INSTITUTO TECNOLOGICO AUTONOMO DE MEXICO

PROBLEMAS NP-COMPLETOS.

T E S I S

QUE PARA OBTENER EL TITULO DE

LICENCIADA EN MATEMATICAS APLICADAS

P R E S E N T A

MARIA CRAVIOTO LAGOS

MEXICO, D.F. 2008

Problemas $\mathcal{NP}$ – *completos*

María Cravioto Lagos

A mis papás y hermanos por su apoyo y paciencia. A toda mi familia por formar una parte importante de mi vida. A todos mis amigos, dentro y fuera del ITAM, por su valiosa amistad y por su ánimo.

Al Dr. Ramón Espinosa, por dirigir este trabajo, por su ayuda y su apoyo. Al Mtro. Javier Alfaro, a la Dra. Carmen López y al Dr. Edgar Possani por sus correcciones, observaciones y comentarios. A todos los profesores que me dieron clases por su conocimiento.

A todos aquellos con los que compartí esta etapa de mi vida.

Prefacio

La teoría de complejidad computacional surge como una forma de clasificar los problemas de acuerdo a su dificultad. Específicamente, los problemas se dividen en clases de acuerdo al tiempo en que pueden resolverse. La clase de problemas $\mathcal{NP}-completos$ es una clase importante, ya que son problemas que a la fecha no pueden resolverse en tiempo polinomial pero pueden verificarse en tiempo polinomial. Además, estos problemas tienen una propiedad importante, si alguno de ellos pudiera resolverse de manera eficiente, entonces todos los problemas dentro de la clase $\mathcal{NP}$, incluidos los $\mathcal{NP}-completos$, podrían resolverse de manera eficiente. En este trabajo buscamos mostrar parte de la teoría de complejidad computacional, así como explicar cuáles son los problemas $\mathcal{NP}-completos$ y su importancia en esta teoría.

Muchos de los conceptos que se manejan pueden encontrarse enunciados de formas similares en diferentes libros y artículos. En este trabajo se utilizaron varias fuentes para algunas definiciones y demostraciones. A continuación se presenta una pequeña descripción de cada capítulo, indicando la bibliografía general en cada caso.

En el primer capítulo se presentan primero las formas en las que se ha intentado mecanizar el proceso de resolver y clasificar diferentes problemas. Vemos también como este proceso lleva a la clasificación de los problemas en diferentes clases de acuerdo a su posible resolución. Esta parte del capítulo fue tomada de los libros [6] y [5]. Después se definen algunos conceptos que se utilizarán a lo largo de este trabajo. Dichos conceptos se obtuvieron, en su mayoría, de los libros [12] y [4]. Finalmente, se presenta una sección acerca de las funciones recursivas parciales, ya que éstas muestran cuáles son las funciones que podemos resolver con una computadora y están relacionadas con la tesis de Church-Turing que se presenta en el capítulo 2. Esta parte del capítulo se obtuvo de [1] y un poco de [13].

El segundo capítulo está dedicado a las máquinas de Turing. Primero se trata la tesis de Church-Turing, que nos permite equiparar a dichas máquinas con los algoritmos. Esta parte del capítulo fue obtenida de los libros [1] y [5], además de los artículos [17] y [18]. Después se entra de lleno a lo que son las máquinas de Turing, dando una explicación de qué son y mostrando ejemplos. La siguiente sección muestra cómo se puede codificar una máquina de Turing, así como sus entradas. Ambas secciones se obtuvieron en su mayoría de [5], aunque algunas formulaciones vienen de [4]. Finalmente, se consideran algunas máquinas de Turing especiales. En particular se considera la máquina de Turing no determinista, ya que ésta se utilizará para definir los problemas $\mathcal{NP}$. Esta parte del capítulo se obtuvo de los libros [9], [10] y [1], y una pequeña parte de [5].

En el tercer capítulo se habla de los diferentes tipos de problemas que hay. Primero se explica cuáles son los problemas indecidibles e intratables. Esta parte fue obtenida de [5] y [6]. A continuación se describen las clases $\mathcal{P}$, $\mathcal{NP}$ y $\mathcal{NP}-completo$. La definición de las diferentes clases se obtuvo, en general, de [4], [12] y [3]. Por último, se explican brevemente las clases $Co\mathcal{NP}$ y $\mathcal{NP}-difíciles$, las cuales se obtuvieron de [13] y [9] respectivamente.

El cuarto capítulo presenta algunos problemas $\mathcal{NP}$ – *completos* importantes junto con sus demostraciones. En cada prueba se indica de dónde se obtuvo la demostración.

El quinto capítulo habla un poco más acerca de los problemas $\mathcal{NP}$ – *completos*. Primero se muestan técnicas en las que se puede demostrar que un problema es de este tipo. Esta parte fue tomada de [6]. Después se muestran algunos otros problemas $\mathcal{NP}-completos$. Estos problemas se obtuvieron de [13], [12], [14] y [6]. Más adelante se presentan formas de trabajar con los problemas $\mathcal{NP}$ – *completos*. Éstas se obtuvieron de [6] y [14]. Por último, se habla de la conjetura $\mathcal{P}$ vs. $\mathcal{NP}$ y su relación con los problemas $\mathcal{NP}$ – *completos*. Esta sección fue obtenida de [16], [3], [4].

Ambos apéndices explican diferentes conceptos que se utilizan en algunos de los problemas $\mathcal{NP}-completos$. El primer apéndice define los diferentes conceptos de teoría de gráficas que se utilizan. El material para dicho apéndice fue obtenido de los libros [8] y [2]. Por otro lado, el apéndice de expresiones booleanas fue obtenido de los libros [4], [9], [15], [11]. Los conceptos dados en este apéndice se utilizan sobre todo para los problemas SAT y 3-SAT.

Índice general

Capítulo 1

Introducción

1.1. Contexto

A grandes rasgos, un problema se puede plantear como una pregunta que queremos resolver. La resolución de problemas es algo que generalmente se percibe como un procedimiento mecánico. Es por esto que desde el siglo XVII se han desarrollado máquinas que ayudan a resolver los problemas de una manera más rápida y precisa. En particular. Gottfried Leibniz (1646-1716) diseñó una máquina que calculaba sumas. restas. multiplicaciones y divisiones. Con esta máquina pretendía que los investigadores en los diferentes campos dedicaran su tiempo a procesos intelectuales y creativos, dejando los cálculos a las máquinas.

Esta idea se llevó inclusive más lejos. En 1928, David Hilbert (1862-1943) plantea el *Entscheidungsproblem*, que literalmente quiere decir problema de decisión. Este problema plantea la búsqueda de un método que pueda comprobar la validez expresiones lógicas (o fórmulas matemáticas) en un número finito de pasos. La solución de este problema sería un procedimiento o algoritmo por medio del cual podríamos conocer la validez de todas las cuestiones matemáticas, aunque no obtendríamos una demostración para las mismas. Básicamente. una solución para el *Entscheidungsproblem* implicaría que todo el razonamiento humano es simplemente un proceso mecánico, capaz de realizarse con una máquina. Sin embargo. en 1936 Alonzo Church (1903-1995) y Alan Turing (1912-1954) demuestran, de manera independiente, que el *Entscheidungsproblem* no tiene solución. En el caso de Alan Turing, el tra-

bajo que realizó para demostrarlo lo llevó al desarrollo de las máquinas de Turing.

Aun cuando quedó demostrado que no todos los problemas se pueden resolver por medio de algoritmos, se siguen buscando algoritmos para los problemas que sí podemos resolver. Al enfrentarnos con un problema, siempre vamos a buscar la mejor forma de resolverlo. Esto es, buscamos la manera más práctica. o bien, aquella que cubra mejor nuestras necesidades. Muchas veces buscamos algoritmos que resuelvan el problema en un tiempo relativamente corto. Sin embargo, encontrar dichos algoritmos puede ser bastante complicado, por lo que debemos conocer el problema para atacarlo de la mejor manera posible.

En 1964 Cobham y en 1965 Edmonds introducen la noción de algoritmos polinomiales, los cuales quedan identificados por Edmonds como algoritmos "buenos". Estos son los algoritmos que consideramos rápidos. Apoyado en estos conceptos, las bases para la teoría de los problemas $\mathcal{NP}$ – *completos* fueron dadas por Stephen Cook en 1971.

Hay diferentes maneras en las cuales podemos clasificar un problema. A muy grandes rasgos, si un problema se puede resolver de manera rápida se encuentra en la clase $\mathcal{P}$. Si un problema se puede corroborar o verificar de manera rápida se encuentra en la clase $\mathcal{NP}$. Podemos ver que $\mathcal{P}$ es subconjunto de $\mathcal{NP}$, y lo demostraremos más adelante. Los problemas $\mathcal{NP}$ – *completos* son los problemas más difíciles dentro de $\mathcal{NP}$. Estos problemas se caracterizan por el hecho de que todos los problemas en $\mathcal{NP}$ pueden transformarse de manera rápida a cualquier problema $\mathcal{NP}$ – *completo*.

A la fecha, no se conocen algoritmos polinomiales que resuelvan los problemas $\mathcal{NP}$ – *completos*. Sin embargo, tampoco se ha demostrado que dichos algoritmos no existan. Por estas razones, si sabemos que un problema es $\mathcal{NP}$ – *completo*, podemos buscar otras maneras de trabajar con él. Por ejemplo, podemos dejar de buscar un algoritmo óptimo y nos concentraremos en alguno que funcione bien en la mayoría de los casos. También podemos buscar aproximaciones a la solución. En otras palabras, buscaremos algoritmos que nos den soluciones buenas. aunque no necesariamente óptimas.

Con la clase de los problemas $\mathcal{NP}$ – *completos* podremos atacar el problema $\mathcal{P}$ vs $\mathcal{NP}$ directamente, que consiste en demostrar la diferencia o la equivalencia de las clases $\mathcal{P}$ y $\mathcal{NP}$. Aunque en general se cree que estas clases son diferentes. no se ha podido demostrar todavía que esto sea cierto.

Dicho problema es considerado por el Clay Mathematics Institute como uno de los siete problemas del milenio, y hay un premio de un millón de dolares para la persona que pueda resolverlo.

1.2. Conceptos básicos

Alfabetos, palabras y lenguajes

Un **alfabeto** es un conjunto finito con al menos dos elementos. Algunos autores, como Korte y Vygen en [12], consideran que el espacio en blanco, denotado $\sqcup$, no debe de encontrarse dentro del alfabeto, mientras que otros, como Cook, et al en [4] lo creen conveniente, aunque no necesario. Denotaremos Σ como un alfabeto que no incluye al espacio en blanco, mientras que $\Sigma_\sqcup$ denotará al mismo alfabeto, pero con el espacio en blanco, i.e. $\Sigma_\sqcup = \Sigma \cup \{\sqcup\}$.

A cualquier secuencia finita y ordenada construida a partir de los elementos del alfabeto Σ se le llama **palabra**. La **longitud** o **tamaño** de una palabra w, denotada $long(w)$, es una función que da el número de elementos de Σ que la conforman. tomando en cuenta repeticiones. Por ejemplo, si $\Sigma = \{0, 1\}$, y $w = 01011101$ es una palabra construida a partir de Σ, entonces $long(w) = 8$. Podemos tener una palabra cuya longitud sea 0. Esta palabra, conocida como la **palabra vacía**, es una palabra sin "letras" o símbolos del alfabeto.

Denotaremos Σ^n al conjunto de todas las palabras construidas a partir de Σ que tienen longitud n. De esta forma, denotaremos Σ^* como el conjunto de todas las palabras que pueden construirse a partir de Σ, por lo que $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$. Dentro de Σ^* también se encuentra la palabra vacía. Un **lenguaje** construido a partir de Σ es un subconjunto de Σ^*

Problemas

Un **problema** se puede ver como una pregunta general, la cual está definida por medio de parámetros libres. Esta pregunta puede estar formulada de dos maneras, una en la que se pide que se busque y encuentre un resultado como respuesta, y otra en la que se conteste únicamente con una afirmación o una negación. Muchas veces, los problemas que buscan resultados pueden

ser construidos o interpretados a partir del segundo tipo de problemas. Para construirlos, se toman diferentes posibles resultados, o bien se encuentra un resultado parcial. Después se pregunta si estos resultados cumplen con las condiciones del problema. De esta forma, cada vez que se toma una de estas posibilidades, estamos resolviendo el segundo tipo de problemas.

Los problemas, en general, buscan todas las soluciones para cualquier grupo de parámetros. Si especificamos todos los parámetros de un problema, entonces estamos considerando una **instancia** del mismo. Así. podemos ver a las diferentes instancias como preguntas específicas o casos particulares del problema.

Para explicar esto mejor, se utiliza un ejemplo obtenido de [19]. Un problema sería. dado un número, encontrar todos los factores primos del mismo. Una instancia del mismo problema sería encontrar todos los factores primos del número 15. Este problema entra dentro de la primera categoría de problemas. Si queremos verlo como el segundo tipo de problemas, tendríamos que preguntar si cada número primo es factor del número dado. De esta manera podemos resolver el primer tipo de problemas a partir del segundo.

Los **problemas de decisión** son aquellos problemas que sólo admiten dos respuestas posibles: o lo que dice el problema es cierto o es falso. Para resolverlos, buscamos todas aquellas instancias para las cuales lo que dice el problema es cierto. Si una instancia responde el problema de manera afirmativa, diremos que dicha instancia es una solución del problema, o bien una **si-instancia**. En cambio, si lo responde de manera negativa, diremos que es una **no-instancia**. El hecho de que los problemas de decisión admitan únicamente dos respuestas nos permite dividir a las instancias en dos conjuntos ajenos.

Podemos codificar las instancias de un problema como palabras de un alfabeto dado. La forma en la que se hace una codificación se explica en el segundo capítulo de este trabajo. De esta forma, un problema puede ser visto como un conjunto de palabras que representan instancias del mismo. Si tenemos un lenguaje L que representa a las instancias de un problema, cada palabra del lenguaje va a encontrarse en uno de dos subconjuntos ajenos de L, uno representando a las si-instancias y otro a las no-instancias.

De esta forma, obtenemos la siguientes definiciones. basadas en las definiciones dadas por Korte y Vygen en [12]:

Definición 1.1. *Sean $L \subseteq \Sigma^*$ un lenguaje, y P tal que $P \subseteq L$. Un problema*

de decisión Π *es una pareja* $\Pi = (L, P)$ *donde:*
Los elementos de L *son instancias de* Π.
Los elementos de P *son si-instancias.*
Los elementos de $L \backslash P$ *son no-instancias.*

Tenemos entonces que si $x \in P$ entonces x es una solución del problema, por lo que P es el conjunto de palabras en L para las cuales se afirma el problema Π.

Si tenemos un problema de decisión $\Pi = (L, P)$, su **complemento**, denotado $\overline{\Pi}$, es un problema de decisión tal que $\overline{\Pi} = (L, L \backslash P)$. De esta forma, las instancias de ambos problemas son palabras del lenguaje L; las no-instancias de Π son las si-instancias de $\overline{\Pi}$ y viceversa. Si un problema busca las instancias que cumplan con ciertas características, su complemento pregunta por aquellas instancias que no las cumplen, o bien, que cumplen con las características opuestas. Por ejemplo, si Π pregunta si un número es primo, entonces $\overline{\Pi}$ pregunta si un número es compuesto.

Algoritmos y funciones calculables

Intuitivamente, un **algoritmo** es una lista de instrucciones que debe ser seguida paso a paso para resolver un problema. También se puede ver como un procedimiento mecánico finito que se utiliza para resolver problemas. Se dice que un algoritmo resuelve un problema si encuentra una solución para cualquier instancia del mismo.

El algoritmo recibe una entrada en la cual se especifican los parámetros libres del problema. definiendo así la instancia a resolver. Una vez dada la entrada, el algoritmo lleva a cabo determinadas instrucciones en un cierto orden hasta llegar al resultado. Para cada entrada, el proceso que desarrolla el algoritmo queda completamente determinado, y el resultado será siempre el mismo. Al resultado que regresa el algoritmo también se le conoce como salida.

Muchos de los problemas que buscamos resolver deben ser representados por medio de funciones para que los diferentes algoritmos puedan trabajar con ellos. En otras palabras, si $f(x)$ es la función que representa al problema, un algoritmo que lo resuelva recibe como entrada x y regresa como resultado $f(x)$. Si dicho algoritmo existe, entonces la función correspondiente va a

ser una función calculable. De esta manera. las **funciones calculables** son aquellas que pueden ser calculadas por un algoritmo.

Por ejemplo, en [12], Korte y Vygen nos muestran una función calculable para un problema de decisión $\Pi = (L, P)$:

$$f : X \longrightarrow \{0, 1\}$$

$$f(x) = \begin{cases} 1 & \text{si } x \in P \\ 0 & \text{si } x \in L \backslash P \end{cases}$$

El dominio de esta función es el conjunto de entradas que representan a las diferentes instancias del problema, es decir, las entradas que puede recibir un algoritmo para dicho problema. Así, dicho algoritmo recibe como entrada x y hace las operaciones necesarias para calcular el valor de $f(x)$, indicando de esta manera si x se encuentra en P o en $L \backslash P$.

Eficiencia, tiempo de ejecución y complejidad

Para un problema dado pueden haber varios algoritmos distintos que lo resuelvan. Cuando trabajamos con problemas o instancias sencillas, no hay mucha diferencia entre usar un algoritmo u otro, ya que el tiempo en el que se resolverá será muy corto. Sin embargo, conforme los problemas se van complicando, el algoritmo utilizado puede afectar de manera importante el tiempo en el que vamos a encontrar la solución. Es por esto que para cada problema queremos encontrar el algoritmo más eficiente posible.

Existen diferentes formas de medir la eficiencia de un algoritmo. Las medidas que generalmente se toman en cuenta son el tiempo y el espacio que el algoritmo necesita para resolver el problema. A grandes rasgos, el espacio que requiere se refiere a la cantidad de memoria que ocupa la máquina para resolver el problema, mientras que el tiempo se refiere al número de pasos que debe llevar a cabo. En este trabajo únicamente consideraremos el tiempo como medida de eficiencia.

La **eficiencia** de un algoritmo se mide con respecto al tiempo que tarda en resolver el problema, conocido como el **tiempo de ejecución**. Para calcular este tiempo no se toma en cuenta la velocidad de la máquina con la que se trabaja, ya que ésta varía dependiendo de la máquina. Sólo se toman en cuenta para el cálculo el número de operaciones elementales que realiza el

algoritmo. Algunas operaciones elementales, mencionadas por Korte y Vygen en [12], son: asignación de valores a una variable, referencia a una variable cuyo índice es otra variable, condicionales, suma, resta, multiplicación y comparación entre variables.

El número de operaciones elementales que realiza un algoritmo está relacionado con el tamaño de las entradas con las que se está trabajando. Este tamaño a su vez está relacionado con la forma en la que se introducen las entradas al algoritmo. Para introducirlas, se escoge un alfabeto, usualmente $\Sigma = \{0, 1\}$, y se codifican las entradas como palabras de Σ^*. La razón por la cual se escoge este alfabeto en particular es porque es el alfabeto con el cual trabajan las computadoras. El tamaño de la entrada va a ser igual a la longitud de la palabra que la codifica. En el capítulo 2 se explicará como se lleva a cabo esta codificación.

Como las entradas definen las instancias, el número de operaciones que realizará el algoritmo depende de la instancia que se busca resolver, por lo que éste varía entre las diferentes instancias de un mismo problema. Es por esto que el tiempo de ejecución solo toma en cuenta el peor caso posible. En otras palabras, el tiempo de ejecución del algoritmo nos indica cual es el mayor número de operaciones que pudiera llegar a necesitar para resolver alguna de las instancias del problema. También existen enfoques que consideran el tiempo promedio en vez del peor caso posible como medida de eficiencia.

Tenemos entonces que el tiempo de ejecución de un algoritmo es una función $f(n)$, en donde n es el tamaño de la entrada. Muchas veces es difícil calcular explícitamente esta función, por lo que es mejor buscar una cota superior para ella. El conjunto $O(g(x))$ es el conjunto de todas las funciones que quedan acotadas superiormente por la función $g(x)$. Esto quiere decir que si $f(x)$ está dentro de $O(g(x))$ entonces, a partir de cierto punto x_0, el valor de $g(x)$ va a ser mayor que el de $f(x)$ salvo por una constante. Formalmente, el conjunto $O(g(x))$ queda definido como [20]:

$$O(g(x)) = \{f(x) : \text{existen } c, x_0 > 0 \text{ tales que } \forall x \geq x_0,\ 0 \leq \mid f(x) \mid \leq c \mid g(x) \mid\}$$

Debido a que el tiempo de ejecución de un algoritmo cuenta el número de pasos que éste realiza, necesariamente $f(x) \geq 0$, por lo que en este caso se pueden quitar los valores absolutos de la definición. Si $f(x) \in O(g(x))$, se dice que $f(x)$ es del orden de $g(x)$ y se escribe $f(x) = O(g(x))$.

Sea $g(n)$ tal que $f(n) = O(g(n))$. Si $g(n)$ es un polinomio, entonces se dice que $f(n)$ es de orden polinomial y el algoritmo resuelve el problema

en un tiempo polinomial. De manera análoga, si $g(n)$ fuera exponencial o factorial, diríamos que el algoritmo lo resuelve en tiempo exponencial o factorial respectivamente. De esta forma, el orden es el que da la complejidad del algoritmo.

A continuación se presenta parte de una tabla obtenida de [12]. En ella se muestra el tiempo que tardaría una computadora en resolver instanacias con diferentes tamaños de entrada para diferente número de operaciones elementales. Se considera que cada operación elemental toma un nanosegundo (10^{-9} seg) en llevarse a cabo.

$n \backslash f(n)$	n^2	$n^{3.5}$	2^n	$n!$
10	.0000001 seg	.000003 seg	.000001 seg	.004 seg
20	.0000004 seg	.000036 seg	.001 seg	76 años
30	.0000009 seg	.000148 seg	1 seg	$8 * 10^{15}$ años
40	.0000016 seg	.000404 seg	1100 seg	-
50	.0000025 seg	.000884 seg	13 días	-
60	.0000036 seg	.002 seg	37 años	-

Vemos como para instancias sencillas con entradas de tamaños de pequeños no hay mucha diferencia entre el tipo de algoritmo que se utilice. Sin embargo, para tamaños de entradas más grandes el tiempo de los algoritmos exponenciales y factoriales los vuelve poco útiles en la práctica. Es por esto que en general se prefiere tener un algoritmo con complejidad polinomial. Sin embargo, no siempre es posible encontrar este tipo de algoritmos.

Claramente vamos a preferir algoritmos "rápidos", por lo que consideramos como más eficiente a un algoritmo con un tiempo de ejecución menor. Sin embargo, podemos encontrar algoritmos funcionen bien en la práctica, aunque existan instancias excepcionales que den tiempos de ejecución muy altos. Un claro ejemplo de esto es el método Simplex. Éste tiene un tiempo de ejecución exponencial, pero en la gran mayoría de los casos funciona en tiempo polinomial.

Transformación polinomial

Una manera de encontrar soluciones a problemas de decisión es convertirlos a problemas que ya han sido resueltos. Así, cada instancia del problema que queremos resolver se convierte en una instancia para la cual ya existe

un algoritmo que la soluciona. Como queremos métodos eficientes para encontrar las respuestas a los problemas, buscamos que tanto el algoritmo que transforma el problema original como el que resuelve el problema conocido sean algoritmos eficientes.

La resolución se centra entonces en la forma de convertir un problema en otro. Si tenemos dos problemas de decisión, Π_1 y Π_2, y contamos con un algoritmo que resuelve Π_2, queremos encontrar una función calculable en tiempo polinomial que transforme Π_1 en Π_2. Lo que hace esta función es convertir si-instancias de Π_1 en si-instancias de Π_2, y análogamente con las no-instancias. De esta manera, podemos obtener soluciones para Π_1 utilizando el algoritmo que resuelve Π_2. A esta función calculable se le llama **transformación polinomial**, y también se le conoce como reducción polinomial o reducción de Karp. A continuación se presenta una definición más formal, dada por Korte y Vygen en [12].

Definición 1.2. *Sean $\Pi_1 = (L_1, P_1)$ y $\Pi_2 = (L_2, P_2)$ dos problemas de decisión. Decimos que Π_1 se puede transformar polinomialmente a Π_2 si existe una función calculable en tiempo polinomial $f : L_1 \longrightarrow L_2$ tal que si $x \in P_1$, entonces $f(x) \in P_2$ y si $x \in L_1 \backslash P_1$ entonces $f(x) \in L_2 \backslash P_2$. A f se le conoce como una transformación polinomial*

Al juntar el algoritmo que realiza la transformación con el algoritmo que resuelve Π_2 obtenemos un algoritmo para Π_1. Nos interesa saber si dicho algoritmo es un algoritmo eficiente. Claramente, si el algoritmo para Π_2 no es eficiente, entonces el algoritmo que buscamos tampoco lo será, ya que se utiliza como base para resolver el problema. La siguiente proposición nos dice que si el algoritmo para Π_2 tiene complejidad polinomial, entonces el algoritmo resultante para Π_1 también tendrá complejidad polinomial. La proposición es análoga a una que presentan Korte y Vygen en [12]; la demostración, dada por Hopcroft, et al, se obtuvo de [9].

