUNTA: ¿Existe una asignación de verdad satisfactible para C?

Teorema 2.6. *3SAT es NP-completo.*

Demostración. Como SAT está en *NP* 3SAT también los estará. Para probar que 3SAT es *NP*-completo se dará una reducción de SAT a 3SAT. Sea $U = \{u_1, u_2, \ldots, u_n\}$ un conjunto finito de variables y $C = \{c_1, c_2, \ldots, c_m\}$ un conjunto de cláusulas de un problema SAT. El objetivo será construir una colección de cláusulas de tres literales C' sobre un conjunto de variables U' cumpliendo que C' es satisfactible si y solo si C lo es.

La construcción de C' consiste en remplazar cada cláusula original $c_j \in C$ por una colección C'_j de tres literales "equivalente", que dependen de las variables de U y algunas variables auxiliares $y_j \in U_j$ que solo aparecerán en las cláusulas C'_j y cuya función es que las nuevas cláusulas sean equivalentes a las de C. Esto se hará fijando:

$$U' = U \cup \left\{ \bigcup_{j=1}^{m} U'_j \right\}$$

y

$$C' = \bigcup_{j=1}^{m} C'_j$$

A continuación se mostrará como construir C'_j y U'_j a partir de c_j.

Sea la cláusula dada por $\{z_1, z_2, \ldots, z_k\}$ donde z_i son literales sobre las variables de U. La manera de construir C'_j y U'_j dependerá de los valores de k.

Caso 1. $k = 1$. $U'_j = \{y_j^1, y_j^2\}$
$$C'_j = \{\{z_1, y_j^1, y_j^2\}, \{z_1, y_j^1, \overline{y_j^2}\}, \{z_1, \overline{y_j^1}, y_j^2\}, \{z_1, \overline{y_j^1}, \overline{y_j^2}\}\}$$

Caso 2. $k = 2$. $U'_j = \{\{y_j^1\}\},\ C'_j = \{\{z_1, z_2, y_j^1\}, \{z_1, z_2, \overline{y_j^1}\}$

Caso 3. $k = 3$. $U'_j = \emptyset, C'_j = \{\{c_j\}\}$

Caso 4. $k \geq 4$. $U'_j = \{y_j^i : 1 \leq i \leq k-3\}$
$$C'_j = \left\{\{z_1, z_2, y_j^1\}\right\} \cup \left\{\{\overline{y_j^i}, z_{i+2}, y_j^{i+1}\} : 1 \leq i \leq k-4\right\}$$
$$\cup \left\{\{\overline{y_j^{k-3}}, z_{k-1}, z_k\}\right\}$$

Para probar que esto es una reducción, hay que ver que el conjunto de cláusulas C' es satisfactible si y solo si lo es C. Sea $t : U \to \{T, F\}$ una asignación de verdad satisfactible para C. A continuación se extenderá t a $t' : U' \to \{T, F\}$, una asignación de verdad satisfactoria para C'. Como las variables de $U' - U$ se particionan en los conjuntos U_j y las variables de U'_j solo están en los conjuntos C'_j, basta probar que t se puede extender a los conjuntos U_j y, en cada caso, hay que comprobar que cada cláusula en las C'_j se satisface. Para ello se plantea la siguiente solución distinguiendo cada caso:

- Si U_j proviene de una cláusula de Caso 1 o el Caso 2, entonces sus variables pueden, debido a la manera en la que están construidas, tomar cualquier valor. Una asignación posible es, por ejemplo, $t'(y) = T$ para todo $y \in U_j$.

- Si U_j provine del caso 3 no hay nada que modificar.

- Si U_j proviene de una cláusula de Caso 4, es decir corresponde a una cláusula del tipo $\{z_1, z_2, \ldots, z_k\}$ de C con $k \geq 4$. Como t es una asignación satisfactible, debe existir un literal z_l con $1 \leq l \leq k$ tal que $t(z_l) = T$. Si $l = k-1$ o $l = k$, entonces tomamos $t'(y_j^i) = F$ para todo $1 \leq i \leq k-3$. Si $l = 1$ o $l = 2$, entonces tomamos $t'(y_j^i) = T$ para todo $1 \leq i \leq k-3$. En otro caso, tomamos $t'(y_j^i) = T$ para $1 \leq i \leq l-2$ y $t'(y_j^i) = T$ para $l-1 \leq i \leq k-3$.

Seguidamente, se va a comprobar que la fórmula para construir las nuevas cláusulas es la que se busca:

- En el Caso 1, como c_j tiene un único literal, z_l con $1 \leq l \leq n$ la asignación de verdad t de SAT debe cumplir que $t(z_l) = T$. Para que todas las cláusulas de C_j se verifique debe ocurrir que $t'(z_1) = T$, pues si fuera falso por la forma de añadir los literales no hay una asignación para y_j^1, y_j^2 que satisfaga las cuatros cláusulas simultáneamente.

- En el Caso 2, ocurre algo parecido, pues al añadir un literal en una cláusula y en otro su negación, las cláusulas de C_j se satisfacen si y solo si se satisface la cláusula c_j de las que provienen.

- En el Caso 3, puesto que la cláusula ya tiene tres literales, no hay nada que hacer.

- Para el Caso 4, se utilizará ejemplo para su compresión. Sea $c_j = \{z_1, z_2, z_3, z_4, z_5\}$. Está cláusula da lugar al siguiente conjunto de cláusulas:

$$C_j = \left\{ \{z_1, z_2, y_j^1\}, \{\overline{y_j^1}, z_3, y_j^2\}, \{\overline{y_j^2}, z_4, z_5\} \right\}$$

  Entonces, si $t(z_1) = T$ o $t(z_2) = T$, para que se satisfagan las dos cláusulas restantes asignamos $t'(y_j^1) = t'(y_j^2) = F$. Análogamente se procede en los otros casos.

Por otra parte, si t' es una asignación de verdad satisfactible para C', su restricción a U, t, es una asignación de verdad satisfactible para C, como se quería ver.

Falta probar, que la reducción se da en tiempo polinomial. En el peor de los casos los subconjuntos de cláusulas C_j con $1 \leq j \leq m$ tienen de cuatro cláusulas o más. Esto ocurre si el número de literales de las cláusulas es mayor o igual que 6, $k \geq 6$. En este caso $|C_j| = k-2$ y $k-2 \leq n-2$. Como hay m subconjuntos de este tipo tenemos que $|C'| \leq m(n-2) \leq mn$. Como la instancia de 3SAT está acotada superiormente por un polinomio en función de la instancia de SAT se tiene el resultado.

Se concluye que SAT$\leq_p$3SAT.

□

2.4.2. 3-Dimensional Matching

3-Dimensional Matching (3DM) es otro problema fundamental en la Teoría de la Complejidad Computacional. Este problema también es NP-completo (hecho que se probará en esta sección) y se utiliza comúnmente como ejemplo de un problema de combinación.

El problema 3DM se define de la siguiente manera: Dados tres conjuntos X, Y y Z, cada uno con q elementos, y un conjunto W de w ternas ordenadas (x, y, z), donde x pertenece a X, y pertenece a Y y z pertenece a Z, se pregunta si existe un conjunto M de q ternas ordenadas, cada una de las cuales tiene un elemento de X, un elemento de Y y un elemento de Z, de tal manera que cada elemento X, Y y Z aparece en exactamente una terna.

El problema de 3DM tiene aplicaciones en la informática teórica, en la investigación operativa y en la teoría de grafos. Por ejemplo, se ha usado para resolver problemas de asignación de recursos y en la optimización de rutas en sistemas de transporte. Además, se ha utilizado para modelar problemas en la biología molecular, como la alineación de secuencias de ADN.

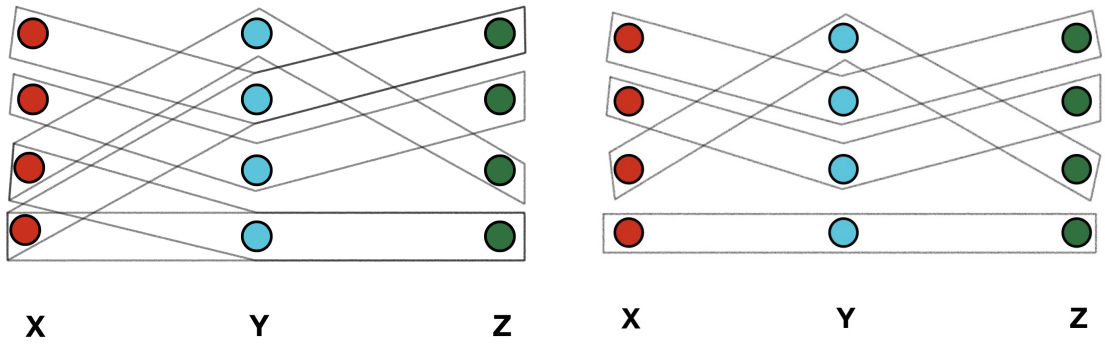
3-Dimensional Matching (3DM)

INSTANCIA: Sean X, Y y Z tres conjuntos finitos disjuntos de igual cardinalidad q, y sea $W \subset X \times Y \times Z$

PREGUNTA: ¿Existe un subconjunto $M \subset W$ de cardinalidad q tal que ninguno de sus elementos coincida en ninguna coordenada?

A tal subconjunto M se le llama correspondencia.

La siguiente figura 2.6a representa el problema 3DM. Las bolas son los elementos de X, Y y Z, y la unión de todos los recuadrado es el conjunto T. La pregunta equivale a si se pueden encontrar un subconjunto de T de cardinalidad q de manera que ningún recuadro tenga puntos en común con los demás. Una posible solución del problema se puede ver a la derecha en 2.6b.



(a) Ejemplo de 3DM. (b) Posible solución del ejemplo.

Figura 2.6

Teorema 2.7. *3-Dimensional Matching es NP-completo.*

Demostración. Sean X, Y y Z tres conjuntos finitos disjuntos de igual cardinalidad q, y sea $W \subset X \times Y \times Z$. En primer lugar, para ver que 3DM está en NP basta dar un algoritmo que seleccione un subconjunto de cardinalidad q de W en la fase no determinista y comprobar que las coordenadas de ninguno de sus elementos coincidan en tiempo polinomial.

A continuación se dará una reducción de 3SAT a 3DM. Sea $U = \{u_1, u_2, \ldots, u_n\}$ un conjunto finito de variables y $C = \{c_1, c_2, \ldots, c_m\}$ un conjunto de cláusulas de una instancia de 3SAT. A partir se estos se construirán tres conjuntos X, Y y Z de cardinalidad q y un conjunto $W \subset X \times Y \times Z$ que contenga una correspondencia si y solo si C es satisfactible.

El conjunto W se particionará en tres clases:

1. Las componentes de **"establecimiento de verdad"** se corresponden con una única variable $u \in U$, y su estructura depende del número total m de cláusulas en C. Esta estructura se ilustra para el caso de $m = 4$ en la figura 2.7. En general, la componente de establecimiento de verdad para una variable u, está compuesta por los elementos "internos" $a_i[j] \in X$ y $b_i[j] \in Y$, $1 \leq j \leq m$, que no aparecen en ninguna terna fuera de esta componente, y elementos "externos" $u_i[j], \overline{u_i[j]} \in Z$, $1 \leq j \leq m$, que aparecerán en otros ternas. Las ternas que conforman esta componente se pueden dividir en dos conjuntos:

$$T_i^t = \left\{(a_i[j], b_i[j], \overline{u_i[j]}) : 1 \leq j \leq m\right\}$$
$$T_i^f = \{(a_i[j+1], b_i[j], u_i[j]) : 1 \leq j < m\} \cup \{(a_i[1], b_i[m], u_i[m])\}$$

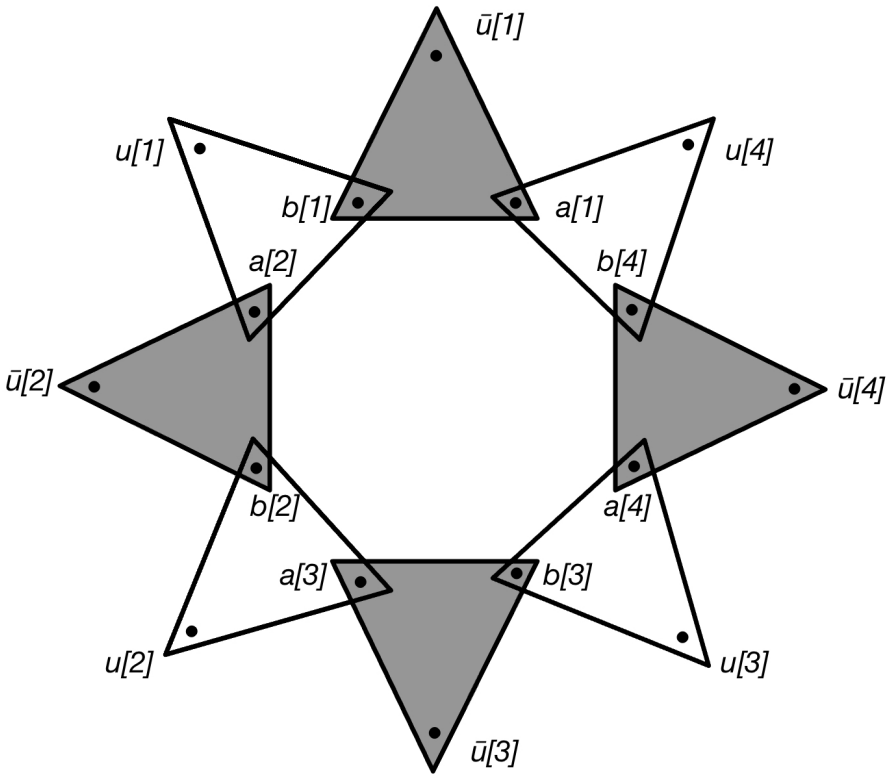


Figura 2.7: Componente de establecimiento de verdad T_i para $m = 4$. Todo los conjuntos de T_i^t o todos los de T_i^f se deben seleccionar dejando sin cubrir a todos los $u_i[j]$ o todos los $\overline{u_i[j]}$, respectivamente.

