



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

FACULTAD DE MATEMÁTICAS

Complejidad computacional del conteo de descascaramientos de un grafo simple

Autora: María Alejandra Schild Zerene

Supervisor: José Alejandro Samper Casas

Tesis presentada a la Facultad de Matemáticas de la
Pontificia Universidad Católica de Chile
para optar al grado académico de magíster en Matemática

Comisión de defensa de tesis:

Pablo Barceló (Pontificia Universidad Católica de Chile),
Federico Castillo (Pontificia Universidad Católica de Chile).

Agosto de 2024

Santiago, Chile

Agradecimientos

En primer lugar, quiero darle las gracias al profesor Jose Samper por todo lo que me ha enseñado, por su comprensión y apoyo durante estos tres años, y por haber confiado en mí. Me siento afortunada de haber contado con un supervisor que siempre estuvo dispuesto a escucharme y aconsejarme cuando lo necesitaba, y que me dio la libertad de dar varios giros en ciento ochenta grados durante el desarrollo de esta tesis.

Le agradezco al profesor Federico Castillo por haberme transmitido su entusiasmo por la geometría discreta y la programación; y al profesor Pablo Barceló, por haberme recibido en múltiples ocasiones para conversar sobre el problema de los descascaramientos. Muchas gracias a ambos por haber aceptado revisar esta tesis. Agradezco, además, a los profesores José Verschae y Juan Reutter por sus valiosos comentarios. También aprecio mucho los consejos y la orientación que he recibido de los profesores Mario Ponce y Mariel Sáez.

Una de las cosas que más disfruté y que más me mantuvieron motivada durante este magíster fue el seminario de combinatoria algebraica. Le mando un abrazo a Juan, a Felipe, y especialmente a Nachín, que me convenció de integrarme. Gracias por tantos recuerdos bonitos, lo pasé muy bien con ustedes.

Durante estos años en la UC he tenido la oportunidad de conocer a personas geniales. En particular, aprecio mucho la amistad que he construido con Vicente y con Cecilia: muchas gracias a ambos por haber estado siempre ahí para mí. También le mando un abrazo a los cabros de la oficina M25, especialmente a Coke, a Benja y a Juan Pablo.

Un saludo muy (muy) especial a Sofía, que ha sido mi amiga y compañera en este proceso. Ya no recuerdo bien cómo sucedieron las cosas, pero el giro computín que tomó esta tesis fue, en gran parte, gracias a ella.

Por último, un gran abrazo a mis padres. Gracias por su apoyo y amor incondicional durante todos estos años. Los quiero mucho.

Los estudios que condujeron a la elaboración de esta tesis fueron financiados por la Subdirección de Capital Humano de la ANID (Agencia Nacional de Investigación y Desarrollo) a través de la beca de Magíster Nacional del año 2022. El número de folio correspondiente es 22221414.

Índice

Introducción	3
Notación general y convenciones	5
1 Preliminares de complejidad computacional	6
1.1 Máquinas decisoras: Las clases P y FP	6
1.2 Codificaciones binarias	8
1.3 Verificadores de tiempo polinomial	11
1.4 Máquinas oráculo	13
1.5 La complejidad de contar: La clase #P	15
1.6 #P-completitud	18
1.7 Reducciones parsimoniosas y #P-completitud fuerte	20
2 Preliminares de complejos simpliciales	21
2.1 Definiciones básicas sobre complejos simpliciales	21
2.2 Descascaramientos de un complejo simplicial puro	22
2.3 Codificación binaria de un complejo simplicial puro	25
2.4 El problema de contar descascaramientos	25
3 Conteo de descascaramientos de un grafo simple	27
3.1 Terminología básica	27
3.2 Recursiones del problema	28
3.3 Ejemplos notables	29
3.4 Algoritmo de tiempo polinomial para grafos con h_2 acotado	32
3.5 #P-completitud de variante con un árbol generador prescrito	34
3.6 Ordenamientos conexos de vértices	40
3.7 Vectores de partición	43
4 Apéndices	50
4.1 Código Python para calcular descascaramientos de grafos	50
4.2 Invertibilidad de la matriz de decoraciones	51
4.3 Código Python para calcular vectores de partición	52
Referencias	55

Introducción

Un *descascaramiento* es una forma organizada de construir un complejo simplicial puro. Más precisamente, se trata de un ordenamiento de sus facetas que permite entender la estructura del complejo por medio de una secuencia de pasos sencillos y bien comportados. A los complejos simpliciales que admiten un descascaramiento se les llama *descascarables*. Este concepto tiene múltiples aplicaciones, tanto en topología computacional como en combinatoria algebraica. El estudio de la topología de los complejos simpliciales descascarables es sencillo, pues estos son homotópicamente equivalentes a una cuña de esferas [BW96]. Respecto al álgebra, se sabe que el anillo de Stanley-Reisner de un complejo simplicial descascarable es Cohen-Macaulay sobre cualquier campo infinito [Sta18a]. Estos son solo dos ejemplos que ilustran la importancia de poder contar con criterios para decidir si un complejo simplicial es descascarable.

Se sigue directamente de la definición que es posible verificar en tiempo polinomial si un ordenamiento de las facetas de un complejo simplicial es un descascaramiento. Por lo tanto, el problema de la descascarabilidad pertenece a la clase de complejidad **NP**. En el año 2018, Xavier Goaoc, Pavel Paták, Zuzana Patáková, Martin Tancer y Uli Wagner demostraron que decidir si un complejo simplicial puro de dimensión 2 es descascarable es **NP-completo** [GPP⁺19]. Puesto que el número de descascaramientos se preserva al tomar conos, la **NP**-dificultad del caso 2-dimensional se propaga inmediatamente a cualquier dimensión superior. En la práctica, bajo suposiciones estándar en la teoría de la complejidad computacional, este resultado muestra que no puede existir un criterio de tiempo polinomial para resolver el problema de la descascarabilidad en general.

Una consecuencia de la **NP**-completitud del problema de la descascarabilidad es que, a menos que $\mathbf{P} = \mathbf{NP}$, no existe un algoritmo de tiempo polinomial para contar el número de descascaramientos de un complejo simplicial de dimensión 2 (pues tal algoritmo permitiría, en particular, decidir descascarabilidad). Sería deseable contar con algún resultado para computar esto último, pues entonces sería posible calcular la probabilidad exacta de que un ordenamiento aleatorio de las facetas sea un descascaramiento. Existe muy poca literatura sobre este problema, y se desconoce su ubicación en la jerarquía de clases de complejidad.

Es importante mencionar que un problema de conteo es distinto a un problema de enumeración. El primero consiste en determinar el número de objetos que cumplen cierta propiedad, mientras que el segundo exige que estos sean listados explícitamente (lo que, por supuesto, es al menos igual de difícil que poder contarlos). Por ejemplo, no es posible enumerar en tiempo polinomial los árboles generadores de un grafo general, sencillamente porque el número de ellos no está acotado por ningún polinomio en el número de vértices del grafo (el grafo completo de n vértices tiene n^{n-2} árboles generadores). Por otro lado, el problema de contar los árboles generadores de un grafo general sí puede resolverse en tiempo polinomial. Esto último se debe al teorema de Kirchhoff, que reduce el conteo de árboles generadores al cálculo del determinante de una matriz que puede construirse en tiempo polinomial.

El objetivo de esta tesis es estudiar la complejidad computacional del problema de conteo de descascaramientos en el caso de dimensión 1, que corresponde a grafos simples. Se sabe que el problema de decisión asociado está en **P**, pues es equivalente al problema de decidir conexidad. No obstante, hasta donde llega nuestro conocimiento, el problema de conteo solo ha sido estudiado para familias particulares. Hacemos notar que una prueba de **#P**-completitud para este problema significaría una fuerte evidencia de intratabilidad, y elevaría la complejidad computacional conocida de los problemas de conteo de descascaramientos en dimensiones superiores.

Nuestra contribución puede resumirse en los siguientes puntos:

- Describimos un algoritmo de tiempo polinomial para el problema restringido a instancias en que la diferencia entre el número de aristas y el número de vértices está acotada por una constante fija.
- Demostramos la $\#\mathbf{P}$ -completitud de una variante del problema, que consiste en prescribir que las aristas que conecten vértices por primera vez al descascaramiento pertenezcan a un árbol generador fijo. La prueba se basa en una correspondencia entre descascaramientos que cumplen dicha condición y extensiones lineales de ciertos conjuntos parcialmente ordenados, cuya $\#\mathbf{P}$ -completitud fue establecida en 2020 por Samuel Dittmer e Igor Pak [DP20].
- Presentamos un algoritmo para contar descascaramientos en tiempo $\mathcal{O}(n^2 \cdot n!)$, donde n es el número de vértices del grafo. En grafos densos, esto es significativamente más rápido que el algoritmo de *backtracking* que explora los posibles ordenamientos de aristas. La idea se basa en un argumento de conteo utilizado por Richard Stanley para calcular el número de descascaramientos de los grafos completos [Sta18b].
- Mostramos que, si las aristas de un grafo pueden particionarse en subgrafos completos, entonces el número de descascaramientos será divisible en un producto de superfactoriales que dependen de la cardinalidad de dichos subgrafos. Creemos que, restringiendo a instancias adecuadas, esto puede dar lugar a un criterio en ambas direcciones.
- Definimos dos matrices que particionan los descascaramientos de un grafo según el paso en el que conectan a un elemento específico, y usamos una técnica de interpolación para mostrar que ambas son computables en tiempo polinomial a partir de un oráculo para contar descascaramientos. Estas matrices entregan información más fina y local que el oráculo original; creemos que pueden ser útiles para una futura reducción que evidencie la probable intratabilidad del problema.

Notación general y convenciones

- $\mathbb{N} = \{0, 1, 2, \dots\}$.
- Dado $n \in \mathbb{N}$, definimos $[n] := \{1, 2, \dots, n\}$. En particular, $[0] = \emptyset$.
- Dado un entero positivo n , S_n denota el grupo de biyecciones $[n] \rightarrow [n]$.
- Dado un conjunto finito expresado en notación de corchetes, escribiremos $\#\{\cdot\}$ en lugar de $|\{\cdot\}|$ para denotar su cardinalidad.
- La notación $A \subseteq B$ significa que A es un subconjunto de B . Si, además, sabemos que $A \neq B$, escribiremos $A \subsetneq B$. No usaremos el símbolo \subset .
- Dado un conjunto E , usaremos la notación 2^E para denotar a su conjunto potencia.
- Usaremos la notación $A - B$ para denotar al subconjunto de A que consiste de los elementos que no pertenecen a B .
- Dada una función $f : A \rightarrow B$ y subconjuntos $X \subseteq A$ e $Y \subseteq B$, consideramos
$$f(X) := \{y \in B \mid \exists x \in X \ f(x) = y\} \quad , \quad f^{-1}(Y) := \{x \in A \mid f(x) \in Y\}.$$
- Dadas dos funciones $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$, escribiremos $f(n) = \mathcal{O}(g(n))$ si existe una constante $c > 0$ y un natural $n_0 \in \mathbb{N}$ tal que $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$.
- Dada una función $h : \mathbb{N} \rightarrow \mathbb{N}$, escribiremos $h(n) = \text{poly}(n)$ si existe un entero positivo k y un natural n_0 tal que $h(n) \leq n^k$ para todo $n \geq n_0$.
- Dados enteros positivos n, m , escribimos K_n para denotar al grafo completo de n vértices, y $K_{n,m}$ para denotar al grafo bipartito completo con n y m vértices en cada parte.
- Dado un grafo $G = (V, E)$ y un vértice $v \in V$, escribimos $\text{deg}(v)$ para denotar su grado, es decir, el número de aristas incidentes con él.
- Dados enteros $m_1 + m_2 + \dots + m_k = n$, definimos el coeficiente multinomial

$$\binom{n}{m_1, m_2, \dots, m_k} := \frac{n!}{m_1! m_2! \dots m_k!}.$$

1. Preliminares de complejidad computacional

Asumiremos que el lector está familiarizado con el concepto de máquina de Turing determinista: ese será el modelo de computación que utilizaremos. También asumiremos conocimiento sobre las nociones de decibilidad y de máquina universal. El tipo de complejidad computacional que estudiaremos es el tiempo de ejecución en el peor de los casos. Hemos decidido utilizar las caracterizaciones vía verificadores polinomiales para las clases no deterministas, de modo que omitimos discutir el modelo de la máquina de Turing no determinista.

Para aquellos lectores que no están familiarizados con estos conceptos, recomendamos la lectura del capítulo 1 del libro *Computational complexity: a modern approach*, de Sanjeev Arora y Boaz Barak [AB09], y también del capítulo 2 de *Computational complexity*, de Christos Papadimitriou [Pap94]. Nos guiaremos por esos dos libros durante este capítulo.

No nos preocuparemos por fijar una arquitectura particular del modelo de computación, pues las modificaciones habituales de la máquina de Turing solo implican un cambio polinomial en su tiempo de ejecución, y las clases de complejidad temporales que estudiaremos son cerradas bajo transformaciones polinomiales. Sin embargo, resultará conveniente establecer que, además de las cintas de trabajo, la máquina de Turing siempre cuenta con una cinta especial de entrada (en la que solo puede leer) y otra de salida (en la que solo puede escribir).

Tampoco le daremos mayor importancia a la implementación a bajo nivel de los algoritmos: la descripción de un pseudocódigo nos bastará. Asumiremos que el tiempo de computación de un proceso efectivo es proporcional a la cantidad de operaciones elementales que se deben realizar como máximo, donde una «operación elemental» es cualquier acción que una máquina de Turing pueda realizar en un número fijo de pasos (independientemente de la instancia).

1.1. Máquinas decisoras: Las clases P y FP

Dado un conjunto finito \mathcal{A} , denotaremos por \mathcal{A}^* al conjunto de tuplas finitas formadas por elementos de \mathcal{A} . Diremos que \mathcal{A} es un «alfabeto» y que sus elementos son «símbolos». Además, diremos que \mathcal{A}^* es un «vocabulario», y a sus elementos los llamaremos «palabras». En particular, a los símbolos del alfabeto $\{0, 1\}$ los llamamos «bits». A la palabra vacía la denotaremos con el símbolo ϵ . Dada una palabra $x \in \mathcal{A}^*$, denotamos por $|x| \in \mathbb{N}$ a su longitud. Un «lenguaje» es un subconjunto de \mathcal{A}^* .

Trabajar con más de dos símbolos no nos dará más poder computacional que el alfabeto binario. En efecto, supongamos que tenemos una máquina \mathcal{M} que lee palabras escritas en el alfabeto $\mathcal{A} = \{0, \dots, m-1\}$, con $m > 2$. Sea $k = \lceil \log_2 m \rceil$, y notemos que cada elemento de \mathcal{A} puede reescribirse como una palabra de k bits. Esta traducción escala el largo de cualquier palabra en \mathcal{A}^* por una constante fija. Podemos modificar \mathcal{M} para que funcione en binario: lo que hacemos es crear un nuevo estado interno por cada símbolo del alfabeto \mathcal{A} . La nueva máquina debe leer la entrada binaria en trozos de k bits y, para cada uno de ellos, calcular el símbolo en \mathcal{A} que le corresponde, entrar al estado interno respectivo y reemplazar ese trozo de k bits por el trozo de k bits correspondiente al nuevo estado interno que indique la función de transición. Aparte de esa traducción, el funcionamiento de ambas máquinas es idéntico, y el costo computacional de la transformación corresponde a multiplicar por una constante. Por lo tanto, de ahora en adelante consideraremos que $\mathcal{A} = \{0, 1\}$.

No involucraremos ningún símbolo para denotar la concatenación de palabras, salvo que se trate de la repetición de un mismo símbolo, en cuyo caso utilizaremos la notación de potenciación. Siempre que escribamos la expresión 0^k o 1^k con $k \in \mathbb{N}$ nos referiremos a las

correspondientes palabras de k bits.

Definición 1.1.1. Diremos que una máquina de Turing \mathcal{M} es un «decisor» si se detiene en toda entrada $x \in \{0, 1\}^*$. Escribiremos $\mathcal{M}(x) = 1$ cuando \mathcal{M} acepta la entrada x , y $\mathcal{M}(x) = 0$ cuando la rechaza. Definimos el lenguaje

$$\mathcal{L}(\mathcal{M}) := \{x \in \{0, 1\}^* \mid \mathcal{M}(x) = 1\}.$$

Notemos que dos decisores distintos pueden tener el mismo lenguaje asociado. Por otro lado, existen lenguajes que no son el lenguaje de palabras aceptadas por ningún decisor. Esto último se sigue de un argumento de numerabilidad: no es difícil probar que el conjunto de todas las posibles máquinas de Turing es numerable, mientras que el conjunto de posibles lenguajes binarios no lo es.

La siguiente definición es fundamental para definir las clases de complejidad temporal.

Definición 1.1.2. Todo decisor \mathcal{M} tiene asociada una función $t_{\mathcal{M}} : \mathbb{N} \rightarrow \mathbb{N}$ que indica el tiempo de ejecución de \mathcal{M} en el peor caso: dado un $n \in \mathbb{N}$, $t_{\mathcal{M}}(n)$ se define como el número de pasos de la ejecución más larga entre todas las palabras $x \in \{0, 1\}^*$ con $|x| = n$. Diremos que el decisor \mathcal{M} «trabaja en tiempo polinomial» si existe un $k \in \mathbb{N}$ tal que $t_{\mathcal{M}}(n) \in \mathcal{O}(n^k)$.

Definición 1.1.3. \mathbf{P} es el conjunto de lenguajes $L \subseteq \{0, 1\}^*$ tales que $L = \mathcal{L}(\mathcal{M})$ para algún decisor \mathcal{M} que trabaja en tiempo polinomial.

En otras palabras, que un lenguaje L esté en \mathbf{P} significa que existe un algoritmo que permite decidir si una palabra pertenece a L o no, y que el número de pasos que requiere tal algoritmo (en el peor de los casos) crece como un polinomio en función de la longitud de la palabras que testeamos.

Recordemos que, además de aceptar o rechazar, las máquinas de Turing pueden devolver una palabra. Esta corresponde al contenido que esté escrito en la cinta de salida en el momento en que la ejecución finaliza.

Definición 1.1.4. Decimos que un decisor \mathcal{M} «calcula» una función $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ si $\mathcal{L}(\mathcal{M}) = \{0, 1\}^*$ y, para todo $w \in \{0, 1\}^*$, el contenido de la cinta de salida de \mathcal{M} tras la ejecución de w es exactamente $f(w)$.

La condición de que $\mathcal{L}(\mathcal{M}) = \{0, 1\}^*$ en la definición 1.1.4 es irrelevante: siempre podemos modificar la máquina \mathcal{M} para que todos sus estados de detención sean aceptantes.

Definición 1.1.5. Diremos que una función $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ es «computable en tiempo polinomial» si existe un decisor que la calcula y que trabaja en tiempo polinomial. Al conjunto de funciones computables en tiempo polinomial se le denota \mathbf{FP} .

Notemos que las clases \mathbf{FP} y \mathbf{P} no son comparables, pues la primera consiste en funciones de $\{0, 1\}^*$ en sí mismo, mientras que la segunda consiste en subconjuntos de $\{0, 1\}^*$. Es usual identificar a un lenguaje $L \subseteq \{0, 1\}^*$ con la función $f : \{0, 1\}^* \rightarrow \{0, 1\}$ tal que $f(x) = 1$ si y solo si $x \in L$.

Una propiedad que usaremos frecuentemente es que la clase \mathbf{FP} es cerrada bajo composición de funciones: basta construir una nueva máquina que simule a los decisores de ambas funciones de forma secuencial.

1.2. Codificaciones binarias

Imaginemos que, dado un conjunto de objetos combinatorios¹ Σ , nos interesa poder decidir qué elementos de Σ cumplen cierta propiedad. A esto se le conoce como un «problema de decisión», y se asocia con una función $f : \Sigma \rightarrow \{0, 1\}$. Para poder aplicar la teoría de la complejidad computacional en este contexto, debemos poder escribir f como la composición de una inyección de Σ a $\{0, 1\}^*$ con una función de $\{0, 1\}^*$ a $\{0, 1\}$. La primera puede pensarse como una traducción al alfabeto binario, por lo que debe ser la parte que tome menos tiempo. Formalizaremos esto a continuación.

Definición 1.2.1. Sea Σ un conjunto numerable. Una «codificación» Σ es una inyección $\text{cod} : \Sigma \rightarrow \{0, 1\}^*$ tal que $\text{cod}(\Sigma) \in \mathbf{P}$.

La condición de que $\text{cod}(\Sigma) \in \mathbf{P}$ nos asegura que es posible verificar rápidamente si cierta palabra binaria es o no la codificación válida de algún objeto combinatorio. A una máquina que realice esta tarea la llamaremos un «análizador sintáctico»². Esto es necesario para poder plantear los problemas de decisión que nos interesen en términos de lenguajes.

En lo que respecta a esta tesis, la condición de existencia de un analizador sintáctico de tiempo polinomial no será limitante. Sin embargo, no afirmamos que este sea el caso para todos los objetos combinatorios relevantes. En el caso de querer estudiar algún problema para el que no exista un *parser* polinomial, una posible solución sería asumir que contamos con un oráculo para chequear la sintaxis (ver sección 1.4).

Nuestra intuición de lo que sería una codificación adecuada es que esta debería permitir, en la menor longitud posible, que una máquina pueda decidir preguntas sobre el objeto leyendo directamente la entrada. Consideraremos que una codificación comprende al menos la misma información que otra si podemos computar la segunda en tiempo polinomial a partir de la primera. Formalizaremos esto a continuación:

Definición 1.2.2. Sean cod_1 y cod_2 dos codificaciones de un conjunto Σ . Diremos que cod_2 «comprende polinomialmente» a cod_1 , y escribiremos $\text{cod}_1 \leq_{cp} \text{cod}_2$ si la siguiente función (que llamaremos «función de traducción») pertenece a \mathbf{FP}

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^* , \quad f(x) = \begin{cases} \text{cod}_1 \circ \text{cod}_2^{-1}(x) & \text{si } x \in \text{cod}_2(\Sigma) \\ \epsilon & \text{si no} \end{cases} .$$

Si se cumple además que $\text{cod}_2 \leq_{cp} \text{cod}_1$, escribiremos $\text{cod}_1 \equiv_{cp} \text{cod}_2$ y diremos que son codificaciones «polinomialmente equivalentes». En caso contrario, escribiremos $\text{cod}_1 <_{cp} \text{cod}_2$.

Proposición 1.2.3. Dado un Σ fijo, \leq_{cp} es una relación reflexiva y transitiva en el conjunto de codificaciones de Σ .

Demostración. Para la reflexividad, sea $\text{cod} : \Sigma \rightarrow \{0, 1\}^*$ una codificación. La función de traducción de cod hacia sí misma es

$$f(x) = \begin{cases} x & \text{si } x \in \text{cod}(\Sigma) \\ \epsilon & \text{si no} \end{cases} ,$$

que pertenece a \mathbf{FP} porque $\text{cod}(\Sigma) \in \mathbf{P}$.