Proposición 1.3. *Sean Π_1 y Π_2 dos problemas de decisión. Si Π_1 se puede transformar polinomialmente a Π_2 y existe un algoritmo de complejidad polinomial para Π_2, entonces existe un algoritmo polinomial para Π_1*

Demostración. Sea A_t un algoritmo de orden polinomial $O(n^j)$ para la transformación polinomial de Π_1 a Π_2. Sea A_2 un algoritmo de orden polinomial $O(n^k)$ para Π_2. El algoritmo que buscamos para Π_1, al que llamaremos A_1,

va a construirse a partir de los algoritmos A_t y A_2. A grandes rasgos, A_1 recibe una entrada, la transforma por medio de A_t y después determina el resultado por medio de A_2.

Queremos demostrar que A_1 resuelve el problema Π_1. Sea w una entrada para A_1. El algoritmo A_t recibe a w como entrada y hace la transformación polinomial, dando como resultado, o salida, w'. Por la forma en la que se define la transformación polinomial, w' va a ser una entrada para A_2, definiendo así una instancia de Π_2. Si w es una si-instancia de Π_1, entonces w' es una si-instancia de Π_2, por lo que A_2 indicará que es una si-instancia. De esta manera, si w es una si-instancia, el algoritmo A_1 indicará en el resultado que lo es. El procedimiento es análogo si w es una no-instancia. Por lo tanto, A_1 es un algoritmo que resuelve Π_1.

Ahora sólo queda demostrar que A_1 tiene complejidad polinomial. Supongamos que w es la entrada de A_1 y $long(w) = n$. Por definición de $O(n^j)$, sabemos que el número de pasos de A_t queda acotado cn^j, donde c es una constante. Escribir un elemento de la salida de A_t es un paso del algoritmo, por lo que si w' es la salida de A_t, entonces w' va a tener a lo más longitud cn^j. Esto es $long(w') \leq cn^j$.

A su vez, w' va a ser la entrada del algoritmo A_2. El orden de A_2 es $O(n^k)$ para una entrada de longitud n. Como w' tiene longitud a lo más cn^j, entonces el orden de A_2 es $O((cn^j)^k)$.

La complejidad de A_1 va obtenerse como el máximo número de pasos al juntar los algoritmos A_t y A_2. De esta forma, obtenemos que la complejidad de A_1 es:

$$O(n^j) + O((cn^j)^k) = O(n^j + (cn^j)^k) = O(n^j + cn^{jk})$$

Como c, j, k son constantes, tenemos un polinomio en n, por lo que el tiempo de ejecución A_1 es polinomial. □

1.3. Funciones recursivas parciales

Como las funciones calculables son aquellas que se pueden resolver por medio de algoritmos, nos interesa conocer cuáles son estas funciones. La tesis de Church, que se enunciará formalmente en el capítulo 2, equipara las funciones calculables con las funciones recursivas parciales. Es por esta

razón que buscamos definir estas funciones y explicar cómo se generan para así conocerlas.

Para poder explicar lo que son las funciones recursivas parciales tenemos que definir algunos conceptos. Una **función total** es una función definida sobre todos los enteros no negativos. Una **función parcial** es una función que no necesariamente está definida sobre todos los enteros no negativos. Es decir, en una función parcial pueden existir argumentos para los cuales la función no está definida.

Para poder construir la clase de las funciones recursivas parciales, necesitamos ciertas funciones totales, a las que llamaremos funciones básicas. Además. requerimos de algunas reglas generadoras las cuales aplicaremos a dichas funciones.

Las funciones básicas son: la función sucesión, la función constante igual a cero y las proyecciones. Utilizando la misma notación que utiliza Beckman en [1]. estas funciones quedan definidas como:

$$\begin{aligned} S(x) &= x+1 \\ N(x) &= 0 \\ U_i^{(n)}(x_1,\ldots,x_n) &= x_i \qquad 0<i\leq n \end{aligned}$$

Las primeras reglas generadoras que utilizaremos son la **composición** y la **recursión primitiva**. La composición de funciones permite que una función reciba el valor obtenido por medio de otra función como argumento. La recursión primitiva toma una función total $r(x)$ para la cual el valor en $r(0)$ es conocido y una función total $g(x,y)$, tal que $r(k+1) = g(k, r(k))$. Es decir, la función $g(x,y)$ nos da los valores de $r(x)$ a partir de valores anteriores. De esta manera podemos encontrar el valor de $r(k)$ para cualquier k por medio de iteraciones de la función $g(x,y)$.

Por ejemplo, si tomamos las funciones $S(x)$ y $N(x)$ definidas anteriormente, podemos generar por medio de la composición una nueva función $h(x)$ definida como

$$h(x) = S(N(x)) = N(x) + 1 = 1.$$

En el caso de la recursión primitiva, sea $r(0) = 1$ y $g(k, r(k)) = (k+1)\times(r(k))$, entonces la función generada queda definida como

$$r(k+1) = (k+1) \times r(k).$$

Con las funciones básicas y las reglas generadoras presentamos la siguiente definición, obtenida de [1]:

Definición 1.4 (Funciones Recursivas Primitivas). *La clase de las funciones recursivas primitivas consiste de aquellas funciones que pueden ser obtenidas a partir de la composición y la recursión primitiva de las funciones básicas* $S(x)$, $N(x)$ *y* $U_i^{(n)}(x_1, \ldots, x_n)$.

Al componer dos funciones totales, la función que se obtiene también es una función total. Lo mismo ocurre con la recursión primitiva. Es así como sabemos que las funciones recursivas primitivas son funciones totales.

Podemos extender la clase de las funciones recursivas primitivas si agregamos la **minimización** dentro de las reglas generadoras. Si tenemos alguna función a la cual queremos aplicar esta regla, obtenemos una nueva función que da el valor más pequeño para la cual la función original se hace cero. Es decir, dada una función total $f(y, x_1, \ldots, x_n)$, la minimización da una función $h(x_1, \ldots, x_n)$ cuyo valor es la mínima y tal que $f(y, x_1, \ldots, x_n) = 0$. La función $h(x_1, \ldots, x_n)$ no es necesariamente una función total, por lo que la minimización nos puede llevar a generar funciones parciales. Por ejemplo, si tomamos la función $f(y, x) = |y^2 - x|$, la función $h(x) = min_y(|y^2 - x| = 0)$ obtenida con la minimización queda definida únicamente para las x que son cuadrados perfectos.

Una vez que introducimos la minimización a las reglas generadoras debemos tener cuidado al utilizar la composición. Esto es porque las funciones que se buscan componer pueden ser funciones parciales, por lo que pueden no estar definidas para algunos argumentos. Así, decimos que la composición $f(g(x))$ está definida en x_0 si y solo si $g(x)$ está definida en x_0 y $f(y)$ está definida en $g(x_0)$.

Con esta nueva regla obtenemos entonces la clase de las **funciones recursivas parciales**, obtenida también de [1], definida como:

Definición 1.5 (Funciones Recursivas Parciales). *La clase de las funciones recursivas parciales consiste de aquellas funciones que pueden ser obtenidas a partir de la composición, la recursión primitiva y la minimización de las funciones básicas* $S(x)$, $N(x)$ *y* $U_i^{(n)}(x_1, \ldots, x_n)$.

Ésta es la clase de funciones que nos interesa, ya que es la que equipararemos con las funciones calculables. Sin embargo, si introducimos un argumento que

no esté definido, podríamos tener algoritmos que no terminan. Es por esto que hay que tener cuidado en el manejo de estas funciones. A veces se prefiere trabajar con la clase de las **funciones recursivas**, también conocida como funciones recursivas generales. Estas funciones son funciones totales que pertenecen a la clase de las funciones recursivas parciales.

Capítulo 2

Máquinas de Turing

2.1. Tesis de Church-Turing

En el primer capítulo dimos una definición intuitiva de lo que es un algoritmo. Aunque el concepto de algoritmo o procedimiento efectivo se puede entender fácilmente, la definición no suele ser enteramente formal. De [17] obtenemos que, para que un procedimiento sea considerado como un procedimiento efectivo, o bien un algoritmo, debe tener ciertas características:

- Consiste de un número finito de instrucciones exactas, cada una descrita por un número finito de símbolos.
- Si las instrucciones se siguen sin errores, termina en un número finito de pasos.
- Una persona puede, teórica o prácticamente, seguir las instrucciones utilizando únicamente lápiz y papel.
- La persona que lo sigue no requiere de ingenio o comprensión para seguirlo.

El *Entscheidungsproblem* planteado por Hilbert buscaba como solución un procedimiento de este tipo que indicara la verdad o falsedad de proposiciones lógicas. Alan Turing quería demostrar que dicha solución no existe, por lo que necesitaba comprender este tipo de procedimientos a fondo. Para lograr esto,

en vez de ver lo que el algoritmo pedía que se hiciera, Turing se concentró en lo que la persona que lo estaba siguiendo realmente hacía, ya que, en el momento en el que se plantea el *Entscheidungsproblem*, los algoritmos eran ejecutados por personas. De esta manera, Turing conservó únicamente lo esencial de los algoritmos, y comenzó el desarrollo de las máquinas que más adelante llevarían su nombre.

La forma en la que surgen las máquinas de Turing llevó a Turing a formular su tesis. Aunque hay muchas formulaciones, la **Tesis de Turing** básicamente dice que las máquinas de Turing pueden realizar cualquier cosa que pueda ser descrita como un procedimiento efectivo. Otra formulación de la tesis, dada por Beckman en [1], dice que las funciones calculables y las funciones que pueden ser calculadas por máquinas de Turing son las mismas. Recordemos del capítulo 1 que las funciones calculables son aquellas para las cuales existe un algoritmo que las calcula. En otras palabras, si tenemos una función calculable que represente un problema. entonces éste puede ser resuelto tanto por un algoritmo como por una máquina de Turing.

Además, también en el desarrollo del *Entscheidungsproblem* Alonzo Church llegó a una conclusión similar. La **Tesis de Church** nos dice que el conjunto de las funciones recursivas parciales es equivalente al de las funciones calculables [1].

Al conocer los resultados de Church, Turing mostró que el método utilizado por Church y las máquinas de Turing eran equivalentes, por lo que ambos resuelven exactamente los mismos problemas. Es decir, el conjunto de las funciones parcialmente recursivas es equivalente al conjunto de las funciones calculables por una máquina de Turing.

Al ser equivalentes ambas tesis. se les conoce únicamente como tesis de Church, o bien **Tesis de Church-Turing**. Una de las versiones más sencillas y utilizadas obtenida de [18] dice:

> Todo algoritmo o procedimiento efectivo es Turing computable.

Dicha tesis aun no ha sido demostrada, sobre todo por falta de definiciones formales para algoritmo o procedimiento efectivo. Sin embargo, en general la tesis de Church-Turing se considera como cierta. Así, las máquinas de Turing pasan a ser las formalizaciones matemáticas de los algoritmos. Esto es, si tenemos un algoritmo, en teoría siempre lo vamos a poder describir

como una máquina de Turing, aunque en la práctica esto puede ser muy complicado.

2.2. Descripción de las máquinas de Turing

Una **máquina de Turing** consta de una cinta infinita dividida en celdas y de un lector que se coloca sobre la cinta. Al comenzar, no hay símbolos escritos en las celdas de la cinta. Se podría decir que cada celda de la cinta tiene el símbolo ⊔, que representa al espacio en blanco. Al igual que los algortimos, las máquinas de Turing reciben entradas que definen la instancia del problema a resolver. Estas entradas se presentan como palabras de un alfabeto Σ. Cada símbolo de las palabras que codifican las entradas se escribe en una de las celdas de la cinta. Si se recibe más de una entrada, éstas se separan con espacios en blanco.

El lector tiene un número finito de estados en los que puede encontrarse, que son los que le indican cómo tiene que actuar. Los estados del lector buscan simular los estados mentales de las personas que siguen los algoritmos. Por ejemplo, al hacer una multiplicación de dos números con más de dos dígitos, se llevan a cabo dos etapas: en la primera se multiplica y en la segunda se suma. Así, cuando multiplicamos 125 por 67, lo que hacemos es:

$$\begin{array}{rrrrr} & & 1 & 2 & 5 \\ & & \times & 6 & 7 \\ \hline & & 8 & 7 & 5 \\ + & 7 & 5 & 0 & \\ \hline & 8 & 3 & 7 & 5 \end{array}$$

La persona que está haciendo la multiplicación sabe en que etapa se encuentra, y realiza las operaciones correspondientes. En la primera etapa, su mente le indica que debe multiplicar, mientras que en la segunda le indica que debe sumar. Sin embargo, si queremos que una máquina haga la misma multiplicación, debemos indicarle de alguna forma cuando se encuentra en las diferentes etapas y qué hacer en cada una de ellas. Queremos que la máquina "sepa" cuándo tiene que sumar y cuándo tiene que multiplicar. Los estados del lector nos sirven para indicarle a la máquina los diferentes estados mentales en los que se puede encontrar.

El lector se coloca sobre una y sólo una celda de la cinta. Después lee el símbolo en dicha celda y, dependiendo del estado en el que se encuentre y del símbolo leído, actúa de determinada manera. Cuando el lector actúa, se ven afectados el estado en el que se encuentra, el símbolo de la celda leída y la posición sobre la cinta. Las formas en las que puede actuar son: cambiar el símbolo o dejarlo igual, cambiar de estado o permancecer en el mismo, y desplazarse una celda, ya sea a la derecha o a la izquierda o quedarse en la misma celda.

Las máquinas de Turing deben tener un estado inicial, y muchas veces también presentan un conjunto de estados finales. Cuando la máquina de Turing empieza a funcionar, el lector, que se encuentra en el estado inicial, se coloca sobre la primera celda de la cinta que contenga un símbolo del alfabeto Σ. La máquina lee dicha celda y actúa como se describió anteriormente. Después. ya que se realizaron los cambios correspondientes, repite el proceso en la celda en la que ahora se encuentra. La máquina de Turing sólo se detiene cuando no hay una instrucción que seguir para algún caso dado, o bien cuando llega al estado final. Se dice que una máquina de Turing acepta una entrada si la máquina se detiene. Sin embargo, si se introduce una palabra que no sea aceptada por la máquina de Turing, ésta podría seguir trabajando eternamente.

Al conjunto de entradas, o palabras, que acepta la máquina se le llama el lenguaje aceptado por la máquina de Turing. Dependiendo de lo que se necesite en cada caso, puede ser conveniente que el lenguaje aceptado por una máquina sea el conjunto de instancias o de si-instancias de un problema. Así, si tenemos un problema de decisión $\Pi = (L, P)$, podemos construir para él una máquina T que acepte como lenguaje L o P, según nos convenga.

Si la máquina cuenta con un conjunto de estados finales, éstos pueden determinar el resultado del problema. Por ejemplo, si tenemos una máquina de Turing que resuelva un problema de decisión, ésta podría contar con un estado final de aceptación y otro de rechazo. Así, el estado en el que se detenga la máquina indica si la entrada representa una si-instancia o un no-instancia. Sin embargo, existen otras formas de determinar el resultado sin necesidad de un estado final. Lo que queda escrito en la cinta de la máquina cuando ésta se detiene puede indicar el resultado del problema. Esta salida nos da el resultado de la instancia. Es por esto que un estado final no es estrictamente necesario, pero puede llegar a ser conveniente.

Basado en el libro de Cook, et al [4], obtenemos la siguiente definición de una máquina de Turing como una función matemática.

Definición 2.1 (Máquina de Turing). *Sea M el conjunto de posibles estados, $D = \{\rightarrow, \bullet, \leftarrow\}$ el conjunto de los movimientos posibles ($\bullet$ denota que no hay movimiento) y un alfabeto $\Sigma_\sqcup$, una máquina de Turing es una función T tal que:*

$$T : M \times \Sigma_\sqcup \longrightarrow M \times \Sigma_\sqcup \times D$$
$$T(m, \sigma) = (m', \sigma', d)$$

Dado que M, $\Sigma_\sqcup$ y D son conjuntos finitos, podemos expresar una máquina de Turing como el conjunto de quintetos $(m, \sigma, m', \sigma', d)$ que la conforman. Dichos quintetos definen las posibles situaciones en los diferentes estados, y las acciones que se pueden tomar en cada una de ellas. En otras palabras, podríamos ver una máquina de Turing como un conjunto finito de instrucciones o reglas a seguir.

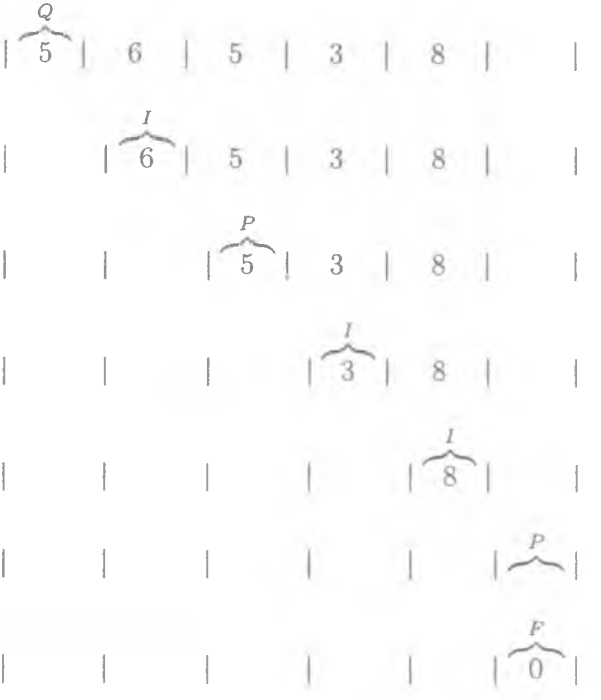
Con el fin de dejar esto más claro, a continuación se muestran dos ejemplos dados por Davies en [5].

Ejemplo 2.2. Una máquina de Turing que revisa si un entero positivo es par o impar: Sea $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ el alfabeto utilizado, $M = \{Q, P, I, F\}$, donde Q es el estado inicial y F es el estado final. Las instrucciones que sigue, o bien, los quintetos que conforman la máquina son:

Estado inical Q	Estado P (par)	Estado I (impar)
$Q0 : \sqcup \rightarrow P$	$P0 : \sqcup \rightarrow P$	$I0 : \sqcup \rightarrow P$
$Q1 : \sqcup \rightarrow I$	$P1 : \sqcup \rightarrow I$	$I1 : \sqcup \rightarrow I$
$Q2 : \sqcup \rightarrow P$	$P2 : \sqcup \rightarrow P$	$I2 : \sqcup \rightarrow P$
$Q3 : \sqcup \rightarrow I$	$P3 : \sqcup \rightarrow I$	$I3 : \sqcup \rightarrow I$
$Q4 : \sqcup \rightarrow P$	$P4 : \sqcup \rightarrow P$	$I4 : \sqcup \rightarrow P$
$Q5 : \sqcup \rightarrow I$	$P5 : \sqcup \rightarrow I$	$I5 : \sqcup \rightarrow I$
$Q6 : \sqcup \rightarrow P$	$P6 : \sqcup \rightarrow P$	$I6 : \sqcup \rightarrow P$
$Q7 : \sqcup \rightarrow I$	$P7 : \sqcup \rightarrow I$	$I7 : \sqcup \rightarrow I$
$Q8 : \sqcup \rightarrow P$	$P8 : \sqcup \rightarrow P$	$I8 : \sqcup \rightarrow P$
$Q9 : \sqcup \rightarrow I$	$P9 : \sqcup \rightarrow I$	$I9 : \sqcup \rightarrow I$
$Q\sqcup : \sqcup \rightarrow Q$	$P\sqcup : 0 \bullet F$	$I\sqcup : 1 \bullet F$

Esta máquina de Turing recibe un número entero positivo como entrada; las cifras que lo conforman se escriben en celdas consecutivas de la cinta. El lector se coloca inicialmente en la cifra que se encuentra más a la izquierda. Después, el lector lee esa cifra, revisa si es par o impar, cambia al estado correspondiente, la borra y se mueve a la siguiente cifra a la derecha. Este proceso se repite, hasta que llega a un espacio en blanco, lo cual indica que el número terminó. De esta manera, si el último dígito del número fue par, la máquina se encontrará en el estado P y escribirá un número 0 en la cinta. En otro caso se encontará en el estado I y escribirá un 1. Al final, esta cifra es la que nos indicará si el número introducido es par o impar. Esta máquina en particular no realiza movimientos hacia la izquierda.

Por ejemplo, si se introduce el número 56538 la máquina de Turing haría lo siguiente:



Si esta máquina de Turing recibe como entrada a la palabra vacía, entonces el lector se desplazará eternamente hacia la derecha. Esto es porque, al no encontrar un número en la celda, el lector se moverá a la derecha hasta

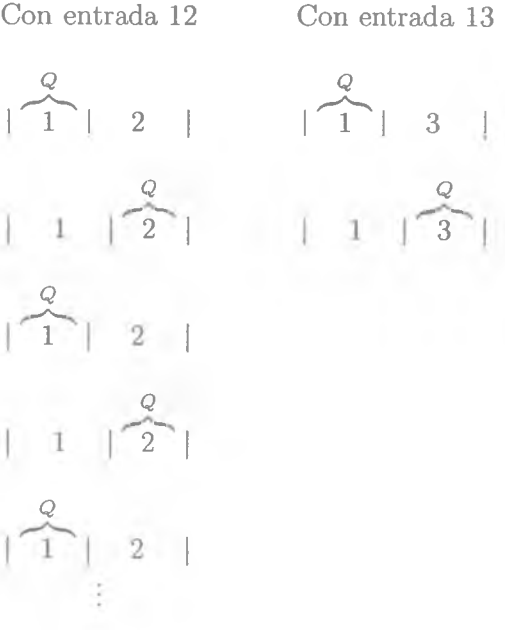
encontrarlo de acuerdo con la instrucción

$$Q\sqcup : \sqcup \rightarrow Q.$$

Ejemplo 2.3. Una máquina de Turing que consta únicamente de un estado inicial Q y de dos quintetos:

$Q1 : 1 \rightarrow Q$	$Q2 : 2 \leftarrow Q$

Si introducimos como dato de entrada 12, esta máquina no se detendrá, sino que avanzará y retrocederá eternamente. Si introducimos cualquier otro dato como entrada, la máquina se detendrá en algún momento.



2.3. Codificación

Los elementos de $\Sigma_{\sqcup}$, como los estados y las direcciones se pueden codificar utilizando palabras construidas a partir del alfabeto, así como cualquier símbolo utilizado dentro de la descripción de la máquina de Turing. Se pueden

asignar cadenas de caracteres que indiquen el comienzo y el fin de las palabras representantes de cada cosa. De esta manera, podemos tener prefijos y sufijos que nos indiquen si lo que se va a leer a continuación es un elemento de $\Sigma_{\sqcup}$, un estado o una dirección. También podemos utilizar codificaciones especiales para indicar el fin de un quinteto o el comienzo del otro.

Aunque se puede usar cualquier alfabeto para dar una codificación, es conveniente que la codificación sea dada en el mismo alfabeto con el que trabaja la máquina. De esta manera, dada una codificación podremos expresar una máquina de Turing como una cadena de caracteres, o bien, un elemento de Σ^*. Sabemos que siempre vamos a poder hacer esto ya que el conjunto de instrucciones es finito.

Podemos utilizar la maquina de Turing dada en el ejemplo 2.3 para mostrar como podría ser una codificación, obtenida también de [5]:

Símbolo	Código	Símbolo	Código
0	8008	$\sqcup$	8558
1	8018	$\rightarrow$	616
2	8028	$\leftarrow$	626
3	8038	$\bullet$	636
4	8048	:	646
5	8058	;	77
6	8518	Q	99
7	8528		
8	8538		
9	8548		

El símbolo de (;) representa el fin de un quinteto, y es denotado en la codificación por 77. El prefijo y sufijo 8 indican que lo que se leerá a continuación es un elemento de $\Sigma_{\sqcup}$, mientras que el prefijo y sufijo 6 indican direcciones, así como el símbolo (:), que separa las entradas de las salidas de la máquina de Turing. Finalmente, el prefijo y sufijo 9 denota estados. Con esto, podemos describir la máquina de Turing 2.3

$$Q1 : 1 \rightarrow Q; Q2 : 2 \leftarrow Q$$

como:

$$99801864680186169977998028646802862699.$$

La codificación de una máquina de Turing no es única. Se pueden elegir diferentes alfabetos o diferentes sufijos y prefijos. Lo mismo ocurre con su representación. Se pueden utilizar diferentes símbolos para denotar los mismos estados, o incluso alterar el orden en el que se presentan los quintetos. En otras palabras, la descripción y la codificación de una máquina de Turing dependen de la persona que la esté dando, por lo que es posible dar diferentes cadenas que representen a la misma máquina.

La codificación no se utiliza únicamente en las máquinas de Turing. Ésta también es utilizada, como ya se vio, para las entradas de los algoritmos, lo cual nos permite conocer el tamaño de las mismas. Diferentes alfabetos dan diferentes tamaños de entrada. Sin embargo, el alfabeto elegido no va a alterar el orden del tiempo de ejecución del algoritmo. El alfabeto más comúnmente utilizado es $\Sigma = \{0, 1\}$.

2.4. Casos especiales de máquinas de Turing

Máquina universal de Turing

Como ya vimos, cada algoritmo nos sirve para resolver un problema. Al ser formalizaciones matemáticas de algoritmos, cada máquina de Turing también resuelve únicamente un problema. Esto nos lleva a tener que hacer una máquina de Turing particular para cada problema que queramos resolver. Turing quería encontrar una máquina que englobara a todas las máquinas de Turing posibles. Es así como surge la **máquina universal de Turing**.