Como los elementos internos $\{a_i[j], b_i[j] : 1 \leq j \leq m\}$ solo aparecerán en ternas de $T_i = T_i^t \cup T_i^f$, cualquier correspondencia M tendrá exactamente m ternas T_i, los cuales pertenecerán todos a T_i^t o a T_i^f. De ahí que la componente T_i pueda ser entendida como establecer la variable u_i cierta o falsa. Por lo tanto, en general una correspondencia $M \subset W$ responde a una asignación de verdad t para U, con $t(u)$ verdadera si y solo si $M \cap T_i = T_i^t$.

2. Las componente de **"testeo de satisfacción"** en W se corresponden con una cláusula $c_j \in C$. En ellas intervienen dos elementos internos $s_1[j] \in X$ y $s_2[j] \in Y$, y un elemento externo del conjunto $\{u_i[j], \overline{u_i[j]} : 1 \leq j \leq m\}$, dependiendo de los literales que aparezcan en c_j. El conjunto de ternas que forman esta componente se definen como sigue:

$$C_j = \{(s_1[j], s_2[j], u_i[j]) : u_i \in c_j\} \cup \left\{(s_1[j], s_2[j], \overline{u_i[j]}) : \overline{u_i} \in c_j\right\}$$

En la figura 2.8 podemos ver un ejemplo de una componente C_j.

Esta componente se encarga de que en la correspondencia $M \subset W$ cada terna aparezca solo una vez. Esto solo puede ocurrir si para algún literal $u_i[j]$ (o $\overline{u_i[j]}$) correspondiente al literal $u_i \in c_j$ ($\overline{u_i} \in c_j$) no aparece en la terna de $M \cap T_i$, y esto se dará si y solo si la asignación de verdad determinada por M satisface c_j.

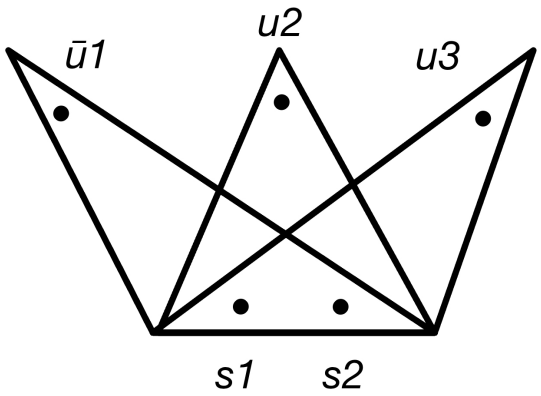


Figura 2.8: Componente de testeo de satisfacción C_j para la cláusula $c = \{\overline{u_1}, u_2, u_3\}$.

3. Por último, para completar W, se define la componente de **"residuos"** G tal y como se ve en la figura 2.9, que contiene elementos internos $g_1[k] \in X$ y $g_2[k] \in Y$, $1 \leq k \leq m(n-1)$, y elementos externos de la forma $u_i[j]$ y $\overline{u_i[j]}$ de Z. Está formada por el siguiente conjunto de ternas:

$$\begin{aligned} G = &\{(g_1[k], g_2[k], u_i[j]), (g_1[k], g_2[k], \overline{u_i[j]}) : \\ &1 \leq k \leq m(n-1), 1 \leq i \leq n, 1 \leq j \leq m\} \end{aligned}$$

Cada par $g_1[k], g_2[k]$ debe emparejarse a una única $u_i[j]$ o $\overline{u_i[j]}$ que no aparezca en ningún terna de $M - G$. La estructura de G permite que los $m(n-1)$ elementos

externos que permanecen en este momento sin cubrir puedan ser cubierto eligiendo adecuadamente $M \cap G$. Por lo tanto, G nos garantiza que, una vez que un subconjunto de $W - G$ satisfaga las restricciones de la componentes de establecimiento de verdad, este conjunto se pueda extender a la correspondencia que se busca.

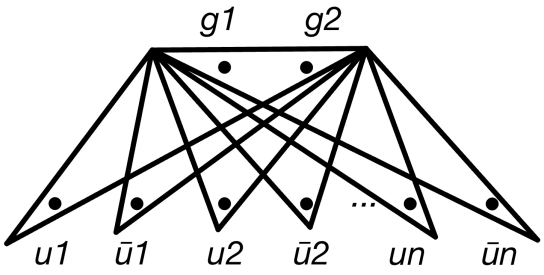


Figura 2.9: Componente de "residuos".

En resumen se tiene:

- $Z = \{u_i[j], \overline{u_i[j]} : 1 \leq i \leq n, 1 \leq j \leq m\}$
- $X = A \cup S_1 \cup G_1$
  donde:
  - $A = \{a_i[j] : 1 \leq i \leq n, 1 \leq j \leq m\}$
  - $S_1 = \{s_1[j] : 1 \leq j \leq m\}$
  - $G_1 = \{g_1[k] : 1 \leq k \leq m(n-1)\}$
- $Y = B \cup S_2 \cup G_2$
  donde:
  - $B = \{b_i[j] : 1 \leq i \leq n, 1 \leq j \leq m\}$
  - $S_2 = \{s_2[j] : 1 \leq j \leq m\}$
  - $G_2 = \{g_2[k] : 1 \leq k \leq m(n-1)\}$

  y
- $W = \{\bigcup_{i=1}^{n} T_i\} \cup \left\{\bigcup_{j=1}^{m} C_j\right\} \cup G$

Por construcción tenemos que $W \subset X \times Y \times Z$ y, además, W tiene

$$2mn + 3m + 2m^2n(n-1)$$

ternas y puede ser construida en tiempo polinomial a partir de una instancia de 3SAT.

Falta ver que la existencia de una una asignación de verdad satisfactible para C implica que W tenga una correspondencia.

Sea $t : U \to \{T, F\}$ asignación de verdad satisfactible para C. M se construye de la siguiente manera: Para cada cláusula $c_j \in C$, sea $z_j \in \{u_i, \overline{u_i} : 1 \leq i \leq n\} \cap c_j$ un literal

cuya asignación de verdad sea verdadera (al menos debe existir uno pues t satisface C). A continuación, tomamos:

$$M = \left\{ \bigcup_{t(u_i)=T} T_i^t \right\} \cup \left\{ \bigcup_{t(u_i)=F} T_i^f \right\} \cup \left\{ \bigcup_{i=1}^{m} \{(s_1[j], s_2[j], z_j[j])\} \right\} \cup G'$$

Donde G' contiene a los $g_1[k]$, los $g_2[k]$ y los $u_i[j]$ y $\overline{u_i[j]}$ restantes.
Se concluye que 3SAT$\leq_p$3DM.

□

2.4.3. Vertex Cover y Clique

A pesar de que Vertex Cover y Clique son útiles para probar resultados de *NP*-completitud por separado, no dejan de ser dos formas distintas de mirar el mismo problema. Para verlo, se introducirá un tercer problema, Independent Set.

Vertex Cover(VC)

Instancia: un grafo $G = (V, E)$ y un natural $K \leq |V|$.

Pregunta: ¿Existe un recubrimiento de vértices de tamaño menor o igual que K, es decir, un subconjunto $V' \subset V$ tal que $|V'| \leq K$ tal que para toda arista $\{u, v\} \in E$ al menos u o v pertenezca a V'?

Clique

Instancia: un grafo $G = (V, E)$ y un natural $J \leq |V|$.

Pregunta: ¿Contiene G un clique de tamaño mayor o igual que J, es decir, un subconjunto $V' \subset V$ tal que $|V'| \geq J$ y todos sus vértices estén unidos por una arista?

Independent Set

Instancia: un grafo $G = (V, E)$ y un natural $J \leq |V|$.

Pregunta: ¿Contiene V un conjunto independiente de tamaño mayor o igual que J, es decir, un subconjunto $V' \subset V$ tal que $|V'| \geq J$ tal que, para todo $u, v \in V'$, la arista $\{u, v\} \notin E$?.

El siguiente lema muestra las relaciones entre estos tres problemas:

Lema 2.8. *Sea $G = (V, E)$ un grafo y $V' \subset V$, son equivalentes*

(a) V' es un recubrimiento de vértices de G.

(b) $V - V'$ es un conjunto independiente de G.

(c) $V - V'$ es un clique en G^c.

El lema 2.8 implica que si es posible conseguir una solución para uno de los tres problemas se tendrá una solución para los otros dos. Por ejemplo, para reducir VC a Clique, dada la instancia de VC: un grafo $G = (V, E)$ y un natural $K \leq |V|$; se corresponde con la instancia de Clique: un grafo G^c y un natural $J = |V| - K$. Por esta razón, se probará únicamente la NP-completitud de VC.

Teorema 2.9. VERTEX COVER *es NP-completo.*

Demostración. Para ver que VERTEX COVER está en *NP*, se que comprueba el algoritmo no determinista que procede de la siguiente manera resuelve VC en tiempo polinomial. En la fase no determinista se genera un conjunto V' de todos los subconjuntos posibles de V de tamaño menor o igual que K. En la fase determinista, se toma un vértice de V' y se eliminan las aristas adyacentes, consiguiendo así $G' = (V, E')$, donde E' es el conjunto que se obtiene eliminando las aristas adyacentes a v. Se repite este proceso en G' hasta remover K vértices. Si en ese momento $E' = \emptyset$, entonces existe un recubrimiento de vértices de G; en caso contrario V' no es un recubrimiento de vértices de G.

En segundo lugar, se reducirá 3SAT a VC. Sea $U = \{u_1, u_2, \ldots, u_n\}$ un conjunto finito de variables y $C = \{c_1, c_2, \ldots, c_m\}$ un conjunto de cláusulas de una instancia de 3SAT. Se construirá, seguidamente, un grafo $G = (V, E)$ y un número natural $K \leq |V|$ tal que G tenga un recubrimiento de vértices de tamaño K si y solo si C es satisfactible.

Al igual que en la reducción de 3SAT a 3DM, se procederá mediante la creación de componentes que dependen de las variables y las cláusulas. En este caso se añadirán, además, aristas que nos permitan conectar las diferentes componentes:

- A cada variable $u_i \in U$ se le asigna una componente de **"establecimiento de verdad"** $T_i = (V_i, E_i)$, donde $V_i = \{u_i, \overline{u_i}\}$ y $E_i = \{\{u_i, \overline{u_i}\}\}$. Es decir, dos vértices unidos por una arista. Cualquier recubrimiento de vértices de tener al un vértice de T_i para $1 \leq i \leq n$.

Figura 2.10: Componente establecimiento de verdad asociada a la variable u_i.

- A cada cláusula $c_j \in C$ se le asigna una componente de **"testeo de sastisfacción"** $S_j = (V'_j, E'_j)$. S_j está compuesta por tres vértices y tres aristas en forma de triángulo:

$$V'_j = \{a_1[j], a_2[j], a_3[j]\}$$
$$E'_j = \{\{a_1[j], a_2[j]\}, \{a_1[j], a_3[j]\}, \{a_2[j], a_3[j]\}\}$$

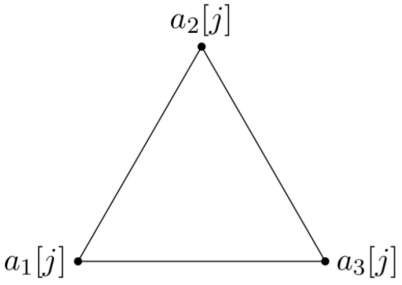


Figura 2.11: Componente de testeo de satisfacción C_j para la cláusula c_j.

Obsérvese que para recubrir las aristas de E'_j son necesarios al menos 2 vértice de V'_j.

- Por último, la parte que se encarga de conectar todas las componentes, el conjunto de aristas de comunicación. Sea $c_j \in C$ una clásula formada por los literales x_j, y_j y z_j. Se construye el siguiente conjunto de aristas:

$$E''_j = \{\{a_1[j], x_j\}, \{a_2[j], y_j\}, \{a_3[j], z_j\}\}$$

Estableciendo $K = n + 2m$ y $G = (V, E)$, donde

$$V = \left(\bigcup_{i=1}^{n} V_i\right) \cup \left(\bigcup_{j=1}^{m} V'_j\right)$$

y

$$E = \left(\bigcup_{i=1}^{n} E_i\right) \cup \left(\bigcup_{j=1}^{m} E'_j\right) \cup \left(\bigcup_{j=1}^{m} E''_j\right)$$

se concluye la construcción de la instancia de VC.

En la figura 2.12 se muestra un ejemplo del grafo que se obtiene a partir de $U = \{u_1, u_2, u_3, u_4\}$, $C = \{\{u_1, \overline{u_3}, \overline{u_4}\}, \{\overline{u_1}, u_2, \overline{u_4}\}\}$.