¹Esta es una noción imprecisa para referirnos a cualquier colección de objetos discretos que sean finitos al fijar ciertos parámetros, como los grafos o las fórmulas en la lógica proposicional.

²En inglés, «*parser*».

Para la transitividad, consideremos $\text{cod}_1, \text{cod}_2, \text{cod}_3 : \Sigma \rightarrow \{0, 1\}^*$ codificaciones con $\text{cod}_1 \leq_{cp} \text{cod}_2$ y $\text{cod}_2 \leq_{cp} \text{cod}_3$. Sean $f_{2,1}, f_{3,2}$ y $f_{3,1}$ las funciones de traducción de cod_2 a cod_1 , de cod_3 a cod_2 , y de cod_3 a cod_1 , respectivamente. Note que $f_{3,1} = f_{2,1} \circ f_{3,2} \in \mathbf{FP}$, y por lo tanto $\text{cod}_1 \leq_{cp} \text{cod}_3$. \square

Notemos que el hecho de que la función f en la definición 1.2.2 pueda definirse en todo el vocabulario binario depende de que contemos con un analizador sintáctico de tiempo polinomial. En otro caso, solo podríamos pedir que la función con dominio $\text{cod}_1(\Sigma)$ sea computable en tiempo polinomial.

Es sencillo pensar en lo que no sería una buena descripción. Por ejemplo, si Σ es un conjunto infinito numerable, una simple enumeración sería trivialmente inyectiva y *parseable*. Sin embargo, probablemente para poder realizar algún procedimiento efectivo la máquina tendría que conocer cómo funciona la enumeración y reconstruir el objeto en cuestión antes de comenzar a trabajar con él. Ese objeto reconstruido en forma de una palabra binaria sería probablemente un mejor candidato a ser una codificación adecuada.

Algunos problemas tienen instancias que consisten en más de un objeto, y es necesario que una codificación adecuada de dichas instancias permita a la máquina identificar cada uno de ellos. Por ejemplo, si una máquina recibe una cadena de bits que codifica un par de grafos, el programa debe ser capaz de identificar en qué punto termina la codificación del primer grafo y comienza la del segundo. Esto motiva la siguiente definición.

Definición 1.2.4. Diremos que una codificación $\text{cod} : \Sigma \rightarrow \{0, 1\}^*$ es «inequívoca» si, para todo $x \in \text{cod}(\Sigma)$, el único $y \in \{0, 1\}^*$ tal que $xy \in \text{cod}(\Sigma)$ es $y = \epsilon$.

Si tenemos dos codificaciones inequívocas de dos conjuntos Σ_1 y Σ_2 , podremos concatenarlas sin ambigüedad. Veremos que siempre podemos modificar levemente una codificación binaria para garantizar esta propiedad, y con el único costo de aumentar en forma no significativa la longitud de las palabras.

Definición 1.2.5. Dado $n \in \mathbb{N}$, definimos $\text{bin}(n) \in \{0, 1\}^*$ como la expansión binaria de n . En particular, $\text{bin}(0) = \epsilon$. Esta función nos da una inclusión de \mathbb{N} en $\{0, 1\}^*$.

Definición 1.2.6. Dada una palabra $x \in \{0, 1\}^*$, definimos su «prefijo de longitud de entrada binaria» como la palabra binaria $\text{pleb}(x) := 1^{|x|}0y$, donde $y = \text{bin}(|x|)$.

Este prefijo le indica a la máquina la longitud exacta de la palabra que le sigue.

Proposición 1.2.7. Si $\text{cod} : \Sigma \rightarrow \{0, 1\}^*$ es una codificación, entonces $\text{cod}' : \Sigma \rightarrow \{0, 1\}^*$ dada por $\text{cod}'(\sigma) = \text{pleb}(\text{cod}(\sigma))\text{cod}(\sigma)$ es una codificación inequívoca.

Demostración. Sea $x \in \text{cod}'(\Sigma)$ e $y \in \{0, 1\}^*$ tal que $xy \in \text{cod}'(\Sigma)$. Luego existen $\sigma_1, \sigma_2 \in \Sigma$ tales que $x = \text{pleb}(\text{cod}(\sigma_1))\text{cod}(\sigma_1)$ y $xy = \text{pleb}(\text{cod}(\sigma_2))\text{cod}(\sigma_2)$. Entonces

$$1^{|x|} = 1^{|xy|} = 1^{|x|+|y|} = 1^{|x|}1^{|y|},$$

de lo que concluimos que $1^{|y|} = \epsilon$, y luego $y = \epsilon$. \square

Notemos que agregar el prefijo pleb solo aumenta la longitud de una entrada x en $\mathcal{O}(\log_2|x|)$ bits. Con esto, ahora podemos hacer sentido a expresiones de la forma

$$\mathcal{M}(\text{cod}(\sigma_1), \dots, \text{cod}(\sigma_m)),$$

³Recuerde que la notación $1^{|y|}$ se refiere a la palabra de $|y|$ bits iguales a 1.

que representa la aceptación o rechazo de una máquina \mathcal{M} frente una entrada que codifica a los objetos $\sigma_1, \dots, \sigma_m$, respectivamente. Cuando la codificación esté clara por contexto (o bien no sea relevante), escribiremos simplemente $\mathcal{M}(\sigma_1, \dots, \sigma_m)$.

También puede ocurrir que una tupla contenga objetos de distinta naturaleza. En dicho caso, podemos agregar otro prefijo que codifique el tipo de objeto que se codificará a continuación, y que en función de esto la máquina entre a un modo o a otro. En todo caso, ninguno de estos detalles es realmente relevante. Podríamos incluso asumir que las máquinas de Turing cuentan con símbolos especiales para delimitar las entradas y salidas, y esto no alteraría las clases de complejidad.

Veamos algunos ejemplos de codificaciones.

Ejemplo 1.2.8. Sea G un grafo dirigido sin aristas paralelas, aunque permitiendo aristas opuestas. Sea n el número de vértices de G . Una posible codificación sería la matriz de adyacencia⁴ de G , que consistirá de n^2 bits. Si G fuese un grafo no dirigido, ni siquiera sería necesario especificar las entradas de la matriz que están debajo de la diagonal principal. Esto dividiría la longitud de la entrada aproximadamente en un factor de 2. Sin embargo, este cambio no es significativo para nuestros propósitos: la codificación sigue siendo cuadrática en términos del número de vértices.

Ejemplo 1.2.9. Sea G un grafo dirigido sin aristas paralelas, aunque permitiendo aristas opuestas. Sea n el número de vértices de G , y sea m el número de aristas. Otra codificación que podríamos considerar es la matriz de incidencia⁵ de G . Como cada entrada de la matriz puede tomar tres valores posibles, necesitaremos dos bits para representar cada una de ellas. Por lo tanto, la longitud de la codificación en este caso será $2nm$. Como ya mencionamos anteriormente, el factor constante 2 no es relevante para nuestros propósitos. Notemos que m está acotado por $n(n-1) \in \mathcal{O}(n^2)$, por lo que esta codificación es cúbica en términos del número de vértices. Si bien esta cota es peor que la cuadrática del ejemplo 1.2.8, no hará una diferencia cuando estudiemos un problema de grafos respecto a una clase de complejidad que es cerrada bajo transformaciones polinomiales.

Ejemplo 1.2.10. Sea G un grafo dirigido, posiblemente con aristas repetidas. Sea C la entrada de mayor valor en la matriz de adyacencia⁶ de G . Necesitaremos $\lceil \log_2 C \rceil$ bits para representar cada entrada de dicha matriz, por lo que la codificación consistirá de $n^2 \lceil \log_2(C) \rceil$ bits. En este caso sí será necesario especificar el valor de C como un prefijo, pues de lo contrario podría haber ambigüedad para deducir el tamaño de la matriz. De todos modos, este prefijo no cambiará el orden de la codificación: $\mathcal{O}(n^2 \log C)$. Lo que sí es problemático es que, para un n fijo, podríamos tener valores de C arbitrariamente grandes, por lo que no existe una cota para la longitud de esta codificación que dependa solo del número de vértices, como sí ocurre en los ejemplos 1.2.8 y 1.2.9. Lo que podríamos afirmar es que la longitud de la codificación está acotada por un polinomio en $n + m$, donde m es el número de aristas de G .

En los ejemplos 1.2.8, 1.2.9 y 1.2.10 hemos acotado la longitud de la codificación en términos de ciertos parámetros. Podemos formalizar esta idea de la siguiente manera.

⁴La matriz de adyacencia de G es una matriz A de $n \times n$ tal que la entrada $A_{i,j}$ es igual a 1 si existe la arista dirigida desde el vértice v_i al vértice v_j , y es igual a 0 si no.

⁵La matriz de incidencia de G es una matriz A de $n \times m$ tal que la entrada $A_{i,j}$ es igual a 1 si la arista e_j tiene al vértice v_i como vértice final; es igual a -1 si la arista e_j tiene a vértice v_i como vértice inicial, y es igual a 0 en otro caso.

⁶En este caso, la matriz de adyacencia indica cuántas aristas desde el vértice v_i al vértice v_j hay en G .

Definición 1.2.11. Sea Σ un conjunto, $\text{cod} : \Sigma \rightarrow \{0, 1\}^*$ una inyección, k un entero positivo, y $f : \Sigma \rightarrow \mathbb{N}^k$, $g : \mathbb{N}^k \rightarrow \mathbb{N}$ funciones. Diremos que cod está «acotada por $\mathcal{O}(g)$ » en términos de los parámetros de f si existe una constante $c > 0$ y un natural $n_0 \in \mathbb{N}$ tal que, para todo $\sigma \in \Sigma$ tal que cada coordenada de $f(\sigma)$ es mayor o igual a n_0 , se tiene que $|\text{cod}(\sigma)| \leq c \cdot g(f(\sigma))$.

Debemos tener especial cuidado cuando las longitudes de las codificaciones de nuestras instancias dependan de dos o más parámetros. En ese caso, asumir que alguno de ellos está fijo podría cambiar la complejidad del problema. Veamos el siguiente ejemplo.

Ejemplo 1.2.12. Dado un natural $k \geq 3$ fijo, el problema k -CLIQUE consiste en decidir si un grafo G tiene un k -clique⁷. Consideremos el algoritmo de fuerza bruta que revisa todos los posibles subconjuntos de k vértices de G y chequea si estos forman un k -clique o no. Se realizan $\mathcal{O}(n^k)$ verificaciones, y cada una de estas toma $\mathcal{O}(k^2)$ pasos, por lo que la complejidad temporal del algoritmo es $\mathcal{O}(n^k k^2)$. Esto prueba que k -CLIQUE $\in \mathbf{P}$ para todo $k \geq 3$. Ahora consideremos el problema CLIQUE, que consiste en, dada la entrada (G, k) , decidir si G tiene un k -clique. La diferencia entre el problema CLIQUE y la familia de problemas $\{k\text{-CLIQUE}\}_k$ es que en el primero el número k ya no está fijo, sino que es parte de la entrada. En este caso, el algoritmo de tiempo $\mathcal{O}(n^k k^2)$ ya no es polinomial, por lo que no podemos concluir que CLIQUE está en \mathbf{P} .

1.3. Verificadores de tiempo polinomial

En el ejemplo 1.2.12 discutimos que un posible algoritmo para resolver CLIQUE para la instancia (G, k) sería revisar todos los posibles conjuntos de k vértices de G y chequear si inducen un subgrafo completo. Este no es un algoritmo de tiempo polinomial; sin embargo, tiene la particularidad de que, si tenemos *a priori* un candidato a ser el k -clique, entonces podemos chequear ese caso en tiempo polinomial. En otras palabras, lo que hace el algoritmo es transformar el problema en varias verificaciones que tardan un tiempo polinomial, y decidir si alguna de ellas resultó en aceptación. La siguiente clase de complejidad captura a los problemas que tienen una solución de esa naturaleza.

Definición 1.3.1. [AB09, 2.1] Decimos que un lenguaje $L \subseteq \{0, 1\}^*$ pertenece a la clase \mathbf{NP} si existe una función polinomial $p : \mathbb{N} \rightarrow \mathbb{N}$ y un decisor \mathcal{M} (llamado un «verificador» para L) que trabaja en tiempo polinomial y tal que, para todo $x \in \{0, 1\}^*$, se cumple que

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

Si $x \in L$ y $u \in \{0, 1\}^{p(|x|)}$ satisfacen $\mathcal{M}(x, u) = 1$, decimos que u es un «certificado» o «testigo» para x (con respecto al lenguaje L y el verificador \mathcal{M}).

Es importante notar que la definición anterior involucra dos polinomios: uno que acota el número de pasos que el decisor \mathcal{M} tardará en detenerse (en términos de la longitud de la entrada que reciba), y otro que corresponde a la longitud de los posibles certificados.

La definición 1.3.1 puede debilitarse levemente al relajar la condición de que la longitud de los certificados sea exactamente $p(|x|)$. De eso se trata la siguiente proposición, que puede facilitar las demostraciones de pertenencia a la clase \mathbf{NP} de ciertos problemas concretos.

⁷Un « k -clique» de G es un conjunto de k vértices de G que inducen un subgrafo completo.

Proposición 1.3.2. *Un lenguaje $L \subseteq \{0, 1\}^*$ pertenece a **NP** si y solo si existe una función polinomial $p : \mathbb{N} \rightarrow \mathbb{N}$ y un decisor \mathcal{M} que trabaja en tiempo polinomial y tal que, para todo $x \in \{0, 1\}^*$, se cumple que*

$$x \in L \iff \exists u \in \{0, 1\}^* \left(|u| \leq p(|x|) \wedge \mathcal{M}(x, u) = 1 \right).$$

Ejemplo 1.3.3. Veamos que el lenguaje 3COL de los grafos simples que son 3-coloreables pertenece a la clase **NP**. Sea n el número de vértices de G . Un certificado consistirá de una palabra que codifique el color que le corresponda a cada vértice. Necesitamos 2 bits por cada color, así que dicho certificado tendrá longitud $\mathcal{O}(n)$. Por otro lado, la verificación también puede realizarse en tiempo polinomial: podemos recorrer las aristas del grafo y, para cada una de ellas, revisar que los dos vértices que conecta no tengan el mismo color. Este algoritmo claramente puede realizarse en tiempo polinomial, por lo que el problema pertenece a **NP**.

En estricto rigor, para hablar del lenguaje 3COL deberíamos primero establecer una codificación. Sin embargo, como discutimos en la sección 1.2, cualquier codificación adecuada nos permitirá seguir el mismo argumento. Lo importante es que todas las codificaciones que podamos considerar sean polinomialmente equivalentes entre sí (ver definición 1.2.2).

Proposición 1.3.4. **P** \subseteq **NP**.

Demostración. Basta tomar p como el polinomio nulo en la proposición 1.3.2. □

Se desconoce si la inclusión de la proposición 1.3.4 es estricta o no.

La siguiente definición nos permitirá comparar la dificultad de dos problemas de decisión.

Definición 1.3.5. Sean $L_1, L_2 \subseteq \{0, 1\}^*$. Diremos que L_1 es «reducible en tiempo polinomial» a L_2 si existe $f \in \mathbf{FP}$ tal que, para todo $x \in \{0, 1\}^*$, se cumple que $x \in L_1$ si y solo si $f(x) \in L_2$. En dicho caso, decimos que f es una «reducción de Karp» de L_1 a L_2 , y escribimos $L_1 \leq_p L_2$.

La idea de la definición 1.3.5 es que, si tuviésemos un algoritmo para decidir L_2 , entonces inmediatamente tenemos también un algoritmo para decidir L_1 : dado un $x \in \{0, 1\}^*$, basta calcular $f(x)$ y copiar la respuesta del algoritmo para L_2 con la entrada $f(x)$.

Como la clase **FP** es cerrada bajo composición de funciones, tenemos que la relación \leq_p es transitiva.

Proposición 1.3.6. *Sean L_1 y L_2 dos lenguajes tales que $L_1 \leq_p L_2$. Entonces:*

(i) *Si $L_2 \in \mathbf{P}$, entonces $L_1 \in \mathbf{P}$.*

(ii) *Si $L_2 \in \mathbf{NP}$, entonces $L_1 \in \mathbf{NP}$.*

Demostración de (i). Sea $f \in \mathbf{FP}$ una reducción de Karp de L_1 a L_2 con tiempo de computación $\mathcal{O}(h)$, donde $h : \mathbb{N} \rightarrow \mathbb{N}$ es una función polinomial. Sea \mathcal{M} un decisor para L_2 que trabaja en tiempo polinomial $\mathcal{O}(p)$. Ahora consideramos un decisor \mathcal{M}' que, con entrada $x \in \{0, 1\}^*$, calcula $f(x)$, simula \mathcal{M} con entrada $f(x)$ y responde igual que \mathcal{M} . Tenemos que $\mathcal{L}(\mathcal{M}') = L_1$, pues

$$x \in L_1 \iff f(x) \in L_2 \iff \mathcal{M}(f(x)) = 1 \iff \mathcal{M}'(x) = 1 \iff x \in \mathcal{L}(\mathcal{M}').$$

Veamos que \mathcal{M}' también trabaja en tiempo polinomial. Notemos que $|f(x)| = \mathcal{O}(h(|x|))$, pues una máquina no puede imprimir una palabra de longitud mayor que el número de pasos de la ejecución. Por lo tanto, el tiempo de ejecución de \mathcal{M}' con entrada x es a lo más $\mathcal{O}(h(|x|) + p(h(|x|)))$. Como $h + (p \circ h)$ es un polinomio, esto prueba que $L_1 \in \mathbf{P}$. \square

Demostración de (ii). Sea $f \in \mathbf{FP}$ una reducción de Karp de L_1 a L_2 con tiempo de computación $\mathcal{O}(h)$, donde $h : \mathbb{N} \rightarrow \mathbb{N}$ es una función polinomial. Sea \mathcal{M} un verificador polinomial para L_2 , y sea $p : \mathbb{N} \rightarrow \mathbb{N}$ la función polinomial correspondiente a las longitudes de los certificados para \mathcal{M} . Tendremos que $x \in L_1$ si y solo si $f(x) \in L_2$, lo que a su vez equivale a que exista un certificado $u \in \{0, 1\}^{p(|f(x)|)}$ tal que $\mathcal{M}(f(x), u) = 1$. Por lo tanto, podemos considerar un verificador para L_1 que, dada la entrada x , primero computa $f(x)$ y luego verifica de forma idéntica a \mathcal{M} . Como $|f(x)| = \mathcal{O}(h(|x|))$, los certificados para $f(x)$ también estarán acotados por un polinomio en $|x|$. Por lo tanto, concluimos que $L_1 \in \mathbf{NP}$. \square

A continuación definiremos la noción de **NP-completitud**, que pretende capturar la idea de que un problema sea un representante de la dificultad de su clase.

Definición 1.3.7. Diremos que un lenguaje L es «**NP-difícil**»⁸ si $L' \leq_p L$ para todo $L' \in \mathbf{NP}$. Si además $L \in \mathbf{NP}$, diremos que L es **NP-completo**.

Notemos que, en virtud de la proposición 1.3.6, la existencia de un algoritmo polinomial para un problema **NP-difícil** implicaría que $\mathbf{P} = \mathbf{NP}$. Como se cree que esta última igualdad es falsa, lo usual es razonar en el sentido inverso: una demostración de la **NP-dificultad** de un problema suele interpretarse como evidencia de que probablemente dicho problema no está en \mathbf{P} . En ese sentido, podemos considerar que los problemas **NP-completos** son los más difíciles de la clase **NP**.

La siguiente proposición es una consecuencia inmediata de la transitividad de las reducciones de Karp.

Proposición 1.3.8. Sean L_1 y L_2 dos lenguajes tales que $L_1 \leq_p L_2$. Si L_2 es **NP-completo**, entonces L_1 también lo es.

El problema **NP-completo** más conocido es SAT, que consiste en decidir si una fórmula de la lógica proposicional es satisfacible. Al resultado de la **NP-completitud** de SAT se le conoce como el teorema de Cook-Levin. Mediante reducciones de Karp se puede propagar esa completitud hacia otros problemas. Por ejemplo, los problemas CLIQUE (ejemplo 1.2.12) y 3COL (ejemplo 1.3.3) son también **NP-completos**.

1.4. Máquinas oráculo

Las máquinas oráculo son útiles cuando buscamos entender la complejidad computacional de un problema A , pero bajo el supuesto de que somos capaces de resolver eficientemente otro problema B . En otras palabras, asumiendo que tenemos un algoritmo para resolver B , estudiamos cómo podemos aprovecharlo al máximo para construir un algoritmo para A .

Definición 1.4.1 (Oráculos de decisión). Sea $L \subseteq \{0, 1\}^*$ un lenguaje. Decimos que \mathcal{M} es una «máquina de Turing con acceso a un oráculo para L » si es una máquina de Turing

⁸En inglés, «**NP-hard**»

equipada con una cinta especial, llamada «cinta de consulta» (distinta a las cintas de trabajo, de entrada y de salida), y tres estados especiales: $q_?$, $q_{sí}$ y q_{no} . El funcionamiento de \mathcal{M} es idéntico al de una máquina de Turing tradicional, excepto cuando entra al estado $q_?$. En dicho caso, en un solo paso la máquina cambia al estado $q_{sí}$ o q_{no} , dependiendo de si lo que esté escrito en la cinta de consulta sea una palabra en L o no, respectivamente.

Ejemplo 1.4.2. Consideremos el problema TAU de decidir si una fórmula de la lógica proposicional es una tautología. Se cree que $\text{TAU} \notin \mathbf{NP}$, por lo que es aún más improbable que pertenezca en \mathbf{P} . Sin embargo, podemos resolver este problema en tiempo polinomial si contamos con un oráculo para SAT. Supongamos que queremos decidir si $\varphi \in \text{TAU}$. Lo que hacemos es escribir $\neg\varphi$ en la cinta del oráculo, y la respuesta al problema será la negación de la respuesta del oráculo. En efecto, tenemos que φ es una tautología si y solo si $\neg\varphi$ no es satisficible, por lo que debemos aceptar φ si y solo si el oráculo rechaza $\neg\varphi$.

También existen oráculos para funciones binarias, como veremos a continuación.

Definición 1.4.3 (Oráculos funcionales). Sea $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ una función. Decimos que \mathcal{M} es una «máquina de Turing con acceso a un oráculo para f » si es una máquina de Turing equipada con dos cintas especiales adicionales, llamadas «cinta de consulta» y «cinta de respuesta», y dos estados especiales, llamados «estado de consulta» y «estado de respuesta». El funcionamiento de \mathcal{M} es idéntico al de una máquina de Turing tradicional, excepto cuando entra al estado de consulta. En dicho caso, si x es la palabra que está escrita en la cinta de consulta, entonces la máquina en un solo paso cambia al estado de respuesta y escriba en la cinta de respuesta la palabra $f(x)$.

La definición de oráculos funcionales generaliza a la de oráculos de decisión, pues un lenguaje puede entenderse como una función $\{0, 1\}^* \rightarrow \{0, 1\} \subseteq \{0, 1\}^*$. Teniendo esto en cuenta, continuaremos este capítulo considerando solo oráculos funcionales.