Esta máquina resuelve cualquier problema que pueda ser resuelto por alguna máquina de Turing ordinaria. En otras palabras, una máquina universal de Turing puede calcular cualquier cosa que sea calculable. Esta máquina podría ser vista como una computadora o bien como un intérprete. De hecho, se han programado computadoras como máquinas universales de Turing [1].

La forma en la que funciona es la siguiente. La máquina universal de Turing recibe como entradas dos cadenas de caracteres. La primera es la codificación de una máquina de Turing y la segunda es la codificación de la entrada que se le quiere dar a dicha máquina. La máquina universal primero decodifica la primera entrada, para así saber las instrucciones que va a seguir. Después. decodifica la segunda entrada para comenzar a trabajar en el problema. De esta forma puede simular cualquier máquina de Turing ordinaria.

Máquinas con varias cintas

Las **máquinas de Turing con varias cintas** funcionan de manera muy similar a las máquinas de Turing de una sola cinta. Constan, como su nombre lo indica, de varias cintas divididas en celdas, cada una con su lector. Además, cuenta con una unidad de control, que es la que regula el estado en el que se está y las acciones que debe tomar el lector de cada cinta. Los lectores de las diferentes cintas son independientes unos de otros, es decir, pueden moverse en diferentes direcciones, así como escribir distintos símbolos. Al comenzar a funcionar, la máquina de varias cintas recibe la entrada en la primera cinta, mientras que las demás cintas se encuentran vacías. De la misma forma, la unidad de control se encuentra en el estado inicial.

Las máquinas de Turing de varias cintas facilitan el cómputo y la programación. Sin embargo, cualquier máquina con un número finito de cintas puede ser simulada por una máquina de una sola cinta. Todo lo que se escribe en las diferentes cintas puede ser escrito en una sola cinta, acomodándolo y poniendo señalizaciones adecuadas. De la misma manera, el lector se puede mover a todas las celdas de la cinta. Es por esto que cualquier problema que pueda ser resuelto por una máquina de Turing de varias cintas puede también ser resuelto por una máquina de Turing ordinaria.

Máquinas deterministas y no deterministas

Como ya vimos, una máquina de Turing consiste de un conjunto de quintetos de la forma $(m, \sigma, m', \sigma', d)$. Los primeros dos elementos del quinteto son el estado en el que se encuentra la máquina antes de actuar y el símbolo leido. Estos dos elementos describen las situaciones en las que puede actuar una máquina, mientras que los siguientes tres elementos indican las acciones que se toman en cada caso.

En una **Máquina de Turing determinista** los primeros dos elementos de cada quinteto son únicos. Es decir, si un quinteto comienza con un estado y símbolo determinados, ningún otro quinteto de la máquina comenzará con esa pareja de estado y símbolo. De esta manera, para cada situación, la máquina de Turing sólo puede actuar de una manera. Así, al dar una entrada a la máquina, los pasos que se llevarán a cabo quedan completamente determinados.

Una **Máquina de Turing no determinista** puede tener dos o más quintetos que comiencen con el mismo estado y el mismo símbolo. Es decir, pueden existir situaciones para las cuales la máquina de Turing podría tomar dos o más acciones diferentes. Al encontrarse en una situación de este tipo, la máquina escoge al azar alguna de las acciones posibles. De esta forma, al recibir una entrada la máquina podría seguir varias secuencias de pasos distintas, por lo que la secuencia no está determinada. Para este tipo de máquinas el número de quintetos que presentan también es finito.

Como plantea Beckman en [1], en teoría es posible construir una máquina de Turing no determinista que dé como resultado cualquier palabra escogiendo símbolos del alfabeto al azar, por lo que podría resolver problemas que una máquina determinista no puede. Sin embargo, no hay forma de saber si el resultado obtenido es el correcto. Es por esto que, en general, las máquinas de Turing no deterministas se utilizan únicamente para resolver problemas de decisión. Dado un problema $\Pi = (\Sigma^*, P)$ y una entrada $x \in \Sigma^*$, la máquina de Turing seguirá alguna de las posibles secuencias para determinar si $x \in P$. Si alguna de estas posibles secuencias me lleva a que $x \in P$, entonces x es solución de Π. En cambio, si ninguna de las posibles secuencias me llevan a aceptar, ya sea porque se rechaza o porque no se detiene, entonces x no es solución de Π.

Los problemas de decisión que pueden resolver las máquinas deterministas y las no deterministas son los mismos. Es decir, si existe una máquina no determinista que resuelva algún problema, podemos encontrar una máquina determinista que también lo resuelva. Para probar esto, necesitamos demostrar que, si tenemos una máquina de Turing no determinista, podemos construir una máquina de Turing determinista que acepte el mismo lenguaje. Claramente, la máquina determinista deberá ser capaz de llevar a cabo cualquier secuencia de acciones que puedan ser realizadas por la máquina no determinista.

El siguiente teorema. junto con su demostración, fue obtenido de [9] y [10]. En el teorema suponemos que si las máquinas se detienen es porque alcanzaron un estado final. Aunque esto no es necesariamente cierto para todas las máquinas de Turing. como se ve en la máquina del ejemplo 2.3, facilita la demostración y no afecta el resultado. También suponemos que las máquinas reciben una sola entrada. En caso de que reciban más de una entrada, se puede generalizar la demostración, tomando como alfabeto $\Sigma_\sqcup$ y como única entrada la cadena de caracteres conformada por las diferentes

entradas separadas por espacios.

Teorema 2.4. *Sea M_N una máquina de Turing no determinista y sea $L \subset \Sigma^*$ el lenguaje aceptado por dicha máquina. Entonces existe una máquina de Turing determinista M_D para la cual L es también el lenguaje aceptado.*

Demostración. Al ser una máquina no determinista, M_N puede tener diferentes quintetos en los cuales se repitan las parejas de estados y símbolos con las que comienzan. Como el número de quintetos es finito, existe una pareja de estado y símbolo que se repite un mayor número de veces que las demás. Llamaremos m al número máximo de veces que se repite cualquier pareja de estado y símbolo al comienzo de un quinteto. En otras palabras, m es el mayor número de opciones entre las cuales M_N podría escoger en dada circunstancia.

Como tenemos la máquina M_N, contamos con el listado de quintetos que la conforman. Podemos agrupar los quintetos de este listado, de forma que aquellos que comiencen con la misma pareja de estado y símbolo queden juntos. Después, podemos enumerar los quintetos de cada grupo con números de 1 hasta m, ya que a lo más hay m quintetos que comiencen de la misma forma.

Así, dada una entrada w, podemos describir cada uno de los diferentes caminos que puede seguir M_N como una cadena conformada por $\{1, 2, \ldots, m\}$. Por ejemplo, si tenemos la cadena 12, entonces entre los quintetos que podemos elegir al principio, eligiremos el que queda enumerado como 1. Después, entre las opciones a la que nos lleve esa elección, eligiremos la 2 y así sucesivamente. A estas cadenas las llamaremos **configuraciones**. Claramente, no todas las configuraciones son posibles. Por ejemplo, si para la cadena anterior después del primer paso sólo existe una opción o se llega a un estado final, esta configuración no será posible.

Vamos a construir la maquina determinista M_D como una máquina de tres cintas. Esto no afecta en la demostración, ya que sabemos que podemos simular esta máquina con una máquina determinista de una sola cinta. Al comenzar, en la primera cinta se escribe la entrada, denotada como w. La segunda cinta comenzará en blanco, y en ella se generarán las configuraciones posibles de M_N.

Las configuraciones se generarán de manera ordenada, encontrándose primero las configuraciones más cortas, y en orden numérico en caso de haber

varias de la misma longitud. Este orden será en el que iremos trabajando con ellas para ver si nos llevan a un estado final, y por lo tanto a aceptar a w. Además, debe haber una forma de señalar la configuración con la que se está trabajando, a la cual llamaremos configuración actual. De esta manera, las configuraciones que se encuentren a la izquierda de la configuración actual serán configuraciones que ya se revisaron, con las cuales no se llegó a un estado final.

En la tercera cinta se escribe de nuevo w y se simula M_N de acuerdo con la configuración actual. Es decir, se sigue el camino que indica la configuración hasta que ésta termine. Si al terminar esta simulación se llega a un estado final, entonces detenemos la máquina M_D, por lo que se acepta w como entrada.

En el caso en el que no se llegue a un estado final, se revisa la pareja de estado y símbolo en la cual se terminó la configuración. Se cuenta el número de quintetos que comienzan con esa pareja, y se llama a ese número k. Se copia entonces la configuración actual k veces al final de la segunda cinta, agregando al final de cada copia un símbolo de $\{1, 2, \ldots, k\}$ de manera ordenada. De esta manera, se agregan a la lista de configuraciones los k posibles caminos que puede seguir la máquina. Al terminar este procedimiento, la señalización que marca la configuración actual se desplazará a la siguiente configuración a la derecha, indicando que la configuración con la que se estaba trabajando no llegó a un estado final.

Es así como la máquina M_D va revisando todas las configuraciones posibles de M_N. Primero se revisan todas las configuraciones que requieren 0 movimientos, luego las que requieren 1 y así sucesivamente. Debido las configuraciones se forman a partir de un número finito de caracteres, las configuraciones para cada número de movimientos, o bien, las configuraciones de cierta longitud, son finitas.

Sea L es el lenguaje aceptado por M_N y $w \in L$. Como w está en L, entonces para alguna configuración finita de M_N, la máquina se detendrá, llegando a un estado final. Si introducimos w como entrada de M_D, ésta simulará las configuraciones finitas de M_N hasta llegar a la configuración con la cual M_N se detiene. Por lo tanto M_D también se detendrá, y aceptará a w como entrada.

Ahora, si tomamos w' tal que $w' \notin L$, entonces no existe una configuración finita a partir de w' para la cual M_N se detenga. Esto implica que ninguna

configuración finita de M_N me lleva a un estado final. Como M_D simula las configuraciones finitas de M_N, si introducimos w' como entrada, entonces M_D no se detendrá, por lo que nunca llegará a un estado final. Por lo tanto, w' no es una palabra que sea aceptada por M_D.

Tenemos entonces que si una palabra está en L entonces es aceptada por M_D. En cambio, si tenemos una palabra que no está en L, dicha palabra no es aceptada por M_D. Esto nos lleva a que L es el lenguaje aceptado por M_D, por lo que M_N y M_D aceptan el mismo lenguaje. □

Al existir máquinas deterministas que reconocen exactamente el mismo lenguaje que las no deterministas, vemos de esta forma que no existen problemas de decisión que puedan ser resueltos por máquinas no deterministas pero no por máquinas deterministas. Si esto no ocurriera, tendríamos un procedimiento efectivo que no es Turing calculable, y tendríamos un contraejemplo para la tesis de Church-Turing.

Si tomamos una máquina no determinista M_N y una máquina determinista M_D construida como en la demostración anterior, vemos como M_N va a ser más rápida. De acuerdo con Hopcroft, et al en [9]. por la forma en la que se construye la máquina determinista a partir de la no determinista, el tiempo que podría emplear la primera es exponencialmente mayor. Vemos entonces que la diferencia entre usar alguna de estas dos máquinas no radica en los problemas que pueden resolver, sino en el tiempo en el que los resuelven.

Capítulo 3

Clases de Complejidad

3.1. Problemas indecidibles y problemas intratables

Al enfrentarnos con un problema, queremos que el algoritmo que lo resuelva sea el más eficiente posible. Los algoritmos poco eficientes suelen revisar todos los casos posibles, de manera que, al aumentar el tamaño de la entrada, el tiempo de ejecución se incrementa de manera muy rápida. En cambio, los algoritmos polinomiales se desarrollan a partir de una comprensión más completa del problema. Esto permite que el algoritmo se concentre únicamente en aquellas características que engloban la esencia del problema, lo cual lleva a una mejor resolución.

Sin embargo, existen muchos problemas para los cuales no se conoce un algoritmo polinomial, y otros para los cuales se sabe que no existe solución alguna. Es por esto que nos interesa conocer el tipo de problema con el que estamos tratando antes de comenzar a buscar un algoritmo que lo solucione. De esta manera podemos enfocar nuestros esfuerzos a buscar algoritmos que cubran mejor nuestras necesidades.

Los problemas para los cuales no existen algoritmos que los resuelvan se conocen como **problemas indecidibles**. En la búsqueda de una solución para el *Entscheidungsproblem*, Turing probó que el conjunto de estos problemas es no vacío. En su demostración, Turing exhibió un problema, el problema de la parada, y mostró que ningún algoritmo puede resolverlo. Es

decir. Turing demostró que el problema de la parada es un problema indecidible. A continuación se explica en que consiste este problema. Tanto la explicación como demostración de su indecibilidad fueron dadas por Davies en [5].

Recordemos del capítulo anterior que podemos codificar una máquina de Turing de manera que ésta quede representada como una palabra de Σ^*. El alfabeto Σ que utilizaremos para dar esta codificación será el mismo con el que trabaja la máquina. De esta forma, la palabra que describe a la máquina puede ser una entrada para la misma. Llamaremos a esta palabra el código de la máquina de Turing.

Por otro lado, el conjunto de paro de una máquina de Turing es el conjunto de entradas para las cuales la máquina se detiene. En otras palabras, el conjunto de paro equivale al lenguaje aceptado por una máquina de Turing. El problema de la parada consiste en ver si, dada una máquina de Turing, su código está o no dentro de su conjunto de paro.

Proposición 3.1. *El problema de la parada es un problema indecidible.*

Demostración. Sea D un subconjunto del conjunto de todos los posibles códigos de las máquinas de Turing. Este subconjunto D queda caracterizado por la siguiente propiedad:

El código de una máquina de Turing es un elemento de D si y sólo si el código de la máquina no se encuentra dentro del conjunto de paro de la misma.

Entonces tenemos que, si tomamos una máquina de Turing cualquiera, o bien su código está dentro de su conjunto de paro o no. Si sí se encuentra dentro de su conjunto de paro, entonces el código no es elemento de D, en otro caso, el código está en D. Por lo tanto, D va a ser diferente al conjunto de paro de cualquier máquina de Turing.

El problema de la parada es equivalente a ver si la entrada dada se encuentra en D. Lo que queda por demostrar es que no existe un algoritmo que determine si una palabra de Σ^* se encuentra en el conjunto D. Esta demostración se hará por contradicción.

Sea $\Pi_{parada} = (\Sigma^*, D)$ el problema de la parada. Supongamos que existe una máquina de Turing H que resuelve Π_{parada}. Esta máquina se va a detener al llegar a un estado final F, regresando un 1 si la entrada está en D y un 0

si la entrada no está en D. Al ser un estado final, no hay quintetos en H que comiencen con F. Si la máquina H no se detiene al recibir alguna entrada es porque la entrada no representa al código de alguna máquina.

A partir de esta máquina H vamos a construir otra máquina, denotada H', que no se detenga si la entrada no está en D. La máquina H' va a tener todos los quintetos de H, pero va a contar además con los siguientes dos quintetos:

$F0 : \sqcup \rightarrow F$	$F\sqcup : \sqcup \rightarrow F$

Así, si la entrada no se encuentra en D, H' no se detendrá. De esta forma, D va a ser el conjunto de paro de la máquina H', ya que ésta sólo se detiene si la entrada está en D.

Sin embargo, esto es una contradicción, ya que, por la forma en la que se construyó, D es diferente al conjunto de paro de cualquier máquina de Turing. Por lo tanto, no existe un algoritmo que resuelva el problema de la parada. □

Los **problemas intratables** son aquellos problemas para los cuales se ha demostrado que no existe una solución eficiente. En otras palabras, si tenemos un problema intratable, no existe un algoritmo polinomial que nos de una solución. Algunos problemas intratables pueden contar con algoritmos no polinomiales que los resuelvan. Sin embargo, el tiempo que tardan puede ser tan alto que en la práctica no se pueden utilizar.

La intratabilidad nos permite enfocar nuestros esfuerzos únicamente a ciertos aspectos de los problemas, atacándolos de una mejor manera. De esta forma. no perdemos tiempo buscando soluciones exactas y eficientes, ya que sabemos que no vamos a poder encontrarlas. Una vez que sabemos que cierto problema es intratable, podemos buscar algoritmos que funcionen bien en la mayoría de los casos, o bien que encuentren soluciones buenas, aunque no óptimas.

No siempre sabemos de antemano si un problema es intratable, y demostrar que lo es puede ser muy complicado. El que no se haya podido encontrar un algoritmo eficiente para un problema no implica necesariamente que éste no exista. Es por esto que existen muchos problemas que se sospecha son intratables pero no se ha podido probar que lo son.

Aquellos problemas que se ha demostrado son intratables entran en dos categorías. La primera categoría es la de los problemas indecidibles. Al no

existir un algoritmo que los resuelva, claramente no existe un algoritmo polinomial para ellos. En la segunda categoría se encuentran aquellos problemas que no pueden ser resueltos en tiempo polinomial por una máquina de Turing no determinista.

Por otro lado tenemos los problemas $\mathcal{NP}$ – *completos*. Estos problemas pueden ser resueltos por máquinas no deterministas en un tiempo polinomial, pero se sospecha que son intratables. Aun cuando no se han encontrado algoritmos polinomiales que los resuelvan, tampoco se ha podido demostrar que éstos no existen. Es por esto que normalmente estos problemas se manejan como problemas intratables.

Para poder definir los problemas $\mathcal{NP}$ – *completos*, debemos primero conocer las diferentes clases de complejidad. Los diferentes problemas se clasifican de acuerdo al tiempo que tardan en resolverse, dándonos una idea de su dificultad. De esta forma podemos averiguar de que tipo es cada problema, y si es $\mathcal{NP}$ – *completo*, podemos buscar la mejor forma de trabajar con él.

3.2. Clase $\mathcal{P}$

A la clase de problemas de decisión resolubles en tiempo polinomial por una máquina de Turing determinista se le conoce como **clase** $\mathcal{P}$. Como consideramos la tesis de Church-Turing como cierta. un problema de decisión se encuentra en esta clase si y sólo si existe un algoritmo de complejidad polinomial que lo resuelva.

En general se considera que un problema está bien resuelto si existe un algoritmo polinomial para él. En este sentido, la clase $\mathcal{P}$ está compuesta por los problemas de decisión "fáciles", o bien, tratables. Aquí daremos como ejemplos dos problemas que se encuentran en esta clase. Ambos ejemplos, así como muchos de los ejemplos dados en este trabajo. son de teoría de gráficas. En el apéndice A se encuentran los diferentes conceptos de teoría de gráficas que se utilizan.

Ejemplo 3.2. Dada G una gráfica simple conexa con m aristas, encontrar un árbol generador en G.

Este problema puede ser visto como un problema de decisión de la siguiente manera. Dada G, queremos saber si tiene un árbol generador. El algoritmo que lo resuelva debe dar dicho árbol en la salida.

Para mostrar que este problema se encuentra en $\mathcal{P}$ necesitamos dar un algoritmo de complejidad polinomial que lo resuelva. Dicho algoritmo va tomando diferentes aristas de la gráfica y las va colocando en un conjunto E'. Sólo se agregan a E' aquellas aristas que no se hayan agregado antes y que no generen ciclos en la subgráfica que tiene como conjunto de aristas a E'. Al conjunto de aristas que se pueden agregar se le denota E''. El algoritmo sigue corriendo hasta que no queden aristas que se puedan agregar, esto es, hasta que $E'' = \emptyset$.

Paso 0: $E' = \emptyset$

Paso 1: Sea $E'' = \{e \in E(G) \backslash E' \mid G' = (V(G), E') + e$ sea acíclica$\}$

Si $E'' = \emptyset$, ALTO

En otro caso, $E' = E' + e_i$ para algún $e_i \in E''$

Paso 2: Regresar al Paso 1

En el peor de los casos, vamos a tener que hacer m iteraciones del algoritmo. En cada iteración se añade a lo más una arista al conjunto E'. Además, el proceso para revisar que una gráfica es acíclica es de orden polinomial. De esta forma obtenemos que el orden del algoritmo es $O(m)$, por lo que el algoritmo tiene complejidad polinomial. Por lo tanto, este problema pertenece a la clase $\mathcal{P}$.

Ejemplo 3.3 (Problema del Conector). Sea G una gráfica simple conexa y sea $C : E(G) \longrightarrow \mathbb{N} \cup \{0\}$ una función de costos que asigna un peso a cada arista. Queremos hallar una árbol generador T tal que $C(T) = \Sigma_{e \in E(T)} C(e)$ sea mínimo.

Para formular este problema como un problema de decisión, damos como entrada, además de la gráfica G, un entero k. El problema consiste en ver si la gráfica G tiene un arbol generador tal que la suma de los costos de las aristas del árbol sea a lo más k.

El algoritmo de Kruskal (1956) encuentra el árbol generador de costo mínimo en un tiempo polinomial [2]. Este algoritmo toma en cada paso una arista para formar un árbol generador. Dentro de las aristas que no forman ciclos con la gráfica que se va construyendo, se escoge aquella que tenga el costo más pequeño. Podemos utilizar este algoritmo y después comparar el valor encontrado con k, resolviendo de esta forma el problema de decisión en tiempo polinomial. Este algoritmo también es polinomial si los costos de las aristas son números reales.

Proposición 3.4. *Si tenemos un problema $\Pi \in \mathcal{P}$, entonces su complemento, $\overline{\Pi}$, también se encontrará en $\mathcal{P}$.*

Demostración. Sea Π un problema en $\mathcal{P}$. Esto implica que existe una máquina de Turing T tal que T resuelve Π en un tiempo polinomial. Es decir, si tenemos una si-instancia de Π de longitud n, entonces T se detiene antes de $p(n)$ pasos, donde $p(x)$ es un polinomio. Por otro lado, si tenemos una no-instancia de Π, entonces T se detendrá en un estado de rechazo o bien hará más de $p(n)$ pasos.

Vamos a construir una máquina T' para el problema $\overline{\Pi}$ a partir de la máquina T. En esta máquina, intercambiamos los estados finales de aceptación y rechazo de T. Es decir, si T terminaba en un estado de rechazo, entonces T' seguirá el mismo procedimiento pero terminará por aceptar la entrada, y análogamente si terminaba en un estado de aceptación. Por otro lado, agregamos una nueva instrucción que haga que T' se detenga en un estado de aceptación después de $p(n)$ pasos. De esta forma, si tenemos una no instancia de Π, la máquina T' indicará que es una si-instancia de $\overline{\Pi}$ en un número de pasos menor o igual a $p(n)+1$. Por lo tanto, T' resuelve el problema $\overline{\Pi}$ en un tiempo polinomial, por lo que $\overline{\Pi} \in \mathcal{P}$. □

3.3. Clase $\mathcal{NP}$

En la clase de problemas $\mathcal{NP}$ se encuentran aquellos problemas cuya respuesta positiva puede ser verificada en un tiempo polinomial. Esto es, si damos una si-instancia del problema, podemos verificar que efectivamente es una si-instancia con un algoritmo de orden polinomial. La siguiente definición fue obtenida a partir de las definiciones dadas por Korte y Vygen en [12]; Cook, et al en [4]; y Stephen Cook en [3]. En ella se presenta un problema dentro de $\mathcal{P}$ para cada problema en $\mathcal{NP}$. Este problema tiene como instancias palabras compuestas por palabras del problema en $\mathcal{NP}$ y un "certificado" c, separadas por el símbolo $\#$.

Definición 3.5. *Un problema $\Pi = (L, P)$ se encuentra en la clase $\mathcal{NP}$ si existen un polinomio $p(x)$ y un problema $\Pi' = (L', P')$ en la clase $\mathcal{P}$, donde*

$$L' = \{w\#c : w \in L,\ long(c) \leq p(long(w))\},$$

de manera que

$$w \in P \iff \text{Existe } c \text{ tal que } w\#c \in P' \text{ y } long(c) \leq p(long(w)).$$

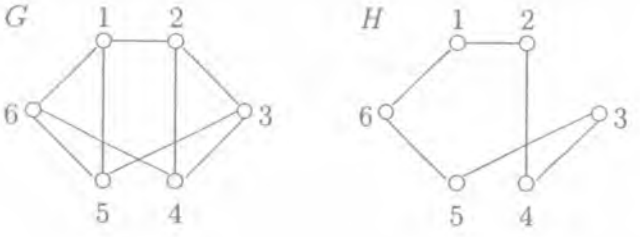
En la definición anterior, suponemos que w es una palabra de Σ^*, mientras que c será una palabra Σ_1^*, donde Σ_1 es un alfabeto. El símbolo $\#$ no se encuentra en los alfabetos Σ ni Σ_1. Éste se utiliza unicamente para unir w con c, formando la palabra $w\#c$ dentro del lenguaje L', que a su vez es un subconjunto de $\Sigma^* \cup \Sigma_1^* \cup \{\#\}$.

La palabra c se utiliza para demostrar que w es una si-instancia de Π. Como el problema Π' se encuentra en la clase $\mathcal{P}$, podemos saber si $w\#c$ es una si-instancia de Π' en tiempo polinomial. De ser así, sabemos inmediatamente que w es una si-instancia de Π. En este caso se dice que c es un **certificado** para w.

A continuación se presenta un ejemplo de un problema de la clase $\mathcal{NP}$. En éste se ve de manera más clara cómo es que actúa un certificado para alguna instancia. Dicho ejemplo fue dado por Cook, et al en [4].