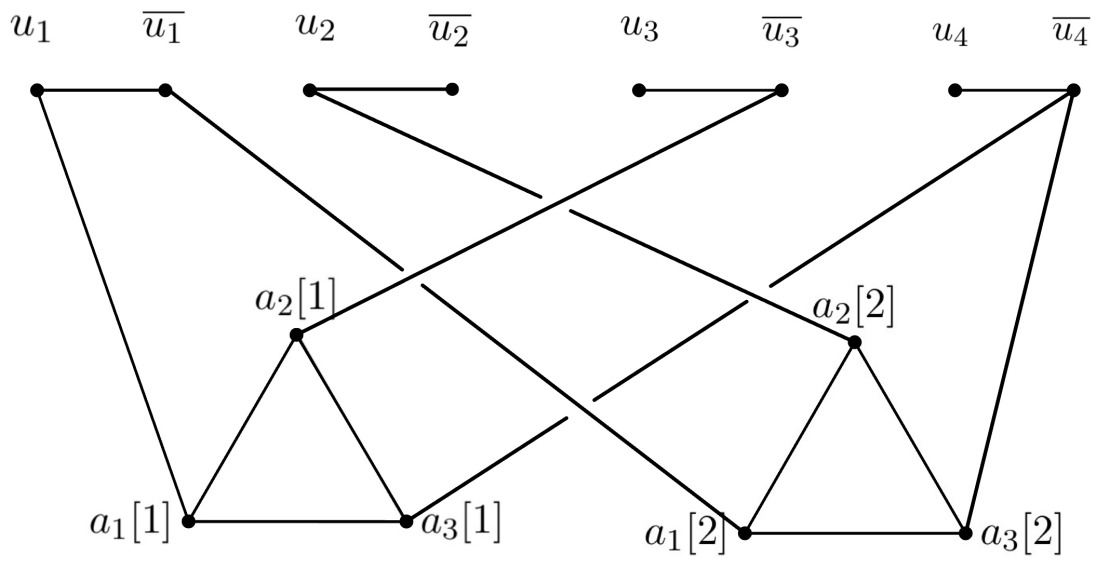


Figura 2.12: Instancia de VC resultante de la instancia de 3SAT $U = \{u_1, u_2, u_3, u_4\}$, $C = \{\{u_1, \overline{u_3}, \overline{u_4}\}, \{\overline{u_1}, u_2, \overline{u_4}\}\}$. Aquí $K = n + 2m = 8$.

La reducción puede completarse en tiempo polinomial pues de la construcción resultan $2n+3m$ vértices (dos vértices por variables y tres por cláusula) y $n+3m+3m$ aristas (una arista por variable, 3 por cada componente de testeo y $3m$ por las aristas de comunicación). Lo único que queda por probar es que C es satisfactible si y solo si G tiene un recubrimiento de vértices de tamaño K o menor.

En primer lugar, sea $V' \subset V$ un recubrimiento de vértices de G con $|V'| \leq K$. Según se señaló en los párrafos anteriores, V' debe contener al menos un vértice en cada T_i y al menos dos en cada S_j. Como esto da que V' contiene $n + 2m = K$ vértices, ocurre que V'

tiene un vértice en cada T_i y dos en cada S_j, exactamente. Este hecho permite considerar la siguiente asignación de verdad: $t(u_i) = T$ si $u_i \in V'$ y $t(u_i) = F$ si $\overline{u_i} \in V'$. Para ver que esta asignación de verdad satisface todas las cláusulas de $c_j \in C$, se consideran la tres aristas de E''_j. Solo dos de estas aristas pueden ser recubiertas por vértices de $V'_j \cap V'$, así que al menos una de ellas es cubierta por un vértice de algún V_i que pertenezca a V'. Pero esto supone que el literal correspondiente a ese vértice, ya sea u_i o $\overline{u_i}$, de la cláusula c_j debe ser verdadero bajo la asignación de verdad t, y, por lo tanto, c_j se satisface bajo t. Como esto se tiene para todo $c_j \in C$, t es una asignación de verdad satisfactoria.

Recíprocamente, sea t una asignación de verdad satisfactoria para C. El recubrimiento de vértices correspondiente V' contiene un único vértice de cada T_i y dos de cada S_j. El vértice de T_i en V' es u_i si $t(u_i) = T$ y u_i si $t(\overline{u_i}) = F$. Esto garantiza que al menos una de las tres arista de E''_j esté cubierta, pues t satisface todas las cláusulas $c_j \in C$. Por consiguiente, solo necesitamos incluir en V' los vértices extremos de S_j de las otras dos aristas de E''_j, obteniendo así el recubrimiento de vértices deseado.

Se concluye que 3SAT$\leq_p$VC.

□

2.4.4. Ciclo Hamiltoniano

En teoría de grafos se conoce como ciclo Hamiltoniano a aquel que recorre todos los vértices del grafo una única vez. Es decir, dado $G = (V, E)$ un grafo existe una sucesión $\langle v_1, v_2, \ldots, v_n \rangle$ que contenga todos los vértices de V una única vez y que $\{v_i, v_{i+1}\} \in E$ para $1 \leq i < n$ y $\{v_n, v_1\} \in E$.

El problema del ciclo Hamiltoniano consiste en dado un grafo, determinar si este contiene o no un ciclo Hamiltoniano:

Ciclo Hamiltoniano (CH)

INSTANCIA: un grafo $G = (V, E)$.

PREGUNTA: ¿Contiene G un ciclo Hamiltoniano?

Teorema 2.10. CICLO HAMILTONIANO *es NP-completo.*

Demostración. CH es NP pues solo se necesita crear un ciclo de los posibles (fase no determinista) y comprobar si recorre una vez todos los vértices del grafo (fase determinista), y recorrer los vértices del grafo se completa en tiempo polinomial.

A continuación, se reducirá VERTEX COVER a CH. Dada una instancia de VC ,un grafo $G = (V, E)$ y un natural $K \leq |V|$, se construirá una instancia $G' = (V', E')$ tal que G' tenga un ciclo Hamiltoniano si y solo si G tiene un recubrimiento de vértices.

De nuevo la reducción se hará en términos de componentes. En primer lugar G' constará de K **"vértices selectores"**: $a_1, a_2, .., a_K$ cuya función será seleccionar K de V, los vértices de G. En segundo lugar para cada arista de E, G' tendrá una **"componente de testeo de recubrimiento"** que garantizará que al menos uno de los extremos de la arista está entre los K vértices seleccionados. A la arista $e = \{u, v\} \in E$ le corresponderá la componente que aparece en la figura 2.13. Tendrá 12 vértices,

$$V'_e = \{(u, e, i), (v, e, i) : 1 \leq i \leq 6\}$$

y 14 aristas,

$$\begin{aligned} E'_e = &\{\{(u, e, i), (u, e, i+1)\}, \{(v, e, i), (v, e, i+1)\} : 1 \leq i \leq 5\} \\ &\cup \{\{(u, e, 3), (v, e, 1)\}, \{(v, e, 3), (u, e, 1)\}\} \\ &\cup \{\{(u, e, 6), (v, e, 4)\}, \{(v, e, 6), (u, e, 4)\}\} \end{aligned}$$

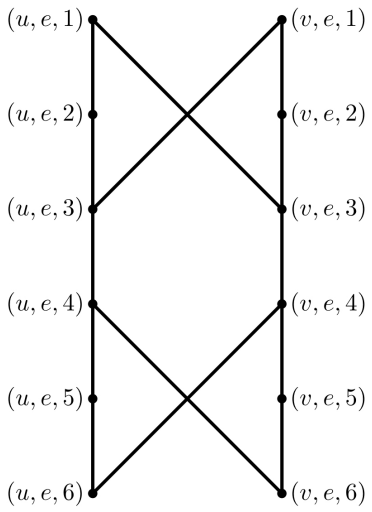


Figura 2.13: Componente de testeo de recubrimiento correspondiente a la arista $\{u, v\}$.

Los únicos vértices de la componente de testeo de recubrimiento que aparecerán en otras aristas son :$(u, e, 1), (v, e, 1), (u, e, 6)$ y $(v, e, 6)$. Esto supone que cualquier ciclo Hamiltoniano de G' debe unirse con E'_e en alguna de las tres configuraciones que se muestran en la figura 2.14. Por ejemplo, si el ciclo entra por el vértice $(u, e, 1)$ tendrá que salir por $(u, e, 6)$ y visitará los 12 vértices o solo los 6 del tipo (u, e, i) con $1 \leq i \leq 6$.

El siguiente tipo de aristas que intervienen en la construcción tendrán la función de unir componentes de testeo de recubrimiento con vértices selectores. Para cada vértice $v \in V$, sean $e_{v[1]}, e_{v[2]}, \ldots, e_{v[gr(v)]}$, donde $gr(v)$ denota el *grado* de v en G, es decir, el número de aristas que llegan a v. Todas la componentes de testeo de recubrimiento de estas aristas (que tienen a v como extremo) se unen con las siguientes aristas de conexión:

$$E'_v = \big\{\{(v, e_{v[i]}, 6), (v, e_{v[i+1]}, 1)\} : 1 \leq i \leq gr(v)\big\}$$

Como se observa en la figura 2.15, esto crea un único camino en G' que incluye exactamente los vértices (x, y, z) donde $x = v$.

El último grupo de aristas de comunicación en G' une el primer y el último vértice de dichos caminos con los vértices selectores $a_1, a_2, .., a_K$. Estas tienen la siguiente forma:

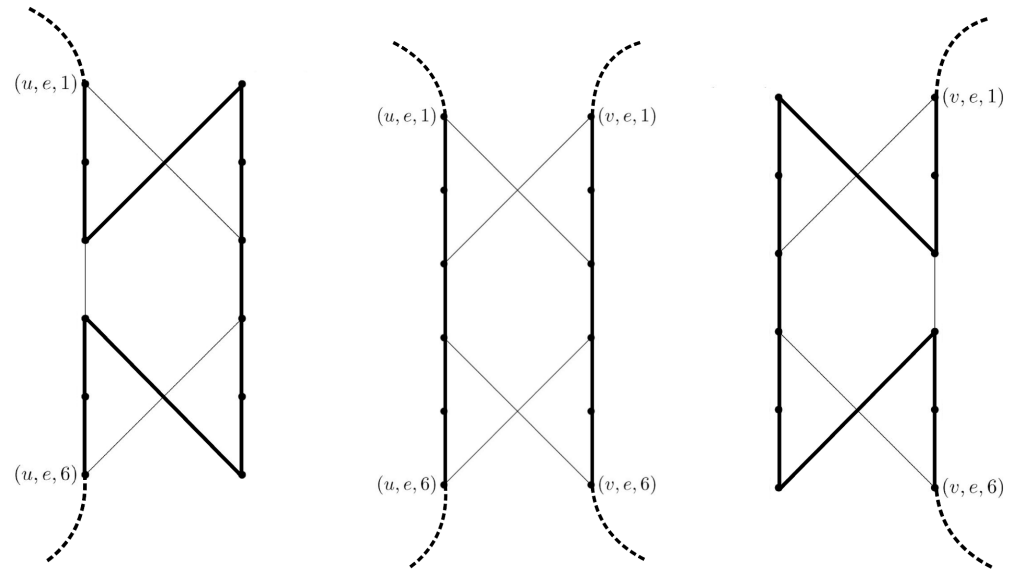
$$E'' = \big\{\{a_i, (v, e_{v[1]}, 1)\}, \{a_i, (v, e_{v[gr(v)]}, 6)\} : 1 \leq i \leq K, v \in V\big\}$$

El grafo $G' = (V', E')$ se completa de la siguiente manera:

$$V' = \{a_i : 1 \leq i \leq K\} \cup \left(\bigcup_{e \in E} V'_e\right)$$

y

$$E' = \left(\bigcup_{e \in E} E'_e\right) \cup \left(\bigcup_{v \in v} E'_v\right) \cup E''$$



(a) u pertenece al recubrimiento de vértices pero no lo hace v.

(b) u y v pertenece al recubrimiento de vértices.

(c) v pertenece al recubrimiento de vértices pero no lo hace u.

Figura 2.14: Tres posibles configuraciones de un ciclo Hamiltoniano en la componente de testeo correspondiente a la arista $\{u, v\}$.

El número de vértices y aristas del nuevo grafo G' depende de los datos de G de forma polinomial ya que hay $K + 12|E| \leq |V| + 12|E|$ vértices y, a lo más, $14|E| + |V||E| + 2|K||V| \leq 14|E| + |V||E| + 2|V|^2$ aristas (el sumando $|V||E|$ viene de que para todo vértice $v \in V$, $gr(v) \leq |E|$).

Para concluir, se probará que G' tiene un ciclo Hamiltoniano si y solo si G tiene un recubrimiento de vértices de tamaño K o menor. Sea $\langle v_1, v_2, \ldots, v_n \rangle$, donde $n = |V'|$, un ciclo Hamiltoniano de G'. Se considera una parte de este ciclo que empiece en un vértice del conjunto $\{a_1, a_2, \ldots, a_K\}$, acabe en un vértice del conjunto $\{a_1, a_2, \ldots, a_K\}$ y que no se encuentra ningún vértice interno. Debido a las restricciones con las que un ciclo Hamiltoniano recorre la componente de testeo de recubrimiento, esta parte del ciclo debe pasar por un conjunto de componentes de testeo de recubrimiento correspondientes exactamente a una de las aristas de E que son incidentes en algún vértice $v \in V$. Cada componente de testeo de recubrimiento se recorre de una de las formas 2.14a, 2.14b o 2.14c que aparece en la figura 2.14 y no se encuentra con ningún otro vértice de otra componente. Por lo tanto, los K vértices de $\{a_1, a_2, \ldots, a_k\}$ dividen el ciclo Hamiltoniano en K caminos cada uno de los cuales corresponde a un vértice distinto $v \in V$. Como cada ciclo Hamiltoniano debe incluir todo los vértices de cada componente de testeo de recubrimiento y como los vértices de estas componentes por cada arista $e \in E$ pueden pasar solo por un camino correspondiente a un extremo de e, cada arista de E debe tener al menos un extremo entre los K vértices seleccionados. En consecuencia, este conjunto de K vértices forma el recubrimiento buscado para G.