Según la definición 1.4.3, la máquina es capaz de escribir $f(x)$ en un solo paso en la cinta de respuesta, independientemente de la longitud de dicha palabra. Sin embargo, para que un algoritmo pueda hacer uso de esa respuesta, necesariamente deberá acceder al contenido de la cinta para leerla, y esa acción sí será sensible a la longitud de $f(x)$.

Podríamos agregar la condición de que la máquina escriba el prefijo $\text{pleb}(f(x))$ (ver definición 1.2.6) antes de cada respuesta al oráculo. Esto último sería para tener una forma de asegurarnos de poder leer la respuesta hasta su final. Sin embargo, como ya discutimos en la sección 1.2, estos detalles no son relevantes para el análisis de complejidad.

Definición 1.4.4. Dada una función $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, definimos las siguientes clases:

- \mathbf{P}^f es el conjunto de lenguajes decidibles en tiempo polinomial por una máquina de Turing con acceso a un oráculo para f .
- \mathbf{FP}^f es el conjunto de funciones de $\{0, 1\}^*$ en $\{0, 1\}^*$ computables en tiempo polinomial por una máquina de Turing con acceso a un oráculo para f .

La siguiente proposición nos dice que contar con un oráculo para una subrutina de tiempo polinomial no altera las clases \mathbf{P} y \mathbf{FP} .

Proposición 1.4.5.

- Si $L \in \mathbf{P}$, entonces $\mathbf{P}^L = \mathbf{P}$.
- Si $f \in \mathbf{FP}$, entonces $\mathbf{FP}^f = \mathbf{FP}$.

Ejemplo 1.4.6. Consideremos el lenguaje L de las fórmulas en lógica proposicional que son satisfechas por al menos dos valuaciones distintas. Por otro lado, consideremos una función $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ que, dada la codificación de una fórmula φ , devuelve la codificación de alguna valuación que satisface φ en caso de que φ sea satisfacible, y devuelve ϵ en otro caso. Veamos que $L \in \mathbf{P}^f$. Sea φ una fórmula en lógica proposicional, con conjunto (finito) de variables X . Queremos decidir si $\varphi \in L$ usando el oráculo para f . Lo primero que haremos es usar el oráculo para calcular $f(\varphi)$. Si $f(\varphi) = \epsilon$, ya sabemos que $\varphi \notin L$. Supongamos entonces que $f(\varphi) \neq \epsilon$, de modo que tenemos la codificación de una valuación $v : X \rightarrow \{0, 1\}$ que satisface φ . Sean $\{y_1, \dots, y_m\} \subseteq X$ las variables de X que son verdaderas bajo la valuación v . Ahora definimos

$$\psi = \varphi \wedge (\neg y_1 \vee \neg y_2 \vee \dots \vee \neg y_m).$$

Notemos que $f(\psi) = \epsilon$ si y solo si φ no es satisfecha por ninguna valuación salvo v , por lo que tendremos que $\varphi \in L$ si y solo si $f(\psi) \neq \epsilon$. Este algoritmo es de tiempo polinomial, por lo que $L \in \mathbf{P}^f$.

1.5. La complejidad de contar: La clase $\#\mathbf{P}$

Supongamos que nos interesa contar el número de valuaciones que satisfacen una fórmula de la lógica proposicional. Llamemos $\#\text{SAT}$ a este problema. Podemos pensar que esto corresponde a computar una función de $\{0, 1\}^*$ en sí mismo, donde las salidas son codificaciones de un número natural. Este es nuestro primer ejemplo de un problema de conteo.

Más generalmente, dado un conjunto de objetos combinatorios Σ , un «problema de conteo» será una función de Σ en \mathbb{N} . Notemos que todos los problemas \mathbf{NP} -completos que hemos discutido tienen naturalmente asociado un problema de conteo. Por ejemplo, podemos considerar el problema $\#\text{3COL}$, que consiste en contar las 3-coloraciones de un grafo simple.

Formalizaremos esta idea a continuación. Lo primero que debemos hacer es relacionar los problemas de conteo con las clases de complejidad funcionales.

Definición 1.5.1. Tenemos la siguiente sobreyección $\text{num} : \{0, 1\}^* \rightarrow \mathbb{N}$: dado cualquier $x \in \{0, 1\}^*$, definimos $\text{num}(x)$ como el único natural $n \in \mathbb{N}$ tal que $x = 0^k \text{bin}(n)$ para algún $k \in \mathbb{N}$.

Notemos que, en el contexto de un problema de conteo, podemos identificar los conjuntos $\{0, 1\}^*$ y \mathbb{N} a través de las funciones bin y num . Esta identificación nos hace perder la información sobre los bits 0 que están a la izquierda, pero estos son irrelevantes cuando la palabra codifica un número natural.

La siguiente clase captura a los problemas de conteo que están naturalmente asociados a un problema de decisión en \mathbf{NP} .

Definición 1.5.2. Decimos que una función $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ pertenece a la clase⁹ $\#\mathbf{P}$ si existe una función polinomial $p : \mathbb{N} \rightarrow \mathbb{N}$ y un decisor \mathcal{M} que trabaja en tiempo polinomial y tal que, para todo $x \in \{0, 1\}^*$, se cumple que

$$\text{num}(f(x)) = \#\left\{u \in \{0, 1\}^{p(|x|)} \mid \mathcal{M}(x, u) = 1\right\}.$$

En otras palabras, las funciones en $\#\mathbf{P}$ cuentan el número de certificados para cada instancia de un problema en \mathbf{NP} . Se sigue directamente de la definición que $\#\text{SAT}$ y $\#\text{3COL}$ pertenecen a $\#\mathbf{P}$.

⁹Léase «numeral-P», o «sharp-P» en inglés.

En la definición 1.5.2 tendremos que $\text{num}(f(x)) \leq 2^{p(|x|)}$, por lo que, si ignoramos los bits 0 a la izquierda, entonces $f(x)$ consiste de a lo más $p(|x|) = \text{poly}(|x|)$ bits. Resumiremos esta observación en la siguiente definición.

Definición 1.5.3. Diremos que $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ tiene «longitud polinomialmente acotada» si existe un polinomio $p : \mathbb{N} \rightarrow \mathbb{N}$ tal que $|f(x)| < p(|x|)$ para todo $x \in \{0, 1\}^*$.

Como ya comentamos, todas las funciones en $\#\mathbf{P}$ tienen longitudes polinomialmente acotadas. Necesitaremos esta condición en la sección 1.6.

Todos los problemas de conteo tienen naturalmente un lenguaje asociado, como definimos a continuación.

Definición 1.5.4 (Lenguaje asociado a un problema de conteo). Dada $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, definimos el lenguaje

$$\mathcal{L}_f := \left\{ x \in \{0, 1\}^* \mid \text{num}(f(x)) > 0 \right\}.$$

La siguiente proposición se sigue directamente de las definiciones.

Proposición 1.5.5. Si $f \in \#\mathbf{P}$, entonces $\mathcal{L}_f \in \mathbf{NP}$.

Podemos usar la proposición 1.5.5 como un criterio necesario para chequear la pertenencia de una función a la clase $\#\mathbf{P}$. Tenemos un resultado análogo para las clases de tiempo polinomial:

Proposición 1.5.6. Si $f \in \mathbf{FP}$, entonces $\mathcal{L}_f \in \mathbf{P}$.

Demostración. Sea \mathcal{M} un decisor de tiempo polinomial que computa f . Entonces un algoritmo de tiempo polinomial para decidir si $x \in \mathcal{L}_f$ consiste en calcular $f(x)$ y decidir si dicha palabra contiene un bit 1 o no. \square

Notemos que, a diferencia de la contención $\mathbf{P} \subseteq \mathbf{NP}$, la siguiente proposición no se sigue inmediatamente de las definiciones.

Proposición 1.5.7. Se cumple que $\mathbf{FP} \subseteq \#\mathbf{P}$. La igualdad implicaría que $\mathbf{P} = \mathbf{NP}$.

Demostración. Para la primera parte, sea $f \in \mathbf{FP}$ vía un decisor \mathcal{M} de tiempo polinomial. Modificamos dicha máquina para que reciba una entrada auxiliar u de un largo mayor o igual al número de bits necesarios para codificar al natural $\text{num}(f(x))$. Esta nueva máquina, que llamaremos \mathcal{M}' , aceptará todos los pares (x, u) tales que $\text{num}(u) < \text{num}(f(x))$, y claramente sigue siendo de tiempo polinomial. Luego $f \in \#\mathbf{P}$ vía \mathcal{M}' .

Para la segunda parte, basta notar que si, $\mathbf{FP} = \#\mathbf{P}$, entonces $\#\text{SAT} \in \mathbf{FP}$. Luego, por la proposición 1.5.6, tendríamos que $\text{SAT} = \mathcal{L}_{\#\text{SAT}} \in \mathbf{P}$. Como SAT es \mathbf{NP} -completo, esta última pertenencia implicaría inmediatamente que $\mathbf{P} = \mathbf{NP}$. \square

Se cree que la igualdad $\mathbf{FP} = \#\mathbf{P}$ es más fuerte que $\mathbf{P} = \mathbf{NP}$; hasta la fecha, no se ha demostrado que la segunda igualdad implique la primera.

A continuación veremos un ejemplo de un problema de conteo que probablemente no pertenece a \mathbf{FP} , pero cuyo problema de decisión asociado sí pertenece a \mathbf{P} . Esto muestra que el paralelismo entre las clases $\mathbf{P} \subseteq \mathbf{NP}$ y $\mathbf{FP} \subseteq \#\mathbf{P}$ no es exacto: existen problemas de conteo difíciles cuyos lenguajes asociados pueden decidirse en tiempo polinomial.

Supongamos que nos interesa contar el número de ciclos dirigidos (inyectivos en vértices) de un grafo simple y dirigido G . Llamemos $\#\text{CICLOS}$ a este problema. Notemos que el lenguaje asociado, que corresponde a decidir la existencia de un ciclo dirigido, pertenece a \mathbf{P} . En efecto, basta realizar una búsqueda en profundidad en cada componente conexas, y chequear que el algoritmo no detecte aristas que apunten hacia vértices que ya fueron visitados. Este algoritmo es polinomial en términos de la codificación del grafo. Podríamos esperar, entonces, que $\#\text{CICLOS}$ perteneciera a \mathbf{FP} , por lo que la siguiente proposición puede resultar sorprendente. La demostración es una leve modificación de la presentada en [AB09, 17.4].

Proposición 1.5.8. *Si $\#\text{CICLOS} \in \mathbf{FP}$, entonces $\mathbf{P} = \mathbf{NP}$.*

Demostración. Supongamos que contamos con una máquina \mathcal{M} que resuelve el problema $\#\text{CICLOS}$ en tiempo polinomial. Veremos que esto implicaría la existencia de un algoritmo polinomial para el problema \mathbf{NP} -completo HAM de decidir si un grafo dirigido tiene un ciclo hamiltoniano¹⁰. Más precisamente, lo que haremos es reducir HAM al siguiente lenguaje:

$$\mathcal{L} = \left\{ \text{cod}(G) \mid G \text{ es un grafo dirigido con } n \text{ vértices y al menos } n^{n^2} \text{ ciclos} \right\}.$$

A pesar de que parece demasiado grande, podemos escribir el número n^{n^2} en binario usando $\lceil n^2 \log_2 n \rceil = \text{poly}(n)$ bits, por lo que $\mathcal{L} \in \mathbf{P}$ vía \mathcal{M} .

A continuación detallaremos la reducción. Sea G_0 un grafo dirigido con n_0 vértices. Sea n_1 la menor potencia de 2 tal que $n_0 \leq n_1$. Si $n_0 = n_1$, no hacemos nada y definimos $G_1 := G_0$. En otro caso, elegimos cualquier arista de G_0 y la subdividimos hasta agregar $k := n_1 - n_0$ vértices, obteniendo un nuevo grafo G_1 con n_1 vértices. Más precisamente, lo que hacemos es suprimir una arista (a, c) y luego agregar vértices b_1, b_2, \dots, b_k y aristas $(a, b_1), (b_1, b_2), \dots, (b_{k-1}, b_k), (b_k, c)$. Claramente $G_0 \in \text{HAM}$ si y solo si $G_1 \in \text{HAM}$, y este paso no es computacionalmente significativo. Definimos $m := n_1 \log_2 n_1$.

Ahora construiremos otro grafo G_2 reemplazando cada arista (u, v) en G_1 por el artefacto¹¹ ilustrado en la figura 1.

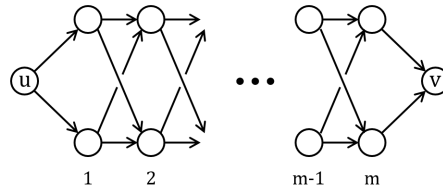


Figura 1: Artefacto que reemplaza a la arista (u, v) .

Afirmamos que $G_1 \in \text{HAM}$ si y solo si $G_2 \in \mathcal{L}$. Notemos que el artefacto es acíclico, que hay 2^m formas de recorrerlo, y que un ciclo que entra en un artefacto necesariamente debe terminar de recorrerlo. De esta forma, a cada ciclo en G_2 le corresponde un único ciclo en G_1 , y cada ciclo en G_1 de largo ℓ da lugar a $(2^m)^\ell$ ciclos en G_2 .

¹⁰Un ciclo hamiltoniano es un ciclo que pasa exactamente una vez por cada vértice del grafo. La existencia de un ciclo hamiltoniano dirigido es uno de los 21 problemas cuya \mathbf{NP} -completitud fue demostrada en 1972 por Richard Karp [Kar72].

¹¹En inglés, *gadget*.

Supongamos que $G_1 \in \text{HAM}$. Entonces G_1 tiene un ciclo de largo n_1 , y por lo tanto G_2 tiene al menos

$$(2^m)^{n_1} = (n_1^{n_1})^{n_1} = n_1^{n_1^2}$$

ciclos. Luego $G_2 \in \mathcal{L}$.

Ahora supongamos que $G_1 \notin \text{HAM}$, de modo que el mayor largo posible de un ciclo de G_1 es $n_1 - 1$. El número de ciclos de G_1 está trivialmente acotado superiormente por $n_1^{n_1 - 1}$, por lo que el total de ciclos de G_2 está acotado superiormente por

$$n_1^{n_1 - 1} \cdot (2^m)^{n_1 - 1} = n_1^{n_1 - 1} \cdot (n_1^{n_1})^{n_1 - 1} = n_1^{n_1^2 - 1} < n_1^{n_1^2}.$$

Luego $G_2 \notin \mathcal{L}$.

Por último, veamos que la reducción es polinomial. G_1 tiene a lo sumo n_1^2 aristas, y cada artefacto agrega $2m$ vértices. Por lo tanto, la transformación agrega a lo sumo $n_1^3 \log_2 n_1 = \text{poly}(n_1)$ vértices. □

1.6. #P-completitud

En esta sección estudiaremos una noción de completitud para la clase #P.

Definición 1.6.1. Una función $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ es «#P-difícil» si $\#P \subseteq \mathbf{FP}^f$. Si además se tiene que $f \in \#P$, se dice que el problema asociado a f es «#P-completo».

Según la definición 1.6.1, una función se considera completa para la clase #P si un oráculo para ella permitiría resolver en tiempo polinomial cualquier otro problema en #P. La siguiente definición nos permitirá comparar la dificultad de calcular dos funciones. Nos restringiremos a funciones polinomialmente acotadas para asegurarnos de que las salidas de los oráculos sean manejables.

Definición 1.6.2. Sean $f, g : \{0, 1\}^* \rightarrow \{0, 1\}^*$ funciones con longitudes polinomialmente acotadas. Escribiremos $f \leq_{\text{FP}} g$ si $f \in \mathbf{FP}^g$. Escribiremos $f \equiv_{\text{FP}} g$ si $f \leq_{\text{FP}} g$ y $g \leq_{\text{FP}} f$.

Proposición 1.6.3. [Val79b] Sean $f, g, h : \{0, 1\}^* \rightarrow \{0, 1\}^*$ funciones con longitudes polinomialmente acotadas. Si $f \in \mathbf{FP}^g$ y $g \in \mathbf{FP}^h$, entonces $f \in \mathbf{FP}^h$. En otras palabras, \leq_{FP} es una relación transitiva.

La transitividad de la relación \leq_{FP} nos indica que es posible propagar la #P-dificultad desde una función particular hacia otra, igual que como ocurre con la NP-dificultad.

Corolario 1.6.4. Sean $f, g : \{0, 1\}^* \rightarrow \{0, 1\}^*$ funciones con longitudes polinomialmente acotadas. Si $f \leq_{\text{FP}} g$ y f es #P-difícil, entonces g también es #P-difícil.

Demostración. Sea $h \in \#P$. Como f es #P-difícil, tenemos que $\#P \subseteq \mathbf{FP}^f$ y, en particular, $h \in \mathbf{FP}^f$. Como $f \in \mathbf{FP}^g$, por la proposición 1.6.3 deducimos que $h \in \mathbf{FP}^g$. Como $h \in \#P$ era arbitrario, concluimos que $\#P \subseteq \mathbf{FP}^g$, y g es #P-difícil. □

La transitividad de \leq_{FP} también implica la segunda parte de la proposición 1.4.5, que decía que $\mathbf{FP}^g = \mathbf{FP}$ para toda $g \in \mathbf{FP}$. En efecto, dada cualquier $f \in \mathbf{FP}^g$, podemos concluir que $f \in \mathbf{FP}$ usando la proposición 1.6.3 (considerando h como la función identidad en $\{0, 1\}^*$, de modo que trivialmente $\mathbf{FP}^h = \mathbf{FP}$). La otra dirección es inmediata.

Como consecuencia de estos hechos, tenemos también el siguiente corolario.

Corolario 1.6.5. *Si existe una función $f \in \mathbf{FP}$ que es $\#\mathbf{P}$ -completa, entonces $\mathbf{FP} = \#\mathbf{P}$.*

Demostración. Como f es $\#\mathbf{P}$ -difícil, tenemos que $\#\mathbf{P} \subseteq \mathbf{FP}^f = \mathbf{FP}$. \square

Se cree que $\mathbf{FP} \neq \#\mathbf{P}$, pues lo contrario implicaría la dudosa igualdad $\mathbf{P} = \mathbf{NP}$. Bajo esas suposiciones, el corolario 1.6.5 indica que la $\#\mathbf{P}$ -completitud de una función implica la imposibilidad de que esta sea calculada por algún algoritmo de tiempo polinomial.

Se conocen varios problemas de conteo $\#\mathbf{P}$ -completos, entre los que está $\#\text{SAT}$. Sin embargo, esto último podría resultar esperable, pues su problema de decisión asociado es \mathbf{NP} -completo. Lo que es más interesante es que existen problemas de conteo difíciles (según esta noción de completitud) cuyos problemas de decisión asociados pertenecen a \mathbf{P} . En breve describiremos un ejemplo de este fenómeno.

Leslie G. Valiant definió en 1979 la clase $\#\mathbf{P}$ con el objetivo de demostrar la dificultad de calcular permanentes de matrices, que definiremos a continuación.

Definición 1.6.6. Sea A una matriz de $n \times n$ con entradas en cierto anillo. Su «permanente» se define como

$$\text{perm}(A) := \sum_{\sigma \in \mathcal{S}_n} \prod_{i=1}^n A_{i, \sigma(i)}.$$

Esta es casi la misma definición que la del determinante, pero omitiendo los signos.

Teorema 1.6.7 (Teorema de Valiant). [*Val79a*] *Calcular permanentes de matrices con entradas en $\{0, 1\} \subseteq \mathbb{Z}$ es un problema $\#\mathbf{P}$ -completo.*

El teorema 1.6.7 contrasta con el hecho de que existen algoritmos de tiempo polinomial para calcular determinantes. A continuación, usaremos el teorema de Valiant para exhibir un ejemplo de un problema de conteo difícil cuyo problema de decisión asociado es tratable en tiempo polinomial.

Definición 1.6.8. Sea $B = (U, V, E)$ un grafo bipartito, donde $E \subseteq U \times V$ y $|U| = |V| = n$. Definimos un «emparejamiento (perfecto)»¹² como un $M \subseteq E$ con $|M| = n$ y tal que, si (u, v) y (u', v') son elementos distintos en M , entonces $u \neq u'$ y $v \neq v'$. Al problema de decidir si un grafo bipartito dado tiene un emparejamiento se le denota MATCHING , y al problema de conteo asociado se le denota $\#\text{MATCHINGS}$.

Proposición 1.6.9. [*Pap94*, p. 12] $\text{MATCHING} \in \mathbf{P}$.

Proposición 1.6.10. $\#\text{MATCHINGS}$ es $\#\mathbf{P}$ -completo.

Demostración. La pertenencia a $\#\mathbf{P}$ es clara. Por el teorema 1.6.7, bastaría con probar que un algoritmo polinomial para $\#\text{MATCHINGS}$ implicaría la existencia de un algoritmo de tiempo polinomial para calcular permanentes de matrices con entradas en $\{0, 1\} \subseteq \mathbb{Z}$. Sea $A = (a_{i,j})_{i,j \in [n]}$ una matriz de $n \times n$ con entradas en $\{0, 1\}$. Consideremos un grafo bipartito simple $B = (U, V, E)$, donde $E \subseteq U \times V$, $U = \{u_1, \dots, u_n\}$, $V = \{v_1, \dots, v_n\}$ y $(u_i, v_j) \in E$ si y solo si $a_{i,j} = 1$. Entonces el número de emparejamientos perfectos de B es exactamente $\text{perm}(A)$. Como la construcción del grafo bipartito puede hacerse en tiempo polinomial, esto prueba que $\#\text{MATCHINGS}$ es $\#\mathbf{P}$ -completo. \square

¹²En inglés, «(perfect) matching».

1.7. Reducciones parsimoniosas y #P-completitud fuerte

Sabemos que hay problemas #P-completos cuyos problemas de decisión pertenecen a P. Una pregunta natural, entonces, es si existe algún resultado en la otra dirección. Podríamos esperar que la siguiente afirmación fuese un teorema:

Sea $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ una función en la clase #P. Si \mathcal{L}_f es NP-completo, entonces f es #P-completa.

Hasta donde llega nuestro conocimiento, no existen contraejemplos a la afirmación anterior; pero tampoco ha sido demostrada en general (en [CH96] se presenta una demostración de la conjetura para problemas de satisfacibilidad generalizados).

Hay ocasiones en que de la misma demostración de que cierto problema de decisión es NP-completo se desprende que el problema de conteo asociado es #P-completo. Esto ocurre cuando la NP-dificultad se prueba por medio de una reducción de Karp que además preserva el número de certificados para los verificadores. Veamos la siguiente definición:

Definición 1.7.1. Dadas $f, g: \{0, 1\}^* \rightarrow \{0, 1\}^*$, decimos que $\sigma: \{0, 1\}^* \rightarrow \{0, 1\}^*$ es una «reducción parsimoniosa» de f a g si $\sigma \in \mathbf{FP}$ y además

$$\forall x \in \{0, 1\}^* \quad \text{num}(f(x)) = \text{num}(g(\sigma(x))).$$

Note que una reducción parsimoniosa de f a g induce una reducción de Karp de \mathcal{L}_f a \mathcal{L}_g . El recíproco no es cierto.