Ejemplo 3.6 (Problema del Ciclo Hamiltoniano)**.** El problema del ciclo hamiltoniano consiste en ver si dada una gráfica simple y conexa ésta tiene un ciclo hamiltoniano. Sea $\Pi = (L, P)$ el problema del ciclo hamiltoniano. Para este problema, L es el conjunto de todas las palabras que representan gráficas, mientras que P es el conjunto de todas palabras que representan gráficas hamiltonianas.

Sea h una palabra que representa un ciclo hamiltoniano H para una gráfica G representada por la palabra g. Claramente, la gráfica G va a ser hamiltoniana, ya que cuenta con un ciclo hamiltoniano H. Es en este sentido en el cual h es un certificado, mientras que $g\#h$ es una instancia de Π'. Por ejemplo, sea G la gráfica que se presenta en el siguiente esquema y sea H un ciclo hamiltoniano para ella.



Si tomamos los alfabetos $\Sigma = \{1,2,3,4,5,6,[,]\}$ y $\Sigma_1 = \{1,2,3,4,5,6,\}$ podemos representar la gráfica G y el ciclo hamiltoniano H como

$$g = [12][15][16][23][24][34][35][46][56]$$

$$h = 1243561.$$

Los corchetes de Σ se utilizan en la codificación para indicar las aristas de G; los números dentro de los corchetes son los vértices que une cada arista. Debido a que la gráfica G es simple y conexa, esto es suficiente para describir la gráfica. La palabra h simplemente indica el orden en el que el ciclo hamiltoniano va visitando los vértices de G. De esta manera, la palabra que representa a la instancia de Π' es

$$g\#h = [12][15][16][23][24][34][35][46][56]\#1243561.$$

Para comprobar que h representa un circuito hamiltoniano, se revisa que todas las aristas del ciclo sean aristas de $E(G)$ y que todos los vértices de $V(G)$ se encuentren representados dentro de h exáctamente una vez, con excepción del vértice donde comienza y termina el ciclo que aparece dos veces. Esto se puede hacer en un tiempo polinomial, por lo que Π' pertenece a $\mathcal{P}$. Por lo tanto Π pertenece a $\mathcal{NP}$.

El nombre de la clase $\mathcal{NP}$ representa "No determinístico Polinomial". Esto es porque originalmente esta clase se definía como la clase de los problemas que pueden ser resueltos por una máquina de Turing no determinista en un tiempo polinomial. Más adelante se introdujo la definición 3.5 para esta misma clase de problemas. Ambas definiciones son equivalentes, lo cual se demostrará a continuación.

Proposición 3.7. *La definición dada en términos de máquinas de Turing no deterministas para la clase $\mathcal{NP}$ es equivalente a la definición 3.5.*

Demostración. Sea X el conjunto de problemas que pueden ser resueltos por una máquina de Turing no determinista en tiempo polinomial. Queremos demostrar que $X = \mathcal{NP}$.

Sea $\Pi \in X$, queremos demostrar que $\Pi \in \mathcal{NP}$. Sabemos que existe una máquina de Turing no determinista que resuelve Π, y que el número de pasos que realiza queda acotado por un polinomio $p(x)$. Podemos enumerar los

diferentes quintetos de dicha máquina de la misma manera que se hizo en la demostración del teorema 2.4. De esta forma podemos describir las posibles configuraciones para cada entrada.

Construimos ahora una máquina determinista a partir de la no determinista. La máquina determinista incluirá todos los quintetos de la no determinista, y éstos quedarán diferenciados de acuerdo a la enumeración anterior. Esta nueva máquina recibirá como entradas w y c, donde w es una entrada para la máquina no determinista y c es una configuración posible para dicha entrada. La máquina seguirá los pasos indicados por c para determinar si w resuelve Π.

Cada elemento de c indica un paso de la máquina no determinista, por lo que $long(c)$ es el número total de pasos que realiza. Además de leer cada elemento de c, la nueva máquina hace estos mismos pasos, dando un tiempo de ejecución polinomial. Si $long(c) > p(long(w))$ sabemos de antemano que esa configuración no llevará a aceptar a w. Es por esto que sólo tomaremos como posibles entradas aquellas configuraciones c tales que $long(c) \leq p(long(w))$, y llamaremos a este conjunto C.

La máquina determinista resuelve un nuevo problema de decisión Π'. Como la máquina tiene un tiempo de ejecución polinomial, Π' pertenece a la clase $\mathcal{P}$. Las instancias de este problema son de la forma $w\#c$, donde w y c son las entradas aceptadas por la máquina y $\#$ es un símbolo que une a ambas.

Si w' es una si-instancia de Π, entonces sabemos que existe al menos una configuración $c' \in C$ con la cual la máquina no determinista llega a un estado de aceptación. Además, si introducimos w' y c' como entradas de la máquina determinista, entonces llegaremos a un estado de aceptación, y $w'\#c'$ será una si-instancia de Π'.

De la misma forma, si tenemos una configuración $c'' \in C$ para una entrada w'', y sabemos que $w''\#c''$ es una si-instancia de Π', entonces podemos tomar c'' como una configuración en la máquina no determinista que lleva a aceptar w''. De esta manera, w'' es una si-instancia de Π. De esta forma demostramos que $\Pi \in \mathcal{NP}$, y por lo tanto $X \subseteq \mathcal{NP}$.

Falta ahora demostrar que $\mathcal{NP} \subseteq X$. Sea Π un problema en $\mathcal{NP}$. Queremos construir una máquina de Turing no determinista que resuelva Π en un tiempo polinomial. Esta máquina recibirá una entrada w y construirá diferentes certificados de manera aleatoria pero ordenada. Después utilizará estos

certificados para determinar si w resuelve Π. Los posibles certificados son palabras de Σ_1^*, donde Σ_1 es un alfabeto.

Por definición, sabemos que existe un problema Π' con el cual podemos verificar las si-instancias de Π. Como $\Pi' \in \mathcal{P}$, existe una máquina de Turing determinista M_D que lo resuelve en un tiempo polinomial. Llamemos $q(x)$ al polinomio que acota el número de pasos de dicha máquina.

El primer certificado c que se prueba es la palabra vacía. Después se utiliza la máquina M_D para determinar si $w\#c$ resuelve el problema Π'. En caso de que lo resuelva, entonces hemos encontrado un certificado para w, por lo que la máquina se detiene indicando que w es una si-instancia de Π. Si no lo resuelve, entonces agregamos un símbolo de Σ_1 a la palabra c de manera aleatoria. Ésta es la parte no determinística de la máquina, ya que tiene tantas opciones como elementos de Σ_1. El procedimiento se repite con el nuevo certificado c. Como sabemos que $long(c) \leq p(long(w))$ para un polinomio $p(x)$, en el momento en que esta desigualdad no se cumpla la máquina se detiene. De esta forma, la máquina no determinista puede generar y probar cualquier palabra $c \in \Sigma_1^*$ tal que $long(c) \leq p(long(w))$.

Si w es una si-instancia de Π, sabemos que existe una c tal que $w\#c$ resuelve Π'. Además, existe una manera de generar dicha c con la máquina no determinista, por lo que ésta aceptará a w como solución. En cambio, si w es una no-instancia de Π, no existe una c con $long(c) \leq p(long(w))$ tal que $w\#c$ resuelva Π', por lo que ninguna secuencia que siga la máquina me llevará a aceptar w. Por lo tanto, la máquina no determinista resuelve el problema Π.

En cada iteración, la máquina no determinista utiliza a la máquina M_D y agrega un elemento a c. Como M_D recibe como entrada $w\#c$, el número de pasos que realiza queda acotado por $q(long(w) + long(c) + 1)$. De esta manera, en cada iteración se hacen a lo más $q(long(w) + long(c) + 1) + 1$ pasos. Sabemos que $long(c) \leq p(long(w))$, por lo que en el peor de los casos $long(c) = p(long(w))$. Así, el mayor número de pasos que puede haber en una iteración queda acotado por

$$q(long(w) + p(long(w)) + 1) + 1.$$

Además. sabemos que a lo más se harán $p(long(w))$ iteraciones. Por lo tanto, el número total de pasos que realiza la máquina no determinista queda

acotado por

$$p(long(w))(q(long(w) + p(long(w)) + 1) + 1),$$

que es un polinomio en $long(w)$. Esto nos indica que el tiempo de ejecución de dicha máquina es polinomial. Por lo tanto, $\mathcal{NP} \subseteq X$. □

3.4. Problemas $\mathcal{NP}$ – *completos* y otras clases de complejidad

Problemas $\mathcal{NP}$ – *completos*

Si tenemos un problema en $\mathcal{NP}$ tal que cualquier otro problema de $\mathcal{NP}$ puede ser transformado polinomialmente a él, entonces se dice que dicho problema es $\mathcal{NP}$ – *completo*. Es decir, los **problemas $\mathcal{NP}$ – completos** son aquellos problemas tales que cualquier problema de $\mathcal{NP}$ puede ser transformado polinomialmente a ellos.

Esta definición nos indica que los problemas $\mathcal{NP}$ – *completos* son al menos tan difíciles como cualquier otro problema de $\mathcal{NP}$. Esto sucede porque, por la proposición 1.3, si resolvemos uno de estos problemas en tiempo polinomial, entonces podemos resolver cualquier problema de $\mathcal{NP}$ en tiempo polinomial. Es por esto que estos problemas son considerados los más difíciles dentro de $\mathcal{NP}$. De hecho, como no se conocen algoritmos polinomiales para ellos, muchas personas consideran que estos problemas son intratables.

La forma de demostrar que un problema es $\mathcal{NP}$ – *completo* es reduciendo algún otro de estos problemas a él. Es decir, si queremos ver que un problema Π_2 es de este tipo y contamos con otro problema Π_1 en $\mathcal{NP}$ – *completo*, entonces podemos buscar una transformación polinomial de Π_1 a Π_2. De esta forma. damos una transformación polinomial de cualquier problema en $\mathcal{NP}$ a Π_2, demostrando que es un problema $\mathcal{NP}$ – *completo*.

Sin embargo, para llevar a cabo este proceso, primero debemos encontrar algún problema que se encuentre en $\mathcal{NP}$ – *completo*. El primer problema que se encontró en esta clase fue el problema de satisfacibilidad booleana, mejor conocido como SAT. En el siguiente capítulo se demostrará que dicho problema es $\mathcal{NP}$ – *completo*.

Clase $Co\mathcal{NP}$

La definición 3.5 para la clase $\mathcal{NP}$ sólo hace referencia a las si-instancias de un problema. El hecho de que un problema esté en $\mathcal{NP}$ no implica que un certificado para una no-instancia pueda ser verificado en tiempo polinomial. De esta forma se genera una nueva clase de problemas, los problemas $Co\mathcal{NP}$.

La clase de problemas $Co\mathcal{NP}$ está compuesta por los completmentos de los problemas de $\mathcal{NP}$. Es decir, si tenemos un problema $\Pi = (L, P)$ dentro de $\mathcal{NP}$, entonces $\overline{\Pi} = (L, L \backslash P)$ se encontrará en la clase $Co\mathcal{NP}$. De esta manera, sabemos que una no-instancia de un problema en $Co\mathcal{NP}$ puede ser verificada en tiempo polinomial.

Por la proposición 3.4 sabemos que la clase $\mathcal{P}$ se encuentra en la intersección de $\mathcal{NP}$ y $Co\mathcal{NP}$. Sin embargo, no se ha podido demostrar que $\mathcal{NP} = Co\mathcal{NP}$, y en general se cree que son diferentes. De hecho, $\mathcal{NP} \neq Co\mathcal{NP}$ implica que $\mathcal{P} \neq \mathcal{NP}$, la cual es una conjetura muy importante de la que se hablará más adelante.

Dentro de la clase $Co\mathcal{NP}$ encontramos también los problemas $Co\mathcal{NP}$ – *completos*. Estos problemas son los complementos de los problemas $\mathcal{NP}$ – *completos* y se puede demostrar que también son completos para $Co\mathcal{NP}$. Es decir, cualquier problema en $Co\mathcal{NP}$ se puede transformar polinomialmente a ellos. Debido a que estos problemas son los más difíciles de los problemas $Co\mathcal{NP}$, podemos ver que el complemento de algún problema $\mathcal{NP}-completo$ se encuentra en $\mathcal{NP}$ si y sólo si $\mathcal{NP} = Co\mathcal{NP}$. Esto se hace transformando polinomialmente cualquier problema de $Co\mathcal{NP}$ en el problema $Co\mathcal{NP}$ – *completo* que se está considerando.

Problemas $\mathcal{NP}$–*difíciles*

Los **problemas $\mathcal{NP}$–*difíciles*** son aquellos problemas tales que cualquier problema en $\mathcal{NP}$ puede transformarse polinomialmente a ellos. Por definición, todos los problemas $\mathcal{NP}$ – *completos* son $\mathcal{NP}$–*difíciles*. Sin embargo, un problema debe encontrarse necesariamente en $\mathcal{NP}$ para ser considerado $\mathcal{NP}-completo$. Ésta es la diferencia entre los problemas $\mathcal{NP}-completos$ y los $\mathcal{NP}$–*difíciles*. Un problema no necesariamente tiene que encontrarse en $\mathcal{NP}$ para ser $\mathcal{NP}$–*difícil*. Es decir, podemos tener problemas $\mathcal{NP}$–*difíciles* que no se encuentren en $\mathcal{NP}$. Estos problemas reciben este nombre por que son al menos tan difíciles como cualquier problema en $\mathcal{NP}$.

Capítulo 4

Problemas $\mathcal{NP}$ – *completos*

4.1. Teorema de Cook

En 1971, Stephen Cook demostró que el conjunto de problemas $\mathcal{NP}$ – *completos* es no vacío. En su demostración, mostró que el problema de satisfacibilidad booleana es un problema $\mathcal{NP}$ – *completo*. Este problema busca determinar si existen valores para los cuales cierta expresión booleana es verdadera. En el anexo de expresiones booleanas se explica de manera más clara cuáles son estas expresiones y cómo se construyen. Todos los conceptos referentes a este tipo de expresiones se encuentran definidos en dicho apéndice.

El problema de satisfacibilidad booleana, mejor conocido como SAT, tiene como conjunto de sus si-instancias al subconjunto de todas las expresiones booleanas que se pueden satisfacer. Es decir, si existen valores para los cuales la expresión booleana se satisface, entonces dicha expresión es una si-instancia de SAT.

Por ejemplo, si tenemos la expresión boolena $f(x) = x$, si $x = 1$ tenemos que $f(x) = 1$. Por lo tanto. esta expresión se puede satisfacer. por lo que es una si-instancia de SAT. En cambio, la expresión $f(x) = x \wedge \overline{x}$, al ser una contradicción, es falsa para cualquier valor de x, por lo que no se puede satisfacer. Esta expresión es por lo tanto una no-instancia de SAT.

Queremos demostrar que el problema SAT es $\mathcal{NP}$ – *completo*. La idea general que se presenta de dicha demostración fue dada por Papadimitriou y

Steiglitz en [14], aunque también se tomaron elementos de [4] y [12]. Antes de comenzar con la demostración, necesitamos probar el siguiente resultado, obtenido de [4], que se utilizará en ella. Supongamos que tenemos un conjunto W de palabras construidas a partir del alfabeto $\{0,1\}$, todas ellas de longitud n. La siguiente proposición nos dice que existe una función booleana que recibe exactamente n variables para las cuales cada palabra de W es una asignación de valores. En particular, esta función tiene la propiedad de tomar un valor de verdadero si y solo si la asignación que recibe equivale a una palabra de W.

Proposición 4.1. *Sean $n \in \mathbb{N}$ y $W \subseteq \{0,1\}^n$, entonces existe una función booleana $f(x_1, x_2, \ldots, x_n)$ tal que para una palabra $w = \alpha_1\alpha_2 \ldots \alpha_n$, con $\alpha_i \in \{0,1\}$, $i = 1, \ldots, n$, se tiene que*

$$w \in W \iff f(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$$

Demostración. La demostración se hace por inducción matemática.

Para probar que la proposición se cumple para $n = 1$, necesitamos considerar todos los subconjuntos posibles de $\{0,1\}^1$. Esto nos lleva a estudiar los siguientes casos:

$$W = \{\},\ W = \{0\},\ W = \{1\} \text{ y } W = \{0,1\}.$$

Si $W = \{\}$, entonces la proposición se cumple por vacuidad.

Si $W = \{0\}$, tomamos la función booleana $f(x) = \overline{x}$. Así podemos ver que

$$\begin{array}{c} \text{Si } x \in W,\ x = 0 \text{ entonces } f(0) = \overline{0} = 1 \\ \text{Si } x \notin W,\ x = 1 \text{ entonces } f(1) = \overline{1} = 0. \end{array}$$

Si $W = \{1\}$, tomamos $f(x) = x$. De esta forma tenemos que

$$\begin{array}{c} \text{Si } x \in W,\ x = 1 \text{ entonces } f(1) = 1 \\ \text{Si } x \notin W,\ x = 0 \text{ entonces } f(0) = 0. \end{array}$$

Si $W = \{0,1\}$, tomamos $f(x) = x \vee (\overline{x})$. En este caso, $f(x)$ es una tautología, por lo que para cualquier valor $w \in W$ se tiene que $f(w) = 1$. Por otro lado, si $f(x) = 1$, esto implica que $x = 0$ o bien $x = 1$, y tanto 0 como 1 son elementos de W. Por lo tanto, la proposición es cierta si $n = 1$.

Supongamos ahora que existe alguna $k \in \mathbb{N}$ para la cual la proposición se cumple. Debemos ahora demostrar que la proposición se cumple para $k+1$.

Sea $W \subseteq \{0,1\}^{k+1}$. Vamos a particionar a W en dos conjuntos, W_0 y W_1. En W_0 vamos a colocar todos aquellos elementos de W tales que su último símbolo es 0. Es decir, todas las palabras de W que terminen en 0 van a estar dentro de W_0. De la misma forma, todas las palabras de W que terminen en 1 van a estar en W_1.

A partir de estos conjuntos construimos dos nuevos conjuntos W_0' y W_1' de la siguiente manera. A todas las palabras de W_0 les quitamos la última cifra, creando así W_0'. Es decir, si tenemos $w = \alpha_1 \ldots \alpha_k \alpha_{k+1}$ una palabra en W_0, entonces $\alpha_1 \ldots \alpha_k \in W_0'$. La construcción de W_1' es análoga. De esta forma obtenemos W_0' y W_1' dos subconjuntos de $\{0,1\}^k$.

Por hipótesis de inducción sabemos que existen dos funciones boolenas f_0 y f_1 tales que

$$\alpha_1\alpha_2 \ldots \alpha_k \in W_0' \iff f_0(\alpha_1, \ldots, \alpha_k) = 1$$
$$\alpha_1\alpha_2 \ldots \alpha_k \in W_1' \iff f_1(\alpha_1, \ldots, \alpha_k) = 1.$$

Con estas dos funciones construimos la función f que buscamos:

$$f(\alpha_1, \ldots, \alpha_k, \alpha_{k+1}) = (f_0(\alpha_1, \ldots, \alpha_k) \wedge (\overline{\alpha}_{k+1})) \vee (f_1(\alpha_1, \ldots, \alpha_k) \wedge \alpha_{k+1})$$

Sea $w = \alpha_1 \ldots \alpha_k \alpha_{k+1}$. Si $w \in W$, entonces sabemos que $w \in W_0$ o bien $w \in W_1$, dependiendo de cual sea el valor de α_{k+1}. Si $w \in W_0$ entonces sabemos que $\alpha_1 \ldots \alpha_k \in W_0'$, por lo que $f_0(\alpha_1, \ldots, \alpha_k) = 1$. Además, sabemos que $\alpha_{k+1} = 0$, por lo que $\overline{\alpha}_{k+1} = 1$. De esta manera obtenemos que $f(\alpha_1, \ldots, \alpha_k, \alpha_{k+1}) = 1$. Ahora, si $w \in W_1$, obtenemos de manera análoga que $f_1(\alpha_1, \ldots, \alpha_k) \wedge \alpha_{k+1} = 1$. Por lo tanto, si $w \in W$ entonces $f(\alpha_1, \ldots \alpha_k, \alpha_{k+1}) = 1$.

Por otro lado, si $w \notin W$ entonces $w \notin W_0$ y $w \notin W_1$, por lo que $\alpha_1\alpha_2 \ldots \alpha_k \notin W_0'$ y $\alpha_1 \ldots \alpha_k \notin W_1'$. Esto nos lleva a que $f_0(\alpha_1, \ldots \alpha_k) = 0$ y $f_1(\alpha_1, \ldots \alpha_k) = 0$, por lo tanto $f(\alpha_1, \ldots \alpha_k, \alpha_{k+1}) = 0$. De esta forma, la proposición se cumple para $k+1$, por lo que la proposición se cumple para todo $n \in \mathbb{N}$. □

La prueba que se da del teorema de Cook fue obtenida en su mayoría de la demostración dada por Papadimitriou y Steiglitz en [14]. Sin embargo,

también se utilizaron las demostraciones dadas por Cook, et al en [4], Korte y Vygen en [12] Moret en [13].

Es fácil ver que el problema de SAT se encuentra en $\mathcal{NP}$. Si una expresión $f(x_1, x_2, x_3, \dots)$ es satisfacible, entonces existen valores $\alpha_1, \alpha_2, \alpha_3, \dots$ tales que $f(\alpha_1, \alpha_2, \alpha_3, \dots) = 1$. Podemos tomar como certificado para esta expresión los valores $x_i = \alpha_i$. La evaluación en estos valores se puede hacer en tiempo polinomial, por lo que SAT $\in \mathcal{NP}$.

Una vez que sabemos que SAT $\in \mathcal{NP}$, queremos ver que todo problema en NP puede ser transformado polinomialmente a SAT. Para poder demostrar esto, primero probaremos que cualquier problema en $\mathcal{P}$ puede transformarse polinomialmente a SAT. Después veremos que lo mismo ocurre con los problemas en $\mathcal{NP}$, lo que nos lleva a concluir que SAT es $\mathcal{NP}$ – *completo*.

En la demostración se explica cómo se construye una transformación polinomial de cualquier problema Π en $\mathcal{P}$ a SAT. Dicha transformación construye una función booleana $F(x)$ para cada instancia de Π. Esta función va a representar exactamente lo que hace la máquina de Turing que resuelve Π para la instancia dada. De esta forma, para cada instancia el algoritmo construye una función booleana que es satisfacible si y sólo si la máquina de Turing llega a un estado de aceptación, dando así transformación polinomial del Π a SAT.

Teorema 4.2. *Cualquier problema $\Pi \in \mathcal{P}$ puede ser transformado polinomialmente al problema SAT.*

Demostración. Sea $\Pi = (\Sigma^*, P)$ un problema en $\mathcal{P}$. Queremos demostrar que existe un algoritmo polinomial que relaciona las si-instancias de Π con expresiones boolenas satisfacibles, y no-instancias con expresiones no satisfacibles. En otras palabras, el algoritmo que buscamos toma $w \in \Sigma^*$ una instancia de Π, y con ella debe construir una función booleana que sea satisfacible si y sólo si $w \in P$.

Tomamos entonces $w \in \Sigma^*$, y llamamos $n = long(w)$. Además, podemos describir a w como una cadena de símbolos de Σ. Es decir. $w = \alpha_1\alpha_2 \dots \alpha_n$, donde $\alpha_j \in \Sigma$ para $j = 1, \dots, n$. Como $\Pi \in \mathcal{P}$, sabemos que existe una máquina de Turing T que resuelve Π en un tiempo polinomial. Esto es, T determinará si w es una si-instancia de Π en un número de pasos menor a $p(n)$, donde $p(x)$ es un polinomio.

Sabemos que T es de la siguiente forma:

$$T : M \times \Sigma_{\sqcup} \longrightarrow M \times \Sigma_{\sqcup} \times D$$

$$T(m, \sigma) = (m', \sigma', d)$$

Aquí. M es el conjunto de estados. Σ es el alfabeto con el que se trabaja, que será el mismo sobre el que está definido el problema Π. y $D = \{-1, 0, 1\}$ es el conjunto de direcciones posibles. El movimiento del lector a la derecha se representa por un 1, a la izquierda por -1 y si el lector no se mueve se representa con 0. La máquina T es una máquina determinista, por lo que, al ser finitos todos los conjuntos anteriores, el número de quintetos posibles también es finito. Sea N este número.

Numeraremos las celdas de la cinta de T. Al escribir la entrada c en la cinta, la celda donde se encuentra α_1, el primer símbolo de w, será la celda 0. El número de la celda irá incrementando hacia la derecha y disminuyendo hacia la izquierda. Sabemos que a lo más T realiza $p(n)$ pasos, por lo que la celda más a la derecha que se puede alcanzar es la celda $p(n)$, y hacia la izquierda la celda $-p(n)$. En otras palabras, el lector sólo se puede encontrar dentro de estas celdas, ya que si se encuentra afuera de este rango, sabemos que la instancia w es una no-instancia.

Sin pérdida de generalidad, contamos con un estado inicial y un estado final. La máquina T se detendrá si y sólo si alcanza un estado final, en cuyo caso se aceptará a w como una si-instancia de Π. Vamos a enumerar también todos los estados en M. El estado inicial va a recibir el número 1, mientras que el estado final va a recibir el número $|M|$.