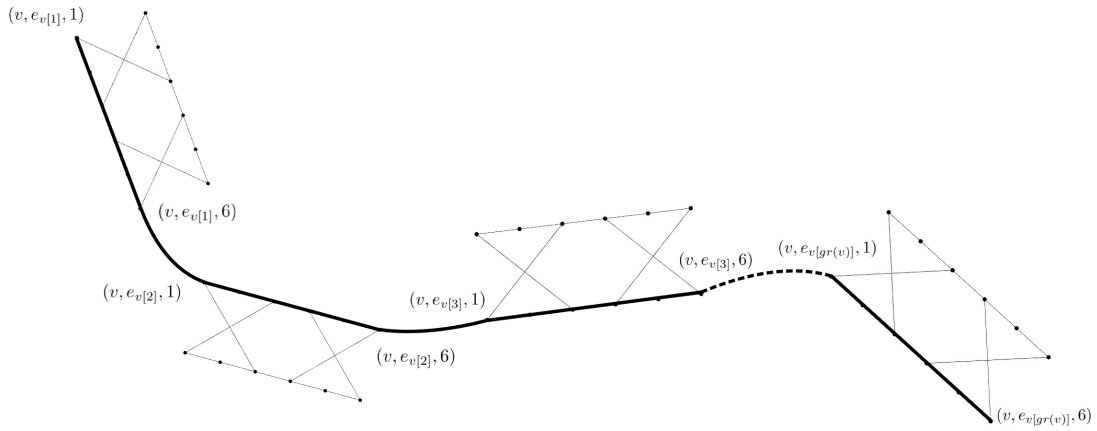


Figura 2.15: Unión por caminos de las componentes de teste de cubrimiento correspondientes a las aristas de E que tienen al vértice v como extremo.

Recíprocamente, sea $V^* \subset V$ un recubrimiento de vértices de G con $|V^*| \leq K$. Al añadir vértices a V^* hasta conseguir $|V^*| = K$ se seguirá teniendo un recubrimiento de vértices de G, luego podemos suponer que $|V^*| = K$. Sean $v_1, v_2, \ldots, v_K$ los elemento de V^*. Se tomarán las siguientes aristas de G' para formar el ciclo Hamiltoniano para G'. Para la componente correspondiente a $e = \{u, v\}$ se elige la forma de recorrer las aristas 2.14a,2.14b o 2.14c dependiendo de si $\{u, v\} \cap V^*$ es $\{u\}, \{u, v\}$ o $\{v\}$, respectivamente. Estas son las tres únicas posibilidades posibles ya que V^* es un recubrimiento de vértices de G'. Finalmente, se toman las aristas:

$$\{a_i, (v_i, e_{v_i[1]}, 1)\}, 1 \leq i \leq K, \quad \{a_i, (v_{i+1}, e_{v_i[gr(v)]}, 6)\}, 1 \leq i \leq K$$

y

$$\{a_1, (v_K, e_{v_k[gr(v_K)]}, 6)\}$$

Por lo tanto VC $\leq_p$ CH.

□

2.4.5. Partición

En esta sección se verá que Partición es un problema de optimización NP-completo. El problema de Partición se define de la siguiente manera: dado un conjunto de números enteros positivos, ¿es posible dividirlos en dos subconjuntos de manera que la suma de los elementos en cada conjunto sea la misma?

Para entender mejor el problema, supongamos que tenemos una lista de números enteros positivos como $\{1, 2, 3, 4, 5, 6, 7\}$. En este caso, una solución podría ser dividir la lista en $\{1, 2, 4, 7\}$ y $\{3, 5, 6\}$, ya que la suma de los elementos en cada subconjunto es 14.

El problema Partición tiene interés pues es útil a la hora de analizar la *NP*-completitud de problemas en los que aparecen parámetro numéricos tales como medidas de peso o altura, costes, capacidades, etc.

Partición
INSTANCIA: un conjunto finito $A \in \mathbb{Z}$ y una función $s : A \to \mathbb{Z}^+$.
PREGUNTA: ¿Contiene A un subconjunto A' tal que

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) \ ?$$

La función s que aparece en la instancia comúnmente tiene que ver con la forma de medir la magnitud de los elementos de A.

Teorema 2.11. PARTICIÓN *es NP-completo.*

Demostración. Sea un $A \in \mathbb{Z}$ conjunto finito. Para ver que PARTICIÓN es *NP*-completo basta generar un subconjunto de A en la fase no determinista y comprobar que la suma de s aplicada a sus elementos es igual la suma s aplicada a los elementos de su complementario, lo cual se realiza en tiempo polinomial.

A continuación se reducirá 3DM a PARTICIÓN. Sean X, Y y Z tres conjuntos finitos disjuntos de igual cardinalidad q, y sea $W \subset X \times Y \times Z$ correspondientes a una instancia arbitraria de 3DM. Se denotarán los elementos de cada conjunto como sigue:

$$\begin{aligned} X &= \{x_1, x_2, \ldots, x_q\} \\ Y &= \{y_1, y_2, \ldots, y_q\} \\ Z &= \{z_1, z_2, \ldots, z_q\} \\ W &= \{w_1, w_2, \ldots, w_k\} \end{aligned}$$

donde $k = |W|$. El objetivo para completar la prueba será construir un conjunto A y una función $s : A \to \mathbb{Z}^+$, tal que A contenga un subconjunto A' satisfaciendo :

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$$

si y solo si W contiene una correspondencia.

El conjunto A contendrá $k + 2$ elementos en total y se construirá en dos pasos. Los primeros k elementos de A son $\{a_i : 1 \leq i \leq k\}$, donde a_i está asociado con la terna $w_i \in W$. La función s se concretará dando su representación en binario, en términos de una cadena de $0's$ y $1's$ divididos en $3q$ "zonas" de $p = \lceil \log_2(k+1) \rceil$ bits cada una. De nuevo, cada una de estas zonas está identificada por un elemento de $X \cup Y \cup Z$, como se muestra en la figura 2.16.

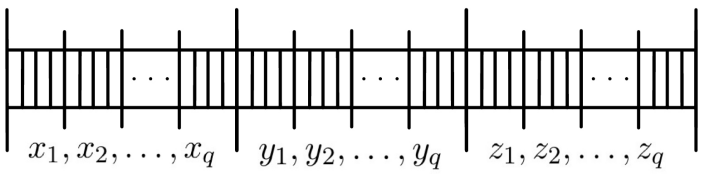


Figura 2.16: Etiquetado de las $3q$ zonas, cada una de las cuales contiene $p = \log_2(k+1)$ bits de la representación en binario de $s(a)$, usado para reducir 3DM a PARTICIÓN.

La representación de $s(a_i)$ depende de la terna $w_i = (x_{f(i)}, y_{g(i)}, z_{h(i)}) \in W$ (donde f, g y h son funciones de los subíndices de los w_i). Esta tiene un 1 en el bit más a la derecha de las zonas identificada con $x_{f(i)}, y_{g(i)}$ y $z_{h(i)}$, y $0's$ es las demás posiciones. Equivalentemente, esto se puede escribir como:

$$s(a_i) = 2^{p(3q-f(i))} + 2^{p(2q-g(i))} + 2^{p(q-h(i))}$$

Como para expresar $s(a_i)$ en binario se usan a lo más $3pq$ bits, $s(a_i)$ puede ser construido en tiempo polinomial dada una instancia de 3DM.

Lo que se debe tener en cuenta de esta parte de la construcción es que si sumamos todas las entradas de cualquier zona sobre todos los elementos $\{a_i : 1 \leq i \leq k\}$, el total nunca superará $k = 2^p - 1$. Por lo tanto, al sumar $\sum_{a \in A'} s(a)$ para cualquier subconjunto $A' \subset \{a_i : 1 \leq i \leq k\}$, no habrá ninguna llevada de una zona a la siguiente. Se sigue que si hacemos

$$B = \sum_{j=0}^{3q-1} 2^{pj}$$

(el cual es un número cuya representación en binario tiene un 1 en el lugar más a la derecha de cada zona), entonces cualquier subconjunto $A' \subset \{a_i : 1 \leq i \leq k\}$ satisfará

$$\sum_{a \in A'} s(a) = B$$

si y solo si $M = \{a_i : a_i \in A'\}$ es una correspondencia para W. El último paso de la construcción describe los dos últimos elementos de A. Estos serán denotados como b_1 y b_2, s actúa sobre ellos de la siguiente manera:

$$s(b_1) = 2\left(\sum_{i=1}^{k} s(a_i)\right) - B$$

y

$$s(b_2) = \left(\sum_{i=1}^{k} s(a_i)\right) + B$$

Como ambos pueden ser escritos en binario con no más de $3pq + 1$ bits, se pueden construir en tiempo polinomial dada una instancia de 3DM.

Sea $A' \subset A$ tal que

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$$

Se tiene que las dos sumas valen $2\sum_{i=1}^{k} s(a_i)$ y uno de los dos conjuntos, A' o $A - A'$, contiene a b_1 pero no a b_2. Se sigue que los elementos restantes del conjunto forman un subconjunto de $\{a_i : 1 \leq i \leq k\}$ cuya suma al aplicarle s vale B, y por tanto este subconjunto es una correspondencia de W. Recíprocamente, si $M \subset W$ es una correspondencia, entonces el conjunto $\{b_1\} \cup \{a_i : w_i \in M\}$ forma el conjunto A' buscado en la instancia de PARTICIÓN. Se concluye que 3DM $\leq_p$ PARTICIÓN.

□

2.5– Problemas adicionales: Métodos para probar la *NP*-completitud de un problema

Las técnicas usadas para probar la *NP*-completitud de un problema son tan variadas como la cantidad de problemas NP-completos existentes. Sin embargo, en muchos casos es útil saber alguna de estas tres reglas: restricción, reemplazamiento local o diseño de componentes.

En esta sección, se explicará como se usan en las demostraciones las técnicas anteriores y se darán algunos ejemplos. Mientras que en algunas ocasiones varias de estas reglas aparecen en algún paso de la demostración, en otras, puede que ninguna sea válida a la hora de afrontar el problema. Por lo tanto, esto no da una clasificación de los problemas NP-completos, simplemente es una ayuda para detectarlos.

En el transcurso de las siguientes secciones se omitirá probar que los problemas pertenecen a *NP* con el motivo de aligerar las demostraciones. Los algoritmos para ver si un problema está en *NP* son similares al de los problemas tratados anteriormente así que en este caso no se parará a mostrarlos explícitamente.

2.5.1. Restricción

Las pruebas por restricciones suelen ser las más simples y, aún así, las más usadas. Dado $\Pi \in NP$, una prueba de *NP*-completitud por restricción consiste en demostrar que Π contiene un problema *NP*-completo Π' como caso particular. El groso de esta prueba está en dar las restricciones adecuadas a las instancias de Π para que el problema resultante sea equivalente a Π'. No es necesario que los dos problemas sean idénticos, sino que exista una correspondencia uno a uno entre sus instancias que respete las respuestas "sí" y "no".

El problema Recubrimiento exacto por conjunto de tres constituye un ejemplo de este tipo. El problema dice lo siguiente:

Cubrimiento exacto por conjunto de tres (X3C)

Instancia: Un conjunto finito X tal que $|X| = 3q$ con $q \in \mathbb{N}$ y una colección C de subconjuntos de X de tres elementos.

Pregunta: ¿Contiene C un subconjunto que recubra exactamente a X, es decir, existe $C' \subset C$ tal que cada elemento de X aparece únicamente en un elemento de C'?

Teorema 2.12. *X3C es NP-completo.*

Demostración. Se probará la NP-completitud del problema reduciendo 3DM a X3C. Para ello solo hay que darse cuenta que toda instancia de 3DM puede ser vista como una de X3C considerándola como un subconjunto conjunto no ordenado de $X \cup Y \cup Z$, es decir, las ternas de W pasan a ser subconjuntos de C (figura 2.17). De esta forma las correspondencias de la instancias de 3DM se identifican uno a uno con los conjuntos recubridores de tres elementos como podemos ver en la figura 2.18.

□

Otro de los problemas en los que se aplica este método es en Planificación de Multiprocesadores. Este problema consiste en estudiar, dados un conjunto de tareas con una determinada duración, un conjunto de procesadores y un tiempo límite, si es posible distribuir las tareas entre los distintos procesadores de manera que todas se finalicen antes del tiempo establecido.

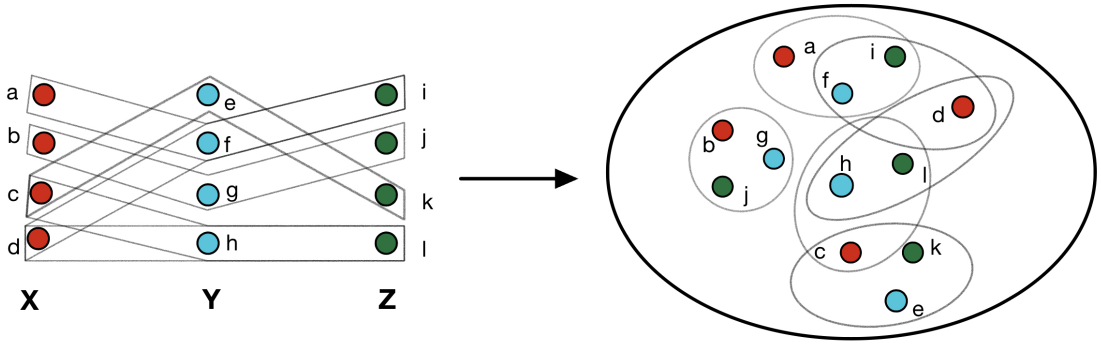


Figura 2.17: Transformación de una instancia de 3DM a una instancia de X3C. $X \times Y \times Z$ pasa ser $X \cup Y \cup Z$ y las ternas del conjunto W se convierten en los subconjuntos de tres elementos de C.

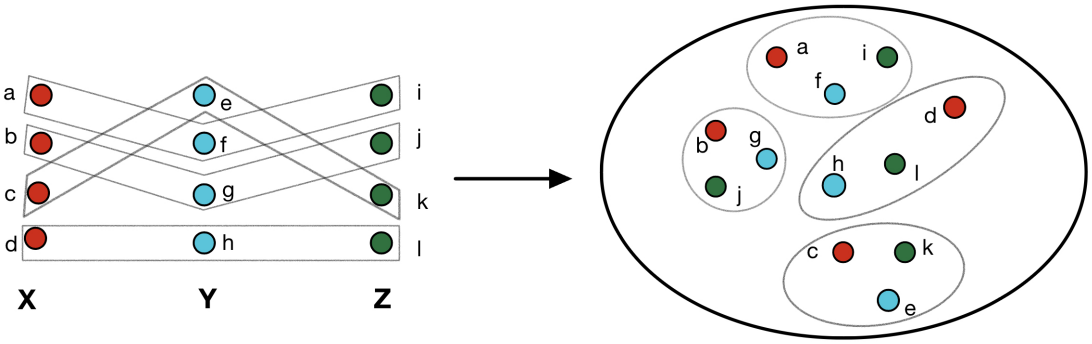


Figura 2.18: Solución del problema. Se observa que la respuesta "sí" para 3DM se mantiene para X3C.