Si tenemos una reducción parsimoniosa de f a g , y contamos con un oráculo para g , entonces podremos calcular f en tiempo polinomial. Enunciamos eso en la siguiente proposición.

Proposición 1.7.2. Sean $f, g: \{0, 1\}^* \rightarrow \{0, 1\}^*$ funciones con longitudes polinomialmente acotadas. Si existe una reducción parsimoniosa de f a g , entonces $f \leq_{\mathbf{FP}} g$.

Se puede definir una noción de completitud en #P por medio de reducciones parsimoniosas, de forma similar a como se define la propiedad de NP-completitud.

Definición 1.7.3. Una función $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ se dice «fuertemente #P-completa» si $f \in \#P$ y, para todo $g \in \#P$ existe una reducción parsimoniosa de g a f .

Ambas nociones de completitud en #P no son equivalentes, a menos que $\mathbf{P} = \mathbf{NP}$. De hecho, como las reducciones parsimoniosas son, en particular, reducciones de Karp, tenemos el siguiente resultado.

Proposición 1.7.4. Si f es fuertemente #P-completa, entonces \mathcal{L}_f es NP-completo.

La mayoría de los problemas NP-completo conocidos tienen variantes de conteo fuertemente #P-completas.

Durante esta tesis, preferiremos la noción de #P-completitud vía oráculos funcionales: consideramos que captura más naturalmente la idea de dificultad para problemas de conteo.

2. Preliminares de complejos simpliciales

Las secciones 2.1 y 2.2 serán un breve resumen de algunas definiciones básicas. Nuestra referencia será el capítulo 13 del libro *Algebraic combinatorics* de Richard Stanley [Sta18a].

2.1. Definiciones básicas sobre complejos simpliciales

Durante esta sección, E será un conjunto finito, que llamaremos «conjunto subyacente», y a cuyos elementos llamaremos «vértices».

Definición 2.1.1. Diremos que $\Delta \subseteq 2^E$ es un «complejo simplicial sobre E » si para todo par de conjuntos $A, B \in 2^E$ tales que $A \subseteq B$ y $B \in \Delta$ se cumple que $A \in \Delta$. A los elementos de Δ se les llama «caras», y a las caras maximales por contención se les llama «facetas». La «dimensión» de una cara $A \in \Delta$ se define como $\dim(A) := |A| - 1$. La «dimensión» de Δ , denotada $\dim(\Delta)$, se define como el máximo de las dimensiones de sus caras.

Notemos que, si Δ es un complejo simplicial no vacío, entonces $\emptyset \in \Delta$ y $\dim(\emptyset) = -1$.

Definición 2.1.2. Dado un complejo simplicial $\Delta \subseteq 2^E$ de dimensión d e $i \in \{-1, 0, 1, \dots, d\}$, denotamos por $f_i(\Delta)$ al número de caras de dimensión i en Δ . Definimos el «f-vector» de Δ como la siguiente tupla:

$$f(\Delta) := (f_{-1}(\Delta), f_0(\Delta), f_1(\Delta), \dots, f_d(\Delta)).$$

Definición 2.1.3. Sean Δ_1 y Δ_2 dos complejos simpliciales sobre E_1 y E_2 , respectivamente. Diremos que Δ_1 y Δ_2 son «isomorfos» si existe una biyección entre E_1 y E_2 que induce una biyección entre Δ_1 y Δ_2 . Más precisamente, requerimos que exista una biyección $\psi : E_1 \rightarrow E_2$ tal que, para todo $X \subseteq E_1$, se cumpla que $X \in \Delta_1$ si y solo si $\psi(X) \in \Delta_2$.

Como todas las propiedades combinatoriales de los complejos simpliciales se conservan bajo isomorfismos, es usual fijar como conjunto subyacente a algún $[n]$ con $n \in \mathbb{N}$.

Definición 2.1.4. Se dice que un complejo simplicial Δ es «puro» si todas sus facetas tienen dimensión $\dim(\Delta)$.

Proposición 2.1.5. Si Δ_1 y Δ_2 son dos complejos simpliciales sobre el mismo conjunto subyacente E , entonces $\Delta_1 \cap \Delta_2$ también es un complejo simplicial.

Un complejo simplicial queda totalmente determinado por el conjunto de sus facetas. Esto motiva la siguiente definición:

Definición 2.1.6. Dado $\Gamma \subseteq 2^E$, definimos $\langle \Gamma \rangle$, el «complejo simplicial generado por Γ », como el menor complejo simplicial que contiene a Γ , es decir:

$$\langle \Gamma \rangle := \left\{ F \in 2^E \mid \exists G \in \Gamma : F \subseteq G \right\}.$$

En particular, se define $\langle \emptyset \rangle = \emptyset$. Si $\Gamma = \{A_1, \dots, A_t\}$, escribiremos $\langle A_1, \dots, A_t \rangle$ en lugar de $\langle \{A_1, \dots, A_t\} \rangle$.

En virtud de la proposición 2.1.5, también podríamos definir $\langle \Gamma \rangle$ como la intersección de todos los complejos simpliciales sobre E que contienen a cada elemento de Γ .

Definición 2.1.7. Sea Δ un complejo simplicial sobre E . Considerando un nuevo vértice v_0 que no está en E , llamado «ápice», podemos definir el «cono» $\Delta * \{v_0\}$ como el complejo simplicial sobre $E \cup \{v_0\}$ dado por

$$\Delta * \{v_0\} := \Delta \cup \{A \cup \{v_0\} \mid A \in \Delta\}.$$

Note que $\Delta * \{v_0\}$ es el complejo simplicial generado por los elementos de la forma $F \cup \{v_0\}$, donde F es una faceta de Δ .

Las siguientes dos proposiciones se siguen directamente de la definición 2.1.7.

Proposición 2.1.8. Si Δ es un complejo simplicial de dimensión d , entonces:

- $f_{-1}(\Delta * \{v_0\}) = f_{-1}(\Delta)$.
- Si $i \in [0, 1, \dots, d]$, entonces $f_i(\Delta * \{v_0\}) = f_{i-1}(\Delta) + f_i(\Delta)$.
- $f_{d+1}(\Delta * \{v_0\}) = f_d(\Delta)$.

Proposición 2.1.9. Si Δ es un complejo simplicial puro de dimensión d , entonces $\Delta * \{v_0\}$ es un complejo simplicial puro de dimensión $d+1$, y además $F \mapsto F \cup \{v_0\}$ es una biyección entre las facetas de Δ y las facetas de $\Delta * \{v_0\}$.

Proposición 2.1.10. Si Δ_1 y Δ_2 son dos complejos simpliciales sobre un mismo conjunto subyacente E , y $v_0 \notin E$, entonces

$$(\Delta_1 \cap \Delta_2) * \{v_0\} = (\Delta_1 * \{v_0\}) \cap (\Delta_2 * \{v_0\}).$$

Demostración. Para la inclusión de izquierda a derecha, sea $F \cup \{v_0\}$ una faceta arbitraria de $(\Delta_1 \cap \Delta_2) * \{v_0\}$, es decir, con F una faceta de $\Delta_1 \cap \Delta_2$. Eso significa que F es una faceta tanto de Δ_1 como de Δ_2 , y por lo tanto $F \cup \{v_0\}$ es una faceta de $(\Delta_1 * \{v_0\}) \cap (\Delta_2 * \{v_0\})$. Como este último es un complejo simplicial, se cumple la inclusión. Para la otra dirección, sea A una cara de $(\Delta_1 * \{v_0\}) \cap (\Delta_2 * \{v_0\})$. Tenemos dos casos:

- El primer caso es que $v_0 \notin A$. Entonces, como $A \in (\Delta_1 * \{v_0\})$, deducimos que $A \in \Delta_1$, y, análogamente, $A \in \Delta_2$. Luego $A \in \Delta_1 \cap \Delta_2 \subseteq (\Delta_1 \cap \Delta_2) * \{v_0\}$.
- El segundo caso es que $v_0 \in A$. Sea $B = A \setminus \{v_0\}$. Entonces, como $A \in (\Delta_1 * \{v_0\})$, deducimos que $B \in \Delta_1$, y, análogamente, $B \in \Delta_2$. Luego $B \in \Delta_1 \cap \Delta_2$, y entonces $A = B \cup \{v_0\} \in (\Delta_1 \cap \Delta_2) * \{v_0\}$.

□

2.2. Descascaramientos de un complejo simplicial puro

Definición 2.2.1. Sea Δ un complejo simplicial puro de dimensión d . Un «descascaramiento»¹³ de Δ es un ordenamiento F_1, \dots, F_t de sus facetas (considerando $t = f_d$) tal que, para todo $j \in [t]$, el conjunto $\langle F_j \rangle \setminus \langle F_1, \dots, F_{j-1} \rangle$ tiene un único elemento minimal G_j , llamado la «restricción» de F_j (con respecto a dicho descascaramiento). Diremos que Δ es «descascaramiento» si admite un descascaramiento.

¹³En inglés, «*shelling order*».

A veces resulta más útil la siguiente caracterización:

Proposición 2.2.2. *Sea Δ un complejo simplicial puro de dimensión d , y sea $t = f_d(\Delta)$. Un ordenamiento F_1, \dots, F_t de las facetas de Δ es un descascaramiento si y solo si para todo $i \in \{2, \dots, t\}$ se tiene que $\langle F_1, \dots, F_{i-1} \rangle \cap \langle F_i \rangle$ es un complejo simplicial puro de dimensión $d - 1$.*

La siguiente proposición será clave en la sección 2.4.

Proposición 2.2.3. *Sea Δ un complejo simplicial puro, sea t el número de facetas de Δ , y sea $\Delta * \{v_0\}$ cono sobre Δ . Entonces Δ y $\Delta * \{v_0\}$ tienen el mismo número de descascaramientos. De hecho, F_1, \dots, F_t es un descascaramiento de Δ si y solo si $F_1 \cup \{v_0\}, \dots, F_t \cup \{v_0\}$ es un descascaramiento de $\Delta * \{v_0\}$.*

Demostración. $F_1 \cup \{v_0\}, \dots, F_t \cup \{v_0\}$ es un descascaramiento de $\Delta * \{v_0\}$ si y solo, si para todo $i \in \{2, \dots, t\}$, el siguiente es un complejo simplicial puro de dimensión $d = (d + 1) - 1$:

$$\begin{aligned} & \langle \{F_1 \cup \{v_0\}, \dots, F_{i-1} \cup \{v_0\}\} \rangle \cap \langle \{F_i \cup \{v_0\}\} \rangle \\ &= \langle (F_1, \dots, F_t) * \{v_0\} \rangle \cap \langle (F_i) * \{v_0\} \rangle \\ &= \langle (F_1, \dots, F_{i-1}) \cap \langle F_i \rangle \rangle * \{v_0\}, \end{aligned}$$

donde la última igualdad se tiene por la proposición 2.1.10. Por la proposición 2.1.9, dicha condición es equivalente a que

$$\langle F_1, \dots, F_{i-1} \rangle \cap \langle F_i \rangle$$

sea un complejo simplicial puro de dimensión $d - 1$. Pero esa es exactamente la condición para que F_1, \dots, F_t sea un descascaramiento de Δ . □

Corolario 2.2.4. *Sea Δ un complejo simplicial puro. Se cumple que $\Delta * \{v_0\}$ es descascaramiento si y solo si Δ es descascaramiento.*

A continuación describiremos una conexión entre el f-vector de un complejo simplicial y sus descascaramientos.

Definición 2.2.5. Sea Δ un complejo simplicial de dimensión d . Definimos números enteros $h_0(\Delta), h_1(\Delta), \dots, h_{d+1}(\Delta)$ según la fórmula

$$\sum_{i=0}^{d+1} f_{i-1}(\Delta) (x-1)^{d+1-i} = \sum_{i=0}^{d+1} h_i(\Delta) x^{d+1-i}.$$

Definimos el «h-vector» de Δ como la siguiente tupla:

$$h(\Delta) := (h_0(\Delta), h_1(\Delta), \dots, h_{d+1}(\Delta)).$$

Es claro que $f(\Delta)$ y $h(\Delta)$ contienen información equivalente: podemos determinar $h(\Delta)$ a partir de $f(\Delta)$, y viceversa. Notemos también que $h_0(\Delta) = 1$ siempre que $\Delta \neq \emptyset$, y que $h_1(\Delta) = f_0(\Delta) - (d + 1)$. Si tomamos $x = 1$ en la fórmula de la definición 2.2.5, obtenemos también que

$$f_d(\Delta) = \sum_{i=0}^{d+1} h_i(\Delta).$$

Teorema 2.2.6. [Sta18a]. Sea Δ un complejo simplicial puro de dimensión d . Si F_1, \dots, F_t es un descascaramiento de Δ , con restricciones respectivas G_1, \dots, G_t , entonces

$$\sum_{i=0}^{d+1} h_i x^i = \sum_{j=1}^t x^{|G_j|}.$$

En otras palabras, h_i es el número de restricciones con exactamente i elementos, independientemente del descascaramiento que consideremos.

Corolario 2.2.7. Sea Δ un complejo simplicial puro de dimensión d . Si Δ es descascarable, entonces $h_i(\Delta) \geq 0$ para todo $i \in \{0, 1, \dots, d+1\}$.

Es importante mencionar que el recíproco del corolario 2.2.7 no es cierto (ver, por ejemplo, el corolario 13.16 de [Sta18a]).

Nos interesará particularmente el caso $d = 1$. Los complejos simpliciales puros de dimensión 1 son exactamente los grafos¹⁴ simples y sin vértices aislados. En ese caso, la condición dada por la proposición 2.2.2 se traduce de la siguiente manera:

Sea Δ un complejo simplicial puro de dimensión 1, y sea $t = f_d(\Delta)$. Un ordenamiento de las aristas F_1, \dots, F_t es un descascaramiento de Δ si y solo si, para todo $i \in \{2, \dots, t\}$, se tiene que $\langle F_1, \dots, F_{i-1} \rangle \cap \langle F_i \rangle \neq \{\emptyset\}$.

En otras palabras, un descascaramiento de un complejo simplicial puro de dimensión 1 es un ordenamiento de las aristas de modo que, en cualquier paso del descascaramiento, el subgrafo inducido por las aristas que ya hemos agregado es conexo.

Proposición 2.2.8. Sea G un grafo simple y sin vértices aislados. Tenemos que G descascarable si y solo si es conexo.

Demostración. Supongamos primero que G es descascarable y desconexo. Entonces existe un descascaramiento F_1, \dots, F_t de las aristas, y además los vértices del grafo se pueden particionar en dos conjuntos no vacíos, A y B , de modo que no existe ninguna arista que conecte un vértice de A con un vértice de B . Supongamos, sin pérdida de generalidad, que F_1 es una arista entre vértices de A . Como G no tiene vértices aislados, existe al menos una arista que conecta dos vértices de B . Sea $k > 1$ el menor natural tal que F_k conecta dos vértices de B . Entonces $\langle F_1, \dots, F_{k-1} \rangle \cap \langle F_k \rangle = \{\emptyset\}$, lo que contradice el hecho de que F_1, \dots, F_t es un descascaramiento. Esto prueba que, si G es descascarable, entonces debe ser conexo.

Para la otra dirección, supongamos que G es conexo. Consideremos un árbol generador T , y fijemos algún vértice v_0 . Para cada $k \in \mathbb{N}$, definimos A_k como el conjunto de los vértices v de G tales que el único camino de v_0 a v que está contenido en T tiene exactamente k aristas. Note que $A_0 = \{v_0\}$, y que existe un $N \in \mathbb{N}$ tal que el conjunto de vértices de G es la unión disjunta de A_0, A_1, \dots, A_N . Entonces el siguiente ordenamiento de las aristas es claramente un descascaramiento: primero agregamos las aristas que conectan a v_0 con los vértices de A_1 (en cualquier orden), luego agregamos las aristas que conectan a algún vértice de A_1 con algún vértice de A_2 (en cualquier orden), y así hasta agregar las aristas que conectan a algún vértice de A_{N-1} con algún vértice de A_N ; finalmente, agregamos todas las aristas que no pertenecían al árbol generador. \square

¹⁴Durante esta tesis, los grafos siempre serán complejos simpliciales con finitos vértices y finitas aristas.

2.3. Codificación binaria de un complejo simplicial puro

No tendría sentido estudiar la complejidad de problemas sobre el conjunto de complejos simpliciales con un conjunto subyacente fijo, pues esta sería una colección finita de objetos. En lugar de eso, definiremos $\Sigma_{d,n}$ como el conjunto de complejos simpliciales Δ sobre $[n]$ que son puros, de dimensión d , y tales que $\{m\} \in \Delta$ para todo $m \in [n]$. Definimos

$$\Sigma_d := \bigcup_{n=d+1}^{\infty} \Sigma_{d,n} \quad , \quad \Sigma := \bigcup_{d=0}^{\infty} \Sigma_d.$$

Notemos que ambas uniones son disjuntas, es decir, dado un $\Delta \in \Sigma$, existen únicos $n = n(\Delta)$ y $d = d(\Delta)$ tales que $\Delta \in \Sigma_{d,n}$.

Consideraremos una codificación de Σ que consiste en listar sus facetas. Esta tendrá al inicio la información de los números $n = n(\Delta)$ y $d = \dim(\Delta)$ (esto es posible en virtud del prefijo pleb descrito en la definición 1.2.6). Como $d < n$, este fragmento inicial tendrá longitud $\mathcal{O}(\log_2 n)$.

Veamos la longitud de la codificación. Cada faceta puede describirse listando los $d+1$ vértices que involucra, y cada vértice puede describirse usando $\mathcal{O}(\log(n))$ bits. Por lo tanto, si $t = f_d(\Delta)$, la codificación tendrá una longitud de $\mathcal{O}(t \cdot d \cdot \log(n))$ bits. Notemos que t puede ser tan grande como $\binom{n}{d+1}$, que no es un polinomio en n y d cuando ninguna de las dos variables están fijas. Por lo tanto, lo más adecuado para codificar a Σ sería considerar a t como un parámetro.

Para los propósitos de esta tesis, solo nos interesarán problemas en que consideramos complejos simpliciales puros con cierta dimensión fija. En dicho caso, como

$$t \leq \binom{n}{d+1} \leq \frac{n^{d+1}}{(d+1)!},$$

la codificación tendrá longitud de $\mathcal{O}(n^{d+1} \log(n))$ bits, que asintóticamente crece más lento que el polinomio n^{d+2} . Por lo tanto, consideraremos al número de vértices como el único parámetro relevante.

La veracidad de la siguiente proposición depende también de que la dimensión esté fija.

Proposición 2.3.1. *Sea d un entero positivo fijo. Dada la codificación de las facetas de un $\Delta \in \Sigma_d$, podemos construir todas sus caras en tiempo polinomial.*

Demostración. Por lo que acabamos de discutir, basta mostrar que existe un algoritmo polinomial en n para generar todas las caras. Para cada $k \in \{2, \dots, d\}$, el número de subconjuntos de Δ de tamaño k es a lo sumo

$$\binom{n}{k} \leq \frac{n^k}{k!} < n^d.$$

Por lo tanto, tenemos a lo sumo $d \cdot n^d = \text{poly}(n)$ candidatos a caras. Para cada uno de ellos, hacemos una pasada por la codificación de Δ , y verificamos si alguna de las facetas lo contiene. Este es un algoritmo de tiempo polinomial. \square

2.4. El problema de contar descascaramientos

Sea d un entero positivo fijo. Llamaremos d -SHELLINGS al problema de, dado un complejo simplicial puro de dimensión d , decidir si es descascarable. Al problema de conteo asociado lo llamaremos $\#d$ -SHELLINGS.

Proposición 2.4.1. $\#d$ -SHELLINGS $\in \#\mathbf{P}$ para todo $d \geq 1$.

Demostración. Sea $\Delta \in \Sigma_{d,n}$, y sea $t = f_d(\Delta)$. Como vimos en la sección 2,3, podemos codificar un ordenamiento F_1, \dots, F_t de las facetas de Δ usando $\mathcal{O}(n^{d+2})$ bits. Por lo tanto, basta ver que la verificación de la caracterización de los descascaramientos dada por la proposición 2.2.2 puede hacerse en tiempo polinomial. Como el número de facetas es $\mathcal{O}(n^{d+1})$, basta comprobar que, para cierto $i \in \{2, \dots, t\}$, se puede verificar en tiempo polinomial que $\langle F_1, \dots, F_{i-1} \rangle \cap \langle F_i \rangle$ es un complejo simplicial puro de dimensión $d-1$. Notemos que

$$\langle F_1, \dots, F_{i-1} \rangle \cap \langle F_i \rangle = \langle F_1 \cap F_i, F_2 \cap F_i, \dots, F_{i-1} \cap F_i \rangle.$$

Las caras $\{F_j \cap F_i\}_{1 \leq j < i}$ tienen dimensión a lo más $d-1$, y entre ellas están todas las facetas del complejo simplicial generado. Por lo tanto, el verificador debe rechazar si y solo si existe un $j \in \{1, \dots, i-1\}$ tal que $\dim(F_j \cap F_i) < d-1$ y, además, no existe un $l \in \{1, \dots, i-1\}$ tal que $F_j \cap F_i \subsetneq F_l \cap F_i$ (es decir, si existe una faceta de dimensión menor a $d-1$). Dicha verificación es claramente polinomial. \square

Corolario 2.4.2. d -SHELLINGS $\in \mathbf{NP}$ para todo $d \geq 1$.

Recordemos que, cuando $d=1$, la propiedad de descascarabilidad es equivalente a la de conexidad (proposición 2.2.8). Por lo tanto, se cumple lo siguiente:

Proposición 2.4.3. 1-SHELLINGS $\in \mathbf{P}$.

En 2018, Xavier Goaoc, Pavel Paták, Zuzana Patáková, Martin Tancer y Uli Wagner [GPP⁺19] demostraron el siguiente teorema:

Teorema 2.4.4. 2-SHELLINGS es \mathbf{NP} -completo.

Notemos que, para todo $k \geq 1$, la proposición 2.2.3 nos da una reducción parsimoniosa de $\#k$ -SHELLINGS en $\#(k+1)$ -SHELLINGS. Por lo tanto, la dificultad de los problemas se propaga inmediatamente hacia las dimensiones superiores. De esta forma, tenemos el siguiente corolario:

Corolario 2.4.5. d -SHELLINGS es \mathbf{NP} -completo para todo $d \geq 2$.

El teorema 2.4.4 fue demostrado con una reducción desde 3SAT¹⁵. La prueba hace uso de un resultado de Masahiro Hachimori [Hac08], que da una condición suficiente y necesaria para que la segunda subdivisión baricéntrica de un complejo simplicial sea descascarable. La naturaleza existencial de dicho criterio significa que la reducción no es parsimoniosa, por lo que no es posible deducir directamente que $\#2$ -SHELLINGS es $\#\mathbf{P}$ -completo. En 2021, Andrés Santamaría-Galvis y Russ Woodroffe presentaron una reducción simplificada usando complejos simpliciales relativos [SGW21]. Si bien esta segunda demostración es más constructiva y los artefactos son más sencillos, de todas maneras se imponen condiciones adicionales al descascaramiento, y esto evita que la reducción pueda volverse parsimoniosa.