La función booleana que queremos construir necesita representar los procesos que lleva a cabo la máquina de Turing T. Es por esto que necesitamos representar las diferentes situaciones de T por medio de variables booleanas. Para poder representar tanto las variables como los procesos de T, utilizamos los siguientes índices:

Índice	Rango	Representación
i	$0 \leq i \leq p(n)$	Paso de la máquina T
j	$-p(n) \leq j \leq p(n)$	Posición en la cinta
k	$1 \leq k \leq \|M\|$	Estado en el que se encuentra T
σ	$\sigma \in \Sigma_{\sqcup}$	Símbolo del alfabeto
q	$1 \leq q \leq N$	Quinteto de T que se ejecuta

Claramente, los índices i, j, k y q son números enteros. En la máquina T, las parejas de estado y símbolo al inicio de cada quinteto determinan las acciones que se toman. Es por esto que necesitamos variables que representen a los elementos de estas parejas. Utilizaremos entonces las variables booleanas $x_{i,j,\sigma}$ y $y_{i,j,k}$ en la función que queremos construir. De esta manera, si en el i-ésimo paso del algoritmo, en la posición j-ésima se encuentra el símbolo σ, entonces $x_{i,j,\sigma}$ toma el valor 1, mientras que si no es cierto, toma el valor 0. De la misma forma $y_{i,j,k}$ da un valor 1 si en el paso i-ésimo, en la posición j-ésima, el lector de T se encuentra en el k-ésimo estado, y un valor de 0 de no ser cierto.

El número de variables de la forma $x_{i,j,\sigma}$ es $(p(n)+1)(2p(n)+1)(|\Sigma|)$. El número de variables de la forma $y_{i,j,k}$ es $(p(n)+1)(2p(n)+1)(|M|)$. Así, el número de variables con las que contamos es

$$(2p(n)^2+3p(n)+1)(|\Sigma|+|M|).$$

Como ya vimos, estas variables representan lo que sucede en cada paso de T. Podemos definir W un subconjunto de $\{0,1\}^{(2p(n)^2+3p(n)+1)(|\Sigma|+|M|)}$, donde los elementos de W representen procesos válidos de T que me lleven a aceptar a w. Por la proposición 4.1, sabemos que, una vez que contamos con el número de variables, siempre va a existir una función booleana que nos ayude a determinar si las variables dadas representan un elemento de W.

Por lo tanto, siempre vamos a poder construir una función booleana con estas propiedades. Esta función, a la que llamaremos $F(\mathbf{x})$, utiliza estas variables. Es decir,

$$F : \{0,1\}^{(2p(n)^2+3p(n)+1)(|\Sigma|+|M|)} \longrightarrow \{0,1\}.$$

La construcción de $F(\mathbf{x})$ va a depender en su mayoría de la longitud de w. Esto es, si tenemos dos entradas del mismo tamaño, la función que se construya será la misma, excepto en una subfunción de $F(\mathbf{x})$. La manera en la que se incluye a w en $F(\mathbf{x})$ es en la representación del paso inicial de T. En este paso, la subfunción indica que la máquina se encuentra en el estado inicial, con los elementos que conforman w en las primeras n celdas.

La construcción de $F(\mathbf{x})$ se lleva a cabo en cuatro etapas, las cuales definen diferentes aspectos de T. Así, construimos cuatro expresiones booleanas más pequeñas que unidas forman $F(\mathbf{x})$. Los índices dados en dichas subfunciones corren dentro de los rangos establecidos anteriormente.

El primer aspecto que describimos es el de unicidad. Esto es, queremos que en cada paso del algoritmo, cada posición dentro del rango tenga un único símbolo. Además, queremos que el lector se encuentre en una sola posición y un solo estado. A esta función la llamaremos $U(\mathbf{x})$

Primero vemos que los símbolos en cada paso y posición sean únicos. Esto lo hacemos con la siguiente expresión:

$$U_1 = \bigwedge_{\substack{i,\ j \\ \sigma \neq \sigma'}} (\overline{x}_{i,j,\sigma} \vee \overline{x}_{i,j,\sigma'})$$

Si existieran dos símbolos diferentes para algún paso i y una posición j, esto implicaría que tendríamos $x_{i,j,\sigma} = 1$ y $x_{i,j,\sigma'} = 1$. De esta forma, $\overline{x}_{i,j,\sigma} = 0$ y $\overline{x}_{i,j,\sigma'} = 0$ por lo que toda la expresión anterior se haría cero. Así, esta expresión nos garantiza que en cada paso hay un solo símbolo en cada posición.

De la misma manera, utilizamos la siguiente expresión para indicar que en un paso dado el lector sólo puede encontrarse en una posición y en un estado:

$$U_2 = \bigwedge_{i} \bigwedge_{\substack{j \neq j' \\ \vee \\ k \neq k'}} (\overline{y}_{i,j,k} \vee \overline{y}_{i,j',k'})$$

Por último, la siguiente expresión indica que en un tiempo dado. debe haber algún símbolo en todas las celdas del rango. Además, el lector debe encontrarse necesariamente en alguna posición y en algún estado.

$$U_3 = \bigwedge_{i} \left(\left(\bigwedge_{j} \bigvee_{\sigma} x_{i,j,\sigma} \right) \wedge \bigvee_{j,\ k} y_{i,j,k} \right)$$

Al juntar todas estas expresiones, obtenemos $U(\mathbf{x})$.

$$U(\mathbf{x}) = U_1 \wedge U_2 \wedge U_3$$

El segundo aspecto que queremos cubrir indica que T se inicia de forma adecuada. Es en este aspecto en el cual se utiliza directamente a la entrada $w = \alpha_1 \ldots \alpha_n$ en la construcción de $F(\mathbf{x})$. Así, la segunda función, denotada $S(\mathbf{x})$ es la siguiente:

$$S(\mathbf{x}) = \left(\left(\bigwedge_{j=0}^{n-1} x_{0,j,\alpha_{j+1}} \right) \wedge y_{0,0,1} \right) \wedge \left(\bigwedge_{(-p(n) \leq j < 0) \wedge (n \leq j \leq p(n))} x_{0,j,\sqcup} \right)$$

Esto nos indica que, al comenzar a funcionar la máquina, los primeros n símbolos en la cinta son los de la entrada w, y el lector se encuentra en la celda 0 en el estado inicial. Además, todos las demás celdas del rango están vacías, o bien, contienen como símbolo al espacio en blanco.

La tercera función que utilizamos indica que la máquina funciona correctamente. Esto es, queremos que las variables indiquen exactamente las acciones que sigue T en cada paso.

En un paso dado la máquina lee el símbolo de la posición en la que se encuentra el lector. En el siguiente paso el estado y el símbolo leído serán cambiados de acuerdo al quinteto correspondiente, mientras que el lector se desplazará a la posición adecuada. Para indicar que esto sucede, vamos a tener la siguiente expresión W_q por cada quinteto de T:

$$W_q = \bigwedge_{i,j} (\overline{x}_{i,j,\sigma_q} \vee \overline{y}_{i,j,k_q} \vee x_{i+1,j,\sigma'_q}) \wedge (\overline{x}_{i,j,\sigma_q} \vee \overline{y}_{i,j,k_q} \vee y_{i+1,j+d_q,k'_q})$$

Agregamos además la siguiente expresión. Esta indica que una vez que llegamos a un estado final, el lector no se va a mover y el símbolo de la celda no va a cambiar.

$$W_f = \left(\bigwedge_{i,j} \overline{y}_{i,j,|M|} \vee y_{i+1,j,|M|} \right) \wedge \left(\bigwedge_{i,j,\sigma} \overline{x}_{i,j\sigma} \vee \overline{y}_{i,j,|M|} \vee x_{i+1,j,\sigma} \right)$$

Por último, la siguiente expresión indica que las celdas que no están siendo leídas permanecen igual:

$$\bigwedge_{\substack{i,\sigma,k \\ j \neq j'}} \overline{x}_{i,j,\sigma} \vee \overline{y}_{i,j',k} \vee x_{i+1,j,\sigma}$$

Así, la expresión $W(\mathbf{x})$ queda de la siguiente forma:

$$W(\mathbf{x}) = \left(\bigwedge_{q=1}^{N} W_q \right) \wedge W_f \wedge \left(\bigwedge_{\substack{i,\sigma,k \\ j \neq j'}} \overline{x}_{i,j,\sigma} \vee \overline{y}_{i,j',k} \vee x_{i+1,j,\sigma} \right)$$

Finalmente, queremos indicar que T termina de manera adecuada. Esto es, queremos que en el último paso el lector se encuentre en cualquier posición,

pero en el estado final. Esto queda indicado por la función $E(\mathbf{x})$:

$$E(\mathbf{x}) = \bigvee_{j=-p(n)}^{p(n)} y_{p(n),j,|M|}$$

Así, el algoritmo que buscamos es el siguiente. Primero toma w y obtiene su longitud. En el siguiente paso calcula $p(n)$, $|\Sigma|$, $|M|$ y N. Después introduce las variables $x_{i,j,\sigma}$ y $y_{i,j,k}$ de acuerdo con los índices indicados. Finalmente, construye las funciónes booleanas anteriores, y a partir de ellas obtiene la función $F(\mathbf{x})$ de la siguiente manera:

$$F(\mathbf{x}) = U(\mathbf{x}) \wedge S(\mathbf{x}) \wedge W(\mathbf{x}) \wedge E(\mathbf{x}).$$

Ahora nos interesa ver que este algoritmo es una trasformación polinomial de Π a SAT. Primero necesitamos ver que $F(\mathbf{x})$ se construye en tiempo polinomial. Como ya vimos, el número de variables es de orden $O(p^2(n))$. Además, estas variables se asocian y relacionan de diferentes maneras en cada una de las subfunciones de $F(\mathbf{x})$. Estas asociaciones quedan acotadas por $O(p^3(n))$. Así, la construcción de $F(\mathbf{x})$ se da en tiempo polinomial.

Por otro lado, sea $w = \alpha_1 \ldots \alpha_n$ una instancia de Π. Queremos ver que $F(\mathbf{x})$ es satisfacible si y sólo si w es una si-instancia de Π. Es importante notar que la función $F(\mathbf{x})$ toma en cuenta los elementos que conforman a w en su construcción.

Supongamos que $F(\mathbf{x})$ es satisfacible. Sea $\mathbf{t}$ una asignación de las variables de $\mathbf{x}$ tal que $F(\mathbf{t}) = 1$. Por lo tanto $U(\mathbf{t})$, $S(\mathbf{t})$, $W(\mathbf{t})$ y $E(\mathbf{t})$ son todas iguales a 1.

Primero vemos que $U(\mathbf{t}) = 1$. Entonces sabemos que para cada i y cada j dentro de los rangos dados, exactamente una variable $x_{i,j,\sigma}$ tiene un valor de verdadero. Esto nos lleva a que en cada paso y posición particular se encuentra un único símbolo de $\Sigma_{\sqcup}$. Por otro lado, para cada i dentro del rango, exactamente una variable $y_{i,j,k}$ es verdadera, por lo que que en cada paso el lector se encuentra en exactamente un estado y una posición. De esta forma, sabemos que $\mathbf{t}$ describe una secuencia de situaciones posibles para la máquina T. Las demás funciones indicarán si dicha secuencia es válida para T.

Como $S(\mathbf{t}) = 1$, sabemos que, al comienzo, la entrada w está escrita correctamente en las celdas correspondientes de la cinta de T, mientras que

el resto de las celdas se encuentran en blanco. Además, el lector se encuentra en la primera celda que tiene un símbolo de Σ, como se indica en la descripción de las máquinas de Turing dada en el capítulo 2.

La siguiente parte, $W(\mathbf{t}) = 1$, nos indica que la secuencia de pasos se lleva a cabo de forma correcta. Cuando el lector se topa con una pareja de estado y símbolo descritos por un quinteto de T, escribe en la celda el símbolo correspondiente, cambia de estado y se desplaza de la manera indicada por dicho quinteto. Cuando alcanza un estado final, los siguientes pasos no cambian a las celdas ni al lector, por lo que podemos decir que la máquina se detiene. Además, todas aquellas celdas sobre las cuales no se encuentra el lector se mantienen iguales, por lo que sabemos que la máquina únicamente actúa sobre la celda en la que se encuentra el lector.

Por último, también tenemos que $E(\mathbf{t}) = 1$. Con esto sabemos que T alcanza un estado final de aceptación en alguna posición en un número de pasos menor o igual a $p(n)$.

Al unir todas las funciones, podemos ver que las secuencias descritas por $\mathbf{t}$ son válidas para la máquina T. Así, al recibir a w como entrada, la máquina T sigue una secuencia de pasos que llevan a aceptar a w. Por lo tanto, w es una si-instancia del problema Π.

Si suponemos que w es una si-instancia de Π, entonces sabemos que la máquina T realiza una secuencia de pasos menor a $p(n)$ que me lleva a aceptar w. De acuerdo con esta secuencia podemos dar una asignación de valores $\mathbf{t}$ para las variables de $\mathbf{x}$ tal que $F(\mathbf{t}) = 1$.

Construimos esta asignación numerando las celdas de la cinta, los estados y los quintetos de T en la forma en la que se describió anteriormente. Después se asignan los valores de $\mathbf{t}$ de acuerdo a lo escrito en la cinta en cada paso, así como la posición y estado del lector. Como $F(\mathbf{x})$ se construyó para representar los procesos que realiza la máquina T, una asignación $\mathbf{t}$ construida a partir de un proceso válido de T necesariamente dará un valor de 1 a dicha función.

De esta forma, si w es una si-instancia de Π, entonces $F(\mathbf{x})$ es una si-instancia de SAT, y si es una no-instancia, $F(\mathbf{x})$ también será una no instancia de SAT. Además, la construcción de $F(\mathbf{x})$ se lleva a cabo en tiempo polinomial. Por lo tanto, para cualquier problema $\Pi \in \mathcal{P}$ existe una transformación polinomial de Π a SAT. □

Una vez que tenemos este resultado para los problemas en $\mathcal{P}$, lo utilizaremos para probar lo mismo para problemas en $\mathcal{NP}$. De esta forma, podemos obtener el siguiente resultado.

Teorema 4.3 (Teorema de Cook, 1971). *El problema de satisfacibilidad booleana, SAT, es $\mathcal{NP}$ – completo.*

Demostración. Para probar el Teorema de Cook, necesitamos demostrar que cualquier problema en $\mathcal{NP}$ puede ser transformado polinomialmente a SAT. Sea $\Pi = (\Sigma^*, P)$ un problema en $\mathcal{NP}$. Entonces, existen un problema Π' en $\mathcal{P}$ y un polinomio $p(x)$ con las características indicadas en la definición 3.5.

Sea w una instancia de Π con $long(w) = n$. Sabemos que w es una si-instancia si y sólo si existe c tal que $w\#c$ es una si-instancia de Π' y $long(c) \leq p(n)$. Por lo tanto, sabemos que existe una máquina de Turing que determina si w es una si-instancia de Π' en un número de pasos menor a $q(long(w\#c)) = q(n + 1 + p(n))$, donde $q(x)$ es un polinomio. Para facilitar la notación, hacemos $Q = q(n + 1 + p(n))$.

Ahora, sabemos por el teorema 4.2 que existe una transformación polinomial de Π' a SAT. Sin embargo, en esta transformación necesitamos contar con el certificado c para la construcción de la función $F(\mathbf{x})$ correspondiente, ya que c es parte de la entrada. Es por esto que debemos hacer algunos cambios a esta función para lograr la transformación polinomial de Π a SAT.

Como se vio en la demostración del teorema 4.2, la parte que involucra a la entrada es la función $S(\mathbf{x})$. Esta será la parte que modificaremos, obteniendo una función $S'(\mathbf{x})$. Como las funciones $U(\mathbf{x})$, $W(\mathbf{x})$ y $E(\mathbf{x})$ dependen únicamente de Q, podemos construirlas de la manera usual. De esta forma, la función que construimos será:

$$F'(\mathbf{x}) = U(\mathbf{x}) \wedge S'(\mathbf{x}) \wedge W(\mathbf{x}) \wedge E(\mathbf{x}).$$

La subfunción $S'(\mathbf{x})$ queda definida de la siguiente manera:

$$S'(\mathbf{x}) = \left(\left(\bigwedge_{j=0}^{n-1} x_{0,j,a_{j+1}} \right) \wedge x_{0,n,\#} \wedge y_{0,0,1} \right) \wedge \left(\bigwedge_{(-Q \leq j < 0) \wedge (n+1+p(n) \leq j \leq Q)} x_{0,j,\sqcup} \right)$$

Esto nos indica que seguimos escribiendo los elementos de w en la cinta de la manera usual, además de agregar el símbolo # al final de w. Sin embargo,

sólo aseguramos que haya espacios en blanco después de $p(n)$ celdas. De esta forma, dejamos $p(n)$ celdas libres en donde se pueden escribir diferentes certificados, sin necesidad de conocer alguno de antemano.

Así, si w es una si-instancia de Π, existe un certificado c apropiado tal que $w\#c$ produce una secuencia de pasos que lleva a aceptar a dicha entrada. Por lo tanto, existe una manera de otorgar valores a las celdas libres de forma que $F'(\mathbf{x}) = 1$.

Por otro lado, si $F'(\mathbf{x})$ es satisfacible entonces existe una asignación $\mathbf{t}$ que la satisface. Esta asignación da valores a las $p(n)$ celdas libres, las cuales darán un certificado c. Dado que $F'(\mathbf{t}) = 1$, sabemos que con este certificado la máquina acepta a $w\#c$, por lo que existe c tal que $w\#c$ es una si-instancia de Π' y $long(c) \leq p(n)$. Así, w es una si-instancia de Π.

La construcción de $F'(\mathbf{x})$ es igual a la de $F(\mathbf{x})$ excepto por algunas diferencias en $S(\mathbf{t})$ y $S'(\mathbf{t})$. Como sabemos que $F(\mathbf{x})$ se construye en un tiempo polinomial, entonces $F'(\mathbf{x})$ también se obtiene en un tiempo polinomial. Por lo tanto, hemos obtenido una transformación polinomial de Π en SAT. □

4.2. Programación lineal entera y 3-SAT

La demostración del teorema de Cook da una transformación polinomial de cualquier problema en $\mathcal{NP}$ a SAT. Como tenemos que considerar cualquier problema, la prueba es muy laboriosa. Sin embargo, una vez que contamos con un problema $\mathcal{NP}$ – *completo*, podemos utilizarlo para demostrar que otros problemas son de este tipo.

Si tenemos una transformación polinomial de un problema en $\mathcal{NP}$ – *completo* a otro problema en $\mathcal{NP}$, podemos asegurar que el segundo problema también es $\mathcal{NP}$ – *completo*. Esto es porque cualquier problema en $\mathcal{NP}$ puede transformarse polinomialmente al primer problema, que a su vez puede transformarse polinomialmente al segundo, obteniendo así la transformación deseada.

En general, la manera de probar que un problema Π_1 es $\mathcal{NP}$ – *completo* es dar una transformación polinomial de un problema $\mathcal{NP}$ – *completo* Π_2 a Π_1. Por la forma en la que definen los problemas $\mathcal{NP}$ – *completos*, Π_2 podría ser cualquier problema de este tipo. Sin embargo, es conveniente utilizar

algún problema cuya transformación a Π_1 sea sencilla, o bien, que tenga una estructura similar a la de Π_1. La demostración tiene la siguiente forma.

- Se demuestra que Π_1 está dentro de $\mathcal{NP}$.
- Se escoge algún problema $\mathcal{NP}$ – *completo* Π_2 para reducirlo a Π_1.
- Se describe la transformación polinomial de Π_2 a Π_1.
- Se demuestra que la transformación manda si-instancias de Π_2 a si-instancias de Π_1, y hace lo mismo con las no-instancias.
- Se verifica que la transformación sea polinomial.

En la demostración de aquellos problemas que se transforman directamente de SAT, tomaremos como instancias únicamente expresiones booleanas en forma normal conjuntiva (FNC). Esto es porque, por la proposición B.1 dada en el apéndice de expresiones booleanas, cualquier expresión booleana puede transformarse polinomialmente a otra en FNC. Existe otra razón por la cual podemos hacer esto. En la transformación dada en el teorema de Cook 4.3, la expresión booleana obtenida está en FNC. Por lo tanto, la transformación que buscamos recibirá como entradas expresiones en FNC.

Programación lineal entera

Un problema de optimización en programación lineal entera tiene la siguiente forma:

$$\begin{array}{ll} \text{minimizar} & c'x \\ \text{sujeto a} & Ax = b \\ & x \geq 0 \\ & x \text{ entero,} \end{array}$$

en donde la matriz A y los vectores b y c' tienen entradas enteras. El **problema de programación lineal entera (PLE)** consiste en determinar si existe un vector x con entradas enteras tal que $Ax = b$ y $x \geq 0$. Es decir, dado un problema de optimización en programación lineal entera, queremos ver si dicho problema es factible. Este problema y la demostración de que es $\mathcal{NP}$ – *completo* fueron dados por Papadimitriou y Steiglitz en [14].

Proposición 4.4. *PLE es $\mathcal{NP}$ – completo.*

Demostración. PLE está dentro de $\mathcal{NP}$, ya que podemos utilizar como certificado un punto factible. Para demostrar que es $\mathcal{NP}$ – *completo* vamos a dar una transformación polinomial de SAT a PLE.

Tomamos una función booleana $f(x_1, \ldots, x_n)$ en FNC con m cláusulas. Vamos a modificar dichas cláusulas para obtener restricciones del problema de PLE. En cada cláusula cambiamos todos los operadores $\vee$ por sumas y las negaciones en literales $\overline{x}_i$ por $(1 - x_i)$. Por ejemplo, si tenemos una cláusula

$$(x_2 \vee \overline{x}_5 \vee \overline{x}_8 \vee x_9)$$

al modificarla obtenemos

$$x_2 + (1 - x_5) + (1 - x_8) + x_9.$$

Sabemos además que, si la cláusula se satisface, al menos una literal en ella debe valer 1. Por lo tanto, al hacer los cambios anteriores, la suma obtenida debe ser mayor o igual a 1. Así, en el ejemplo anterior tenemos

$$x_2 + (1 - x_5) + (1 - x_8) + x_9 \geq 1.$$

De esta forma obtenemos una restricción por cada cláusula de $f(\mathbf{x})$. Agregamos además las restricciones $x_i \leq 1$ y $x_i \geq 0$ para $i = 1, \ldots, n$. Así, despejando las constantes y agregando las holguras necesarias, obtenemos un problema de programación lineal entera.

Falta ver que ésta es una transformación polinomial. El tiempo que toma construir el problema de programación lineal entera es lineal. Si $f(\mathbf{x})$ es satisfacible, existe una asignación de verdad $\mathbf{t}$ tal que $f(\mathbf{t}) = 1$. Esta asignación hace que todas las cláusulas sean verdaderas, por lo que al menos una literal en cada cláusula toma el valor de 1. De esta forma, las restricciones obtenidas de cada cláusula se cumplen. Como los elementos de $\mathbf{t}$ son elementos de $\{0, 1\}$, también se cumplen las restricciónes de $t_i \leq 1$ y $t_i \geq 0$. Por lo tanto, $\mathbf{t}$ representa un punto factible del problema de programación lineal entera.

Por otro lado, si $f(\mathbf{x})$ no es satisfacible, entonces para cualquier asignación al menos una cláusula no se cumple. Esto es, dentro de alguna cláusula, ninguna literal tiene valor de verdadero. Por lo tanto, la restricción obtenida a partir de dicha cláusula será 0, que es menor que 1. Esto nos indica que, para cualquier punto, siempre hay al menos una restricción que no se cumple, por lo que el problema de programación lineal entera es infactible. □

3-SAT

El problema 3-SAT es un caso particular del problema SAT. Este problema busca ver si dada una expresión booleana en FNC con exactamente tres literales en cada cláusula, ésta es satisfacible. Es decir, las instancias de 3-SAT son expresiones booleanas de la forma:

$$f(\mathbf{x}) = E_1 \wedge E_2 \wedge \cdots \wedge E_n$$

donde cada E_i tiene exactamente tres variables en disjunción. Al tener una estructura más restringida que SAT, es más fácil transformarlo a otros problemas. Es en este hecho en el cual radica la importancia de 3-SAT. La demostración que se presenta para ver que este problema es $\mathcal{NP}$ – *completo* fue dada por Moret en [13].

Teorema 4.5. *El problema 3-SAT es $\mathcal{NP}$ – completo.*

Demostración. Claramente 3-SAT se encuentra en $\mathcal{NP}$, ya que es un caso particular de SAT.

Sea $f(\mathbf{x}) = E_1 \wedge E_2 \wedge \cdots \wedge E_n$ una expresión booleana FNC. Vamos a sustituir cada una de las cláusulas E_i, con $i = 1, \ldots, n$, por E_i', en donde E_i' está formado por una conjunción de cláusulas de tres literales cada una. De esta forma obtendremos una función $f'(\mathbf{x}) = E_1' \wedge E_2' \wedge \cdots \wedge E_n'$ que tendrá tres literales en cada cláusula.

Para cada cláusula E_i se pueden tener cuatro situaciones distintas. Estas situaciones son: E_i tiene una literal, E_i tiene dos literales, E_i tiene tres literales y E_i tiene más de tres literales.