Planificación de multiprocesadores

INSTANCIA: un conjunto finito A de tareas, una función $d : A \to \mathbb{N}$ que indica la duración de la tarea, $m \in \mathbb{N}$ procesadores, y D un tiempo límite.

PREGUNTA: ¿Existe un partición $A = A_1 \cup A_2 \cup ... \cup A_m$ de A en m subconjuntos disjuntos tal que

$$max\left\{ \sum_{a \in A_i} d(a) : 1 \leq i \leq m \right\} \leq D \ ?$$

Teorema 2.13. PLANIFICACIÓN DE MULTIPROCESADORES *es NP-completo.*

Demostración. Para probar que PLANIFICACIÓN DE MULTIPROCESADORES es NP-completo bastar darse cuenta de que si restringimos el problema instancias en las que $m = 2$ y $D = \frac{1}{2} \sum_{a \in A} d(a)$ se obtiene el problema PARTICIÓN que es NP-completo.

□

Como se ha podido observar en este tipo de pruebas la dificultad no recae en la prueba si no en saber el problema NP-completo al que se restringirá. Muchos de los problemas que aparecen en la práctica no son más que una versión más completa de algún problema ya

conocido. Por esta razón es importante saber reconocerlos y aplicar el método de restricción para probar su NP-completitud.

2.5.2. Reemplazamiento Local

En las pruebas por reemplazamiento local las reducciones son más elaboradas que en el método anterior. La técnica consiste en tomar elementos básicos de un problema NP-completo conocido y sustituir cada uno de ellos por una estructura distinta. En la reducción de SAT a 3SAT se usó este método; se dividieron los distintos tipos de cláusulas de SAT en grupos y cada uno de ellos pasó a ser un tipo de conjunto de cláusulas que se correspondía con las instancias de 3SAT. La clave es ver que cada reemplazamiento está formado únicamente por una modificación local de la estructura, es decir, son independientes unos de otros en cada estructura.

El problema Partición en triángulos constituye un ejemplo de este tipo.

Partición en triángulos

Instancia: un grafo $G = (E, v)$, con $|V| = 3q$ donde $q \in \mathbb{N}$.

Pregunta: ¿Existe un partición de V en q subconjuntos disjuntos de vértices tal que, para cada $V_i = \{v_{i[1]}, v_{i[2]}, v_{i[3]}\}$, las tres aristas $\{v_{i[1]}, v_{i[2]}\}$, $\{v_{i[1]}, v_{i[3]}\}$ y $\{v_{i[2]}, v_{i[3]}\}$ pertenezcan a E?

Teorema 2.14. Partición en triángulos *es NP-completo.*

Demostración. Para probar la NP-completitud de Partición en triángulos se dará una reducción de X3C a Partición en triángulos. Sea X un conjunto con $|X| = 3q$ y una colección de conjuntos de tres elementos C una instancia de X3C. La meta será construir un grafo $G = (V, E)$, con $|V| = 3q'$, tal que la partición buscada para G exista si y solo si C contiene un recubrimiento exacto.

Los elementos básicos de una instancia $X3C$ son los elementos de C, los subconjuntos de 3 elementos de X. El reemplazamiento local sustituirá cada subconjunto $c_i = \{x_i, y_i, z_i\}$ con $c_i \in C$, por las 18 aristas de E_i como se muestra en la figura 2.19.

Por lo tanto, $G = (V, E)$ se define como:

$$V = X \cup \bigcup_{i=1}^{|C|} \{a_i[j] : 1 \leq j \leq 9\}$$

$$E = \bigcup_{i=1}^{|C|} E_i$$

Se observa que los único vértices que puede aparecer en varias E_i simultáneamente son los que pertenecen a X. Además, $|V| = |X| + 9|C| = 3q + 9|C|$, luego $q' = q + 3|C|$. Por otra parte, $|E| = 18|C|$, por lo tanto las instancia de Partición en triángulos se puede construir en tiempo polinomial.

Si $c_1, c_2, \ldots, c_q$ son los elementos de un recubrimiento exacto C de X, entonces la correspondiente partición $V = V_1 \cup V_2 \cup \cdots \cup V_{q'}$ de V es la dada por la unión de:

$$\{a_i[1], a_i[2], x_i\}, \{a_i[4], a_i[5], y_i\}, \{a_i[7], a_i[8], z_i\}, \{a_i[3], a_i[6], a_i[9]\}$$

de los vértices de E_i si $c_i = \{x_i, y_i, z_i\}$ está en el recubrimiento exacto, y tomando:

$$\{a_i[1], a_i[2], a_i[3]\}, \{a_i[4], a_i[5], a_i[6]\}, \{a_i[7], a_i[8], a_i[9]\}$$

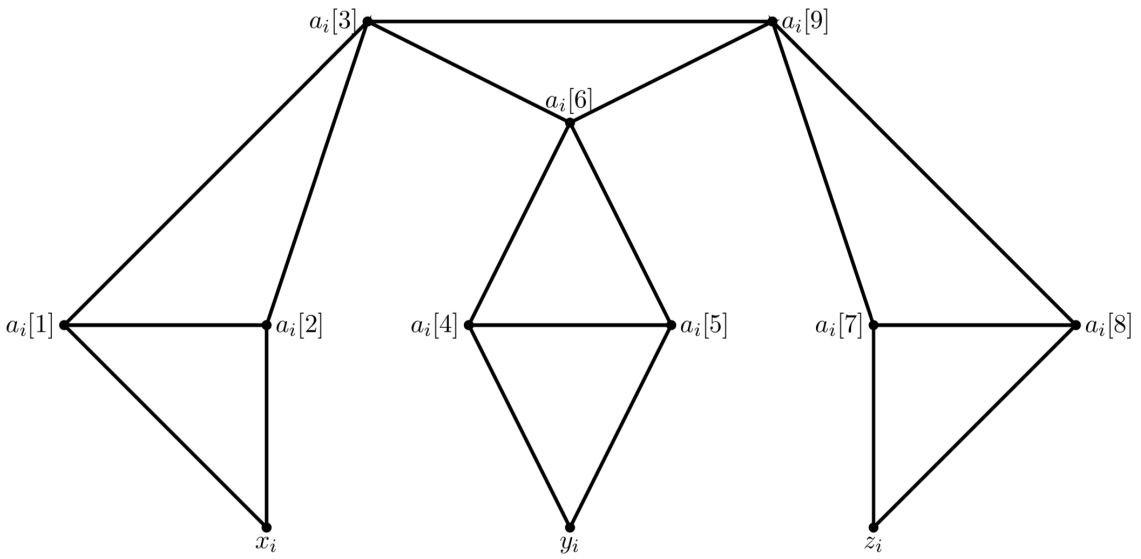


Figura 2.19: Reemplazamiento local de $c_i = \{x_i, y_i, z_i\} \in C$ para reducir X3C a Partición en triángulos.

si c_i no está en el recubrimiento. Esto garantiza que cada elemento de X esté incluido en exactamente uno de los subconjuntos de 3 vértices de la partición.

Recíprocamente, si $V = V_1 \cup V_2 \cup \cdots \cup V_{q'}$ es una partición de G en triángulos, el recubrimiento exacto de X se obtiene tomando los c_i cumpliendo que $\{a_i[3], a_i[6], a_i[9]\} = V_j$ para algún j con $1 \leq j \leq q'$.

□

2.5.3. Diseño de componentes

La última técnica que se expone es la que se usó en problemas como Vertex Cover, 3DM y Ciclo Hamiltoniano. La idea principal consiste en diseñar ciertas componentes que pueden ser combinadas para formar instancias del problema NP-completo conocido. En los tres ejemplos anteriores, hay dos tipos de componentes básicas, unas se encargan de tomar decisiones (por ejemplo, seleccionar vértices o asignar valores de verdad a las variables), mientras otras se encargan de testear propiedades (por ejemplo,comprobar que cada arista está cubierta o cada cláusula se satisface). Estas componentes se unen a los testeadores de propiedades y estos comprueban si las elecciones cumplen las restricciones requeridas. La interacción entre componentes ocurre tanto durante las conexiones (como en las aristas que comunican las componentes de establecimiento de verdad con las de testeo de satisfacción en la reducción de 3SAT a VC) como en las restricciones globales (como en la cota K en la reducción de 3SAT a VC, la cual, junto a la estructura de las componente asegura que cada componente de establecimiento de verdad contiene exactamente un vértice del recubrimiento y cada componente de testeo de satisfacción contiene exactamente dos vértices del recubrimiento).

En general, cualquier prueba en la que las instancias construidas puedan verse como un conjunto de componentes, cada de las cuales desempeña una función específica en la instancia, se puede considerar un prueba por diseño de componentes. La prueba del

Teorema de Cook es un ejemplo de esto, donde cada uno de los seis grupos de cláusulas es un tipo de componente.

CAPÍTULO 3

Aplicaciones de la NP-completitud y fronteras entre las clase P y NP

Como se ha visto en los anteriores capítulos podemos identificar P con los problemas que poseen un algoritmo "eficiente" para su resolución, mientras que los de NP son aquellos que dada una solución, es factible comprobar si esta es correcta o no. A priori, parece que hallar la solución a un problema es más costoso que comprobar si una solución es o no correcta. Por ello la comunidad científica se inclina por pensar que $P \neq NP$.

En este capítulo se expondrá cuales son algunos de los enfoques para demostrar que $P \neq NP$. Aquí es donde juegan un papel esencial los problemas NP-completos, pues si se encuentra que existe un problema NP-completo que no esté en P entonces $P \neq NP$.

En la búsqueda de un algoritmo para la resolución de un nuevo problema de NP pueden darse dos alternativas. La primera de ella es que a simple vista se tenga un algoritmo determinista en tiempo polinomial para su resolución; en este caso el esfuerzo se centrará en optimizar los métodos y obtener el mejor algoritmo posible. La otra opción es que no se encuentre un algoritmo determinista evidente; aquí el groso del trabajo será ver si el problema es o no NP-completo.

En otros muchos casos, ninguna de las dos opciones será fácil. Esto ocurre por que si $P \neq NP$ existen problemas intermedios, es decir, que no pertenecen ni a P ni son NP-completos. En la tercera sección del presente capítulo, se verá que este resultado es cierto si se asume que $P \neq NP$ exponiendo un problema de este tipo.

En otras ocasiones, en el estudio de la complejidad de un problema aparece la duda de qué ocurriría si se imponen restricciones a las instancias de un problema obteniendo así un subproblema. En este caso la intuición puede jugar una mala pasada pues no siempre al restringir un problema NP-completo sigue siéndolo. En la tabla 3.1 se exhiben algunos problemas NP-completos y su versiones en P. Estudiar estos problemas ayuda a entender la frontera entre ambas clases.

La primera sección de este capítulo versará sobre este tema, se expondrán algunos problemas de cada tipo y se verá como se puede estudiar la frontera entre P y NP. Para concluir el capítulo, se expondrán las consecuencias teóricas y prácticas que tendrían las dos posibles soluciones al problema, es decir $P \neq NP$ o $P = NP$.

P	NP-completo
Camino más corto entre dos vértices INSTANCIA: grafo $G = (V, E)$ una longitud d para cada $e \in E$, dos vértices $a, b \in V$ y un entero positivo B. PREGUNTA: ¿Existe un camino entre a y b en G de longitud menor o igual que B?	**Camino más largo entre dos vértices** INSTANCIA: grafo $G = (V, E)$ una longitud d para cada $e \in E$, dos vértices $a, b \in V$ y un entero positivo B. Pregunta: ¿Existe un camino entre a y b en G de longitud mayor o igual que B?
Edge cover INSTANCIA: grafo $G = (V, E)$ un entero positivo K. PREGUNTA: ¿Existe $E' \subset E$ con $\|E'\| \leq K$ tal que para cada $v \in V$ existe $e \in E'$ con $v \in e$?	**Vertex cover** INSTANCIA: grafo $G = (V, E)$ un entero positivo K. PREGUNTA: ¿Existe $V' \subset V$ con $\|V'\| \leq K$ tal que para cada $e \in E$ existe $v \in V'$ con $v \in e$?
Planificación de entrada INSTANCIA: T un conjunto de tareas de duración unitaria, un plazo $d(t)$ para todo $t \in T$, un orden parcial $\lessdot$ en T tal que cada tarea tiene a lo más un sucesor, y un entero positivo m. PREGUNTA: ¿Puede ser T planificada en m procesadores que respeten el orden parcial y cumplan todos los plazos?	**Planificación de salida** INSTANCIA: T un conjunto de tareas de duración unitaria, un plazo $d(t)$ para todo $t \in T$, un orden parcial $\lessdot$ en T tal que cada tarea tiene a lo más un predecesor, y un entero positivo m. PREGUNTA: ¿Puede ser T planificada en m procesadores que respeten el orden parcial y cumplan todos los plazos?

Tabla 3.1: Pareja de problemas similares, se observa como al realizar un cambio en la pregunta la clase del problema pasa de ser P a NP-completo.

3.1– Análisis de subproblemas

Si efectivamente $P \neq NP$, existirá una frontera entre los problemas de P y los problemas NP-completos. El estudio de esta frontera se lleva a cabo mediante el análisis de subproblemas. El objetivo de esta sección es, partiendo de un problema NP-completo, ver cuales son las restricciones sobre las instancias que hacen que el problema deje de ser NP-completo o pase a P, y estudiar ese "salto" entre las dos clase de complejidad. Para ello se definirá lo que es un subproblema y se expondrán un par de ejemplos.