¹⁵El problema de decisión 3SAT es la restricción de SAT a fórmulas que están escritas en forma normal conjuntiva y en las que, además, cada cláusula tiene exactamente 3 literales.

3. Conteo de descascaramientos de un grafo simple

Durante este capítulo estudiaremos la complejidad de #1-SHELLINGS (definido en la sección 2.4). Renombremos dicho problema como #GSH. Recordemos que #GSH consiste en contar el número de descascaramientos de un grafo G que es simple, no vacío y sin vértices aislados. A dicho número lo denotaremos $F(G)$ ¹⁶. Conjeturamos que este problema es #P-completo.

El capítulo está dividido en varias secciones. Las secciones 3.1 y 3.2 son necesarias para entender las demás: en la primera establecemos la terminología básica, y en la segunda probamos algunas propiedades recursivas del problema. Las siguientes secciones no dependen fuertemente una de la otra, pero recomendamos la lectura de 3.3 antes de 3.6, y de 3.6 antes de 3.7. En la sección 3.3 veremos fórmulas para el número de descascaramientos de algunas familias de grafos sencillos. Luego, en la sección 3.4 discutiremos un resultado de Yibo Gao y Junyao Peng [GP21] que permite contar los descascaramientos de árboles en tiempo polinomial, y extendemos el algoritmo a grafos G con $h_2(G)$ acotado. En la sección 3.5 definimos la noción del árbol generador inducido por un descascaramiento, y demostramos que el problema de contar descascaramientos con un árbol generador prescrito es #P-completo. En la sección 3.6 discutimos una variante de #GSH que consiste en ordenamientos de los vértices con cierta propiedad de conexidad; establecemos cuál es la relación entre la complejidad de ambos problemas, y usamos esto para diseñar un algoritmo que cuenta descascaramientos en tiempo $\mathcal{O}(n^2 \cdot n!)$, donde n es el número de vértices del grafo. En esa sección también indicamos una relación entre el número de ordenamientos conexos de vértices de un grafo y la cardinalidad de sus subgrafos completos. En la sección 3.7 mostraremos que la complejidad computacional de #GSH es equivalente a la del problema en la que agregamos la condición de que algún elemento particular del grafo se conecte en un paso fijo del descascaramiento. Creemos que este último resultado podría ser útil para, en el futuro, demostrar la #P-completitud de #GSH.

A menos que explicitemos lo contrario, todos los grafos que estudiaremos de ahora en adelante serán simples, no vacíos y sin vértices aislados.

3.1. Terminología básica

Definición 3.1.1. Dado un grafo $G = (V, E)$, un ordenamiento $\sigma = (e_1, \dots, e_m)$ de E y un índice $i \in \{0, 1, \dots, m\}$, definimos el « i -ésimo subgrafo inducido por σ », denotado $G_{\sigma, i}$, como el subgrafo inducido por las aristas e_1, \dots, e_i . En particular, $G_{\sigma, 0} = \emptyset$.

Según la definición 3.1.1, σ es un descascaramiento de G si y solo si todos los subgrafos $G_{\sigma, 1}, G_{\sigma, 2}, \dots, G_{\sigma, m}$ son conexos.

Como el f-vector de un grafo $G = (V, E)$ es $f(G) = (1, |V|, |E|)$, podemos obtener su h-vector usando la fórmula de la definición 2.2.5:

$$h(G) = (1, |V| - 2, |E| - |V| + 1).$$

Recordemos que, para un descascaramiento fijo, cada arista estará considerada en exactamente una coordenada del h-vector. Esto nos permite clasificar las aristas del grafo.

¹⁶Para evitar confusiones de notación, no volveremos a usar la letra « F » para denotar a facetas de un complejo simplicial. En cambio, usaremos símbolos de « e » minúscula, que es una notación más estándar para aristas de un grafo.

Definición 3.1.2. Dado un descascaramiento $\sigma = (e_1, \dots, e_m)$ de G y un $k \in \{0, 1, 2\}$, diremos que una arista e_i es de «tipo k para σ » si su restricción con respecto a σ tiene cardinalidad k .

Notemos que $\{u, v\} \in E$ es de tipo 0 para un descascaramiento $\sigma = (e_1, \dots, e_m)$ si y solo si $\{u, v\} = e_1$. Por otro lado, si $\{u, v\} = e_i$ con $i > 1$, entonces:

- e_i es de tipo 1 para σ si y solo si exactamente uno de los vértices u, v pertenece al subgrafo $G_{\sigma, i-1}$.
- e_i es de tipo 2 para σ si y solo si tanto u como v pertenecen al subgrafo $G_{\sigma, i-1}$.

Como consecuencia del teorema 2.2.6, tenemos lo siguiente:

Proposición 3.1.3. Dado cualquier descascaramiento σ de $G = (V, E)$, el número de aristas de tipo 1 para σ es $|V| - 2$, y el número de aristas de tipo 2 para σ es $|E| - |V| + 1$.

3.2. Recursiones del problema

La primera recursión de $\#\text{GSH}$ que estudiaremos corresponde a fijar una condición al inicio del descascaramiento. Más precisamente, impondremos la condición de que la primera arista del descascaramiento sea incidente con un vértice en particular.

Definición 3.2.1. Dados $G = (V, E)$ y $v \in V$, escribimos $F(G; v)$ para denotar al número de descascaramientos de G cuya arista inicial es incidente con v . Al problema de calcular $F(G; v)$ dada la tupla (G, v) lo llamaremos $\#\text{GSH}_1$.

Claramente $\#\text{GSH}_1 \in \#\mathbf{P}$ (es esencialmente la misma demostración que la de la proposición 2.4.1).

Proposición 3.2.2.

$$F(G) = \frac{1}{2} \sum_{v \in V} F(G; v).$$

Demostración. Para cada descascaramiento σ de G , tenemos que su arista inicial es incidente con exactamente dos vértices de G . Por lo tanto, cada descascaramiento se considera en exactamente dos sumandos. \square

Notemos que, como consecuencia de la proposición 3.2.2, tenemos que

$$\#\text{GSH} \leq_{\text{FP}} \#\text{GSH}_1.$$

En la sección 3.7 demostraremos que, en realidad,

$$\#\text{GSH} \equiv_{\text{FP}} \#\text{GSH}_1.$$

A continuación, veremos que el problema $\#\text{GSH}_1$ se puede dividir en subproblemas cuando el vértice en cuestión es un punto de articulación. Recordemos que, dado un grafo conexo $G = (V, E)$ y un $v \in V$, se dice que v es un «punto de articulación» de G si el subgrafo inducido por $V \setminus \{v\}$ es desconexo. En otras palabras, un punto de articulación es un vértice cuya supresión desconecta el grafo.

Proposición 3.2.3. Sea $G = (V, E)$ un grafo conexo y $v \in V$ un punto de articulación. Sean A_1, \dots, A_ℓ las componentes conexas del subgrafo inducido por $V \setminus \{v\}$. Para cada $i \in [\ell]$, sea G_i el subgrafo inducido por los vértices A_i junto con v , y sea m_i el número de aristas de G_i . Entonces

$$F(G; v) = \frac{|E|!}{m_1! m_2! \dots m_\ell!} \prod_{i=1}^{\ell} F(G_i; v).$$

Demostración. En primer lugar, notemos que cada arista pertenece a exactamente un G_i , pues, si $i \neq j$, entonces el único vértice que pertenece tanto a G_i como a G_j es v . Por lo tanto, tenemos que $m_1 + \dots + m_\ell = |E|$, y entonces

$$K := \frac{|E|!}{m_1! m_2! \dots m_\ell!} = \binom{|E|}{m_1, m_2, \dots, m_\ell}.$$

Cada descascaramiento de G que comienza en v induce en cada uno de los grafos G_1, \dots, G_ℓ un descascaramiento que comienza en v . Por otro lado, si para cada G_i tenemos un descascaramiento que comienza en v , entonces con ellos podemos armar un descascaramiento para G de K formas (ya que estos descascaramientos no interactúan entre ellos). \square

La siguiente recursión que estudiaremos consiste en prescribir la última arista del descascaramiento. Esta tiene la ventaja de que reduce el problema a varias instancias más pequeñas de #GSH (a diferencia de la recursión que fija una condición inicial, donde los subproblemas que obtenemos no son directamente instancias de #GSH).

Dado un grafo $G = (V, E)$ y una arista $e \in E$, definimos $G \setminus e$ como el grafo que se obtiene suprimiendo la arista e de G , y posiblemente también un vértice aislado¹⁷.

Proposición 3.2.4. Si $G = (V, E)$ es un grafo conexo y $e \in E$, entonces el número de descascaramientos de G que agregan a e en el último paso es igual a $F(G \setminus e)$.

Demostración. Digamos que $|E| = m$. Si e_1, \dots, e_{m-1}, e_m es un descascaramiento de G con $e_m = e$, entonces claramente e_1, \dots, e_{m-1} es un descascaramiento de $G \setminus e$. Por otro lado, si e_1, \dots, e_{m-1} es un descascaramiento de $G \setminus e$, entonces, como G es conexo, alguno de los dos vértices incidentes con e es también incidente con alguna arista de $G \setminus e$, y luego e_1, \dots, e_{m-1}, e es un descascaramiento de G . \square

Corolario 3.2.5. Si $G = (V, E)$ es un grafo conexo, entonces

$$F(G) = \sum_{e \in E} F(G \setminus e).$$

3.3. Ejemplos notables

En esta sección probaremos fórmulas explícitas para el número de descascaramientos de varias familias sencillas de grafos. Comenzamos estudiando los caminos y los ciclos.

Proposición 3.3.1. Sea $G = (V, E)$ un camino de n vértices, es decir:

$$V = \{v_1, \dots, v_n\} \quad , \quad E = \left\{ \{v_i, v_{i+1}\} \mid i \in [n-1] \right\}.$$

Entonces $F(G) = 2^{n-2}$.

¹⁷Agregamos esta restricción para no tener que admitir que grafos con vértices aislados sean descascarambles.

Demostración. Primero notemos que $F(G; v_1) = F(G; v_n) = 1$. Por otro lado, todo vértice v_i con $i \in [n] \setminus \{1, n\}$ es un punto de articulación. Por la proposición 3.2.3, tenemos que

$$F(G, v_i) = \binom{n-1}{i-1} \cdot 1 \cdot 1.$$

Con la proposición 3.2.2 concluimos que

$$F(G) = \frac{1}{2} \left(1 + \sum_{i=2}^{n-1} \binom{n-1}{i-1} + 1 \right) = \frac{1}{2} \sum_{j=0}^{n-1} \binom{n-1}{j} = \frac{2^{n-1}}{2} = 2^{n-2}.$$

□

Proposición 3.3.2. *Sea $C_n = (V, E)$ un ciclo de n vértices, es decir:*

$$V = \{v_1, \dots, v_n\} \quad , \quad E = \left\{ \{v_i, v_{i+1} \mid i \in [n-1]\} \right\} \cup \{ \{v_n, v_1\} \}.$$

Entonces $F(C_n) = n2^{n-2}$.

Demostración. Notemos que $C_n \setminus e$ es isomorfo a un camino de n vértices, independientemente de la arista $e \in E$. Por las proposiciones 3.2.5 y 3.3.1, tenemos que

$$F(C_n) = \sum_{e \in E} F(C_n \setminus e) = n2^{n-2}.$$

□

La demostración de la siguiente proposición ilustra que, incluso agregando una sola arista nueva, el conteo de descascaramientos puede cambiar radicalmente.

Proposición 3.3.3. *Dado un entero $n \geq 3$, sea $Q_n = (V, E)$ el siguiente grafo:*

$$V = \{v_1, \dots, v_n, u\} \quad , \quad E = \left\{ \{v_i, v_{i+1} \mid i \in [n-1]\} \right\} \cup \{ \{v_n, v_1\}, \{v_1, u\} \}$$

(ver figura 2). Entonces $F(Q_n) = (n^2 + 3n + 2) 2^{n-3}$.

Demostración. Para cada $i \in [n+1]$, sea B_i el número de descascaramientos de Q_n que tienen a la arista $\{v_1, u\}$ en el paso i . Claramente $F(Q_n) = \sum_{i \in [n+1]} B_i$.

Notemos que Q_n sin el vértice u es el ciclo C_n de n vértices. Si $i > 1$, los descascaramientos de Q_n que tienen a la arista $\{v_1, u\}$ en el paso i están en biyección con los descascaramientos de C_n tales que el vértice v_1 pertenece al subgrafo inducido por las primeras $i-1$ aristas del descascaramiento. Por otro lado, los descascaramientos de Q_n que tienen a la arista $\{v_1, u\}$ como arista inicial están en biyección con los descascaramientos de C_n tales que la arista inicial es incidente con v_1 .

El grupo cíclico de n elementos actúa como rotaciones en el conjunto de descascaramientos de C_n , y cada órbita tiene tamaño n . Usando esto, y también el hecho de que los primeros $i-1$ pasos de un descascaramiento de C_n necesariamente forman un camino de i vértices cuando $2 \leq i \leq n$, tenemos que

$$B_1 = \frac{2}{n} \cdot F(C_n) \quad , \quad B_{n+1} = F(C_n) \quad , \quad B_i = \frac{i}{n} \cdot F(C_n) \quad \forall i \in \{2, \dots, n\}.$$

Usando la proposición 3.3.2, concluimos que

$$F(Q_n) = \sum_{i=1}^{n+1} B_i = \frac{n^2 + 3n + 2}{2n} \cdot F(C_n) = (n^2 + 3n + 2) 2^{n-3}.$$

□

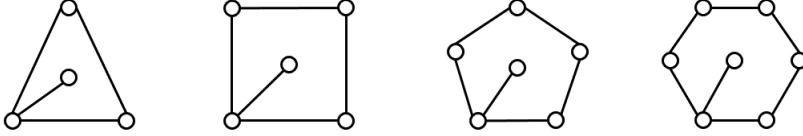


Figura 2: Grafos Q_3 , Q_4 , Q_5 y Q_6 , según la proposición 3.3.3.

Los números $\{B_i\}_{i \in [n+1]}$ en la demostración de la proposición 3.3.3 son una partición de $F(Q_n)$. En la sección 3.7 probaremos que dicha partición siempre se puede calcular en tiempo polinomial a partir de un oráculo para #GSH.

Se conocen fórmulas para el número de descascaramientos de los grafos completos y de los grafos bipartitos completos. La fórmula para los descascaramientos de $K_{m,n}$ (teorema 3.3.5) fue probada por Yibo Gao y Junyao Peng [GP21]. La demostración de la proposición 3.3.4 fue escrita por Richard Stanley en respuesta a una pregunta planteada por Sebastien Palcoux en el sitio web MathOverflow [Sta18b]. El algoritmo que presentaremos en la proposición 3.6.3 está inspirado en el conteo usado en dicha prueba.

Proposición 3.3.4. *Si $n \geq 2$, entonces*

$$F(K_n) = \frac{2^{n-2}}{C_{n-1}} \binom{n}{2}!,$$

donde C_{n-1} es el $(n-1)$ -ésimo número de Catalan.

Demostración. Cada descascaramiento determina un orden en el que los vértices van conectándose a los subgrafos inducidos, salvo por el primer y segundo vértice, que quedan conectados al mismo tiempo. Por lo tanto, hay $\frac{n!}{2}$ formas de ordenar los vértices. Para el k -ésimo vértice que conectamos, hay $(k-1)$ aristas que podemos usar para conectarlo (cualquiera que lo conecte con alguno de los vértices ya conectados).

Ahora debemos colocar las aristas de tipo 2. Solo cuando hemos conectamos el último vértice v_n podemos agregar las $n-2$ aristas de tipo 2 que lo involucran, en cualquier orden. Cuando conectamos el penúltimo vértice v_{n-1} —y no antes— podemos agregar las $n-3$ aristas de tipo 2 que lo involucran y que no involucran a v_n , en cualquier orden, y no importa si se interponen entre las aristas de tipo 2 que involucran a v_n , ni con la última arista de tipo 1. Hay

$$\binom{1 + (n-2) + (n-3)}{n-3} (n-3)! = \binom{2n-4}{n-3} (n-3)!$$

formas de hacer esto último. Cuando colocamos el antepenúltimo vértice v_{n-2} —y no antes— podemos agregar las $n-4$ aristas de tipo 2 que lo involucran y que no involucran a v_{n-1}

ni a v_n , en cualquier orden, y no importa si se interponen entre las aristas de tipo 2 que involucran a v_{n-1} o a v_n , ni con las últimas 2 aristas de tipo 1. Hay

$$\binom{(2 + (n-2) + (n-3) + (n-4))}{n-4} (n-4)! = \binom{3n-6}{n-4} (n-4)!$$

formas de hacer esto último. Continuamos de esta manera. Finalmente, cuando colocamos el tercer vértice v_3 —y no antes— podemos agregar la (única) arista de tipo 2 que lo involucra y que no involucra a ninguno de los vértices v_4, v_5, \dots, v_n , en cualquier orden, y no importa si se interpone con las aristas de tipo 2 que involucran a alguno de los vértices v_4, v_5, \dots, v_n , ni con las últimas $n-3$ aristas de tipo 1. Luego el número de descascaramientos de K_n es

$$\begin{aligned} F(K_n) &= \frac{n!}{2} \cdot (n-1)! \cdot \prod_{k=3}^n \binom{(n-k) + \sum_{j=2}^{n-k+2} (n-j)}{k-2} (k-2)! \\ &= \frac{n!}{2} \cdot (n-1)! \cdot \prod_{k=3}^n \binom{\frac{n^2-n-4+3k-k^2}{2}}{k-2} (k-2)! \\ &= \frac{n!}{2} \cdot (n-1)! \cdot \prod_{k=3}^n \frac{\binom{n}{2} - \binom{k-1}{2} - 1}{\left(\binom{n}{2} - \binom{k}{2}\right)!} \\ &= \frac{n!}{2} \cdot (n-1)! \cdot \left(\binom{n}{2} - 2\right)! \cdot \prod_{k=3}^{n-1} \frac{1}{\binom{n}{2} - \binom{k}{2}} \\ &= \frac{n!}{2} \cdot (n-1)! \cdot \left(\binom{n}{2} - 2\right)! \cdot \prod_{k=3}^{n-1} \frac{2}{(n-k)(n+k-1)} \\ &= \frac{n!}{2} \cdot (n-1)! \cdot \left(\binom{n}{2} - 2\right)! \cdot \frac{2^{n-3} \cdot (n+1)n(n-1)(n-2)}{(2n-2)!} \\ &= \frac{2^{n-2}}{C_{n-1}} \binom{n}{2}!. \end{aligned}$$

□

Teorema 3.3.5. [GP21] Si m y n son enteros positivos, entonces

$$F(K_{m,n}) = \frac{m!n!(mn)!}{(m+n-1)!}.$$

3.4. Algoritmo de tiempo polinomial para grafos con h_2 acotado

En esta sección discutiremos un método propuesto por Yibo Gao y Junyao Peng [GP21] para contar los descascaramientos de cualquier árbol en tiempo polinomial. Esto muestra que el problema #GSH restringido a árboles pertenece a la clase **FP**.

La siguiente definición nos será de utilidad.

Definición 3.4.1. Una «arborescencia» es un árbol dirigido con un vértice especial tal que todas las aristas están orientadas en dirección opuesta a él. Dado un árbol $T = (V, E)$ y un $v \in V$, escribimos T_v para denotar a la arborescencia inducida por el par (T, v) .

Definición 3.4.2. Dado un árbol $T = (V, E)$ y dos vértices $v, u \in V$, definimos el conjunto

$$R(T, v, u) := \{w \in V \mid \text{dist}(w, v) \geq \text{dist}(u, v)\}.$$

Podemos pensar en $R(T, v, u)$ de la siguiente manera: ubicamos el vértice u en la arborescencia T_v , y consideramos todos los vértices $w \in V$ que se pueden alcanzar con un camino dirigido desde u . Notemos que, en general $R(T, v, u) \neq R(T, u, v)$.

Proposición 3.4.3. [GP21] Sea $T = (V, E)$ un árbol con n vértices, y $v \in V$. Entonces

$$F(T; v) = \frac{n!}{\prod_{u \in V} |R(T, v, u)|}.$$

Demostración. Haremos inducción fuerte en la profundidad del árbol enraizado (T, v) , es decir, en el parámetro

$$\text{depth}(T, v) = \max_{w \in V} (\text{dis}(w, v)).$$

Note que el resultado se cumple cuando la profundidad de (T, v) es 1.

Digamos que $\text{depth}(T, v) = k+1$, y supongamos que el resultado se cumple para todos los árboles enraizados con profundidad a lo más $k \geq 1$. Sean a_1, \dots, a_ℓ los vértices de T que son adyacentes a v . Para cada i , sea T_i el subárbol inducido por los vértices $R(T, v, a_i) \cup \{v\}$, y sea m_i el número de aristas de T_i . Si v es un punto de articulación, entonces, por la proposición 3.2.3, tenemos que

$$F(T; v) = \binom{n-1}{m_1, m_2, \dots, m_\ell} \prod_{i=1}^{\ell} F(T_i; v).$$

Por otro lado, si v es una hoja¹⁸, entonces $\ell = 1$, y la fórmula anterior se cumple de todas maneras. Para cada $i \in [\ell]$, sea \widehat{T}_i el subárbol de T_i inducido al suprimir el vértice v . Notemos que, para cada $i \in [\ell]$, $F(T_i; v) = F(\widehat{T}_i; a_i)$, pues la primera arista de un descascaramiento para T_i que comienza en v siempre será $\{v, a_i\}$. Como cada \widehat{T}_i es un árbol con profundidad a lo más k , podemos usar la hipótesis de inducción:

$$\begin{aligned} F(T; v) &= \binom{n-1}{m_1, m_2, \dots, m_\ell} \prod_{i=1}^{\ell} F(\widehat{T}_i; a_i) \\ &= \binom{n-1}{m_1, m_2, \dots, m_\ell} \prod_{i=1}^{\ell} \frac{m_i!}{\prod_{u \in R(T, v, a_i)} |R(\widehat{T}_i, a_i, u)|} \\ &= \binom{n-1}{m_1, m_2, \dots, m_\ell} \prod_{i=1}^{\ell} \frac{m_i!}{\prod_{u \in R(T, v, a_i)} |R(T, a_i, u)|} \\ &= \frac{(n-1)!}{\prod_{u \in V \setminus \{v\}} |R(T, v, u)|} \\ &= \frac{n!}{\prod_{u \in V} |R(T, v, u)|}, \end{aligned}$$

donde la última igualdad se cumple porque $R(T, v, v) = V$. □

¹⁸Todo vértice de un árbol es un punto de articulación o una hoja.

La proposición 3.4.3 muestra que $\#\text{GSH}_1$ restringido a árboles pertenece a la clase **FP**. Podríamos concluir que $\#\text{GSH}$ restringido a árboles pertenece a **FP** usando la proposición 3.2.2. Sin embargo, el siguiente corolario nos da una forma más eficiente.