Si E_i tiene una literal, es decir $E_i = (x)$, vamos a introducir dos nuevas variables $z_{i,1}$ y $z_{i,2}$, y con ellas formaremos cuatro nuevas cláusulas. Así, E_i' queda definido de la siguiente manera:

$$E_i' = (x \vee z_{i,1} \vee z_{i,2}) \wedge (x \vee \overline{z}_{i,1} \vee z_{i,2}) \wedge (x \vee z_{i,1} \vee \overline{z}_{i,2}) \wedge (x \vee \overline{z}_{i,1} \vee \overline{z}_{i,2}).$$

De la misma forma, si E_i tiene dos literales, es decir $E_i = (x_1 \vee x_2)$, introducimos una nueva variable $z_{i,1}$ y dos nuevas cláusulas:

$$E_i' = (x_1 \vee x_2 \vee z_{i,1}) \wedge (x_1 \vee x_2 \vee \overline{z}_{i,1}).$$

Si E_i tiene exactamente tres literales, como la cláusula ya es de la forma que queremos, la dejamos como está. De esta forma tenemos que $E'_i = E_i$.

Si E_i tiene más de tres literales, debemos de separar a las literales y agregar nuevas variables de manera que cada cláusula tenga exactamente tres literales. Supongamos que $E_i = (x_1 \vee x_2 \vee \cdots \vee x_k)$, con $k > 3$. Vamos a separar las literales de E_i en $k-2$ cláusulas utilizando $k-3$ nuevas variables, llamadas $z_{i,j}$, con $j = 1, \ldots, k-3$. De esta forma, E_i se separa en:

$$E'_i = (x_1 \vee x_2 \vee z_{i,1}) \wedge (\overline{z}_{i,1} \vee x_3 \vee z_{i,2}) \wedge (\overline{z}_{i,2} \vee x_4 \vee z_{i,3}) \wedge \cdots \wedge (\overline{z}_{i,k-3} \vee x_{k-1} \vee x_k)$$

Necesitamos ahora demostrar que si $f(\mathbf{x})$ es satisfacible, entonces $f'(\mathbf{x})$ también lo es. Lo mismo debe ocurrir si es no es satisfacible.

Suponemos primero que $f(\mathbf{x})$ es satisfacible. Esto nos dice que existe una asignación de valores $\mathbf{t}$ a las variables de $\mathbf{x}$ tales que $f(\mathbf{t}) = 1$. Es decir, la asignación $\mathbf{t}$ hace que todas las E_i, con $i = 1, \ldots, n$, tengan un valor de 1.

Queremos ver como se comporta E'_i, para $i = 1, \ldots, n$, dada la asignación $\mathbf{t}$ a las literales que no fueron agregadas. En las cláusulas que entran en los primeros dos casos, esta asignación hace que todas las cláusulas que conforman E'_i sean verdaderas, independientemente de los valores que tomen las nuevas variables. En el tercer caso, como $E'_i = E_i$, claramente la asignación dada hará que E'_i tome el valor 1.

Si E_i se encuentra en el cuarto caso, sean $x_{i,1}, \ldots, x_{i,k}$ las literales dentro de E_i. Como $f(\mathbf{t}) = 1$, sabemos que al menos alguna de las literales de E_i tiene valor 1 bajo la asignación $\mathbf{t}$. Sea j el primer índice tal que $x_{i,j} = 1$. Ahora debemos encontrar valores para las variables agregadas. Vamos a hacer

$$z_{i,l} = \begin{cases} 1 & \text{si} \quad 0 \leq l < j-1 \\ 0 & \text{si} \quad j-1 \leq l \leq k \end{cases}$$

Así, la expresión:

$$(x_1 \vee x_2 \vee z_{i,1}) \wedge (\overline{z}_{i,1} \vee x_3 \vee z_{i,2}) \wedge \cdots \wedge (\overline{z}_{i,j-2} \vee x_j \vee z_{i,j-1}) \wedge \cdots \wedge (\overline{z}_{i,k-3} \vee x_{k-1} \vee x_k)$$

dada la asignación $\mathbf{t}$ a los valores de $\mathbf{x}$ con los valores dados para las nuevas variables queda como

$$(0 \vee 0 \vee 1) \wedge (0 \vee 0 \vee 1) \wedge \cdots \wedge (0 \vee 1 \vee 0) \wedge \cdots \wedge (1 \vee x_{k-1} \vee x_k).$$

De esta forma vemos que E'_i es satisfacible.

Por lo tanto $f'(\mathbf{x})$ es satisfacible.

Ahora, suponemos que $f(\mathbf{x})$ no es satisfacible. Sea $\mathbf{t}$ una asignación cualquiera para los valores de $\mathbf{x}$. Como $f(\mathbf{x})$ no es satisfacible, sabemos que $f(\mathbf{t}) = 0$. Esto nos dice que al menos una cláusula E_i no se satisface con los valores dados por $\mathbf{t}$.

Si la cláusula E_i tiene una sola literal, entonces sabemos que el valor de dicha literal es cero. De esta forma, sustituir el valor de cero en E'_i obtenemos

$$E'_i = (0 \vee z_{i,1} \vee z_{i,2}) \wedge (0 \vee \overline{z}_{i,1} \vee z_{i,2}) \wedge (0 \vee z_{i,1} \vee \overline{z}_{i,2}) \wedge (0 \vee \overline{z}_{i,1} \vee \overline{z}_{i,2})$$

$$E'_i = (z_{i,1} \vee z_{i,2}) \wedge (\overline{z}_{i,1} \vee z_{i,2}) \wedge (z_{i,1} \vee \overline{z}_{i,2}) \wedge (\overline{z}_{i,1} \vee \overline{z}_{i,2})$$

lo cual es una contradicción. Por lo tanto, para cualquier valor que tomen las variables extras, $E'_i = 0$. Lo mismo ocurre si E_i tiene dos literales. Además, si E_i tiene exactamente tres literales, $E'_i = E_i = 0$.

Si E_i tiene más de tres literales, entonces sabemos que el valor de todas ellas debe ser cero. Por lo tanto, al sustituir estos valores en E'_i obtenemos

$$(z_{i,1}) \wedge (\overline{z}_{i,1} \vee z_{i,2}) \wedge (\overline{z}_{i,2} \vee z_{i,3}) \wedge \cdots \wedge (\overline{z}_{i,k-4} \vee z_{i,k-3})(\overline{z}_{i,k-3})$$

Si suponemos que esta expresión es satisfacible, entonces $z_{i,1} = \overline{z}_{i,k-3} = 1$. Por otro lado, todas las cláusulas excepto la última nos llevan a la siguiente cadena de implicaciones lógicas:

$$z_{i,1} = 1 \Rightarrow z_{i,2} = 1 \Rightarrow \cdots \Rightarrow z_{i,k-3} = 1$$

Por lo tanto, tenemos que $z_{i,k-3} = 1 = \overline{z}_{i,k-3}$, lo cual es una contradicción. Por lo tanto, $E'_i = 0$.

De esta forma, vemos que si tenemos una no-instancia de SAT, su transformación nos da una no-instancia de 3-SAT.

Tanto el número de cláusulas como el de variables que se agregan quedan acotados por el número de cláusulas originales y el máximo número de literales que presenten éstas. Es por esto que la construcción de la nueva función $f'(\mathbf{x})$ se da en un tiempo polinomial.

Así, acabamos de dar una transformación polinomial de SAT a 3-SAT. Por lo tanto, 3-SAT es $\mathcal{NP}$ – *completo*. □

4.3. Cubiertas, conjuntos independientes y clanes

Los problemas de la **cubierta**, el **conjunto independiente** y el **clan** consisten en, dada una gráfica G y un entero positivo $k \leq |V(G)|$, determinar si G tiene una cubierta, un conjunto independiente o un clan de tamaño k respectivamente. Por la proposición A.3 dada en el apéndice A, estos tres problemas son realmente tres maneras de ver un mismo problema, y la transformación de uno a otro es prácticamente inmediata. Por lo tanto, si demostramos que uno de ellos es $\mathcal{NP} - completo$, sabemos también que los otros dos lo son. Demostraremos entonces que el problema de la cubierta es $\mathcal{NP} - completo$. La demostración siguiente se obtuvo a partir de aquellas dadas en [10], [12] y [9].

Proposición 4.6. *El problema de la cubierta es $\mathcal{NP} - completo$.*

Demostración. El problema de la cubierta está en $\mathcal{NP}$, ya que podemos utilizar una cubierta del tamaño adecuado como certificado. Para demostrar que el problema es $\mathcal{NP} - completo$ vamos a reducir 3-SAT a él.

Sea $f(x_1, \ldots, x_n)$ una instancia de 3-SAT con m cláusulas. Queremos construir una gráfica $G = (V, E)$ con $|V| = 3m$ tal que $f(x_1, \ldots, x_n)$ sea satisfacible si y sólo si G tiene una cubierta de tamaño $2m$.

El conjunto de vértices V queda definido como:

$$V = \{v_{i,j} \text{ tales que } i = 1, \ldots, m \text{ y } j = 1, 2, 3\}.$$

Los vértices de V representan respectivamente a la j-ésima variable de la i-ésima cláusula. Por ejemplo, si x_3 es la segunda variable de la primera cláusula, entonces $v_{1,2}$ representa a esa x_3. Si x_3 se encuentra en alguna otra cláusula, el vértice correspondiente también representará a x_3.

El conjunto de aristas E queda definido como:

$$E = \{[v_{i,j}, v_{i',j'}] \text{ tal que } i = i' \text{ o } v_{i,j} = x_k \text{ y } v_{i',j'} = \overline{x}_k \text{ para alguna } k \leq n\}.$$

De esta forma, todos los vértices que representan a las variables de una misma cláusula son adyacentes. Además, una literal está unida con todas sus negaciones.

Falta ver que es una transformación polinomial. Sea $f(x_1, \ldots, x_n)$ una si-instancia de 3-SAT, y sea t una asignación que la satisface. Esto nos dice que

al menos una literal en cada cláusula tiene valor de 1. Entonces escogemos una literal con valor de 1 de cada cláusula y construimos con los vértices correspondientes el conjunto $I \subset V$. Construimos entonces $C = V \backslash I$; queremos demostrar que C es una cubierta de G. Como $|V| = 3m$ y $|I| = m$, entonces $|C| = 2m$. Cada arista de E incide en al menos un vértice de C. Las aristas de la forma $[v_{i,j}, v_{i,j'}]$ inciden en al menos un vértice de C, ya que, al formar parte de una misma cláusula, sólo se retira uno de estos vértices. Para las aristas cuyos extremos son de la forma $v_{i,j} = x_k$ y $v_{i',j'} = \overline{x}_k$, sabemos que si $x_k = 1$, entonces $\overline{x}_k = 0$, por lo que necesariamente $\overline{x}_k$ se encuentra en C y viceversa. Por lo tanto, C es una cubierta de G.

Por otro lado, supongamos que tenemos una cubierta C de tamaño $2m$ para la gráfica G. Esta cubierta contiene exactamente dos vértices por cada cláusula. Si para alguna cláusula se tuviera representada únicamente una literal dentro de C, entonces habría una arista, la que une a los vértices que representan a las otras dos literales de la cláusula. que no incide en ningún vértice de la cubierta, lo cual es una contradicción. Como tenemos m cláusulas, todas con al menos dos vértices en C y $|C| = 2m$, ninguna cláusula tiene representadas a sus tres literales en la cubierta. Además, una literal y sus negaciones deben estar representadas dentro de C. Esto es porque, si ninguna de las dos estuviera, la arista que las une no incidiría en algún vértice de C.

Damos entonces el valor 1 a las variables que representan a los vértices que no están en C, el resto de las variables pueden tomar cualquier valor. De esta manera, al menos una variable en cada cláusula tiene el valor de 1 y no hay contradicciones. Por lo tanto, tenemos una si-instancia de SAT.

La construcción de la gráfica G es de orden cuadrático, y por lo tanto polinomial. Esto nos lleva a que el problema de la cubierta es $\mathcal{NP}-completo$.

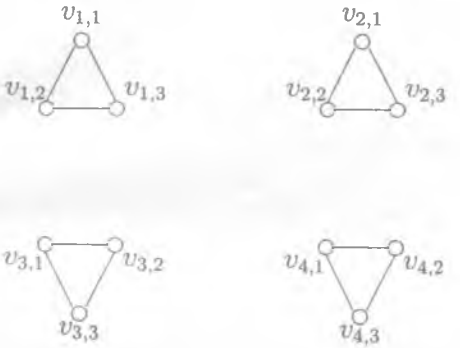
□

Ejemplo 4.7. Para mostrar más claramente cómo se construye la gráfica a partir de la expresión booleana se presenta el siguiente ejemplo obtenido de [9]. Si tenemos la función booleana

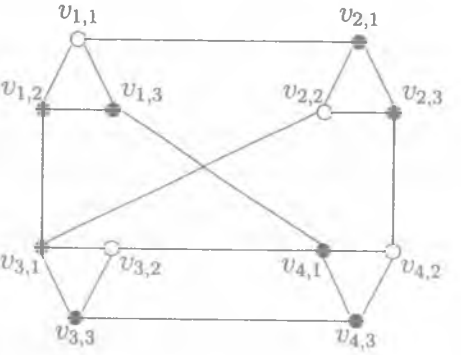
$$f(\mathbf{x}) = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_4) \wedge (\overline{x}_2 \vee x_3 \vee x_5) \wedge (\overline{x}_3 \vee \overline{x}_4 \vee \overline{x}_5)$$

como instancia de 3-SAT, entonces vamos a obtener una gráfica con 12 vértices, tres por cada cláusula de $f(\mathbf{x})$. Primero colocamos las aristas que

unen a los vértices de una misma cláusula entre sí.



Después vamos a agregar las aristas que conectan a las literales con sus negaciones. De esta forma obtenemos la gráfica G que buscamos. Además, como $f(1,1,1,0,0) = 1$, entonces la gráfica G presenta una cubierta de tamaño 8. Los vértices oscuros forman una cubierta para G.



Las transformaciones para los problemas del conjunto independiente y el clan son casi inmediatas. Para el problema del conjunto independiente, tomamos una instancia del problema de la cubierta y simplemente cambiamos k por $|V| - k$. Para el problema del clan, transformamos el problema del conjunto independiente y cambiamos la gráfica por su complemento. Si quisiéramos transformar estos dos problemas a partir de 3-SAT, la demostración sería análoga en ambos casos. En el caso del conjunto independiente, tomamos el conjunto I dado en la demostración. En el caso del problema del clan, simplemente tomamos las aristas no consideradas en la demostración.

Para estos problemas podemos encontrar diferentes aplicaciones prácticas. El siguiente ejemplo fue dado por Hopcroft, et al en [9]. El problema del conjunto independiente podría utilizarse para determinar las fechas de los exámenes finales en una universidad. Los vértices representarían a las materias, y dos vértices estarían conectados por una arista si al menos un estudiante cursa la dos materias. Al encontrar un conjunto independiente para esta gráfica, podemos poner el examen para todas las materias del conjunto el mismo día, ya que sabemos que ningún estudiante cursa más de una materia del conjunto.

4.4. Ciclos hamiltonianos y el problema del agente viajero

Como se mencionó anteriormente, el problema del **ciclo hamiltoniano** consiste en ver si, dada una gráfica ésta tiene un ciclo hamiltoniano. Este problema está muy relacionado con el problema del agente viajero, que es un problema muy importante en optimización combinatoria. Ambos problemas son $\mathcal{NP} - completos$.

Antes de demostrar que el problema del ciclo hamiltoniano es $\mathcal{NP} - completo$, es más fácil demostrar que el problema del **ciclo hamiltoniano dirigido** lo es. Este problema pregunta si una gráfica dirigida tiene un ciclo hamiltoniano. La demostración que se presenta fue dada por Hopcroft, et al, en [9].

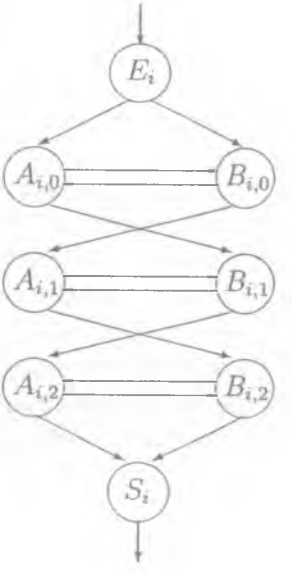
Teorema 4.8. *El problema del ciclo hamiltoniano dirigido es $\mathcal{NP}-completo$.*

Demostración. Es fácil ver que el problema del ciclo hamiltoniano dirigido está en $\mathcal{NP}$. Podemos utilizar un ciclo hamiltoniano como certificado y comprobar en tiempo polinomial que lo es. Para demostrar que es $\mathcal{NP}-completo$, vamos dar una trasformación polinomial desde 3-SAT.

Sea $f(x_1, \ldots, x_n) = C_1 \wedge \cdots \wedge C_m$ una función boolena en 3-SAT con n variables y m cláusulas. Para cada variable, vamos a construir una componente, o bien, una subgráfica que formará parte de la gráfica que buscamos. Al unir las componentes correspondientes a todas las variables, vamos a obtener D', una subgráfica hamiltoniana dirigida de la gráfica D que buscamos.

Para cada variable x_i, la componente que va a representarla va a contar con un vértice de entrada E_i y un vértice de salida S_i. Sea n_i el número de cláusulas en las que aparece x_i. Vamos a incluir en la componente los vértices $A_{i,k}$ y $B_{i,k}$, donde $k = 0, \ldots, n_i$. De esta manera, la componente que representa a x_i va a tener $2n_i + 4$ vértices.

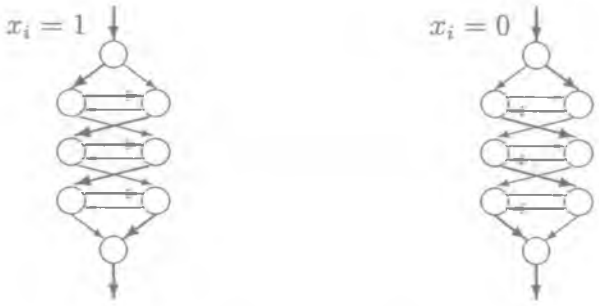
Los arcos que unen a los vértices de la componente son los siguientes. Del vértice E_i salen los arcos $[E_i, A_{i,0}]$ y $[E_i, B_{i,0}]$, mientras que el vértice S_i recibe a los arcos $[A_{i,n_i}, S_i]$ y $[B_{i,n_i}, S_i]$. Estos arcos conectan a los vértices de entrada y salida con el resto de los vértices de la componente. Contamos además con los arcos $[A_{i,k}, B_{i,k+1}]$ y $[B_{i,k}, A_{i,k+1}]$ para $k = 0, \ldots, n_i - 1$. Por último, tenemos los arcos $[A_{i,k}, B_{i,k}]$ y $[B_{i,k}, A_{i,k}]$ para $k = 0, \ldots, n_i$. Por ejemplo, si la variable x_i aparece en dos componentes distintas, entonces la componente que la representa es la siguiente:



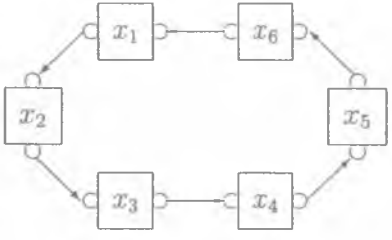
La única manera de entrar a cada componente es por el vértice de entrada, así como la única forma de salir es por el de salida. Una trayectoria hamiltoniana que entra en la componente puede seguir dos caminos distintos, ir al vértice $A_{i,0}$ o al vértice $B_{i,0}$. Una vez que se elige alguno de estos dos vértices, el camino que sigue el ciclo dentro de la componente queda completamente determinado hasta el vértice de salida. Si, por ejemplo, se elige el arco $[E_i, A_{i,0}]$, entonces el siguiente vértice que se visita es necesariamente

$B_{i,0}$, ya que no hay otra forma de llegar a él posteriormente. Los siguientes vértices que se visitan son $A_{i,1}$, $B_{i,1}$, $A_{i,2}$ y así sucesivamente hasta llegar a S_i. El procedimiento es análogo si comenzamos con $B_{i,0}$.

Podemos determinar la trayectoria dentro de la componente de acuerdo a los valores que toma la variable x_i. De esta forma, si $x_i = 1$ entonces el primer vértice que se visita es $A_{i,0}$, mientras que si $x_i = 0$, la trayectoria comenzará por el vértice $B_{i,0}$. Estas dos situaciones se presentan en los siguientes esquemas. En ellos, los arcos más oscuros indican la trayectoria que se sigue en cada caso.



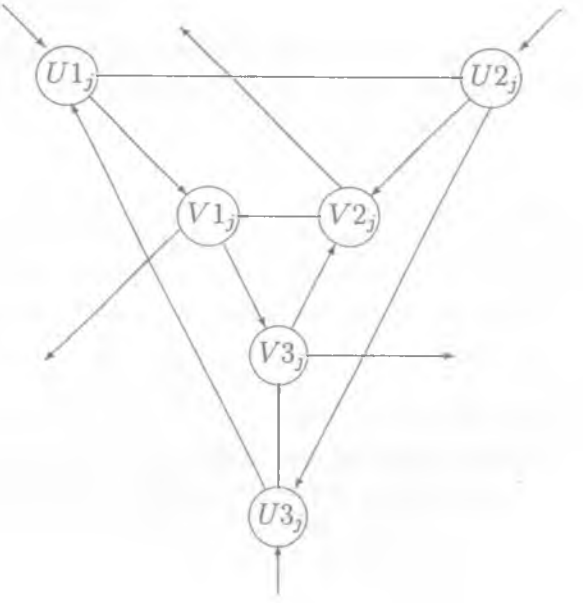
Una vez que construimos las componentes de todas las variables, debemos unirlas en una misma gráfica. Para hacer esto, conectamos el vértice de salida de una componente con el vértice de entrada de la siguiente variable. Es decir, en la gráfica completa se encuentran los arcos $[S_i, E_{i+1}]$ para $i = 1, \ldots, n-1$ y el arco $[S_n, E_1]$, que une a la componente de la última variable con la primera. De esta forma obtenemos la gráfica D'. En el siguiente esquema se representa la gráfica D' de una función con seis variables. Cada componente está representada por un cuadrado, y todas están unidas entre si por medio de sus vértices de entrada y salida.



En cada componente encontramos una trayectoria hamiltoniana del vértice de entrada al vértice de salida. Es por esto que, al unir las componentes de

la manera indicada obtenemos una gráfica hamiltoniana. Como cada componente presenta dos trayectorias distintas, la gráfica D' tiene 2^n ciclos hamiltonianos. Todas las funciones que tengan el mismo número de variables tendrán la misma gráfica D'. Es por esto que debemos agregar mas componentes que representen las cláusulas de la función que queremos transformar. Al unir estas componentes obtendremos la gráfica D que buscamos.

Las componentes que representan a las cláusulas van a tener seis vértices cada una. Llamaremos a estos vértices $U1_j$, $U2_j$, $U3_j$, $V1_j$, $V2_j$ y $V3_j$. Los vértices Uk_j, con $k = 1, 2, 3$, son los vértices por los que se puede entrar a la componente, mientras que los vértices Vk_j son por los que se puede salir. Estos vértices están conectados de la siguiente forma:



Es importante notar que si un ciclo hamiltoniano entra a una componente por Uk_j, necesariamente tendrá que salir por Vk_j. Sin pérdida de generalidad, supongamos que entra por $U1_j$, y queremos ver que siempre sale por $V1_j$. Podemos tener tres casos diferentes.

En el primer caso, el ciclo recorre los seis vértices de la componente al entrar por $U1_j$. Esto ocurre si los siguientes dos vértices en la trayectoria son $U2_j$ y $U3_j$. El siguiente vértice en la trayectoria es necesariamente $V3_j$, ya que el arco $[U3_j, V3_j]$ es el único que podemos tomar. Si la trayectoria saliera

de la componente en $V3_j$, entonces no habría forma de alcanzar los vértices $V2_j$ y $V1_j$. Es por esto que después pasa a $V2_j$. Análogamente se muestra que no puede salir de la componente por $V2_j$, por lo que debe salir por $V1_j$.

En el segundo caso, el ciclo recorre cuatro de los vértices de la componente al entrar por $U1_j$, mientras que en otro momento recorre los dos vértices restantes. Supongamos que el ciclo pasa de $U1_j$ a $U2_2$, que a su vez pasa a $V2_j$. Si el ciclo saliera en $V2_j$, no habría manera de que después se alcanzara $V1_j$, por lo que debe pasar a $V1_j$. En este momento, la única manera de llegar a $U3_j$ es por fuera de la componente. Esto forza a que cuando el ciclo pase por $U3_j$, pase inmediatamente a $V3_j$ para después salir de la componente. Es por esto que de $V1_j$ no podemos pasar a $V3_j$. Por lo tanto, el ciclo debe salir por la componente $V1_j$.

En el último caso, el ciclo entra y sale de la componente en tres ocasiones distitntas. En cada una de estas ocasiones entra por algún vértice Uk_j, pasa al vértice Vk_j y sale de la componente inmediatamente.

Necesitamos ahora conectar a estas componentes con la gráfica G'. Si la primera literal de cláusula C_j es la literal x_i sin negar, entonces conectamos las componentes correspondientes de la siguiente manera. Tomamos $B_{i,k}$ en la componente de x_i, en donde k es el primer índice que no haya sido utilizado en alguna otra cláusula. Vamos a conectar ambas componentes por medio de los arcos $[B_{i,k}, U1_j]$ y $[V1_j, A_{i,k}]$. En cambio, si la primera literal de la cláusula es $\overline{x_i}$, entonces los arcos que agregamos son $[A_{i,k}, U1_j]$ y $[V1_j, B_{i,k+1}]$. De nuevo, k es el primer índice tal que que $A_{i,k}$ no se haya utilizado para conectar alguna otra componente. Hacemos lo mismo con las demás literales de la cláusula, sólo que se utilizarán los vértices $U2_j$ y $V2_j$ para la segunda literal y $U3_j$ y $V3_j$ para la tercera. De esta forma se obtiene la gráfica D. Falta ver que la construcción de D es una transformación polinomial.