Sea Π un problema de decisión del cual se quiere probar su NP-completitud. Posiblemente se haya llegado a este problema partiendo otro más sofisticado habiendo despreciado ciertas restricciones o obviado detalles lo que puede provocar que el problema se pueda resolver en tiempo polinomial. Si no, también cabe la posibilidad de que sigan existiendo otros casos específicos en los que el problema pueda ser tratable. Estos casos se clasifican con el análisis de subproblemas.

Como se describió en los preliminares los problemas de decisión constan de dos partes: un dominio D de que es el conjunto de todas las posibles instancias y un conjunto Y que contiene todas las instancias con respuesta "sí". Se define como *subproblema* de un problema $\Pi = (D, Y)$ como aquel problema $\Pi' = (D', Y')$ cumpliendo $D' \subset D$ e $Y' = Y \cap D'$. Es decir, Π' responde igual que Π a las preguntas pero tiene en cuenta solo un subconjunto de instancias.

Por lo tanto en un subproblema de un problema dado se imponen restricciones a las instancias permitidas del original. En problemas de teoría de grafos un ejemplo de restricción sería imponer que los grafos que intervienen sean planares, bipartitos, acíclicos, o combinaciones de estas. Para problemas que traten con conjuntos se impone restricciones en el tamaño de los conjuntos o en el número de veces que puede aparecer un elemento. De todos estos tipos de subproblemas, se eligen los que nos sean más convenientes según el propósito.

Aparentemente, al tratar con un subproblema de un problema NP-completo no se sabe cual su complejidad. Asumiendo que $P \neq NP$, los subproblemas de un problema NP-completo pueden ser vistos como una escalera entre la intratabilidad y la resolución en tiempo polinomial.

En realidad, es más preciso pensar que existe, en un momento determinado una frontera entre aquellos subproblemas que están en P con aquellos que se conoce que son NP-completos. La frontera está formada por los subproblemas cuya clasificación es desconocida a día de hoy. En la figura 3.1 se representa una posible vista del conocimiento actual de los límites entre P y los problemas NP-completos. Cada vez que se determina la clase de complejidad de un subproblema abierto se estrecha la frontera, moviendo la línea hacia la clase de complejidad a la que se determina la pertenencia.

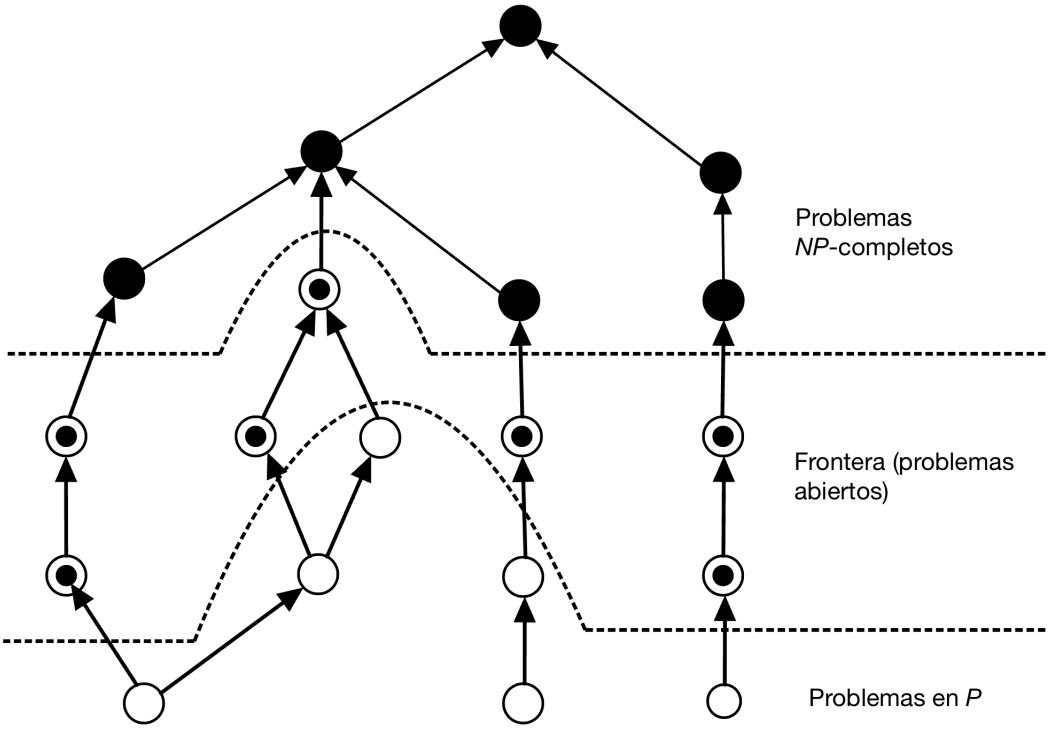


Figura 3.1: Posible imagen de la frontera entre P y NP. Una flecha desde Π_1 a Π_2 significa que Π_1 es subproblema de Π_2.

Para una mejor comprensión del asunto se mostrarán algunos problemas y subproblemas, y se verá donde está el "salto" de una clase de complejidad a otra.

El primer ejemplo será el problema de Programación restringida por Precedentes que es una versión del problema Planificación de Multiprocesadores donde

algunas tareas para poder ser completadas necesitan de la finalización de otras:

Programación restringida por Precedentes

INSTANCIA: Un conjunto T de tareas (de igual duración, se puede suponer $d(t) = 1$ para todo $t \in T$), un orden parcial $\lessdot$ en T, un número m de procesadores, y un plazo $D \in \mathbb{N}$.

PREGUNTA:¿Existe una programación $\sigma : T \to \{1, 2, \ldots, D\}$ tal que, para cada $i \in \{1, 2, \ldots, D\}$, $|\{t \in T \mid \sigma(t) = i\}| \leq m$ y cumpliendo $\sigma(t) \leq \sigma(t')$ si $t \lessdot t'$?

Este problema es el que un estudiante tiene que resolver en su primer año en una universidad en la que cada curso se ofertara cada cuatrimestre y no hubiera solapamiento de horas entre los cursos elegidos. De un conjunto T de asignaturas debe elegir como máximo m de ellas de forma que le de tiempo a acabar el grado menos de D cuatrimestres. Además, algunas de las asignaturas precisan de los conocimientos de otras, se dice que $t \lessdot t'$ si para cursar la asignatura t' se necesita haber cursado t previamente. El objetivo es diseñar un algoritmo que nos permita en tiempo polinomial, a partir de la información anterior, planificar cada cuatrimestre.

Se conoce que el problema de PROGRAMACIÓN RESTRINGIDA POR PRECEDENTES es NP-completo. Se puede consultar una prueba de este hecho en [19]. Por lo tanto, suponiendo que $P \neq NP$, sería improbable que se dé un algoritmo de planificación en tiempo polinomial. Sin embargo, restricciones razonables pueden ser aplicadas a las instancias, como, por ejemplo, que un alumno no curse más de 6 asignaturas por cuatrimestre, es decir, $m \leq 6$. También puede ocurrir que un alumno haya elegido asignaturas tan variadas que no necesiten de de prerrequisitos, luego el orden es vacío. En otras ocasiones cada curso necesita un única prerrequisito dando lugar a un orden parcial en forma de árbol. En la figura 3.2 se describen los subproblemas posibles.

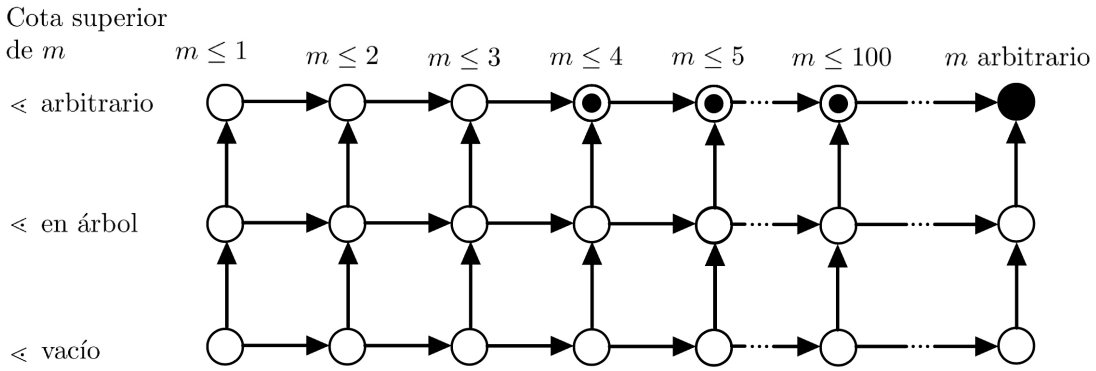


Figura 3.2: Conocimiento actual del problema de PROGRAMACIÓN RESTRINGIDA POR PRECEDENTES.

La figura 3.2 es un ejemplo concreto 3.1, en la que vuelven a aparecer tres tipos de problemas: los $NP-$completos, los P y los de la frontera. Cuando tenemos una red de subproblemas como esta, con frecuencia es posible definir la información de manera más concisa especificando el problema NP-completo "minimal" y el problema P "maximal".

Dada una colección C de subproblemas de un problema NP-completo, se dice que $\Pi \in C$ es un subproblema NP-completo *minimal* si es NP-completo y no se conoce ningún otro subproblema Π' de Π en C. Por otra parte, se dice que un problema $\Pi \in C$ es *maximal* si Π está en P y no existe ningún otro problema de C que tenga como subproblema a Π y

pertenezca a P. Análogamente, se definen los problemas abiertos maximales y minimales.

En la clase de subproblemas de PROGRAMACIÓN RESTRINGIDA POR PRECEDENTES ilustrada en la figura 3.2 , los dos que cumplen "$\prec$ arbitrario, $m \leq 2$" y "$\prec$ en árbol, m arbitrario" son maximales resolubles en tiempo polinomial. El problema general es en sí minimal NP-completo. El problema abierto minimal es el dado por $\prec$ arbitrario, $m \leq 3$" y no existe problema abierto maximal porque $\prec$ arbitrario, $m \leq J$" está abierto para todo lo enteros $J \leq 3$.

Por esto, a la hora de investigar la frontera, puede ser muy útil tratar tanto con los problemas que tienen más pinta de estar en P como con aquellos que pueden ser NP-completos. Tomando como punto de partida los subproblemas que obviamente se pueden resolver en tiempo polinomial y aquellos cuya NP-completitud se siga directamente del problema general, se puede gradualmente aumentar el conjunto de instancias permitidas para los primeros y las restricciones a los segundos. A diferencia de analizar un problema fijo, cuando se pasa de la búsqueda de algoritmo polinomiales a probar NP-completitud, también se cambia de un problema más restringido a uno más relajado.

Como se ha visto en el transcurso del trabajo los problemas de teoría de grafos tienen un gran importancia en el estudio de la NP-completitud. Esta sección concluirá con otro de estos problemas que ejemplificará lo que hemos visto en el párrafo anterior.

3-Colorabilidad

INSTANCIA: un grafo $G = (V, E)$. PREGUNTA:¿Es G 3-coloreable, es decir, existe una función $f : V \to \{1, 2, 3\}$ tal que $f(u) \neq f(v)$ si $\{u, v\} \in E$?

El problema constituye, a su vez, un subproblema de $K-$COLORABILIDAD en la cual la función f toma valores en $\{1, 2, \ldots, K\}$. En la referencia [17] está probada su NP-completitud. A continuación se impondrán restricciones y se estudiaran otros subproblemas.

La primera restricción que se considera es la de acotar el grado de los vértices. La mayoría de los problemas de teoría de grafos pasan a estar en P cuando se restringe a 2 es grado de los vértices. Por ejemplo, bajo esa restricción los problema CICLO HAMILTONIANO, VERTEX COVER y 3$-$COLORABILIDAD pasan trivialmente a poder ser resueltos en tiempo polinomial. Aparece la pregunta: ¿Cuál es la restricción más fuerte que podemos imponer al grado de los vértices para que el subproblema siga siendo NP-completo?

El problema CLIQUE es uno de los ejemplos para los que no existe una cota que preserve la NP-completitud. Sea D la cota para el grado, cualquier clique del grafo no puede contener más de $D + 1$ vértices. Por lo tanto, podemos buscar los cliques examinando los subconjuntos de $D + 1$ o menos elementos, y el número de subconjuntos está polinomialmente acotado pues D es una constante fija.

Para muchos otros problemas de teoría de grafos hay restricciones en el grado para las que los problemas siguen siendo NP-completos. La tabla 3.2 muestra algunos de estos problemas.

	En P para $D \leq$	NP-completo para $D \geq$
VERTEX COVER	2	3
CICLO HAMILTONIANO	2	3
3-COLORABILIDAD	2	4

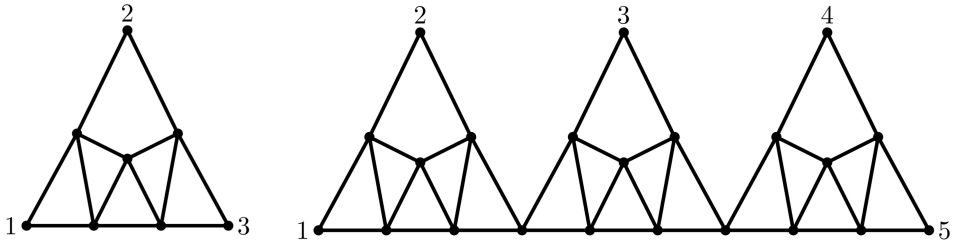
Tabla 3.2: Clasificación de problemas de teoría de grados en función del grado de D de sus vértices.

La NP-completitud de los problemas de la derecha de la tabla puede ser probada por reemplazamiento local de manera similar. Para ejemplificarlo se usará 3−Colorabilidad.