Corolario 3.4.4. [GP21] Si $T = (V, E)$ es un árbol con n vértices, y $\{u, v\} \in E$, entonces

$$\frac{F(T; v)}{F(T; u)} = \frac{|R(T, u, v)|}{|R(T, v, u)|} = \frac{|R(T, u, v)|}{n - |R(T, u, v)|}.$$

Demostración. La segunda igualdad es clara, pues $|R(T, v, u)| + |R(T, u, v)| = n$ siempre que $\{u, v\} \in E$. Para la primera igualdad, notemos que $R(T, u, w) = R(T, v, w)$ para todo $w \in V \setminus \{u, v\}$. Además, tenemos que $|R(T, v, u)| = |R(T, v, u)| = n$. Luego, aplicando la proposición 3.4.3, obtenemos que

$$\frac{F(T; v)}{F(T; u)} = \frac{\prod_{w \in V} |R(T, u, w)|}{\prod_{w \in V} |R(T, v, w)|} = \frac{|R(T, u, u)| \cdot |R(T, u, v)|}{|R(T, v, u)| \cdot |R(T, v, v)|} = \frac{|R(T, u, v)|}{|R(T, v, u)|}.$$

□

El corolario 3.4.4 nos dice que, una vez calculado $F(T; v_0)$ para algún $v_0 \in V$, podemos deducir rápidamente el valor de $F(T; u)$ para cualquier otro $u \in V$.

Los árboles son justamente los grafos conexos G tales que $h_2(G) = 0$. A continuación veremos que, dado un $k \in \mathbb{N}$ fijo, tenemos un algoritmo de tiempo polinomial para contar los descascaramientos de cualquier grafo G con $h_2(G) \leq k$.

Corolario 3.4.5. Si $k \in \mathbb{N}$ es un natural fijo, entonces el problema $\#\text{GSH}$ restringido a grafos G tales que $h_2(G) \leq k$ pertenece a **FP**.

Demostración. Notemos que, dado cualquier grafo G y una arista e de G , tenemos que $h_2(G \setminus e) = h_2(G) - 1$ siempre que $G \setminus e$ sea conexo. Por lo tanto, podemos usar un algoritmo recursivo en base a la fórmula dada por la proposición 3.2.5: dado un grafo G , calculamos $F(G \setminus e)$ para cada arista e tal que $G \setminus e$ sea conexo (de no serlo, tendremos que $F(G \setminus e) = 0$). Continuamos usando la recursión hasta obtener árboles, en cuyo caso ya tenemos un algoritmo polinomial (ver proposición 3.4.3). Como el h_2 disminuye en 1 en cada paso recursivo, la mayor profundidad de recursión que alcanzaremos será k . Por lo tanto, nos reduciremos a menos de $(n^2)^k = n^{2k} = \text{poly}(n)$ instancias del problema de contar descascaramientos de árboles, que pertenece a **FP**. □

3.5. $\#\text{P}$ -completitud de variante con un árbol generador prescrito

El siguiente es un hecho bien conocido. Incluimos una demostración por completitud.

Proposición 3.5.1. Dado cualquier descascaramiento σ de $G = (V, E)$, las aristas que no son de tipo 2 para σ forman un árbol generador.

Demostración. Escribamos $\sigma = (e_1, \dots, e_m)$. Por la proposición 3.1.3, hay $|V| - 1$ aristas que no son de tipo 2 para σ , por lo que basta ver que cualquier vértice de V es incidente con alguna de dichas aristas. Supongamos por contradicción que existe un $v \in V$ que no es incidente con ninguna arista que no es de tipo 2 para σ . Sea $j \in [m]$ el menor índice tal que v pertenece al subgrafo $G_{\sigma, j}$ (dicho j existe porque, por la proposición 2.2.8, la existencia de σ implica que G es conexo). Notemos que v es incidente con e_j . Si $j = 1$,

entonces v es incidente con la arista de tipo 0, absurdo. Si $j > 1$, entonces, e_j es de tipo 2. Pero, por definición, esto último implicaría que v pertenece al subgrafo $G_{\sigma, j-1}$, lo que contradice la minimalidad de j . Concluimos que dicho v no puede existir. Eso termina la demostración. \square

Definición 3.5.2. Dado un descascaramiento σ de G , denotaremos por $T(\sigma)$ al árbol generador descrito en la proposición 3.5.1. Diremos que $T(\sigma)$ es el «árbol generador de G inducido por σ ».

Definición 3.5.3. Dado un árbol generador T_0 de G , escribiremos $F(G; T_0)$ para denotar al número de descascaramientos σ de G tales que $T(\sigma) = T_0$. Al problema de conteo asociado a la función $(G, T_0) \mapsto F(G; T_0)$ lo llamaremos #ST-GSH.

Claramente #ST-GSH \in #P (podemos repetir la demostración de la proposición 2.4.1 agregando a la verificación la condición de que ninguna arista del árbol generador sea de tipo 2).

Proposición 3.5.4. Si $G = (V, E)$ es conexo y T es un árbol generador de G , entonces $F(G; T) > 0$.

Demostración. Como T es un grafo conexo, por la proposición 2.2.8 sabemos que T tiene un descascaramiento. Ahora concatenamos las $|E| - |V| + 1$ aristas de G que nos quedan al final de dicho descascaramiento, en cualquier orden. Eso nos da un descascaramiento σ de G con $T(\sigma) = T$. \square

Notemos también que, si $\mathcal{T}(G)$ es el conjunto de árboles generadores de G , entonces

$$F(G) = \sum_{T \in \mathcal{T}(G)} F(G; T).$$

Esto no implica directamente que GSH \leq_{FP} #ST-GSH, pues el número de árboles generadores de un grafo no está acotado por un polinomio en el número de vértices. Por ejemplo, es un hecho conocido que el grafo completo de n vértices tiene n^{n-2} árboles generadores.

Vamos a probar que el problema #ST-GSH es #P-completo, aun cuando lo restringimos a instancias en que el árbol generador prescrito es isomorfo a la estrella $K_{1, n-1}$. La completitud se sigue de una reducción parsimoniosa desde un problema de extensiones lineales de conjuntos parcialmente ordenados, cuya #P-dificultad fue demostrada en 2020 por Samuel Dittmer e Igor Pak [DP20].

A continuación haremos un breve resumen de la terminología básica de conjuntos parcialmente ordenados.

Definición 3.5.5. Un «conjunto parcialmente ordenado» o «copo»¹⁹ es un par (P, \leq_P) , donde P es un conjunto finito²⁰ y \leq_P es un orden parcial, es decir, una relación binaria que es reflexiva, antisimétrica y transitiva. Dados $x, y \in P$, escribimos $x <_P y$ si y solo si $x \leq_P y$ y $x \neq y$. Cuando el orden parcial esté claro por contexto, escribiremos simplemente P para referirnos al copo (P, \leq_P) . Dados $x, y \in P$, decimos que y «cubre a» x , y escribimos $x <_P y$, si $x <_P y$ y no existe un $z \in P$ tal que $x <_P z <_P y$.

¹⁹«Copo» es una alternativa al término inglés «*poset*», abreviación de «*partially ordered set*».

²⁰Para propósitos de esta tesis, incluimos esta condición de finitud en la definición.

Definición 3.5.6. El «diagrama de Hasse» del copo (P, \leq_P) es un grafo acíclico dirigido cuyos vértices son los elementos de P y tal que existe una arista dirigida del vértice x al vértice y si y solo si $x \prec_P y$.

Note que la relación \leq_P puede recuperarse de la relación \prec_P agregando la condición de reflexividad y tomando la clausura transitiva. Por lo tanto, el diagrama de Hasse contiene toda la información relevante del copo. Además, todo grafo dirigido acíclico es el diagrama de Hasse de algún copo.

Definición 3.5.7. Sea (P, \leq_P) un copo, con $|P| = n$. Una «extensión lineal» de (P, \leq_P) es una biyección $\sigma : P \rightarrow [n]$ tal que $\sigma(a) \leq \sigma(b)$ siempre que $a \leq_P b$. En otras palabras, σ induce un orden total compatible con el orden parcial \leq_P . Al número de extensiones lineales del copo P lo denotaremos $L(P)$.

El problema de conteo asociado a la función $(P, \leq_P) \mapsto L(P)$ pertenece a la clase $\#\mathbf{P}$. La $\#\mathbf{P}$ -completitud del problema fue demostrada por primera vez en 1991 por Graham Brightwell y Peter Winkler [BW91].

Definición 3.5.8. Dado un conjunto finito E y un complejo simplicial $\Delta \subseteq 2^E$, el «copo de caras» de Δ se define como $P_\Delta := (\Delta \setminus \{\emptyset\}, \subseteq)$.

Note que, si G es un grafo simple, no vacío y sin vértices aislados, entonces el diagrama de Hasse de P_G es un grafo bipartito con conjunto de vértices $V \sqcup E$ y tal que todas las aristas apuntan desde V hacia E .

Definición 3.5.9. $\#\text{IPLE}$ es el problema de contar el número de extensiones lineales de P_G , donde G es un grafo simple, no vacío y sin vértices aislados. Llamaremos $\#\text{CIPLE}$ a la variante del problema en la que agregamos la condición de que G sea conexo.

Teorema 3.5.10. [DP20] $\#\text{IPLE}$ es $\#\mathbf{P}$ -completo.

La siguiente proposición muestra que podemos agregar la condición de conexidad sin perder la $\#\mathbf{P}$ -completitud.

Proposición 3.5.11. $\#\text{CIPLE}$ es $\#\mathbf{P}$ -completo.

Demostración. Probaremos que $\#\text{IPLE} \leq_{\text{FP}} \#\text{CIPLE}$. Sea G un grafo simple, no vacío y sin vértices aislados, y sean G_1, \dots, G_k sus componentes conexas. Sea n el número de vértices de G , y, para cada $i \in [k]$, sea m_i el número de vértices de G_i . Entonces

$$L(P_G) = \binom{n}{m_1, m_2, \dots, m_k} \prod_{j=1}^k L(P_{G_j}).$$

El coeficiente multinomial puede ser calculado rápidamente, y el número de llamadas al oráculo para $\#\text{CIPLE}$ está acotado superiormente por n . \square

La siguiente proposición nos da una primera conexión entre extensiones lineales de copos y descascaramientos de grafos. Fue observada por Yibo Gao y Junyao Peng en [GP21].

Proposición 3.5.12. Sea $T = (V, E)$ un árbol y sea $v \in V$. Sea P el copo cuyo diagrama de Hasse es la arborescencia T_v . Entonces

$$L(P) = F(T; v).$$

Demostración. En primer lugar, notemos que todas las extensiones lineales σ de P cumplen que $\sigma(v) = 1$. Si, por el contrario, tuviésemos que $\sigma^{-1}(1) = w \neq v$, entonces habría un camino dirigido de w a v en T_v , lo que es absurdo.

Hay una correspondencia biunívoca entre aristas de T_v y vértices de T distintos a v : basta ver el vértice hacia el que apunta cada arista. Por lo tanto, hay también una correspondencia, digamos φ , entre ordenamientos σ de los elementos de P tales que $\sigma(v) = 1$ y ordenamientos de las aristas de T tales que la primera arista es incidente con v . Veamos que φ induce una biyección entre extensiones lineales de P y descascaramientos de T cuya arista inicial es incidente a v .

Supongamos que existe una extensión lineal σ de P tal que $\varphi(\sigma)$ no es un descascaramiento. Eso significaría que existe una arista (u_1, u_2) (apuntando desde el vértice u_1 hacia el vértice u_2) que queda numerada antes de que se hayan numerado todas las aristas que conforman el camino de v a u_1 . En particular, existirá un vértice w en dicho camino tal que la arista que apunta hacia w queda numerada en σ después que la arista (u_1, u_2) , lo que significa que σ es incompatible con el orden parcial de P , absurdo.

Ahora supongamos que existe un descascaramiento α de T que comienza con una arista incidente con v y tal que $\sigma := \varphi^{-1}(\alpha)$ no es una extensión lineal de P . Eso significaría que existen $u, w \in V$ tales que $\sigma(u) < \sigma(w)$ pero $w \leq_P u$. Eso último implica que en T_v existe un camino dirigido de w a u . Sin embargo, como $\sigma(u) < \sigma(w)$, también tenemos que la arista que apunta hacia w entra al descascaramiento α después que la arista que apunta hacia u , lo que es una contradicción.

Esto prueba la biyección que queríamos. \square

Lo que haremos a continuación es extender la idea de la proposición 3.5.12 a grafos que no necesariamente son árboles. Para ello, consideraremos una variante de #ST-GSH.

Definición 3.5.13. Dado un grafo $G = (V, E)$, un árbol generador T_0 de G y un vértice $v \in V$, escribimos $F(G; T_0, v)$ para denotar al número de descascaramientos σ de G tales que $T(\sigma) = T_0$ y que su arista inicial es incidente con v . Al problema de conteo asociado a la función $(G, T_0, v) \mapsto F(G; T_0, v)$ lo llamaremos #ST-GSH₁.

Definición 3.5.14. Sea $G = (V, E)$ un grafo conexo, T un árbol generador de G y $e = \{u, v\}$ una arista de G que no pertenece a T . Sabemos que existe un único camino de u a v que está contenido en el árbol T . Si al conjunto de aristas de dicho camino le agregamos la arista $\{e\}$, obtendremos un ciclo simple de G , llamado el «ciclo fundamental de e con respecto a T », y que denotaremos $C_T(e)$.

Proposición 3.5.15. Sea G un grafo conexo, T_0 un árbol generador de G y σ un descascaramiento de G . Se cumple que $T(\sigma) = T_0$ si y solo si, para toda arista e de G que no pertenece a T_0 , la última arista de $C_{T_0}(e)$ según el descascaramiento σ es justamente e .

Demostración. Digamos que $\sigma = (e_1, \dots, e_m)$.

Supongamos primero que $T(\sigma) = T_0$. Sea $e = e_j = \{u, v\}$ una arista de G que no pertenece a T_0 , y sea

$$i = \max \{ \ell \in [m] \mid e_\ell \in C_{T_0}(e) \}.$$

Queremos ver que $i = j$. Notemos que en $G_{\sigma, i-1}$ hay un camino de u a v , por lo que la arista e_i es de tipo 2 para σ . Pero la única arista de tipo 2 para σ en $C_{T_0}(e)$ es justamente e , por lo que $e_i = e$ e $i = j$.

Ahora supongamos que $T(\sigma) \neq T_0$. Luego existe una arista $e = e_j = \{u, v\}$ de G que no pertenece a T_0 y que no es de tipo 2 para σ . Entonces en $G_{\sigma, j-1}$ no hay un camino de

u a v y, por lo tanto, $e = e_j$ no puede ser la última arista de $C_{T_0}(e)$ que aparece en el descascaramiento. \square

Definiremos una noción de subdivisión parcial de un grafo, que nos será útil para formalizar algunas construcciones.

Definición 3.5.16. Sea $G = (V, E)$ un grafo y $R \subseteq E$ un subconjunto de aristas de G . La «subdivisión de R » es el grafo $\mathcal{S}(G; R)$ cuyo conjunto de vértices es $V \sqcup R$ y cuyo conjunto de aristas es

$$(E \setminus R) \sqcup \{\{v, e\} \mid v \in V \wedge e \in R \wedge v \in e\}.$$

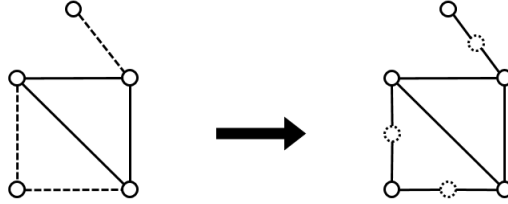


Figura 3: Ejemplo de subdivisión parcial. Las aristas punteadas están siendo subdivididas.

Proposición 3.5.17. Sea G un grafo conexo, T un árbol generador de G y $v \in V$ un vértice. Sea $H(G; T, v)$ el grafo dirigido que se obtiene orientando las aristas de $\mathcal{S}(G; E \setminus T)$ de la siguiente manera: las aristas de T se orientan según la arborescencia T_v , y las aristas de la forma $\{v, e\} \in V \times (E \setminus T)$ se orientan desde v hacia e (vea figura 4). Entonces $H(G; T, v)$ es acíclico.

Demostración. Supongamos por contradicción que $H(G; T, v)$ tiene un ciclo dirigido. Dicho ciclo no puede contener a ningún vértice correspondiente a un elemento de $(E \setminus T)$, pues las dos aristas que inciden en él están orientadas hacia él. Pero si el ciclo no contiene a ningún vértice de esa forma, entonces está contenido en la arborescencia T_v , que es un grafo acíclico, absurdo. \square

La siguiente proposición dice que toda instancia del problema $\#ST\text{-GSH}_1$ corresponde a contar las extensiones lineales de cierto copo.

Proposición 3.5.18. Sea $G = (V, E)$ un grafo conexo, T un árbol generador de G y $v \in V$ un vértice. Sea P el copo cuyo diagrama de Hasse es el grafo acíclico dirigido $H(G; T, v)$ descrito en la proposición 3.5.17. Entonces

$$L(P) = F(G; T, v).$$

Demostración. Las extensiones lineales σ de P cumplen que $\sigma(v) = 1$. Además, hay una correspondencia biunívoca entre aristas de G y vértices de $H(G; T, v)$ distintos a v : si e es una arista de T , la identificamos con el vértice al que apunta en la arborescencia T_v ; por otro lado, si e es una arista de $(E \setminus T)$, la identificamos con el vértice correspondiente en la subdivisión. Por lo tanto, hay también una correspondencia, digamos φ , entre ordenamientos σ de los elementos de P tales que $\sigma(v) = 1$ y ordenamientos de las aristas de G tales que la primera arista es incidente con v . Necesitamos probar que φ induce una biyección entre

extensiones lineales de P y descascaramientos de G cuyo árbol generador inducido es T y cuya arista inicial es incidente con v .

Si σ es una extensión lineal de P , podemos repetir la demostración de la proposición 3.5.12 para probar que $\varphi(\sigma)$ es un descascaramiento de G . Para ver que el árbol generador inducido por $\varphi(\sigma)$ es T , notemos que, por la forma en la que está construido el grafo acíclico dirigido $H(G; T, v)$, tendremos que cada arista en $E \setminus T$ es, según el descascaramiento, la última arista del ciclo fundamental correspondiente, por lo que podemos concluir en virtud de la proposición 3.5.15.

Para la otra dirección, nuevamente podemos repetir lo que hicimos en la proposición 3.5.12, y para la condición del árbol prescrito usamos la otra dirección de la proposición 3.5.15.

Por lo tanto, tenemos la biyección que queríamos. \square

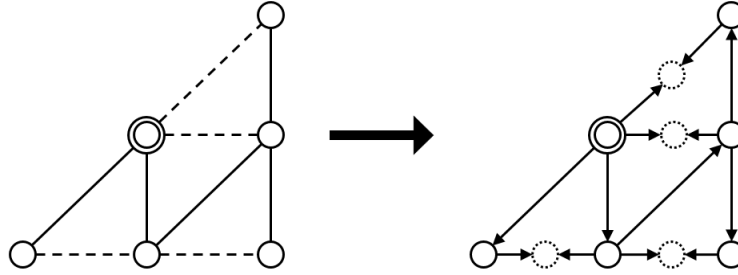


Figura 4: Construcción de $H(G; T, v)$. El vértice con el doble círculo es la raíz v . Las aristas con línea continua corresponden a las del árbol T , y se orientan en sentido opuesto al vértice v . Las aristas punteadas son las que no están en T ; estas se subdividen en dos nuevas aristas, que quedan orientadas hacia el vértice creado al hacer la subdivisión.

No podemos concluir que $\#ST\text{-GSH}_1$ es $\#\mathbf{P}$ -completo usando solo la proposición 3.5.18, pues no sabemos aún si la correspondencia es sobreyectiva sobre alguna clase de copos para la cual contar extensiones lineales es $\#\mathbf{P}$ -completo. De ello nos encargaremos a continuación: usaremos el teorema 3.5.10 y la proposición 3.5.18 para deducir la $\#\mathbf{P}$ -completitud de $\#ST\text{-GSH}_1$.

Definición 3.5.19. Dado un grafo $G = (V, E)$ y un vértice $v_0 \in V$, definimos la «estrella de v_0 », denotada $St_G(v_0)$, como el árbol inducido por todas las aristas de G que inciden con v_0 . Cuando G esté claro por contexto, escribiremos simplemente $St(v_0)$.

Definición 3.5.20. Sea $G = (V, E)$ un grafo, y consideremos un nuevo vértice v_0 que no está en V . Podemos considerar un nuevo grafo $G \star \{v_0\}$ tomando las caras de dimensión 0 y 1 del cono $G \star \{v_0\}$.

Notemos que $St_G(v_0)$ es siempre un árbol generador de $G \star \{v_0\}$.

La siguiente proposición nos dará la conexión que buscábamos entre los problemas $\#\text{CIPLE}$ y $\#ST\text{-GSH}$.

Proposición 3.5.21. Sea $G = (V, E)$ un grafo simple, no vacío y conexo, y consideremos $G \star \{v_0\}$ con $v_0 \notin V$. Entonces

$$L(P_G) = F(G \star \{v_0\}; St(v_0), v_0).$$

Demostración. Sea $H = H(G \star \{v_0\}; \text{St}(v_0), v_0)$ el grafo acíclico dirigido descrito en la proposición 3.5.17, y sea P el copo cuyo diagrama de Hasse es H . Por la proposición 3.5.18 tenemos que

$$L(P) = F(G \star \{v_0\}; \text{St}(v_0), v_0).$$

Por lo tanto, basta probar que $L(P) = L(P_G)$. La observación clave es que P es isomorfo al copo P_G pero con un elemento mínimo²¹ agregado. Si ignoramos dicho elemento (que corresponde al centro de la estrella), las relaciones de cubrimiento en H son exactamente las incidencias del grafo G . Como agregar un elemento mínimo no altera el número de extensiones lineales, concluimos que $L(P) = L(P_G)$, como queríamos probar. \square

Corolario 3.5.22. *#ST-GSH y #ST-GSH₁ son #P-completos.*

Demostración. La proposición 3.5.21 nos da una reducción parsimoniosa de #CIPLÉ en #ST-GSH₁. Como el primero es #P-completo (teorema 3.5.10), concluimos que el segundo también lo es.

Para ver que #ST-GSH también es #P-completo, basta notar que

$$F(G \star \{v_0\}; \text{St}(v_0), v_0) = F(G \star \{v_0\}; \text{St}(v_0)),$$

pues cualquier descascaramiento de $G \star \{v_0\}$ con $\text{St}(v_0)$ como árbol generador prescrito necesariamente tiene como arista inicial a una arista incidente con v_0 . \square

Es importante mencionar que la proposición 3.5.21 nos dice que el problema #ST-GSH es #P-completo incluso si nos restringimos a instancias de una forma muy particular. Por completitud, enunciaremos eso en una proposición aparte.