Sea $f(\mathbf{x})$ una si-instancia de 3-SAT. Esto implica que existe una asignación $\mathbf{t}$ tal que $f(\mathbf{t}) = 1$. Construimos la gráfica D correspondiende a partir de $f(\mathbf{x})$. Después, tomamos la asignación $\mathbf{t}$ para dar el ciclo hamiltoniano correspondiente a D' de la manera que se mostró anteriormente.

Necesitamos extender este ciclo a toda la gráfica D. Como $f(\mathbf{t}) = 1$, sabemos que en cada cláusula al menos una de las literales toma el valor 1. Sea x_i la primera literal en la cláusula C_j con el valor 1. La componente de la cláusula está conectado con la componente de dicha literal de la manera indicada por medio de los vértices $B_{i,k}$ y $A_{i,k+1}$ para alguna k. Por medio de

los arcos que conectan a estos vértices con la componente, podemos introducir el ciclo a la componente de la cláusula pasando por todos sus vértices. Por otro lado, como $x_i = 1$, sabemos que el arco $[B_{i,k}, A_{i,k+1}]$ forma parte del ciclo dado para D'. De esta forma podemos sustituir este arco por el ciclo que inculye a todos los vértices de la componente de la cláusula. El razonamiento es análogo si $x_i = 0$.

Hacemos esto únicamente con la primera literal con valor de verdadero en cada cláusula, ya que con ella se cubren todos los vértices necesarios. Como en todas las cláusulas alguna literal toma el valor de 1 bajo la asignación $\mathbf{t}$, entonces el ciclo pasa por todos los vértices de todas las componentes. Por lo tanto, hemos dado un ciclo hamiltoniano para D una gráfica dirigida.

Ahora, supongamos que D construida de esta manera es hamiltoniana. Sabemos que cuando el ciclo entra a alguna componente por cierto vértice, debe salir necesariamente por el vértice correspondiente. Además. las componentes de las cláusulas se conectan con las de las variables por medio de estas parejas de entrada y salida. De esta forma, la parte del ciclo que pasa por las componentes de las cláusulas sustituye de alguna forma a los arcos que conectan a los vértices de los componentes de las variables.

Podemos considerar entonces a la gráfica D' y dar el mismo ciclo hamiltoniano para la gráfica D, sustituyendo los arcos que conectan a las variables con las cláusulas. Este ciclo de D' da una asignación $\mathbf{t}$ para $f(\mathrm{x})$. Las desviaciones hacia las componentes de las cláusulas sólo pueden ocurrir cuando las literales en las que se encuentran son verdaderas y pertenecen a la cláusula. Como el ciclo hamiltoniano pasa por las componentes de todas las cláusulas, sabemos que al menos una literal es verdadera para cada cláusula. Por lo tanto, $f(\mathbf{t}) = 1$.

Hemos dado una transformación polinomial de 3-SAT a el problema del circuito hamiltoniano dirigido. Por lo tanto, este problema es $\mathcal{NP}-completo$.

□

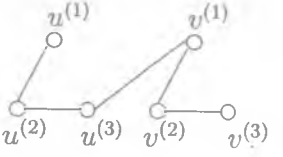
Teorema 4.9. *El problema del ciclo hamiltoniano es $\mathcal{NP}$ – completo.*

Demostración. Por el ejemplo 3.6, sabemos que este problema está dentro de $\mathcal{NP}$. Para demostrar que es $\mathcal{NP}-completo$, vamos a reducir el problema del ciclo hamiltoniano orientado a él.

Sea D una gráfica dirigida. Vamos a transformar esta gráfica a una gráfica G no dirigida que contenga un ciclo hamiltoniano. Para cada vértice v de D,

vamos a sustituirlo con tres vértices $v^{(1)}$, $v^{(2)}$ y $v^{(3)}$. Estos vértices van a estar unidos por las aristas $[v^{(1)}, v^{(2)}]$ y $[v^{(2)}, v^{(3)}]$. Por otro lado, si tenemos un arco en D que va de un vértice v a otro w, entonces en G vamos a tener la arista $[v^{(3)}, w^{(1)}]$. De esta forma, si tenemos los vértices u y v unidos por el arco $[u, v]$

entonces al construir la gráfica G los sustituiremos por



Falta ahora demostrar que esta es una transformación polinomial. Supongamos que D es una gráfica hamiltoniana dirigida. Esto implica que existe un ciclo hamiltoniano dentro de D. Supongamos que $v_1, v_2, \ldots, v_n, v_1$ es un ciclo hamiltoniano para D. Entonces

$$v_1^{(1)}, v_1^{(2)}, v_1^{(3)}, v_2^{(1)}, v_2^{(2)}, v_2^{(3)}, \ldots, v_n^{(1)}, v_n^{(2)}, v_n^{(3)}, v_1^{(1)}$$

será un ciclo hamiltoniano para G.

Por otro lado, supongamos que G es una gráfica hamiltoniana. Cada vértice $v_i^{(2)}$ tienen grado dos, es decir, únicamente hay dos aristas que inciden en él. Por lo tanto, en el ciclo hamiltoniano de G podemos tener la secuencia $v_i^{(1)}, v_i^{(2)}, v_i^{(3)}$ o bien, la secuencia $v_i^{(3)}, v_i^{(2)}, v_i^{(1)}$. Una vez que determinamos alguna de estas secuencias para algún vértice $v_i^{(2)}$, sabemos que todos los vértices van a seguir la misma secuencia. Es decir, si nos fijamos únicamente en los superíndices de los vértices del ciclo hamiltoniano, éstos serán de la forma $\ldots, 1, 2, 3, 1, 2, 3, \ldots$ o bien $\ldots, 3, 2, 1, 3, 2, 1, \ldots$. Estas dos secuencias corresponden a los dos sentidos en los que se puede recorrer el ciclo hamiltoniano. Por la forma en la que fue construida G, si tomamos la secuencia $\ldots, 1, 2, 3, 1, 2, 3, \ldots$ ésta nos va a dar un ciclo hamiltoniano en D.

La construcción de la gráfica G a partir de D toma un tiempo lineal, por lo que tenemos una transformación polinomial. Por lo tanto, el problema del ciclo hamiltoniano es un problema $\mathcal{NP}$ – *completo*. □

Supongamos que una persona quiere recorrer n ciudades y después regresar al lugar de donde partió. Existen diferentes caminos que conectan a

las ciudades, cada uno con una distancia entera diferente. El **problema del agente viajero** consiste en encontrar la manera más eficiente de recorrer las n ciudades y regresar al punto de origen.

Si tomamos las ciudades como vértices y los caminos como aristas, podemos ver este problema como una gráfica con una función de costos en las aristas. El problema consiste entonces en encontrar un ciclo hamiltoniano de costo mínimo. Para plantearlo como un problema de decisión, damos una cota k y preguntamos si existe un ciclo hamiltoniano tal que la suma de los costos de las aristas del ciclo sea menor a k, donde k es un entero. Este problema también es un problema $\mathcal{NP}$ – *completo.*

Teorema 4.10. *El problema del agente viajero es $\mathcal{NP}$ – completo*

Demostración. Este problema se encuentra en $\mathcal{NP}$ ya que, si damos un ciclo como certificado podemos comprobar que pasa por todos los vértices y que su costo es menor a k en un tiempo polinomial. Por otro lado, el problema del ciclo hamiltoniano puede verse como un caso particular del problema del agente viajero en donde todas las aristas tienen costo 1, y $n \leq k$. De esta forma, si una gráfica tiene un ciclo hamiltoniano, entonces el problema del agente viajero correspondiente tendrá un ciclo de costo menor o igual a k y viceversa. □

Capítulo 5

Otros problemas $\mathcal{NP} - completos$

5.1. Técnicas para demostrar que un problema es $\mathcal{NP} - completo$

En general, las demostraciones para cada problema parten de un problema $\mathcal{NP} - completo$ y transforman este problema al que queremos demostrar. Sin embargo, las técnicas para encontrar estas transformaciones pueden variar mucho en cada problema. Existen tres tipos generales de pruebas que son comúnmente utilizadas: restricción, sustitución local y diseño de componentes. Estas técnicas no son las únicas que existen y no clasifican a las pruebas de problemas $\mathcal{NP} - completos$. Tampoco dan una manera fija de demostrar que un problema es $\mathcal{NP} - completo$, por lo que no se pueden definir explícitamente. Con ellas simplemente se muestran algunas formas que comúnmente funcionan para demostrar problemas. Para explicar estas técnicas, supongamos que queremos demostrar que un problema Π es $\mathcal{NP} - completo$ a partir de un problema Π'.

La técnica de **restricción** busca mostrar que el problema Π' es un caso particular del problema Π. Esta técnica es la más sencilla y la más utilizada de las tres. Para poder utilizarla, necesitamos restringir el problema Π de manera que podamos encontrar una correspondencia uno a uno con el problema Π'. Así, la transformación polinomial de Π' al problema restringido Π es fácil de encontrar. Un ejemplo de este tipo de pruebas es la del problema del agente viajero. En este caso, se muestra como el problema del ciclo hamiltoniano es

un caso particular de dicho problema.

La **sustitución local** es la segunda técnica más sencilla. aunque requiere muchas veces de una prueba más formal. En este caso, se toma una característica del problema Π' y la remplazamos por alguna otra estructura del problema Π. Un ejemplo de esta técnica es la demostración de 3-SAT. Aquí, cada cláusula de SAT fue sustituida por un conjunto de cláusulas de 3-SAT siguiendo una regla general. Es importante notar que la sustitución de cada cláusula es independiente del resto de las cláusulas. Es decir, las modificaciones se dan únicamente a nivel local.

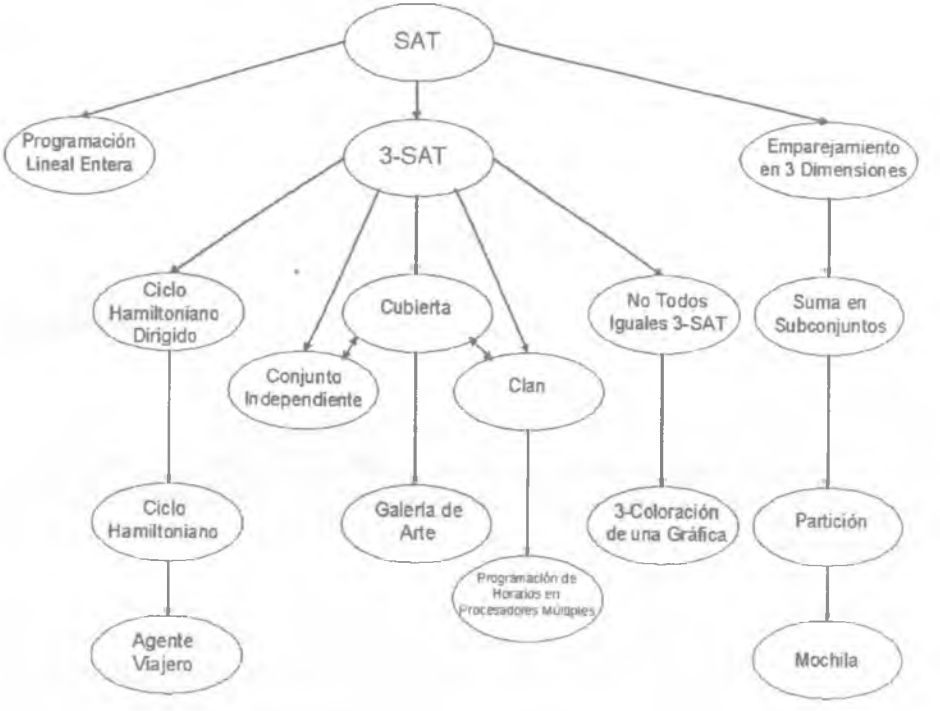
Por último, tenemos la técnica de **diseño de componentes**. Ésta es la técnica más complicada de las tres. En esta prueba se utilizan las características que busca el problema Π, se generan componentes que tengan estas características y se relacionan de manera que puedan representar instancias del problema Π'. Las componentes pueden describir situaciones o determinar las propiedades de Π'. Las diferentes componentes se encuentran conectadas de diferentes formas, y no son ajenas unas a otras. Un ejemplo de este tipo de pruebas es el ciclo hamiltoniano dirigido. En esta prueba, se generan componentes que representan a las variables y a las cláusulas. Además. estas componentes se unen de manera que puedan representar una función booleana de 3-SAT.

5.2. Otros problemas $\mathcal{NP}$ – *completos*

En esta sección se presentan otros problemas $\mathcal{NP}$ – *completos*. Aunque no se dan pruebas formales para estos problemas, en la siguiente página se presenta un esquema que muestra a partir de que otros problemas podemos obtener las demostraciones. En algunos casos se da una idea de como construir la transformación polinomial buscada.

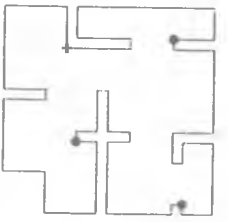
Galería de arte

Un polígono está formado por una secuencia ordenada de puntos $p_1, \ldots, p_n$ con $n \geq 3$, llamados vértices, y por los segmentos de recta que unen p_i con p_{i+1} para $i = 1, \ldots, n-1$ y p_n con p_1, llamados lados. Se dice que un polígono es simple si cualesquiera dos lados no consecutivos no se intersectan. Supongamos que tenemos un polígono simple. Este polígono va a representar una



galería de arte, y los guardias de la misma se van a encontrar localizados en los vértices del polígono. Los guardias pueden ver un punto dentro de la galería si el segmento de recta que une al guardia con el punto dado no intersecta alguno de los lados del polígono. Claramente, necesitamos que todos los puntos de la galería estén vigilados por al menos un guardia y queremos hacer esto con la menor cantidad de guardias posibles. El problema de la **galería de arte** consiste en, dados un polígono simple P con n vértices y una cota entera $B \leq n$, determinar si se pueden situar a lo más B guardias en los vértices de P de manera que todo punto interior del polígono sea visible para al menos un guardia. La demostración de que este problema es $\mathcal{NP}$ – *completo* se da a partir del problema de la cubierta, y puede encontrarse en [13]. Para ilustrar mejor el problema, se presenta una ejemplo obtenido de [21]. La siguiente figura representa una galería de arte que requiere de cuatro guardias para vigilarla. Los guardias se encuentran colocados en los puntos indicados en la

figura.



Programación de horarios en procesadores múltiples

El problema de **programación de horarios en procesadores múltiples** involucra programar los horarios en que se realizarán ciertos trabajos en diferentes máquinas. El tiempo en que se realiza cada trabajo es el mismo para todos ellos. Estos trabajos se hacen en procesadores idénticos y tienen una precedencia en todos los procesadores. Es decir, algunos trabajos sólo se pueden realizar si otros trabajos ya han sido realizados. Contamos además con m máquinas y un tiempo de entrega T. El problema consiste entonces en ver si es posible realizar todos los trabajos antes del tiempo T utilizando m procesadores. La demostración de que este problema es $\mathcal{NP}$ – *completo* se obtiene a partir del problema del clan. Podemos representarlo por medio de una gráfica dirigida, en donde los trabajos a realizarse son los vértices, mientras que los arcos indican la precedencia. Es decir, si tenemos dos trabajos J_1 y J_2, y contamos con el arco $[J_1, J_2]$, entonces necesitamos realizar el trabajo J_1 antes del trabajo J_2. La demostración puede encontrarse en [14].

No-Todos-Iguales 3-SAT

El problema de **no-todos-iguales 3-SAT** (NTI 3-SAT) es un caso particular de 3-SAT. Para este problema, se pide que en cada cláusula las literales no tengan todas el mismo valor. Este caso particular presenta una propiedad muy útil. Si tenemos una asignación que satisface la función booleana entonces su complemento, o bien, la asignación opuesta también satisfacerá la función.

La demostración de que este problema es $\mathcal{NP}$ – *completo* parte de 3-SAT. La idea general de dicha demostración se obtuvo de [13]. Dada una instancia de 3-SAT, se agregan dos nuevas variables por cada variable de la instancia

para representar los valores de verdad y falsedad. Con ellas se hace una nueva función booleana para indicar que no todas pueden ser iguales. Esta función se convierte después a FNC con tres literales por cláusula de la manera en la que se hace en la proposición B.1 y en la demostración del teorema 4.5. Finalmente, se agregan las cláusulas obtenidas a la instancia original.

3-coloración de una gráfica

El prolema de la **3-coloración de una gráfica** consiste, como su nombre lo indica, en determinar si es posible dar una 3-coloración propia para una gráfica dada. Una vez que demostramos que este problema es $\mathcal{NP}-completo$, podemos ver que el problema de la k-coloración de una gráfica es $\mathcal{NP}$ – *completo* para $k \geq 3$. Esto es porque una k-coloración utiliza a lo más k colores, por lo que, si $k \geq 3$ entonces una 3-coloración es una k-coloración.

La idea para esta demostración también se obtuvo de [13]. Podemos reducir el problema NTI 3-SAT a este problema para demostrar que es $\mathcal{NP}$ – *completo*. Por cada variable de la función de NTI 3-SAT vamos a poner dos vértices, uno representando a la variable y el otro a su negación. Estos dos vértices van a estar unidos por una arista. Además, contaremos con un vértice adicional que estará conectado con todos estos vértices. Si queremos dar una 3-coloración propia a la gráfica, este vértice recibirá un color mientras que las variables y sus negaciones recibirán dos colores. Los colores que se utilizan para las variables y las negaciones equivalen a dar una asignación de verdad, ya que los vértices que representan a una variable y a su negación no pueden tener el mismo color. Por otro lado, vamos a incluir en la gráfica un "triángulo" por cada cláusula. Para una cláusula dada, los vértices del triángulo que la representa van a estar unidos con los vértices que representan a las literales dentro de la cláusula. Esta va a ser la gráfica que buscamos. Se puede ver fácilmente que al menos una de las variables de cada cláusula debe ser distinta a las otras para poder dar una 3-coloración de la gráfica.

Emparejamiento en tres dimensiones

Supongamos que contamos con tres conjuntos U, V, W con la misma cardinalidad y con $T \subseteq U \times V \times W$. El problema del **emparejamiento en tres dimensiones** consiste en ver si existe $M \subseteq T$ una manera de agrupar

tercias con $|M| = |U|$ tal que, si tenemos (u, v, w) y (u', v', w') dos elementos de T, entonces $u \neq u'$, $v \neq v'$ y $w \neq w'$. En otras palabras, tenemos T un conjunto de tercias formadas con elementos de U, V y W de manera que cada tercia tiene un elemento de cada uno de estos conjuntos. Queremos ver si podemos encontrar M un subconjunto de T tal que cada elemento de los conjuntos U, V y W se encuentre únicamente en una de las tercias de M. Es decir, queremos ver si a partir de T podemos separar los elementos de los conjuntos en tercias. La demostración de este problema parte de SAT y puede encontarse en [12] y [14].

Suma en subconjuntos

Supongamos que tenemos un conjunto de elementos, cada uno con un peso entero, y un número entero K. Queremos determinar si existe un subconjunto del conjunto original tal que los pesos de los elementos del subconjunto sumen K. A este problema se le conoce como **suma en subconjuntos**. Para demostrar que es $\mathcal{NP}$ – *completo*, transformamos el problema del emparejamiento en tres dimensiones a él. La prueba se encuentra en [12].

Partición

El problema de la **partición** es similar al de suma en subconjuntos. Si tenemos un conjunto C cuyos elementos tienen pesos enteros, entonces queremos encontrar un subconjunto S tal que la suma de los pesos de los elementos de S sea igual a la suma de los pesos de los elementos de $C \backslash S$. En otras palabras, queremos particionar al conjunto C en dos subconjuntos ajenos tales que la suma de los elementos de ambos sea la misma.

Podemos transformar polinomialmente el problema de suma de subconjuntos al de partición. Una instancia del problema de suma de subconjuntos consta de un conjunto C cuyos elementos tienen pesos $c_1, \ldots, c_n$ enteros positivos y un entero positivo K. Agregamos un nuevo elemento x_{n+1} a C tal que su peso sea igual al valor absoluto de la suma de todos los demás elementos de C menos $2K$. Podemos tener dos situaciones distintas, $2K \leq \Sigma_{i=1}^{n} c_i$ o bien $\Sigma_{i=1}^{n} c_i < 2K$. En el primer caso, la suma de los elementos de $I \subseteq C$ es igual a K si y sólo si la suma de los pesos de $I \cup x_{n+1}$ es igual a la suma de los pesos de los elementos de $C \backslash I$. En el segundo caso, los elementos de $I \subseteq C$ tienen un peso K si y sólo si la suma de los pesos de I es igual a la suma

de los pesos de los elementos de $(C \cup x_{n+1}) \backslash I$. En cualquier caso se obtiene una transformación polinomial del problema de suma de subconjuntos al de partición. Esta demostración fue obtenida de [12].

Mochila

Supongamos que tenemos una cantidad de artículos que queremos llevar. Cada uno de los artículos tiene un peso y un valor. Además, tenemos una mochila para transportar los artículos que sólo puede aguantar determinado peso. Queremos ver entonces qué artículos llevar tales que el valor se maximize pero no se exceda la capacidad de la mochila. Visto como un problema de decisión, si tenemos un conjunto U donde cada elemento tiene un peso $s(u)$ y un valor $v(u)$, y contamos también con dos enteros positivios B y K representando la capacidad y el valor objetivo, queremos ver si existe $U' \subseteq U$ tal que $\Sigma_{u \in U'} s(u) \leq B$ y $\Sigma_{u \in U'} v(u) \geq K$. En otras palabras, queremos que los elementos de U' alcancen un valor objetivo K sin exceder la capacidad B. Este es el problema de la **mochila**.

La demostración siguiente se obtuvo de [6]. Podemos ver que el problema de la partición es un caso particular del problema de la mochila, en el cual $s(u) = v(u)$ para toda $u \in U$ y $B = K = \frac{1}{2}\Sigma_{u \in U} s(u)$. De esta forma, podemos obtener la transformación polinomial del problem de partición al problema de la mochila, demostrando así que es $\mathcal{NP}$ – *completo*.

Las maneras que se presentan para demostrar que estos problemas son $\mathcal{NP}$ – *completos* no son únicas. De hecho, teoricamente se podría utilizar cualquier problema $\mathcal{NP}$ – *completo* para dar una transformación polinomial. Por ejemplo, podemos obtener la transformación para el problema de programación lineal entera a partir del problema de partición, o bien podríamos obtener el problema del ciclo Hamiltoniano a partir de 3-SAT como se muesta en [12].

Existen además muchos otros problemas $\mathcal{NP}$ – *completos* importantes. En [6] Garey y Johnson presentan en el apéndice una lista de más de 300 problemas $\mathcal{NP}$–*completos* en diferentes áreas. Podemos encontrar problemas de teoría de gráficas, de secuenciación y programación de horarios, de álgebra y teoría de números y de lógica, entre otros.

5.3. Formas de trabajar con problemas $\mathcal{NP}$ – *completos*

Una vez que sabemos que un problema es $\mathcal{NP}$ – *completo* debemos encontrar una forma adecuada de trabajar con él. Sabemos que no vamos a poder encontrar un algoritmo polinomial que nos de una solución exacta, por lo que debemos buscar otras formas convenientes de atacarlo.

Hay que recordar que el tiempo de ejecución se calcula tomando en cuenta el peor caso posible. Para algunos problemas $\mathcal{NP}$ – *completos* podemos tener un algoritmo que resuelva el problema en tiempo polinomial para la mayoría de las instancias, aunque algunas requieran tiempo exponencial. También podemos tener un problema en el que sólo nos interesen ciertas instancias y éstas puedan resolverse en tiempo polinomial. En estos casos, podemos ignorar el hecho de que el problema sea $\mathcal{NP}$ – *completo* y trabajar con el algoritmo que se tiene.

Existen en general dos enfoques que pueden tomarse cuando se trabaja con problemas $\mathcal{NP}$ – *completos*. En el primer enfoque, se busca avanzar lo más posible dentro de la búsqueda exhaustiva de la solución. Esto se hace dirigiendo la búsqueda de manera inteligente y descartando soluciones que de antemano sabemos que no serán las que buscamos. Las técnicas de corte y cota y programación dinámica son ejemplos de este enfoque. La técnica de corte y cota se puede encontrar en [14], [12] y [?], mientras que la de programación dinámica se puede encontrar en [14].

El segundo enfoque se utiliza para aquellos problemas de optimización que se transforman en problemas de decisión. Para estos problemas lo que se hace es dejar de buscar la solución óptima y conformarnos con una buena solución. Algunas técnicas que podemos encontrar en este enfoque son los algoritmos de aproximación y las búsquedas locales. En muchos de estos casos, sabemos que la solución que encontremos para el problema no diferirá en más de cierto porcentaje de la solución óptima. También tenemos los algoritmos heurísticos dentro de este enfoque. Estos algoritmos no garantizan cercanía a la solución optima, pero pueden llegar a servir en la práctica. Estas técnicas también las podemos encontrar en [14] y en [12].