Teorema 3.1. 3−Colorabilidad *con vértices de grado menor o igual que* 4 *es* NP*-completo.*

Demostración. Como el problema general está en NP este también lo estará. Sea $G = (V, E)$ una instancia del problema general. Se construirá otro grafo $G' = (V', E')$ cuyos vértices tengan grado menor o igual que 4 y que sea 3-coloreable si y solo si G es 3-coloreable, obteniendo así la reducción deseada.

La sustitución realizada se basa en un grafo de 8 vértices H_3 que se ilustra en la figura 3.3a, que tiene 3 salidas, etiquetadas por $1, 2$ y 3 en la figura. Para $k \geq 4$, el vértice de k-salida se sustituye por H_k formado al unir H_{k-1} con una copia de H_3 cuya primera salida coincide con la la salida $k-1$ de H_{k-1}. Los vértices de salida de H_k tienen grado dos. Las salidas originales de H_{k-1} mantienen las mismas etiquetas, uniendo un H_3 en el vértice de salida $k-1$ y convirtiendo la tercera salida de H_3 en la salida k. En la figura 3.3b se muestra la construcción de H_5, es decir, el grafo que sustituye a los vértices de grado 5.



(a) Grafo H_3. (b) Grafo que sustituye a un vértice de grado 5.

Figura 3.3

De la construcción se deducen los siguiente resultados:

1. H_k tiene $7(k-2)+1$ vértices, incluyendo los k vértices de salida etiquetados.
2. Ningún vértice de H_k tiene grado mayor que 4.
3. Los vértices de salida de H_k tienen grado 2.
4. H_k admite una 3-coloración pero no una 2-coloración que, además, asigna el mismo color a los vértices de salida.

Sea $v_1, v_2, \ldots, v_r$ los vértices del grafo G de grado mayor que 4. Se construye la siguiente sucesión de grafos:

$$G = G_0, G_1, G_2, \ldots, G_r = G'$$

por la siguiente recurrencia. Sea d el grado de v_i en G_{i-1} y sean $\{u_1, v_i\}, \{u_2, v_i\}, \ldots, \{u_d, v_i\}$ la aristas incidentes en v_i. Para forma G_i, se elimina de G_{i-1} el vértice v_i, se intercambia por una copia de H_d y la arista $\{u_j, v_i\}$ se reemplaza por una arista que une u_j con el vértice de salida j de H_d.

Por construcción y por las deducciones anteriores G_k con $1 \leq k \leq r$ tiene $r-k$ vértices de grado mayor que 4 y G_k es 3-coloreable si y solo si G lo es. Por lo tanto, $G_r = G'$ es el grafo buscado.

□

Aunque, gran variedad de sustitución de vértices pueden ser aplicadas a los distintos subproblemas de teoría de grafos, este ejemplo da una idea de como proceder con los demás.

3.2– 2SAT está en P

Anteriormente, se estudió la *NP*-completitud del problema SAT. Después se vio como con una reducción adecuada 3SAT también es *NP*-completo. Sin embargo, cuando se restringe a dos el tamaño de las cláusulas, pasando así a 2SAT, resulta un problema perteneciente a la clase *P*. 2SAT ofrece un interesante caso de donde se establecen las posibles fronteras entre *P* y *NP*.

El propósito de esta sección es probar que 2SAT está en P. Para ello se probará, en primer lugar, que el problema Camino es tratable y, posteriormente, se da dará una reducción polinomial de Camino a 2SAT que está inspirada en la que aparece en [11].

Camino

Instancia: Sean $G = (V, E)$ un grafo dirigido y s y t dos nodos.

Pregunta: ¿Existe un camino entre s y t?

En un primer intento de la búsqueda de un algoritmo para Camino se podría pensar en recorrer todos los posibles caminos. Si el grafo tiene n vértices por cada vértice se podrían tener n opciones a seguir, lo que da una complejidad de n^n. En la prueba del teorema 3.2 se obtiene un algoritmo más sofisticado que permite probar que Camino está en P.

Teorema 3.2. Camino *está en* P.

Demostración. Sean $G = (V, E)$ un grafo dirigido, y s y t dos nodos, se tiene que decidir si existe o no un camino entre s y t. El algoritmo utilizará las colas, una estructura de datos en la cual se permiten añadir elementos al final de la cola y eliminar elementos del principio. El algoritmo sigue los siguientes pasos:

1. Marca el nodo s y se añade a la cola.
2. Mientras la cola no esté vacía:
   a) Elimina el primer nodo y de la cola.
   b) Para toda arista (y, u) tal que u no está marcado, marca u y añade u al final de la cola.
3. Aceptar si t está marcado.

El algoritmo es correcto pues si t está marcado quiere decir que hay un camino desde s a t. Se estudia ahora la complejidad en tiempo del algoritmo. La cola puede ser usada pues añadir y eliminar elementos de la misma ocupa tiempo constante. Cada nodo se añade como máximo una vez a la cola. Por lo tanto, una cota superior para el tiempo de ejecución es la dada por la suma sobre todos los nodos y del tiempo requerido para comprobar los nodos adyacentes a y. Si el grafo G tiene n vértices y m aristas, y está dado en términos

de su matriz de adyacencia, se tiene una cota de $\mathcal{O}(n^2)$ y si G está dado por su lista de adyacencia, se obtiene una cota de $\mathcal{O}(n+m)$, pues cada arista solo se comprueba una vez. En ambos casos la complejidad es polinomial, como se quería demostrar.

□

Teorema 3.3. *2SAT está en P*

Demostración. Se reducirá 2SAT a CAMINO. Sean $U = \{u_1, u_2, \ldots, u_m\}$ un conjunto finito de variables booleanas y C una colección de cláusulas de dos literales cada una. A partir de esta instancia, se construirá el siguiente grafo dirigido:

- Por cada variable u_j con $1 \leq j \leq m$ se añaden a G dos vértices, uno correspondiente a u_j y, otro, a $\overline{u_j}$.
- Por cada cláusula $\{x_1, x_2\}$, se añaden a G las aristas $(\overline{x_1}, x_2)$ y $(\overline{x_2}, x_1)$.

Estas últimas aristas hacen el papel de implicaciones lógicas. Por ejemplo, considérese la cláusula $\{\overline{u_1}, u_2\}$ y se supone que la asignación de verdad t estable que $t(u_1) = T$. Como es necesario que al menos uno de los dos literales de la cláusula sea verdadero, esto implicaría que $t(u_2) = V$. En la figura 3.4 se muestra un ejemplo.

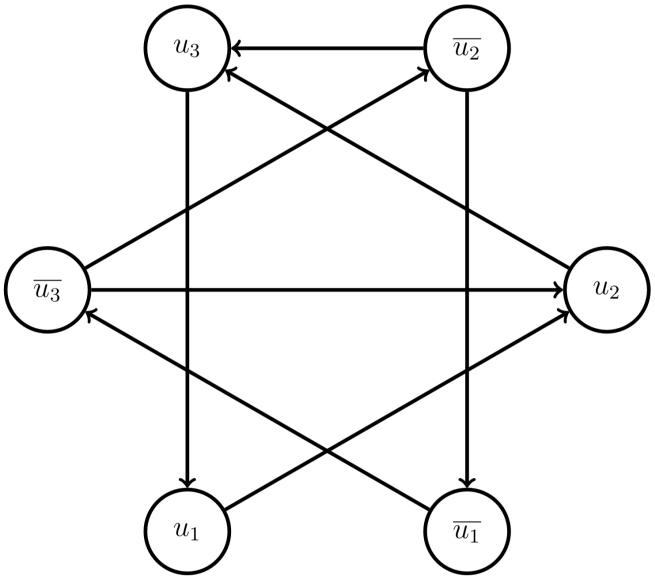


Figura 3.4: Grafo obtenido a partir de $C = \{\{\overline{u_1}, u_2\}, \{\overline{u_2}, u_3\}, \{u_1, \overline{u_3}\}, \{u_2, u_3\}\}$.

Seguidamente, se mostrará que C no es satisfactible si y solo si para algún $1 \leq j \leq m$ existe en G un camino de $\overline{u_j}$ a u_j y de u_j a $\overline{u_j}$

($\Leftarrow$) Por reducción al absurdo, se supone que existe un camino de $\overline{u_j}$ a u_j y de u_j a $\overline{u_j}$. Si $t(u_j) = T$, entonces $t(\overline{u_j}) = 0$, entonces debe existir una arista (x_1, x_2) en el camino desde u_j a $\overline{u_j}$ tal que $t(x_1) = T$ y $t(x_2) = F$. Esto implica que $\{\overline{x_1}, x_2\}$ es una cláusula de C que no es satisfecha por t. Para el caso $t(u_j) = F$, se razona de forma análoga.

(⇒) Se construirá un método para encontrar una asignación de verdad satisfactible para C dando valores de verdad a los vértices de G. Reiteradamente, para cada vértice x que no tiene valor asignado y tal que no existe un camino de x a $\overline{x}$, se le asigna el valor T a x y a los vértices a los que se puede llegar desde x; y se le asigna F a $\overline{x}$ y a la negación de vértices a los que se puede llegar desde x, es decir, a los que se puede llegar desde $\overline{x}$.

En primer lugar, se comprueba que el proceso de construcción está bien definido. Si algún camino desde x a dos vértices y y $\overline{y}$, entonces por simetría de G habría un camino de y y $\overline{y}$ a $\overline{x}$, así que habría un camino de x a $\overline{x}$, contradiciendo la hipótesis. Por lo tanto los valores de verdad se asignan de forma consistente. Además, si hay un camino de x a un vértice y que ha sido asignado con F, por ejemplo, como se puede llegar a y desde x, x debe haber sido asignado con F.

Como se ha asumido que para cada variable u_j, no hay ningún camino de $\overline{u_j}$ a u_j ni de u_j a $\overline{u_j}$, todas las u_j tendrán un valor de verdad asignado. Como la asignación de verdad se produce de tal manera que cada vez que aparezca la arista (x, y), o bien a y se le asigna T o a x se le asigna F, la asignación satisface C.

Una vez hecho esto, se puede finalizar aplicando el algoritmo de CAMINO a cada par u_j y $\overline{u_j}$, lo cual se completa en tiempo polinomial.

□

3.3– Implicaciones de *P≠NP*: Existencia de problemas *NP*-intermedios

Muchos de los problemas de NP que han sido estudiados en las últimas décadas han resultado ser, sorprendentemente, NP-completos. Esté fenómeno sugiere la siguiente conjetura: "Todo problema de NP es NP-completo".

En esta sección, se mostrará que, bajo la asunción de que $P \neq NP$, la conjetura anterior es falsa, es decir existe un lenguaje de $NP - P$ que no es NP-completo. Este resultado es conocido como "Teorema de Ladner". Esta demostración está tomada de [1].

Teorema 3.4. *Si $P \neq NP$, entonces existe un lenguaje $L \in NP - P$ que no es NP-completo.*

Demostración. En esta prueba se usará un esquema de codificación cuyo alfabeto de entrada es $\Sigma = \{0, 1\}$.

Si $P \neq NP$, se conoce al menos un lenguaje que pertenece a $NP - P$; el lenguaje asociado al problema de decisión SAT, L_{SAT}.

Para cada $H : \mathbb{N} \to \mathbb{N}$, se define un lenguage SAT_H que contiene a todos los conjuntos de cláusulas con n cláusulas con $n^{H(n)}$ $1'$s concatenados, esto es:

$$SAT_H = \{\psi 0(1)^{n^{H(n)}} : \psi \in SAT \ y \ n = |\psi|\}$$

Se define una función $H : \mathbb{N} \to \mathbb{N}$ concreta como sigue: $H(n)$ es el menor número $i < \log\log(n)$ tal que para todo $x \in \{0, 1\}^*$ con $|x| \leq \log n$, M_i devuelve $SAT_H(x)$ en $i|x|^i$ pasos.(M_i es la máquina que corresponde a la expansión binaria de i, y $\mathrm{SAT}_H(x) = 1$ si y solo si $x \in \mathrm{SAT}_H$). Si no existe tal i entonces $H(n) = \log\log n$.

H está bien definida pues $H(n)$ determina la pertenencia a SAT_H de cadenas cuya longitud es mayor que n, y la definición de $H(n)$ depende únicamente de comprobar el estado de las cadenas de longitud a los más $\log(n)$.

Se afirma que $SAT_H \in P$ si y solo si $H(n) = \mathcal{O}(1)$(es decir, existe $C > 0$ tal que $H(n) \leq C$ para todo n). Además, si $SAT_H \notin P$ entonces $H(n)$ tiende a infinito cuando n tiende a infinito

Demostración de la afirmación:

($\Rightarrow$) Sea M una máquina que resuelve SAT_H en cn^c pasos como máximo. Como M está representada por infinitas cadenas, debe existir $i > c$ tal que $M = M_i$[1]. La definición de $H(n)$ implica que para $n > 2^{2^i}$ debe ocurrir que $H(n) \leq i$. Por lo tanto, $H(n) = \mathcal{O}(1)$.

($\Leftarrow$) Si $H(n) = \mathcal{O}(1)$ entonces H solo puede tomar una cantidad finita de valores, y por lo tanto, existe un i tal que $H(n) = i$ para infinitos n. Pero esto implica que la máquina de Turing M_i resuelve SAT_H en tiempo in^i: de lo contrario, si habría una entrada x en la que M_i no diera la respuesta correcta en bajo esa cota de tiempo, para todo $n > 2^{|x|}$ se tendría que $H(n) \neq i$. Esto se mantiene si $H(n) \leq C$ para infinitos n, probando así el además de la afirmación.

Usando la afirmación se probará que si $P \neq NP$ entonces SAT_H no está ni en P ni es NP completo:

- Si $\text{SAT}_H \in P$, entonces la afirmación nos dice que $H(n) \leq C$ para alguna constante $C > 0$, esto implica que SAT_H es simplemente SAT con una cantidad polinomial de $1's$ concatenados. Pero entonces un algoritmo polinomial para SAT_H podría usarse para resolver SAT en tiempo polinomial, implicando $P = NP$, lo que contradice la hipótesis.