Proposición 3.5.23. *El problema de conteo asociado a la función $G \mapsto F(G \star \{v_0\}; \text{St}(v_0))$, donde G es un grafo simple, no vacío y sin vértices aislados, y v_0 no es un vértice de G , es #P-completo.*

3.6. Ordenamientos conexos de vértices

Cada descascaramiento de un grafo determina un orden en el que sus vértices se van conectando a los subgrafos inducidos (salvo por los dos vértices incidentes con la primera arista). A continuación estudiaremos la complejidad computacional de contar el número de descascaramientos que tienen algún orden de los vértices prefijo.

Definición 3.6.1. Dado un conjunto V con $|V| = n$, definimos Γ_V como el conjunto de funciones sobreyectivas $\gamma : V \rightarrow \{1, 2, \dots, n-1\}$ tales que $|\gamma^{-1}(1)| = 2$.

Al elemento de Γ_V inducido por un descascaramiento σ lo denotaremos $\gamma(\sigma)$.

Definición 3.6.2. Dado un $\gamma_0 \in \Gamma_V$, escribiremos $F(G; \gamma_0)$ para denotar al número de descascaramientos σ de G tales que $\gamma(\sigma) = \gamma_0$. Al problema de conteo asociado a la función $(G, \gamma_0) \mapsto F(G; \gamma_0)$ lo llamaremos #VO-GSH.

Claramente #VO-GSH \in #P (podemos repetir la demostración de la proposición 2.4.1 agregando a la verificación la condición del orden de conexión de los vértices).

²¹En un copo (P, \leq_P) , un «elemento mínimo» es un $y \in P$ tal que $y \leq_P x$ para todo $x \in P$

Al igual que para la variante con un árbol generador prescrito, este problema también cumple que la suma de todas sus instancias posibles (para un mismo grafo G) es igual al total de descascaramientos. Es decir, se cumple la siguiente fórmula:

$$F(G) = \sum_{\gamma \in \Gamma_V} F(G; \gamma).$$

La siguiente proposición muestra que #VO-GSH puede resolverse en tiempo $\mathcal{O}(n^2)$. La prueba se basa en el método de Richard Stanley para contar los descascaramientos del grafo completo (ver demostración de la proposición 3.3.4).

Proposición 3.6.3. *Sea $G = (V, E)$ un grafo simple y conexo, y sean $n = |V|$ y $m = |E|$. Sea $\gamma_0 \in \Gamma_V$ y, para cada $j \in [n-2]$, sea α_j el número de aristas que conectan al vértice $\gamma_0^{-1}(j+1)$ con algún $v \in V$ tal que $\gamma_0(v) \leq j$. Si $\min(\alpha_1, \dots, \alpha_{n-2}) = 0$, entonces $F(G; \gamma_0) = 0$, y, en otro caso, se cumple que*

$$F(G; \gamma_0) = \prod_{j=1}^{n-2} \binom{m-2-\sum_{i=1}^{j-1} \alpha_i}{\alpha_j-1} \alpha_j! = \frac{(m-2)!}{\prod_{j=1}^{n-2} (m-1-\sum_{i=1}^j \alpha_i)} \prod_{j=1}^{n-2} \alpha_j.$$

Demostración. Para $j \in [n-2]$, sea Λ_j el conjunto de aristas que conectan al vértice $\gamma_0^{-1}(j+1)$ con algún $v \in V$ tal que $\gamma_0(v) \leq j$, de modo que $|\Lambda_j| = \alpha_j$. Notemos que $F(G; \gamma_0) = 0$ si y solo si $\Lambda_j = \emptyset$ para algún j (ver ejemplo 3.6.4).

Dado un descascaramiento σ con $\gamma(\sigma) = \gamma_0$, notemos que cualquier reordenamiento de las aristas de Λ_j (para cualquier j) corresponde a otro descascaramiento σ' con $\gamma(\sigma') = \gamma$. Por lo tanto, para todo $j \in [n-2]$, podemos imponer cualquier orden en las aristas de Λ_j . Esto muestra que $F(G; \gamma_0)$ es divisible en $\alpha_1! \cdot \alpha_2! \cdot (\dots) \cdot \alpha_{n-2}!$.

Calculemos el cociente. Notemos que, una vez ordenados los conjuntos Λ_j , el conteo de los descascaramientos que respetan el orden γ_0 puede hacerse de la siguiente manera. Las $\alpha_1 - 1$ aristas de Λ_1 que son de tipo 2 pueden colocarse en cualquier posición después de haber agregado la arista inicial y la primera arista de Λ_1 , que necesariamente será la segunda arista del descascaramiento. Supongamos que elegimos los índices de dichas aristas: hay $\binom{m-2}{\alpha_1-1}$ maneras de hacer esto. Luego, necesariamente debemos colocar la primera arista de Λ_2 en la primera posición que aún esté disponible (o el orden de conexión de los vértices no sería el dado por γ_0). Las $\alpha_2 - 1$ aristas de Λ_2 que son de tipo 2 pueden colocarse en cualquier posición que aún esté disponible: hay $\binom{m-1-\alpha_1-1}{\alpha_2-1} = \binom{m-2-\alpha_1}{\alpha_2-1}$ maneras de hacer esto. Si continuamos de esta manera, el conteo da como resultado

$$\prod_{j=1}^{n-2} \binom{m-2-\sum_{i=1}^{j-1} \alpha_i}{\alpha_j-1}.$$

La segunda igualdad de la proposición se obtiene expandiendo los coeficientes binomiales y simplificando. \square

La proposición 3.6.3 nos dice que, si iteramos sobre todos los elementos de Γ_G , podemos contar los descascaramientos de G en tiempo $\mathcal{O}(n^2 \cdot n!)$. En el caso de que G sea denso, es decir, que el cociente de dividir m en $\binom{n}{2}$ sea cercano a 1, este algoritmo será significativamente más rápido que el algoritmo de *backtracking* sobre toda los posibles ordenamientos de aristas. Anexamos como apéndice una implementación del algoritmo en código Python (ver sección 4.1).

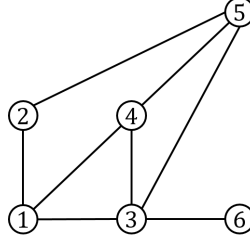


Figura 5: Grafo G del ejemplo 3.6.4.

Ejemplo 3.6.4. Consideremos el grafo $G = ([6], E)$ de la figura 5. Sea $\gamma \in \Gamma_{[6]}$ dado por $\gamma(1) = \gamma(2) = 1$ y $\gamma(k) = k - 1$ para $k \in \{3, \dots, 6\}$. Siguiendo la terminología de la proposición 3.6.3 y de su demostración, tenemos que

$$\Lambda_1 = \{\{1, 3\}\}, \quad \Lambda_2 = \{\{1, 4\}, \{3, 4\}\}, \quad \Lambda_3 = \{\{2, 5\}, \{3, 5\}, \{4, 5\}\}, \quad \Lambda_4 = \{\{3, 6\}\}.$$

Luego $\alpha_1 = 1$, $\alpha_2 = 2$, $\alpha_3 = 3$ y $\alpha_4 = 1$. Usando la fórmula, obtenemos que $F(G; \gamma_0) = 180$.

Observamos que, en el caso de que G sea un árbol, todos los α_j de la proposición 3.6.3 son iguales a 0 o a 1, por lo que el problema de contar descascaramientos coincide con el de contar los $\gamma \in \Gamma_V$ tales que $F(G; \gamma) > 0$. A continuación discutiremos brevemente qué conexiones podemos hacer, en general, entre ambos problemas.

Definición 3.6.5. Dado un grafo $G = (V, E)$, diremos que un ordenamiento v_1, v_2, \dots, v_n de los vértices de V es un «orden conexo de vértices» si para todo $i \in \{2, \dots, n\}$ existe un $j \in [i - 1]$ tal que $\{v_i, v_j\} \in E$. Al número de ordenamientos conexos de G lo denotamos $C(G)$, y al problema asociado a la función $G \mapsto C(G)$ lo llamaremos #CVO.

Se sigue directamente de la proposición 3.6.3 que el número de elementos $\gamma \in \Gamma_V$ tales que $F(G; \gamma) > 0$ es $\frac{C(G)}{2}$.

La siguiente proposición nos da una reducción parsimoniosa de #GSH en #CVO.

Proposición 3.6.6. Sea $G = (V, E)$ un grafo simple y sin vértices aislados. Construimos un grafo²² $L = (E, W)$ de modo que $\{e_1, e_2\} \in W$ si y solo si existe un $v \in V$ tal que tanto e_1 como e_2 son incidentes con v en el grafo G . Se cumple que un ordenamiento de E es un descascaramiento para G si y solo si es un orden conexo de vértices para L . En consecuencia, $F(G) = C(L)$.

Otra consecuencia de la proposición 3.6.3 es que algunas propiedades del grafo G se ven reflejadas en la factorización de $F(G; \gamma_0)$. Esto funciona especialmente bien en lo que respecta a los cliques de G . Recordemos que un « k -clique» de G , con $k \geq 2$ un entero, es un subgrafo de G isomorfo al grafo completo de k vértices. En particular, consideraremos que cada arista es un 2-clique.

Definición 3.6.7. Un «cubrimiento de aristas por cliques» de un grafo $G = (V, E)$ es una partición de E de modo que las aristas de cada componente de la partición formen un clique.

²²A este grafo se le conoce como el «grafo de línea» de G .

Notemos que, si tenemos algún conjunto de cliques de G que son distintos y cuyas aristas son disjuntas dos a dos, entonces siempre podemos extenderlo a un cubrimiento de aristas por cliques considerando que cada una de las aristas que no hemos cubierto están en su propia componente, formando un 2-clique.

Para simplificar la notación de la siguiente proposición, consideraremos el «superfactorial» de un $n \in \mathbb{N}$ como el entero positivo definido por

$$n\$:= \prod_{k=1}^n k! = \prod_{k=1}^n k^{n+1-k}.$$

Proposición 3.6.8. *Supongamos que G tiene un cubrimiento de aristas en cliques con b_k k -cliques para cada $k \in \{2, \dots, n\}$. Entonces, para todo $\gamma_0 \in \Gamma_V$, tenemos que $F(G; \gamma_0)$ es divisible en*

$$\prod_{k=2}^n ((k-1)\$)^{b_k}.$$

Demostración. Primero notemos que, si $F(G; \gamma_0) = 0$, el resultado es trivialmente cierto.

Sea H algún k -clique del cubrimiento, y sean w_1, \dots, w_k sus vértices ordenados según γ_0 (salvo, posiblemente, por w_1 y w_2). Notemos que, para cada $i \in \{3, \dots, k\}$, hay i aristas en H que conectan a w_i con algún w_j con $j < i$. Si σ es un descascaramiento con $\gamma(\sigma) = \gamma_0$, entonces cualquier reordenamiento de esas i aristas corresponde a otro descascaramiento σ' con $\gamma(\sigma') = \gamma_0$. Por lo tanto, solo reordenando aristas de H podemos definir una relación de equivalencia en la que cada clase de equivalencia tendrá exactamente $(k-1)\$$ elementos. Como podemos hacer esto mismo para cada clique del cubrimiento, y los conjuntos de aristas de dichos cliques son disjuntos dos a dos, concluimos el resultado. \square

Notemos que el divisor de la proposición 3.6.8 también será un divisor de $F(G)$. Si elegimos un cubrimiento de aristas en cliques intentando que los cliques sean lo más grande posible, podemos concluir rápidamente que $F(G)$ debe ser un múltiplo de algún número relativamente grande. Creemos que esto, junto con algún método para aproximar el número de descascaramientos del grafo con cotas precisas, podría ser la base de un algoritmo más rápido para el cálculo exacto de $F(G)$.

3.7. Vectores de partición

En esta sección describiremos algunas técnicas que, según creemos, podrían ser útiles en una futura demostración del poder computacional de un oráculo para #GSH.

Definición 3.7.1. Sea $G = (V, E)$ un grafo simple con m aristas, y sea $v \in V$ un vértice. Definimos la «partición de descascaramientos de G en v » como el vector $\mathcal{A}(G; v) \in \mathbb{N}^m$ tal que, para todo $i \in [m]$, $\mathcal{A}(G; v)_i$ es el número de descascaramientos σ de G tales que el vértice v pertenece al subgrafo inducido $G_{\sigma, i}$, pero no a $G_{\sigma, i-1}$. En otras palabras, $\mathcal{A}(G; v)_i$ es el número de descascaramientos que conectan al vértice v por primera vez en el i -ésimo paso.

Los vectores $\mathcal{A}(G; v)$ pueden darnos información valiosa sobre los descascaramientos de G . Por ejemplo, la media aritmética de las coordenadas de $\mathcal{A}(G; v)$ nos indicará, en promedio, en qué paso de un descascaramiento aleatorio debiésemos esperar que quede conectado el vértice v .

La siguiente proposición resume las propiedades básicas de los vectores de partición en vértices.

Proposición 3.7.2. Si $G = (V, E)$ es un grafo conexo con m aristas, entonces:

(i) Para todo $v \in V$, $\sum_{i=1}^m \mathcal{A}(G; v)_i = F(G)$.

(ii) Para todo $v \in V$, $\mathcal{A}(G; v)_1 = F(G; v)$.

(iii) $\frac{1}{2} \sum_{v \in V} \mathcal{A}(G; v)_1 = F(G)$.

(iv) Para todo $i \in \{2, \dots, m\}$, $\sum_{v \in V} \mathcal{A}(G; v)_i$ es igual al número de descascaramientos de G tales que la i -ésima arista del descascaramiento es de tipo 1.

(v) Sea $v \in V$, y sea k el número de aristas de la componente conexa con más aristas de $G \setminus v$, el subgrafo de G inducido por los vértices $V \setminus \{v\}$. Entonces $\mathcal{A}(G; v)_i = 0$ si y solo si $i > k + 1$.

(vi) Si $v \in V$ es un vértice que no es un punto de articulación, entonces

$$\mathcal{A}(G; v)_{n+1-\deg(v)} = \deg(v)! \cdot F(G \setminus v).$$

Demostración. Los primeros cuatro puntos se siguen directamente de la definición.

Para (v), basta notar que cualquier descascaramiento de G que conecta al vértice v en el i -ésimo paso cumplirá que sus primeras $i - 1$ aristas son también aristas de $G \setminus v$. Todas esas aristas deben pertenecer a una misma componente conexa de $G \setminus v$, por lo que $i - 1 \leq k$, es decir, $i \leq k + 1$. Recíprocamente, si $i - 1 \leq k$, tomamos como primeros $i - 1$ pasos al inicio de un descascaramiento de la mayor componente conexa de $G \setminus v$ que además comience en un vértice que sea adyacente a v ; luego agregamos una arista que conecte el vértice v , y terminamos de cualquier forma.

Para (vi), notemos que cada descascaramientos del grafo $G \setminus v$ se corresponde con $\deg(v)!$ descascaramientos de G que conectan a v recién en el paso $n + 1 - \deg(v)$ y que, por lo tanto, tienen de últimas a las $\deg(v)$ aristas incidentes con v , en algún orden. \square

Ejemplo 3.7.3. Consideremos el grafo $G = (V, E)$ con

$$V = \{1, 2, 3, 4\} \quad , \quad E = \{\{1, 2\}, \{2, 3\}, \{3, 1\}, \{1, 4\}\}.$$

La siguiente matriz tiene en la i -ésima fila al vector $\mathcal{A}(G; i)$:

$$\begin{pmatrix} 16 & 4 & 0 & 0 \\ 10 & 6 & 4 & 0 \\ 10 & 6 & 4 & 0 \\ 4 & 4 & 6 & 6 \end{pmatrix}.$$

La segunda y tercera fila son iguales porque G tiene un automorfismo que intercambia los vértices 2 y 3.

Vamos a demostrar que los vectores $\mathcal{A}(G; v)$ pueden calcularse en tiempo polinomial teniendo un oráculo para #GSH. Usaremos el siguiente lema.

Lema 3.7.4. Sea $G = (V, E)$ un grafo con $|E| = m$, $v \in V$ un vértice y $\ell \in \mathbb{N}$. Definimos el grafo $G_{v,\ell} = (V', E')$ como el grafo G con ℓ «decoraciones» en v , es decir:

$$V' = V \sqcup \{u_1, u_2, \dots, u_\ell\} \quad , \quad E' = E \sqcup \{\{v, u_1\}, \{v, u_2\}, \dots, \{v, u_\ell\}\}.$$

Entonces se cumple que

$$\frac{F(G_{v,\ell})}{\ell!} = \binom{m+\ell}{\ell} \mathcal{A}(G;v)_1 + \sum_{j=2}^m \binom{m+\ell-j}{\ell} \mathcal{A}(G;v)_j.$$

La fórmula se cumple incluso si $\ell = 0$.

Demostración. La prueba es un argumento de conteo. Notemos primero que, para todo par $i, j \in [\ell]$, existe un automorfismo de $G_{v,\ell}$ que intercambia u_i con u_j y deja todo lo demás fijo. Por lo tanto, podemos asumir que todos los descascaramientos agregan las aristas de $E' \setminus E$ en algún orden específico: solo debemos dividir $F(G_{v,\ell})$ en $\ell!$.

A cada descascaramiento de $G_{v,\ell}$ con las aristas de $E' \setminus E$ ordenadas le podemos asociar un descascaramiento de G : solo debemos suprimir las aristas de $E' \setminus E$. Esta es una sobreyección. Veremos que la cardinalidad de cada preimagen de un descascaramiento σ de G depende solamente del paso en el que el vértice v se conectó por primera vez. Tenemos dos casos:

- El primer caso es que la primera arista de σ sea incidente con v . En dicho caso, podemos elegir colocar las aristas de $E' \setminus E$ en cualquier posición, por lo que tenemos $\binom{m+\ell}{\ell}$ opciones.
- El segundo caso es que el vértice v se conecte por primera vez en el j -ésimo paso de σ , con $j \in \{2, \dots, m\}$. En dicho caso, podemos elegir colocar las aristas de $E' \setminus E$ en cualquier posición posterior a esta, por lo que tenemos $\binom{m+\ell-j}{\ell}$ opciones.

□

Proposición 3.7.5. *La función $(G, v) \mapsto \mathcal{A}(G;v)$ pertenece a $\mathbf{FP}^{\#\text{GSH}}$.*

Demostración. Sea $G = (V, E)$ un grafo con $|E| = m$ y $v \in V$ un vértice. Aplicando el lema 3.7.4 para todo $\ell \in \{0, 1, \dots, m-1\}$, tenemos que

$$\begin{pmatrix} \binom{m}{0} & \binom{m-2}{0} & \binom{m-3}{0} & \binom{m-4}{0} & \cdots & \binom{0}{0} \\ \binom{m+1}{1} & \binom{m-1}{1} & \binom{m-2}{1} & \binom{m-3}{1} & \cdots & \binom{1}{1} \\ \binom{m+2}{2} & \binom{m}{2} & \binom{m-1}{2} & \binom{m-2}{2} & \cdots & \binom{2}{2} \\ \binom{m+3}{3} & \binom{m+1}{3} & \binom{m}{3} & \binom{m-1}{3} & \cdots & \binom{3}{3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \binom{2m-1}{m-1} & \binom{2m-3}{m-1} & \binom{2m-4}{m-1} & \binom{2m-5}{m-1} & \cdots & \binom{m-1}{m-1} \end{pmatrix} \begin{pmatrix} \mathcal{A}(G;v)_1 \\ \mathcal{A}(G;v)_2 \\ \mathcal{A}(G;v)_3 \\ \mathcal{A}(G;v)_4 \\ \vdots \\ \mathcal{A}(G;v)_m \end{pmatrix} = \begin{pmatrix} F(G_{v,0})/0! \\ F(G_{v,1})/1! \\ F(G_{v,2})/2! \\ F(G_{v,3})/3! \\ \vdots \\ F(G_{v,m-1})/(m-1)! \end{pmatrix}.$$

A la matriz de la izquierda, de tamaño $m \times m$ y con coeficientes enteros, la llamaremos la « m -ésima matriz de decoraciones»²³ P_m (note que esta depende solo de m). La matriz de decoraciones es invertible; decidimos dejar la demostración de este hecho como un apéndice (ver sección 4.2). Como cada uno de los $\left\{ F(G_{v,\ell}) \right\}_{\ell \in \{0,1,\dots,m-1\}}$ puede computarse con el

²³Esta es una variante de una clase de matrices llamadas «matrices de Pascal». Alan Edelman y Gilbert Strang discuten en detalle cómo factorizar matrices de este tipo en [ES04].

oráculo, y podemos calcular la inversa de la matriz en tiempo polinomial, podemos también recuperar el vector $\mathcal{A}(G; v)$ en tiempo polinomial²⁴. \square

Tenemos el resultado análogo a la proposición 3.6.8 para los vectores de partición en vértices:

Proposición 3.7.6. *Sea $G = (V, E)$ un grafo con $|V| = n$ y $|E| = m$. Supongamos que G tiene un cubrimiento de aristas en cliques con b_k k -cliques para cada $k \in \{2, \dots, n\}$. Entonces, para todo $v \in V$ y todo $j \in [m]$, tenemos que $\mathcal{A}(G; v)_j$ es divisible en*

$$\prod_{k=2}^n ((k-1)!)^{b_k}.$$

Demostración. Sea σ un descascaramiento, y sea $\gamma_0 \in \Gamma_V$ tal que $\gamma(\sigma) = \gamma_0$. Sea H algún k -clique del cubrimiento, y sean w_1, \dots, w_k sus vértices ordenados según γ_0 (salvo, posiblemente, por w_1 y w_2). Notemos que, para cada $i \in \{3, \dots, k\}$, hay i aristas en H que conectan a w_i con algún w_j con $j < i$. Cualquier reordenamiento de esas i aristas corresponde a otro descascaramiento en el que los vértices se conectan en los mismos pasos. Como podemos hacer esto mismo para cada clique del cubrimiento, y los conjuntos de aristas de dichos cliques son disjuntos dos a dos, concluimos el resultado. \square

La utilidad de la proposición 3.7.6 es que el máximo común divisor entre todos los $\mathcal{A}(G; v)_i$, variando $v \in V$ e $i \in [m]$, debiese ser significativamente menor que $F(G)$. Además, hemos visto que, teniendo un oráculo para #GSH, dicho número puede calcularse en tiempo polinomial. Esto nos da un criterio necesario (pero no suficiente) para que exista cierto cubrimiento de aristas en cliques.

Ejemplo 3.7.7. El total de descascaramientos de K_5 es 2 073 600. Este número es divisible en $5! = 34 560$. Dado cualquier vértice v de K_5 , tenemos que

$$\mathcal{A}(K_5, v) = (829\,440, 414\,720, 362\,880, 259\,200, 138\,240, 55\,296, 13\,824, 0, 0, 0).$$

El máximo común divisor de las coordenadas del vector anterior es 3456. Este número no es divisible en $5!$, pero sí es divisible en $4! = 288$.

Sería deseable contar con algún resultado recíproco de la proposición 3.7.6, aunque sea para una clase más reducida de grafos. Esto permitiría establecer una relación entre el problema #GSH y el problema de decisión NP-completo CLIQUE.

A continuación estudiaremos un segundo tipo de vector de partición, esta vez respecto a una arista.