5.4. $\mathcal{P}$ vs. $\mathcal{NP}$

Constantemente estamos buscando algoritmos más eficientes para resolver problemas. Aún cuando existan varios algoritmos para un problema, se siguen buscando mejores maneras de resolverlo. En general, se considera que un problema está bien resuelto si existe un algoritmo polinomial que lo resuelva. La importancia de estos algoritmos es tal que aquellos problemas que no los admiten son considerados como intratables. Es por esto que nos interesa saber si podemos encontrar estos algoritmos para diferentes problemas.

Sabemos que todos los problemas que admiten algoritmos polinomiales se encuentran en $\mathcal{P}$, por lo que se considera que están bien resueltos. Lo que no sabemos es si todos los problemas en $\mathcal{NP}$ también admiten este tipo de algoritmos. En otras palabras, nos interesa saber si $\mathcal{P} = \mathcal{NP}$. A esta pregunta se le conoce como $\mathcal{P}$ vs. $\mathcal{NP}$, y consiste en demostrar la diferencia o la equivalencia de las clases $\mathcal{P}$ y $\mathcal{NP}$. En general, se cree que estas dos clases son diferentes, sin embargo no se ha podido demostrar que esto sea cierto.

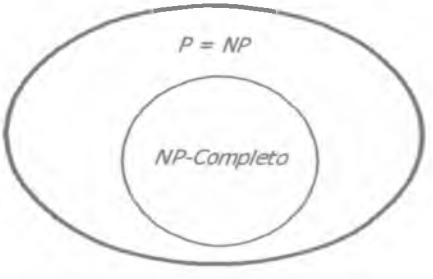
De hecho, $\mathcal{P}$ vs. $\mathcal{NP}$ forma parte de los siete problemas del milenio del Clay Mathematics Institute. Estos siete problemas fueron escogidos por ser preguntas matemáticas para las cuales se han buscado soluciones por varios años sin éxito. Dada la dificultad y la importancia de dichos problemas, existe un premio de un millón de dólares asignado a cada uno ellos.

Es fácil ver que la clase $\mathcal{P}$ es subconjunto de $\mathcal{NP}$. Esto es porque los problemas en $\mathcal{P}$ son aquellos que pueden ser resueltos por máquinas de Turing deterministas, mientras que los problemas $\mathcal{NP}$ pueden ser resueltos por máquinas no deterministas. Sin embargo, las máquinas deterministas son casos particulares de las no deterministas, en las cuales no existen dos quintetos que empiecen con la misma pareja de estado y símbolo. De esta forma, un problema que se encuentre en $\mathcal{P}$, al ser resuelto por una máquina determinista, también es resuelto por una máquina no determinista, y por lo tanto se encuentra en $\mathcal{NP}$.

Utilizando la definición 3.5, la contención se prueba de la siguiente manera. Si tenemos un problema $\Pi = (L, P)$ dentro de $\mathcal{P}$, podemos agregar el símbolo # a todas las palabras de L y de P, formando así L' y P'. De esta forma, tomamos $\Pi' = (L', P')$ y utilizamos como certificado a la palabra vacía.

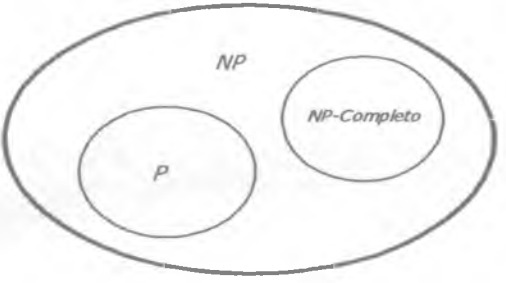
La contención $\mathcal{NP} \subseteq \mathcal{P}$ es la que aún no se ha podido demostrar ni contradecir. La forma en la que se podría contradecir esta contención es mostrando un problema en $\mathcal{NP}$ y demostrando que no se encuentra en P. Sin embargo, muchos de los problemas que no son $\mathcal{NP}-completos$ terminan perteneciendo a $\mathcal{P}$. Este es el caso de los problemas de programación lineal y números primos. Por mucho tiempo no se conocían algoritmos polinomiales para estos problemas. Ahora, para el problema de programación lineal se conoce el método de elipsoides, que es un algoritmo polinomial para este problema [12]. Para el problema de determinar si un número es primo, se acaba de encontrar en agosto del 2002 un algoritmo polinomial que lo resuelve [7]. Sabemos que los problemas $\mathcal{NP}-completos$ son los problemas más difíciles dentro de la clase $\mathcal{NP}$. Es por esto que los intentos de probar $\mathcal{P}$ vs. $\mathcal{NP}$ se basan en estos problemas en particular.

La manera más obvia de probar $\mathcal{P} = \mathcal{NP}$ sería dar un algoritmo polinomial para algún problema $\mathcal{NP}-completo$. Si pudiéramos encontrar un algoritmo polinomial para uno de estos problemas, entonces, por la proposición 1.3 cualquier problema de $\mathcal{NP}$ tendría un algoritmo polinomial. Sin embargo, también cabe la posibilidad de dar una prueba no constructiva de $\mathcal{P} = \mathcal{NP}$, aunque ésta no tendría tantas ventajas. En este caso, el escenario que tendríamos sería el siguiente:



Por otro lado, si quisiéramos probar que $\mathcal{P} \neq \mathcal{NP}$, debemos demostrar que algún problema en $\mathcal{NP}-completo$ es intratable. De esta manera, sabemos que existe un problema en $\mathcal{NP}$ que no está en $\mathcal{P}$, y por lo tanto son diferentes.

El escenario que tendríamos sería el siguiente:



Existen alrededor de 1000 problemas $\mathcal{NP}-completos$ conocidos en diferentes áreas de las matemáticas. Debido a la importancia industrial de muchos de ellos, se han buscado soluciones eficientes para estos problemas durante los últimos 30 años sin éxito. Aún cuando en general se cree que $\mathcal{P} \neq \mathcal{NP}$, es bastante probable que dicha conjetura no podrá ser resuelta sin el desarrollo de una metodología matemática totalmente nueva.

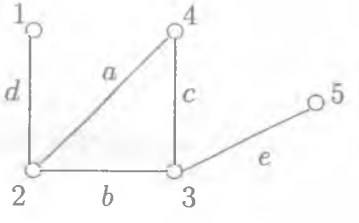
Apéndice A

Conceptos de Teoría de Gráficas

Definición A.1. *Una **gráfica simple** es una pareja ordenada $G = (V(G), E(G))$ donde $V(G)$ es el conjunto finito no vacío de vertices o nodos y $E(G)$ es un conjunto de parejas no ordenadas de vértices distintos llamadas aristas.*

En este trabajo únicamente se utilizan gráficas simples, por lo que al referirnos a una gráfica, ésta será de este tipo. Se dice que dos vértices u y v son **adyacentes** si $e = [u, v]$ es una arista de $E(G)$. En otras palabras, dos vértices son adyacentes si se encuentran unidos por una arista. En este caso, u y v son los **extremos** de e. También se dice que e **incide** en u y en v.

Ejemplo A.2. Sea $G = (V(G), E(G))$ una gráfica con
$V(G) = \{1, 2, 3, 4, 5\}$, $E(G) = \{a, b, c, d, e, f\}$ y
$a = [2, 4]$, $b = [2, 3]$, $c = [3, 4]$, $d = [1, 2]$, $e = [3, 5]$



El **complemento** de una gráfica simple G, denotado G^c, es una gráfica con el mismo conjunto de vértices $V(G)$ pero únicamente tiene aquellas

aristas que no se encuentran en G. Es decir, una arista $[u, v]$ se encuentra en $E(G^c)$ si y sólo si $[u, v]$ no se encuentra en $E(G)$.

Se dice que una gráfica H es una **subgráfica** de G si $V(H) \subseteq V(G)$ y $E(H) \subseteq E(G)$. Una **subgráfica generadora** de G es una subgráfica de G cuyo conjunto de vértices es $V(G)$. Si contamos con $V' \subseteq V(G)$, llamamos **subgráfica inducida por** V' a la subgráfica de G cuyo conjunto de vértices es V' y que tiene como conjunto de aristas a todas las aristas de $E(G)$ que unen a los vertices de V'.

Un **camino** es una secuencia alternante de vértices y aristas

$$v_0, e_1, v_1, e_2, \ldots, e_n, v_n$$

en donde $e_j = [v_{j-1}, v_j]$. Se dice que éste es un camino de v_0 a v_n, y a v_0 y v_n se les llama vértice inicial y final respectivamente. Podemos denotar el camino únicamente con los vértices dados en el orden que los recorre, por ejemplo, $v_0, v_1, \ldots, v_n$.

Un **paseo** es un camino en donde no se repiten aristas. Una **trayectoria** es un paseo en donde no se repiten vértices. Se dice que una gráfica es **conexa** si entre cualesquiera dos vértices u y v existe una trayectoria de u a v. Un **ciclo** es un paseo cuyo vértice inicial y final es el mismo.

Un **árbol** es una gráfica conexa que no contiene ciclos. Un **árbol generador** es una subgráfica generadora que es un árbol.

Una **trayectoria hamiltoniana** es una trayectoria que pasa por todos los vértices de la gráfica. De la misma forma, un **ciclo hamiltoniano** es un ciclo que pasa por todos los vértices de la gráfica. Se dice que G es una **gráfica hamiltoniana** si G contiene un ciclo hamiltoniano.

Una **cubierta** de G, denotada C, es un subconjunto de $V(G)$ tal que cualquier arista de $E(G)$ incide en al menos un vértice de C. Un **conjunto independiente** de G, denotado I, es un subconjunto de $V(G)$ tal que no contiene vértices adyacentes. Un **clan** de G es un subconjunto de $V(G)$ tal que cualquiera dos vértices del clan son adyacentes. Estos tres conceptos están muy relacionados entre si. como se puede ver en la siguiente proposición. Ésta se obtuvo de [12], mientras que la demostración se obtuvo de [2].

Proposición A.3. *Sea G una gráfica simple y $X \subset V(G)$. Entonces son equivalentes:*

1. *X es una cubierta de G*

2. *$V(G)\backslash X$ es un conjunto independiente de G*

3. *$V(G)\backslash X$ es un clan de G^c.*

Demostración. Primero demostraremos que (1) es equivalente a (2). Por definición, X es una cubierta de G si y solo si cada arista tiene al menos un extremo en X. Esto es equivalente a decir que ninguna arista incide sobre dos vértices de $V(G)\backslash X$. Esto significa que no hay vértices adyacentes en $V(G)\backslash X$, lo cual es la definición de un conjunto independiente.

Demostraremos ahora que (2) es equivalente a (3). Sea $V(G)\backslash X$ un conjunto independiente de G. Por definición, no existen vértices adyacentes en $V(G)\backslash X$ en la gráfica G. Esto ocurre si y sólo si todos los vértices de $V(G)\backslash X$ son adyacentes en el complemento de G. Ésta es la definición de un clan en el complemento de G. □

Para ejemplificar este resultado, se muestran las siguientes figuras. Las primeras dos representan a la gráfica G, mientras que la tercera representa a G^c. Los vértices oscuros de la primera figura muestran una cubierta para G, los de la segunda muestran un conjunto independiente de G y los de la tercera dan un clan para G^c.

Una **coloración de vértices** de una gráfica es una función cuyo dominio es $V(G)$ y contradominio es un conjunto $\mathcal{C}$ que representa colores. Es decir, una coloración de vértices es una función que asocia a cada vértice de G un color. Se dice que una coloración es **propia** si cualesquiera dos vértices adyacentes tienen colores distintos. Una **k-coloración** es una coloración de vértices que utiliza a lo más k colores.

Una **gráfica dirigida** o **digráfica** D es una gráfica en la cual las aristas tienen direcciones. Llamamos **arcos** a las aristas dirigidas, y están formados

por parejas ordenadas de vértices distintos. Mientras que en una gráfica simple no dirigida $[u, v]$ o $[v, u]$ eran dos formas de representar una misma arista, en una gráfica dirigida representan dos arcos distintos. Los paseos, caminos y trayectorias en una digráfica se conocen como paseos, caminos y trayectorias dirigidas, y deben seguir el sentido que indican los arcos.

Apéndice B

Expresiones Booleanas

Una **variable booleana** es una variable que únicamente puede tomar los valores de verdadero o falso, representados por los números 1 y 0 respectivamente. Las variables booleanas se relacionan entre sí con los operadores lógicos $\wedge$ y $\vee$, que representan la conjunción y la disyunción. Por otro lado, también tenemos la operación de la negación, representada con el símbolo $\neg$ antes de la variable. Sin embargo, para facilitar la notación hacemos $\overline{x} = \neg x$. La operación con mayor prioridad es la negación ($\neg$), después la conjunción ($\wedge$) y por último la disyunción ($\vee$). Si se quiere modificar la prioridad en una expresión se utilizan paréntesis.

Para facilitar la notación, si tenemos muchas conjunciones o disjunciones juntas, podemos utilizar los operadores de conjunción y disjunción múltiple. Estas operaciones son representadas por los símbolos $\bigwedge$ y $\bigvee$ respectivamente, y se utilizan de la misma manera que las sumas y los productos múltiples. Por ejemplo, si tenemos la expresión booleana $x_1 \vee x_2 \vee \cdots \vee x_n$ podemos escribirla como

$$\bigvee_{i=1}^{n} x_i.$$

Análogamente, si tenemos la expresión $x_1 \wedge x_2 \wedge \cdots \wedge x_n$ podemos escribirla como

$$\bigwedge_{i=1}^{n} x_i.$$

Una **expresión booleana** es una expresión construida a partir de un conjunto de variables booleanas y los operadores lógicos de negación, con-

junción y disyunción. Estas expresiones se construyen de forma recursiva. Las variables booleanas son en sí mismas expresiones booleanas. Si tenemos E_1 y E_2 dos expresiones booleanas, entonces (E_1), $\overline{E_1}$, $E_1 \wedge E_2$ y $E_1 \vee E_2$ son también expresiones boolenas.

A las funciones que pueden ser representadas por medio de expresiones booleanas se les conoce como **funciones booleanas**. Una función booleana

$$f : \{0,1\}^n \longrightarrow \{0,1\}$$

permite asignar valores a las variables de una expresión booleana para obtener un valor de verdad o falsedad. Las funciones booleanas se evalúan utilizando las siguientes identidades:

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 1 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

Si tenemos una función booleana $f(\mathbf{x}) = f(x_1, \ldots, x_n)$, entonces $\mathbf{t} = (t_1, \ldots, t_n)$, donde $t_i \in \{0,1\}$ para $i = 1, \ldots, n$ es una **asignación** de las variables de $\mathbf{x}$. Dada la asignación $\mathbf{t}$, podemos evaluar la función booleana en ella, de manera que $f(\mathbf{t}) = f(t_1, \ldots, t_n)$ nos da un valor de cero o uno. Se dice que una expresión booleana $f(\mathbf{x})$ es **satisfacible** si existe alguna asignación $\mathbf{t}$ tal que $f(\mathbf{t}) = 1$.

Por la forma en la que se construyen, las funciones booleanas en general no tienen una estructura definida. Sin embargo, a veces requerimos una forma más estructurada de dichas funciones, por lo que a continuación damos las siguientes definiciones. Una **literal** es una variable o la negación de una variable. Una **cláusula** es la disyunción de una o más literales. Claramente, la cláusula es verdadera bajo una asignación si y sólo si al menos una de las literales de la cláusula toma el valor de verdadero.

Se dice que una expresión booleana se encuentra en **forma normal conjuntiva** (**FNC**) si se puede expresar como una conjunción de cláusulas. Para que una expresión en FNC sea verdadera bajo una asignación, todas las cláusulas que la conforman deben ser verdaderas bajo dicha asignación.

Dos funciones booleanas con el mismo número de variables son equivalentes si para toda asignación dan el mismo valor. Todas las funciones

booleanas tienen una función en FNC equivalente a ellas. Sin embargo. encontrar dicha función puede tomar un tiempo exponencial.

Si tenemos una función booleana $f(x_1, \ldots, x_n)$, podemos encontrar una función $g(x_1, \ldots, x_n, x_{n+1}, \ldots, x_m)$ en FNC que sea satisfacible si y sólo si la función original lo es. Lo que es más, si tenemos una asignación $\mathbf{t}$ tal que $f(\mathbf{t}) = 1$ entonces existe una extensión $\mathbf{s}$ de $\mathbf{t}$ tal que $g(\mathbf{s}) = 1$. Claramente $\mathbf{s}$ tiene más valores que $\mathbf{t}$. Se dice que es una extensión porque da los mismos valores que $\mathbf{t}$ en las mismas variables, aunque puede asignar otros valores a las variables restantes.

Esta nueva función, aunque no es equivalente a $f(\mathbf{x})$, puede construirse en tiempo polinomial. Para demostrar lo anterior, utilizaremos las leyes de DeMorgan y la ley de doble negación.

$$\begin{aligned} \overline{(x_1 \wedge x_2)} &= \overline{x}_1 \vee \overline{x}_2 \\ \overline{(x_1 \vee x_2)} &= \overline{x}_1 \wedge \overline{x}_2 \\ \overline{(\overline{x})} &= x \end{aligned}$$

Proposición B.1. *Para cualquier función booleana podemos encontrar en tiempo polinomial una función en FNC satisfacible si y sólo si la expresión original lo es.*

No daremos una demostración formal para esta proposición. En cambio, ejemplificaremos como trabaja el algoritmo polinomial que relaciona a una función booleana con una función en FNC.

Para este ejemplo, tomamos la expresión booleana

$$E = ((x_1 \wedge \overline{x}_2 \vee x_3) \vee (x_2 \wedge x_4)) \wedge \overline{\overline{((x_1 \vee x_3 \vee x_4)} \vee x_5)}$$

que representa a una función boolena $f(\mathbf{x})$. Lo primero que hacemos es utilizar las leyes de DeMorgan y la de doble negación para dejar las negaciones únicamente en las literales. Este procedimiento toma un tiempo lineal en realizarse.

$$\begin{aligned} &((x_1 \wedge \overline{x_2} \vee x_3) \vee (x_2 \wedge x_4)) \wedge \overline{(\overline{(x_1 \vee x_3 \vee x_4)}) \wedge \overline{x}_5)} \\ E' &= ((x_1 \wedge \overline{x}_2 \vee x_3) \vee (x_2 \wedge x_4)) \wedge (x_1 \vee x_3 \vee x_4) \wedge \overline{x}_5 \end{aligned}$$

Una vez que tenemos esta expresión, iremos trabajando con los operadores, empezando por los que tienen mayor prioridad. Empezaremos primero

por los paréntesis que se encuentran al interior de la expresión. Si dentro de un paréntesis tenemos únicamente disyunciones, entonces contamos con una cláusula y pasamos al siguiente paréntesis. Asimismo, si contamos únicamente con conjunciones dentro de un paréntesis, tenemos una FNC, por lo que también pasaremos al siguiente paréntesis. Llamaremos F' a la subexpresión con la que estemos trabajando en cada paso.

Cada operador relaciona dos subexpresiones, a las cuales podemos llamar E_1 y E_2. Sabemos que estas subexpresiones se encuentran en FNC, ya que, de no ser cierto, F' sería alguna de estas subexpresiones.

Debido a que tenemos dos operadores, podemos encontrarnos con dos casos distintos. En el primero tenemos $F' = E_1 \wedge E_2$. En este caso, como ambas subexpresiones se encuentran en FNC, la expresión se encuentra en FNC, por lo que podemos pasar al siguiente operador sin hacer cambios. En el segundo caso tenemos $F' = E_1 \vee E_2$. Para transformar esta expresión a FNC es necesario crear una variable y. Esta variable se agregará a cada una de las cláusulas de E_1 formando una nueva subexpresión E_1', mientras que $\overline{y}$ se agregará a las cláusulas de E_2 obteniendo E_2'. Al agregar las literales correspondientes a las cláusulas, podemos sustituir $E_1 \vee E_2$ por $E_1' \wedge E_2'$ sin afectar la satisfacibilidad. Esto es porque si existe alguna asignación que satisfaga E_1 podemos hacer $y = 0$. Esto implica que $\overline{y} = 1$, por lo que las cláusulas de E_2' también se satisfacen. El razonamiente es análogo si alguna asignación satisface E_2.

En el ejemplo, el algoritmo haría lo siguiente:

$$((\underbrace{x_1 \wedge \overline{x}_2}_{F'} \vee x_3) \vee (x_2 \wedge x_4)) \wedge (x_1 \vee x_3 \vee x_4) \wedge \overline{x}_5.$$

Como x_1 y $\overline{x}_2$ son literales, podemos verlas como cláusulas. De esta forma, tenemos una conjunción de cláusulas, por lo que estamos en el primer caso. Dejamos la expresión como está y pasamos al siguiente operador.

$$((\underbrace{x_1 \wedge \overline{x}_2 \vee x_3}_{F'}) \vee (x_2 \wedge x_4)) \wedge (x_1 \vee x_3 \vee x_4) \wedge \overline{x}_5$$

Hacemos $E_1 = x_1 \wedge \overline{x}_2$ y $E_2 = x_3$. Aquí, E_1 y E_2 se encuentran cada una en FNC. Como están separadas por una disyunción, nos encontramos en el

segundo caso. Agregaremos la literal y_1 a cada una de las cláusulas de E_1 y la literal $\overline{y}_1$ a las cláusulas de E_2, obteniendo la expresión

$$(((x_1 \vee y_1) \wedge (\overline{x}_2 \vee y_1) \wedge (x_3 \vee \overline{y}_1)) \vee (x_2 \wedge x_4)) \wedge (x_1 \vee x_3 \vee x_4) \wedge \overline{x}_5.$$

Pasamos a la siguiente operador.

$$\underbrace{(((x_1 \vee y_1) \wedge (\overline{x}_2 \vee y_1) \wedge (x_3 \vee \overline{y}_1)) \vee (x_2 \wedge x_4))}_{F'} \wedge (x_1 \vee x_3 \vee x_4) \wedge \overline{x}_5.$$

Hacemos $E_1 = ((x_1 \vee y_1) \wedge (\overline{x}_2 \vee y_1) \wedge (x_3 \vee \overline{y}_1))$ y $E_2 = (x_2 \wedge x_4)$. De nuevo tenemos una disyunción de expresiones en FNC, por lo que agregaremos una variable y_2 en la forma en la que se hizo anteriormente. Así obtenemos

$$F = (x_1 \vee y_1 \vee y_2) \wedge (\overline{x}_2 \vee y_1 \vee y_2) \wedge (x_3 \vee \overline{y}_1 \vee y_2) \wedge (x_2 \vee \overline{y}_2) \wedge (x_4 \vee \overline{y}_2) \wedge (x_1 \vee x_3 \vee x_4) \wedge \overline{x}_5.$$

El siguiente operador indica una conjunción de FNC, por lo que no se realizarán más cambios. De esta forma, la expresión F es una expresión booleana en FNC. Si tomamos la función booleana $g(\mathrm{y})$ representada por dicha expresión, esta función cumple con las características que buscamos.

Además, esta función se construye en un tiempo polinomial. Como ya vimos, pasar de E a E' toma un tiempo lineal. Sea n el número de operadores de la expresión E' sin contar negaciones. A lo más se agregan n nuevas variables por cada operador. Esto nos lleva a que el algoritmo es de orden cuadrático.

Bibliografía

[1] Beckman, Frank S., *Mathematical Foundations of Programming*, Addison Wesley Publishing Company, USA, 1980.

[2] Bondy, J.A. y Murty, *Graph Thoery with Applications*, North Holland, 1984.

[3] Cook, Stephen, The P versus NP Problem, University of Toronto, http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf

[4] Cook, William J., Cunningham, William H., Pulleyblank, William R., Schrijver, Alexander, *Combinatorial Optimization*, John Wiley & Sons, Inc., EUA, 1998.

[5] Davies, Martin, *The Universal Computer, the road form Leibniz to Turing*, W. W. Norton & Company, EUA, 2000.

[6] Garey, Michael R. y Johnson, David D., *Computers and Intractability, a guide to the theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[7] Granville, Andrew, It is Easy to Determine Whether a Given Integer is Prime, Bulletin (New Series) of the American Mathematical Society, Volume 42, Number 1, Pages 3-38.

[8] Gross, Jonathan L., Yellen, Jay, *Handbook of Graph Theory*, CRC Press, USA, 2004.

[9] Hopcroft, John E., Motwani, Rajeev, Ullman, Jeffrey D., *Introducción a la Teoría de Autómatas, Lenguajes y Computación*, 2 edición, Addison Wesley, Madrid, 2001.

[10] Hopcroft, John E., Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation*, 1 edición, Addison Wesley. USA, 1979.

[11] Jhonsonbaugh, Richard, *Discrete Mathematics*, Fourth Edition, Prentice-Hall, USA, 1997.

[12] Korte, Bernhard y Vygen, Jens, *Combinatorial Optimization*, 2 edición, Springer, Alemania, 2002.

[13] Moret, Bernard M., *The Theory of Computation*, Addison Wesley, USA, 1998.

[14] Papadimitriou, Christos H., Steiglitz, Kenneth, *Combinatorial Optimization, algorithms and complexity*, Dover Publications, Inc., New York, 1998.

[15] Sipser, Michael, *Introduction to the Theory of Computation*, PWS Publishing Company, USA, 1997.

[16] http://www.claymath.org/millennium/

[17] http://plato.stanford.edu/entries/church-turing/

[18] http://es.wikipedia.org/wiki/Tesis_de_Church-Turing

[19] http://es.wikipedia.org/wiki/Complejidad_computacional.

[20] http://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica

[21] http://en.wikipedia.org/wiki/Museum_guard_problem