- Si SAT_H fuera NP-completo existiría una reducción de SAT_H a SAT en tiempo polinomial $\mathcal{O}(n^i)$ para cierto i. Como $\text{SAT}_H \notin P$, por la afirmación anterior se tiene que $H(n)$ tiende a infinito cuando n tiende a infinito. Como la reducción se completa en tiempo $\mathcal{O}(n^i)$, para n suficientemente grande debe hacer corresponder instancias de tamaño n con instancias de SAT_H de tamaño menor que $n^{H(n)}$. Por lo tanto, para fórmulas suficientemente grandes φ, la reducción f las envía en cadenas del tipo $\psi 0(1)^{n^{H(|\psi|)}}$ donde ψ es menor que un factor polinomial adecuado, por ejemplo, $\sqrt[3]{n}$. Pero la existencia de dicha reducción lleva a un algoritmo polinomial para SAT, contradiciendo que $P \neq NP$.

□

3.4– Implicaciones de P=NP

Las anteriores secciones de este capítulo ponen su esfuerzo en probar que $P \neq NP$. Pero, ¿que ocurriría si $P = NP$? En esta sección se desarrollaran algunas de las implicaciones teóricas y prácticas de $P = NP$, para así poder mostrar la importancia del problema.

El hecho de que $P = NP$ es una posibilidad, pero en la práctica no se cuenta con algoritmos polinomiales para problemas NP-completos. Se va suponer que estos algoritmos si existen y se conocen, y se verá que ocurre al hacer uso de ellos.

[1] M_i es la $MTND$ asociada el leguage codificado por i.

Uno de los avances más notables se produciría en el campo de la optimización pues muchos de sus problemas son NP-completos. Esto tendría un impacto significativo en la eficiencia de la sociedad, desde el transporte hasta la producción de bienes. El transporte de personas y mercancías se optimizaría para ser más rápido y económico, y los fabricantes podrían mejorar su producción para aumentar la velocidad y reducir el desperdicio. La inteligencia artificial avanzaría a paso agigantados gracias a la cantidad de problemas de la teoría de grafos que podrían ser resueltos en tiempo polinomial.

Por contra, un problema surgiría en la ciberseguridad actual pues cualquier esquema de encriptación tendría un algoritmo de decodificación trivial, lo que significaría la falta de privacidad en el ámbito digital. No existirían sistemas de dinero digital, SSL, RSA o PGP.

Si se verifica la igualdad, se podrían encontrar demostraciones lógicas cortas para los teoremas matemáticos: se podría encontrar la teoría más simple que explique un conjunto de datos experimentales o información disponible. Esto se basaría en el principio de la Navaja de Occam, que postula que la explicación más simple es probablemente la correcta. Stephen Cook afirmó que si se encontrara un algoritmo eficiente para resolver problemas NP-completos las matemáticas cambiarían drásticamente pues permitiría a los ordenadores dar pruebas a teoremas de longitud razonables, pues las pruebas se pueden comprobar en tiempo polinomial, generalmente.

En resumen, el mundo actual cambiaría drásticamente y habría que comenzar a pensar en nuevas técnicas que permitan tanto tomar ventaja de este problema: encontrar soluciones eficientes para antiguos problemas NP-completos y ponerlas en práctica; como defendernos de los posibles perjuicios: buscar nuevos sistemas de ciberseguridad.

3.5– Otros enfoques para demostrar $P \neq NP$

Como bien se ha mencionado a lo largo del capítulo, la comunidad científica se inclina por pensar que $P \neq NP$, no solo por la cantidad de problemas de los que no se puede hallar un algoritmo eficiente, sino por la Utopía teórica que plantearía. Para concluir el capítulo, se mostrarán diferentes maneras de las que se ha tratado probar $P \neq NP$ y han fallado en el intento. En [5] se pueden encontrar más detalles sobre este tema.

Diagonalización

Este método responde a la pregunta de si puede encontrar un lenguaje L concreto de forma que todo algoritmo polinomial fallara al computar L en alguna entrada. Este planteamiento se conoce como diagonalización y tiene sus inicios en el siglo XIX. La diagonalización es la técnica que en 1874 Cantor usó para demostrar que nos número reales no son numerables. La diagonalización permitió a Cantor[4], dada un conjunto de numero reales numerable, construir otro que no estuviera en la lista.

Turing, en su famoso artículo sobre computación [18], usa una estrategia parecida para demostrar que el problema de la Parada no es computable. En los sesenta los científicos de teoría de la complejidad utilizaron la diagonalización para probar que si se contaba más tiempo y más memoria se podían resolver más problemas. ¿Por qué no usar la diagonalización pa separar NP de P?

La diagonalización requiere de simulación y en la actualidad no se comprende como una máquina no determinista fija puede simular una máquina determinista arbitraria. Además una prueba por diagonalización tiende a relativizarse, es decir, que funcionaría incluso si todas las máquinas involucradas tienen acceso a la misma información adicional

o recursos. Baker, Gill y Solovay, en su trabajo citado [2], demostraron que ninguna prueba relativizable puede resolver el problema P vs NP en ninguna dirección.

Aun así, la diagonalización se ha usado para probar que no pueden existir algoritmos por debajo de cierta complejidad en tiempo y en espacio para SAT, aunque esto dista mucho de probar que $P \neq NP$.

Complejidad de fórmulas Booleanas

Para probar que $P \neq NP$ es suficiente con hallar un problema NP-completo que no pueda ser resuelto con fórmulas Booleanas con AND, OR o NOT relativamente limitas (el número de operadores lógicos está acotado por una cierta constante fija).

En 1984, Furst, Saxe y Sipser [6] demostraron que las fórmulas limitadas no pueden resolver la función de paridad si las fórmulas tienen un número fijo de capas de operadores. En 1985, Razborov [13] mostró que el problema NP-completo CLIQUE no tiene este tipo de fórmulas si solo se permiten operadores AND y OR (sin el operador NOT). Si se extiende el resultado de Razborov a circuitos generales, se habrá demostrado que P es distinto de NP.

Razborov probó más tarde que sus técnicas fallarían si se permitía operadores NOT [14]. Razborov y Rudich [15] desarrollaron una noción de"pruebas naturales" y presentaron evidencia de que estas técnicas no podrían avanzar mucho más.

Complejidad de demostraciones

Una tautología no es mas que una fórmula booleana que se satisface para cualquier asignación de verdad de sus variables.

Existe un método llamado *resolución* que constituye un enfoque estándar para probar tautologías , mediante la búsqueda de cláusulas de cierto formato y agregando nuevas cláusulas. Si se pudiera demostrar que las tautologías no tienen pruebas "cortas", eso implicaría que P es distinto de NP.

En 1985, Haken [8] demostró que las tautologías que codifican el principio del palomar (si hay más palomas que agujeros, entonces algún agujero tiene más de una paloma) no tienen pruebas de resolución cortas.

Desde entonces, los teóricos han demostrado debilidades similares en otros sistemas de demostraciones, como el de planos de corte, sistemas de prueba algebraicos basados en polinomios y versiones restringidas de pruebas utilizando los axiomas de Frege.

Para probar que P es distinto de NP, se necesitaría demostrar que las tautologías no tienen pruebas cortas en un sistema de pruebas arbitrario. Incluso un resultado importante que muestre que las tautologías no tienen pruebas generales de Frege cortas no sería suficiente para separar NP de P.

CAPÍTULO 4

Conclusiones y ampliaciones futuras

A lo largo de este proyecto se ha dado una primera introducción al mundo de la NP-completitud así como al problema del milenio P vs NP, además de haber visto su importancia en las matemáticas y las ciencias de la computación habiendo conseguido ser uno de los siete Problemas del Milenio que se remuneran con 1000000$. A continuación, se aportarán conclusiones y ampliaciones futuras.

Primeramente, se han expuestos resultados de gran relevancia como el Teorema de Cook que consigue reducir toda máquina de Turing no determinista a un problema de lógica Booleana, SAT. También se han explicado, algunas reducciones entre los problemas NP-completos clásicos de manera visual para facilitar su entendimiento. Esto ha confirmado la dificultad del resolver problemas NP-completos y la aparente inexistencia de un algoritmo eficiente para ello.

La Teoría de la Complejidad Computacional es un campo de estudio relativamente moderno que tuvo su inicio a mediados del siglo pasado con el desarrollo de las computadoras. Esto quiere decir que sus conceptos y su metodología están en pleno desarrollo. De hecho, el gran Richard M. Karp[21], científico en esta rama que ha proporcionado numerosas aportaciones a la misma, afirma que si algún día se resuelve este problema será con conceptos que todavía no han sido planteados en la actualidad.

4.1– Ampliaciones futuras

Debido a que la resolución del problema P vs NP tendría implicaciones significativas en áreas tan diversas como la criptografía, la inteligencia artificial, la optimización y la verificación de software. Es fundamental continuar investigando y abordando el problema P vs NP en busca de nuevas soluciones y avances en la teoría de la computación.

Sin embargo, el campo de la teoría de la complejidad es mucho más amplio que únicamente el problema P vs NP. Por ejemplo, las computadoras constan con una memoria limitada, luego es importante que los algoritmos ocupen el menor espacio posible. De esta y otra inquietudes surge el concepto de complejidad en espacio. Por otra parte, las clases de complejidad no tienen por qué ser solo polinomiales sino que también existen algoritmos de otras complejidades como exponenciales o logarítmicas, una imagen de esto puede verse en [9] donde aparecen todas las clases de complejidad y relaciones entre ellas, algunas de ellas probadas y otras conjeturadas.

Por otra parte, el modelo de computación usado en el trabajo es la máquina de Turing. ¿Qué ocurriría si se considerara otro modelo? ¿Ayudaría a proporcionar esas nuevas herramientas de las que se hablaban antes? De momento, esta preguntas siguen abiertas.

No obstante, sí se está trabajando con nuevos modelos. Por ejemplo, se han aprovechado los conceptos físicos tales como la superposición cuántica, la cual acerca a un paralelismo masivo, en la década de los 90, Peter Shor [16] presentó algoritmos cuánticos para factorizar números y resolver el problema del logaritmo discreto, problemas que se usan como base para muchos protocolos criptográficos en las computadoras clásicas. Sin embargo, no se espera que la factorización o el logaritmo discreto sean problemas NP-completos, ni que la computación cuántica sea capaz de resolver problemas como Clique o Vertex Cover en tiempo polinomial.

Otro de los modelos que intenta huir de la computación convencional es la computación natural, basada en modelos bioinspirados, entre estos destaca la computación de membrana[22]. Esta se basa en el funcionamiento de los organismos vivos replicando procesos como la producción de energía, la síntesis de proteínas o los procesos metabólicos para general todo tipo de modelos. La ventaja de este tipo de computación es que las tareas se llevan a cabo paralelamente entre todos los miembros otorgando así una mayor potencia computacional al modelo. Este tipo de computación está comenzando a hacerse camino entre disciplinas como la biología, las ciencias de la computación y la lingüística.

En última instancia, el problema P vs NP sigue siendo uno de los desafíos más apasionantes y enigmáticos en la teoría de la computación. Con un enfoque constante y colaborativo, se podrá avanzar hacia una comprensión más profunda de esta cuestión fundamental y desbloquear nuevas fronteras en las ciencias de la computación.

Bibliografía

[1] S. Arora, B. Barak, *Computational Complexity: A Modern Approach,* Cambridge University Press, 2016.

[2] T. Baker, J. Gill, and R. Solovay. *Relativizations of the P = NP question.* SIAM Journal on Computing, 431–442, 1975.

[3] N. L. Biggs., *Matemática discreta(1a ed., 1a reimp.)*, Vicens-Vives, 1998.

[4] G. Cantor. *Ueber eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen*, Crelle's Journal, 258–262, 1874.

[5] L. Fortnow, *The status of the P versus NP problem*, Illinois Institute of Technology, 2009.

[6] M. Furst, J. Saxe, and M. Sipser. *Parity, circuits and the polynomial-time hierarchy.* Mathematical Systems Theory, 13–27, 1984.

[7] M.R. Garey, D.S. Jonhson, *Compputers and Intractability: A guide to the Theroy of NP-Completeness,* W. H. Freeman, 2003.

[8] A. Haken. *The intractability of resolution.* Theoretical Computer Science, 297–305, 1985.

[9] G. Kuperberg, *Complexity Zoology: Active Inclusion Diagram*, 2006.

[10] F.F. Lara, *Tema 9: Complejidad computacional*, Universidad de Sevilla, 2022.

[11] A. Montanaro, *Computational Complexity Lecture notes*, 2012.

[12] C. H. Papadimitriou, *Computational Complexity*, Pearson, 1994.

[13] A. Razborov. *Lower bounds on the monotone complexity of some boolean functions.* Soviet Mathematics–Doklady, 485–493, 1985.

[14] A. Razborov. *On the method of approximations.* In Proceedings of the 21st ACM Symposium on the Theory of Computing, 167–176. ACM, New York, 1989.

[15] A. Razborov and S. Rudich. *Natural proofs.* Journal of Computer and System Sciences, 24–35, 1997.

[16] P. Shor. *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer.* SIAM Journal on Computing, 1484–1509, 1997.

[17] L. J. Stockmeyer, *Planar 3-colorability in NP-complete*, 1973.

[18] A. Turing. *On computable numbers, with an application to the Etscheidungs problem.* Proceedings of the London Mathematical Society, 230–265, 1936.

[19] J. D. Ullman, *NP-complete scheduling problems*, Journal of Computer and System Sciences, 384-393, 1975.

[20] Wikipedia contributors, *Turing machine*, en Wikipedia, The Free Encyclopedia, 2023.

[21] Wikipedia contributors, *Richard M. Karp*, en Wikipedia, The Free Encyclopedia, 2023.

[22] G. Zhang, M.J Pérez-Jiménez, A. Riscos-Núñez, S. Verlan, S. Konur, T. Hinze, M. Gheonghe, *Mebrane Computing Models: Implementations*, Springer, 2021.