Definición 3.7.8. Sea $G = (V, E)$ un grafo simple con m aristas, y sea $e \in E$ una arista. Definimos la «partición de descascaramientos de G en e » como el vector $\mathcal{B}(G; e) \in \mathbb{N}^m$ tal que, para todo $i \in [m]$, $\mathcal{B}(G; e)_i$ es el número de descascaramientos σ de G tales que la arista e se agrega en el i -ésimo paso del descascaramiento.

La siguiente proposición resume las propiedades básicas de los vectores de partición en aristas.

²⁴Este es un ejemplo de interpolación, un conjunto de técnicas frecuentemente utilizadas para relacionar la dificultad de dos problemas de conteo.

Proposición 3.7.9. Si $G = (V, E)$ es un grafo conexo con m aristas, entonces:

- (i) Para toda $e \in E$, $\sum_{i=1}^m \mathcal{B}(G; e)_i = F(G)$.
- (ii) Para todo $i \in [m]$, $\sum_{e \in E} \mathcal{B}(G; e)_i = F(G)$.
- (iii) Para toda $e \in E$, $\mathcal{B}(G; e)_m = F(G \setminus e)$.
- (iv) Si $e \in E$ y $G \setminus e$ es conexo, entonces $\mathcal{B}(G; e)_i > 0$ para todo $i \in \{1, \dots, m\}$.
- (v) Si $e \in E$ y $G \setminus e$ tiene dos componentes conexas con m_1 y m_2 aristas (de modo que $m_1 + m_2 = m - 1$), entonces $\mathcal{B}(G; e)_i = 0$ si y solo si $i > \max(m_1, m_2) + 1$.
- (vi) Dado un vértice $v \in V$ cuyas aristas incidentes son $e_1, \dots, e_d \in E$, se cumple que $\mathcal{A}(G; v)_1 = \sum_{k=1}^d \mathcal{B}(G; e_k)$.
- (vii) Para toda $e \in E$, $\mathcal{B}(G; e)_1 = \mathcal{B}(G; e)_2$.

Demostración. Los primeros dos puntos se siguen directamente de la definición, y (iii) es la proposición 3.2.4.

Para (iv), tomamos como primeros $i - 1$ pasos al inicio de un descascaramiento de $G \setminus e$ que además comience en un vértice que sea incidente con e ; luego agregamos la arista e , y terminamos de cualquier forma.

Para (v), basta notar que cualquier descascaramiento de G que agrega la arista e en el i -ésimo paso cumplirá que sus primeras $i - 1$ aristas son también aristas de $G \setminus e$. Todas esas aristas deben pertenecer a una misma componente conexa de $G \setminus e$, por lo que $i - 1 \leq \max(m_1, m_2)$. Recíprocamente, si $i - 1 \leq \max(m_1, m_2)$, tomamos como primeros $i - 1$ pasos al inicio de un descascaramiento de la mayor componente conexa de $G \setminus e$ que además comience en un vértice que sea incidente con e ; luego agregamos la arista e , y terminamos de cualquier forma.

Para (vi), notemos que cada descascaramiento que conecta a v en el primer paso necesariamente comienza con una arista que es incidente a v , y viceversa.

Por último, probemos (vii). Para cada par de aristas $a, b \in E$, sea $F_{a,b}$ el número de descascaramientos de G cuya primera arista es a y cuya segunda arista es b . Notemos que $F_{a,b} = F_{b,a}$ para todo par $a, b \in E$, pues la acción de intercambiar las primeras dos aristas es una involución en el conjunto de descascaramientos. Entonces tenemos que

$$\mathcal{B}(G; e)_1 = \sum_{a \in E} F_{e,a} = \sum_{a \in E} F_{a,e} = \mathcal{B}(G; e)_2.$$

□

Ejemplo 3.7.10. Consideremos el grafo $G = (V, E)$ con

$$V = \{1, 2, 3, 4, 5\} \quad , \quad E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{4, 5\}\}.$$

La siguiente matriz tiene de filas a los vectores $\mathcal{B}(G; e)$ en el mismo orden en que presentamos las aristas en el conjunto E :

$$\begin{pmatrix} 6 & 6 & 10 & 14 & 14 \\ 12 & 12 & 10 & 8 & 8 \\ 12 & 12 & 10 & 8 & 8 \\ 16 & 16 & 12 & 6 & 0 \\ 4 & 4 & 8 & 14 & 20 \end{pmatrix}.$$

A continuación mostraremos que los vectores $\mathcal{B}(G; e)$ se pueden construir en tiempo polinomial si tenemos un oráculo para los vectores $\mathcal{A}(G; v)$.

Lema 3.7.11. *Sea $G = (V, E)$ un grafo con $|E| = m$, y sea $e = \{u, v\} \in E$ una arista. Sea $S_e = (V', E')$ el grafo obtenido al subdividir la arista e , agregando un vértice w de modo que $V' = V \sqcup \{w\}$. Entonces se cumple que*

$$\mathcal{B}(G; e)_i = \begin{cases} \frac{1}{2} (\mathcal{A}(G; u)_1 + \mathcal{A}(G; v)_1 - \mathcal{A}(S_e; w)_2) & \text{si } i = 1 \\ \frac{1}{2} (\mathcal{A}(G; u)_i + \mathcal{A}(G; v)_i + \mathcal{A}(S_e; w)_i - \mathcal{A}(S_e; w)_{i+1}) & \text{si } i > 1 \end{cases}$$

Demostración. En primer lugar, afirmamos que

$$\mathcal{A}(S_e; w)_2 = (\mathcal{A}(G; u)_1 - \mathcal{B}(G; e)_1) + (\mathcal{A}(G; v)_1 - \mathcal{B}(G; e)_1) \quad (1)$$

La prueba es un argumento de conteo. En primer lugar, cualquier descascaramiento de S_e que conecta a w por primera vez en el segundo paso debe ser de uno de los siguientes dos tipos:

- Descascaramientos que en el segundo paso agregan a $\{u, w\}$ y que no comienzan con $\{v, w\}$.
- Descascaramientos que en el segundo paso agregan a $\{v, w\}$ y que no comienzan con $\{u, w\}$.

Afirmamos que hay $(\mathcal{A}(G; u)_1 - \mathcal{B}(G; e)_1)$ descascaramientos del primer tipo. Notemos que, por el punto (vi) de la proposición 3.7.9, esta última expresión es igual al número de descascaramientos σ de G que comienzan en una arista incidente a u distinta a e . A cada uno de estos descascaramientos le asociamos el descascaramiento de S_e que comienza igual a σ , luego agrega la arista $\{u, w\}$ y después continúa igual que σ (identificando la arista $\{v, w\}$ de S_e con la arista e de G). Notemos que esta correspondencia es biyectiva entre ambos conjuntos de descascaramientos²⁵. De manera análoga se prueba que $(\mathcal{A}(G; v)_1 - \mathcal{B}(G; e)_1)$ es el número de descascaramientos de S_e que en el segundo paso agregan a $\{v, w\}$ y que no comienzan con $\{u, w\}$. Luego se cumple la ecuación 1, y de esta se deduce que

$$\mathcal{B}(G; e)_1 = \frac{1}{2} (\mathcal{A}(G; u)_1 + \mathcal{A}(G; v)_1 - \mathcal{A}(S_e; w)_2).$$

Más generalmente, si $i \in \{2, \dots, m\}$, entonces se cumple que

$$\mathcal{A}(S_e; w)_i = \sum_{j=1}^{i-1} (\mathcal{A}(G; u)_j - \mathcal{B}(G; e)_j) + \sum_{j=1}^{i-1} (\mathcal{A}(G; v)_j - \mathcal{B}(G; e)_j). \quad (2)$$

En efecto, cualquier descascaramiento de S_e que conecta a w por primera vez en el i -ésimo paso debe ser de uno de los siguientes dos tipos:

- Descascaramientos que en el i -ésimo paso agregan a $\{u, w\}$ y que aún no habían agregado a $\{v, w\}$.

²⁵Lo que estamos usando aquí es que, si contraemos alguna de las dos aristas de $E' \setminus E$, obtenemos un grafo isomorfo a G .

- Descascaramientos que en el i -ésimo paso agregan a $\{v, w\}$ y que aún no habían agregado a $\{u, w\}$.

La primera sumatoria de la ecuación 2 corresponde al número de descascaramientos de G que conectan a u antes del i -ésimo paso, pero que no agregan la arista e antes del i -ésimo paso (el conteo es correcto porque cualquier descascaramiento que conecta a e antes del i -ésimo paso estará considerado exactamente una vez en la suma $\sum_{j=1}^{i-1} \mathcal{A}(G; u)_j$). Podemos usar el mismo argumento biyectivo de antes para mostrar que esta sumatoria también cuenta el número de descascaramientos de S_e que en el i -ésimo paso agregaron a $\{u, w\}$ y que hasta ese punto aún no han agregado a $\{v, w\}$. El argumento para los descascaramientos del otro tipo es análogo.

Finalmente, si $i > 1$, podemos aplicar la ecuación 2 dos veces y obtenemos que

$$\begin{aligned} \mathcal{A}(S_e; w)_{i+1} - \mathcal{A}(S_e; w)_i &= \mathcal{A}(G; u)_i + \mathcal{A}(G; v)_i - 2\mathcal{B}(G; e)_i \\ \therefore \mathcal{B}(G; e)_i &= \frac{1}{2} \left(\mathcal{A}(G; u)_i + \mathcal{A}(G; v)_i + \mathcal{A}(S_e; w)_i - \mathcal{A}(S_e; w)_{i+1} \right). \end{aligned}$$

□

Proposición 3.7.12. *La función $(G, e) \mapsto \mathcal{B}(G; e)$ pertenece a $\mathbf{FP}^{\#\text{GSH}}$.*

Demostración. Sea $G = (V, E)$ un grafo y $e = \{u, v\} \in E$ una arista. Sean $S_e = (V', E')$ y $w \in V' \setminus V$ como en el lema 3.7.11. Usando el oráculo para $\#\text{GSH}$, podemos construir en tiempo polinomial los vectores $\mathcal{A}(G; u)$, $\mathcal{A}(G; v)$ y $\mathcal{A}(S_e; w)$ (proposición 3.7.5). Concluimos usando el lema 3.7.11. □

En cierto sentido, la función $(G, e) \mapsto \mathcal{B}(G; e)$ ofrece información más localizada sobre los descascaramientos. La proposición 3.7.12 establece una equivalencia de oráculos que, según creemos, podría resultar útil en una futura demostración de la intratabilidad de $\#\text{GSH}$.

4. Apéndices

4.1. Código Python para calcular descascaramientos de grafos

Presentaremos un código implementado en Python para calcular el número de descascaramientos de un grafo simple. Este se basa en el algoritmo descrito en la sección 3.6. Codificaremos un grafo $G = (V, E)$ como una lista de 2-tuplas de enteros positivos (que representan las aristas), y asumiremos que $V = [n]$ para algún $n \geq 1$.

Primero definimos una función que sintetiza las adyacencias de G . Esta devuelve una lista $[-1, L_1, L_2, \dots, L_n]$ tal que, para cada $i \in [n]$, L_i es una lista de enteros que describe los vértices adyacentes a $i \in V$. El -1 inicial es solo para simplificar los índices más adelante.

```
def adyacencia(grafo, n):
    ady = [-1]
    for v1 in range(1, n+1):
        fila = []
        for v2 in range(1, n+1):
            # El grafo no es dirigido.
            if ((v1, v2) in grafo) or ((v2, v1) in grafo):
                fila.append(v2)
        ady.append(fila)
    return ady
```

La siguiente función genera todos los órdenes conexos de vértices del grafo, pero forzando un orden en los primeros dos (ver definición 3.6.5). Usamos un algoritmo de *backtracking*. La función `cvo` recibe tres parámetros obligatorios y dos opcionales. Los dos primeros parámetros obligatorios son G y n , y el tercero es la salida de la función `adyacencia` con ese último par como sus argumentos. Los parámetros opcionales se usan en la recursión: el primero es una tupla `vo` de enteros que va registrando el orden de los vértices a medida que avanzamos en la recursión, y el segundo es una tupla `r` con los vértices que aún no hemos visitado.

```
def cvo(grafo, n, ady, vo=(), r=None):
    if len(vo) == n-1:
        # Solo hay un elemento en r.
        yield vo + r
    elif len(vo) == 0:
        r = tuple((vertice for vertice in range(1, n+1)))
        for k in range(1, n+1):
            yield from cvo(grafo, n, ady, vo + (k,), r[:k-1] + r[k:])
    else:
        for k in range(len(r)):
            v = r[k]
            if len(vo) == 1:
                # Forzamos el orden de los primeros dos vértices.
                se_puede_agregar = (vo[0] < v) and (vo[0] in ady[v])
            else:
                se_puede_agregar = len(set(ady[v]) & set(vo)) > 0
            if se_puede_agregar:
                yield from cvo(grafo, n, ady, vo + (v,), r[:k] + r[k+1:])
```

A continuación presentamos la función `descascaramientos`. Esta hace uso del generador `cvo` y aplica la fórmula de la proposición 3.6.3 a cada orden conexo de los vértices del grafo.

```
def descascaramientos(grafo):
    if len(grafo) == 1: return 1
    n = max(max(arista) for arista in grafo)
    ady = adyacencia(grafo, n)
    total = 0
    for orden in cvo(grafo, n, ady):
        alpha, sumando = [], 1
        for j in range(3, n+1):
            conexiones = 0
            for v in ady[orden[j-1]]:
                if orden.index(v) < j-1:
                    conexiones += 1
            alpha.append(conexiones)
            sumando *= conexiones
        saltar = len(grafo) - 1 - alpha.pop(0)
        for i in range(len(grafo)-2, 1, -1):
            if i == saltar:
                saltar -= alpha.pop(0)
            else:
                sumando *= i
        total += sumando
    return total
```

4.2. Invertibilidad de la matriz de decoraciones

A continuación calcularemos el determinante de la matriz P_m que aparece en la demostración de la proposición 3.7.5. Probaremos que $\det(P_m) = (-1)^{(m^2-m)/2} m$, lo que, en particular, demuestra que P_m es invertible²⁶. Usaremos reiteradamente la siguiente identidad: para todo $n \in \mathbb{Z}_{>0}$ y todo $k \in [n-1]$ se cumple que $\binom{n}{k} - \binom{n-1}{k-1} = \binom{n-1}{k}$.

Recordemos que P_m es una matriz de $m \times m$ tal que la entrada en la i -ésima fila y j -ésima coordenada es $\binom{m+i-1}{i-1}$ si $j = 1$, y $\binom{m+i-j-1}{i-1}$ si $j > 1$. Lo que haremos primero es, para cada $i \in [m-1]$, restarle a la $(i+1)$ -ésima fila la fila i -ésima. Entonces

$$\det(P_m) = \begin{vmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ \binom{m}{1} & \binom{m-2}{1} & \binom{m-3}{1} & \cdots & \binom{1}{1} & 0 \\ \binom{m+1}{2} & \binom{m-1}{2} & \binom{m-2}{2} & \cdots & \binom{2}{2} & 0 \\ \binom{m+2}{3} & \binom{m}{3} & \binom{m-1}{3} & \cdots & \binom{3}{3} & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \binom{2m-2}{m-1} & \binom{2m-4}{m-1} & \binom{2m-5}{m-1} & \cdots & \binom{m-1}{m-1} & 0 \end{vmatrix}.$$

²⁶Muchas gracias a Nicolás Vilches por ayudarme a simplificar estos cálculos.

Notemos que $(-1)^{m-1} \det(P_m)$ es igual al determinante de la matriz de $(m-1) \times (m-1)$ que obtenemos al suprimir la primera fila y la última columna de la matriz anterior. Ahora a esta nueva matriz volvemos a aplicarle la misma operación: para cada $i \in [m-2]$, restamos a la $(i+1)$ -ésima fila la fila i -ésima. Obtenemos que

$$\det(P_m) = (-1)^{m-1} \begin{vmatrix} \binom{m}{1} & \binom{m-2}{1} & \binom{m-3}{1} & \cdots & \binom{2}{1} & \binom{1}{1} \\ \binom{m}{2} & \binom{m-2}{2} & \binom{m-3}{2} & \cdots & \binom{2}{2} & 0 \\ \binom{m+1}{3} & \binom{m-1}{3} & \binom{m-2}{3} & \cdots & \binom{3}{3} & 0 \\ \binom{m+2}{4} & \binom{m}{4} & \binom{m-1}{4} & \cdots & \binom{4}{4} & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \binom{2m-3}{m-1} & \binom{2m-5}{m-1} & \binom{2m-6}{m-1} & \cdots & \binom{m-1}{m-1} & 0 \end{vmatrix}.$$

Volvemos a repetir el mismo procedimiento: suprimimos la primera fila y la última columna, y realizamos nuevamente las operaciones en las filas. Entonces

$$\det(P_m) = (-1)^{m-1} (-1)^{m-2} \begin{vmatrix} \binom{m}{2} & \binom{m-2}{2} & \binom{m-3}{2} & \cdots & \binom{3}{2} & \binom{2}{2} \\ \binom{m}{3} & \binom{m-2}{3} & \binom{m-3}{3} & \cdots & \binom{3}{3} & 0 \\ \binom{m+1}{4} & \binom{m-1}{4} & \binom{m-2}{4} & \cdots & \binom{4}{4} & 0 \\ \binom{m+2}{5} & \binom{m}{5} & \binom{m-1}{5} & \cdots & \binom{5}{5} & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \binom{2m-4}{m-1} & \binom{2m-6}{m-1} & \binom{2m-7}{m-1} & \cdots & \binom{m-1}{m-1} & 0 \end{vmatrix}.$$

Si repetimos esto $m-4$ veces más, obtendremos que

$$\det(P_m) = \prod_{k=2}^{m-1} (-1)^k \begin{vmatrix} \binom{m}{m-2} & \binom{m-2}{m-2} \\ \binom{m}{m-1} & 0 \end{vmatrix} = \prod_{k=1}^{m-1} (-1)^k \binom{m}{m-1} = (-1)^{(m^2-m)/2} m.$$

4.3. Código Python para calcular vectores de partición

A continuación presentaremos códigos para calcular los vectores de partición en vértices y en aristas. Al igual que en la sección 4.1, codificaremos un grafo $G = (V, E)$ como una lista de 2-tuplas de enteros positivos, y asumiremos que $V = [n]$ para algún $n \geq 1$.

Para calcular los vectores de partición en vértices, primero definimos una función recursiva `vsh` que genera todos los descascaramientos de G y registra en qué paso fue conectado cada vértice. Dicha función recibe a la codificación del grafo como parámetro obligatorio, y además recibe dos parámetros opcionales, que se usan en la recursión. El primero es un

diccionario `cnx`, de claves y valores enteros, que va registrando en qué paso fue conectado cada vértice (las claves corresponden a los vértices). El segundo parámetro opcional es una tupla `r` con las aristas que aún no hemos agregado.

```
def vsh(grafo, cnx=None, r=None):
    if r is None:
        cnx, r = dict(), tuple(i for i in range(len(grafo)))
        for j in r:
            (a, b) = grafo[j]
            # La arista es de tipo 0.
            yield from vsh(grafo, {**cnx, a: 1, b: 1}, r[:j] + r[j+1:])
    elif len(r) > 0:
        for j in range(len(r)):
            (a, b) = grafo[r[j]]
            k = len(grafo) - len(r)
            ya_conectados = (a in cnx, b in cnx)
            if ya_conectados == (True, True):
                # La arista es de tipo 2.
                yield from vsh(grafo, cnx, r[:j] + r[j+1:])
            elif ya_conectados == (True, False):
                # La arista es de tipo 1 y conecta al vértice b.
                yield from vsh(grafo, {**cnx, b: k+1}, r[:j] + r[j+1:])
            elif ya_conectados == (False, True):
                # La arista es de tipo 1 y conecta al vértice a.
                yield from vsh(grafo, {**cnx, a: k+1}, r[:j] + r[j+1:])
    else:
        yield cnx
```

La siguiente función calcula simultáneamente todos los vectores de partición en vértices. Recibe la codificación de las aristas de $G = ([n], E)$ con $|E| = m$, y devuelve una matriz de $n \times m$ enteros tal que su i -ésima fila es la lista $\mathcal{A}(G; i)$.

```
def particiones_en_vertices(grafo):
    n = max(max(arista) for arista in grafo)
    # Inicializamos una matriz de n por m con entradas 0.
    matriz = [[0] * len(grafo) for _ in range(n)]
    for conexiones in vsh(grafo):
        for (vertice, paso) in conexiones.items():
            matriz[vertice-1][paso-1] += 1
    return matriz
```

Tenemos un algoritmo similar para calcular los vectores de partición en aristas. También hace uso de una función recursiva `esh` que genera todos los descascaramientos de G . Dicha función recibe a la codificación del grafo como parámetro obligatorio, y además recibe tres parámetros opcionales, que se usan en la recursión: el primero de ellos es una tupla `dsc` que, a medida que avanzamos en la recursión, va registrando el orden de las aristas que hemos agregado; el segundo es una tupla `cn` que lleva el registro de los vértices que ya hemos conectado, y el tercero es una tupla `r` con las aristas que aún no hemos agregado.

```

def esh(grafo, dsc=(), cn=(), r=None):
    if r is None:
        r = tuple(i for i in range(len(grafo)))
        for j in r:
            (a, b) = grafo[j]
            n_dsc, n_cn = dsc + (j,), cn + (a, b)
            # La arista es de tipo 0.
            yield from esh(grafo, n_dsc, n_cn, r[:j] + r[j+1:])
    elif len(r) > 0:
        for j in range(len(r)):
            (a, b) = grafo[r[j]]
            ya_conectados = (a in cn, b in cn)
            n_dsc, n_r = dsc + (r[j],), r[:j] + r[j+1:]
            if ya_conectados == (True, True):
                # La arista es de tipo 2.
                yield from esh(grafo, n_dsc, cn, n_r)
            elif ya_conectados == (True, False):
                # La arista es de tipo 1 y conecta al vértice b.
                yield from esh(grafo, n_dsc, cn + (b,), n_r)
            elif ya_conectados == (False, True):
                # La arista es de tipo 1 y conecta al vértice a.
                yield from esh(grafo, n_dsc, cn + (a,), n_r)
    else:
        yield dsc

```

La siguiente función calcula simultáneamente todos los vectores de partición en aristas. Recibe la codificación de las aristas de $G = ([n], E)$ con $|E| = m$, y devuelve una matriz de $m \times m$ enteros tal que su i -ésima fila es la lista $\mathcal{B}(G; e)$, donde e es la i -ésima arista en la codificación de G .

```

def particiones_en_aristas(grafo):
    m = len(grafo)
    # Inicializamos una matriz de m por m con entradas 0.
    matriz = [[0] * m for _ in range(m)]
    for orden_shelling in esh(grafo):
        for j in range(m):
            matriz[orden_shelling[j]][j] += 1
    return matriz

```

Referencias

- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, 2009.
- [BW91] Graham Brightwell and Peter Winkler. Counting linear extensions. *Order*, 8(3):225–242, 1991.
- [BW96] Anders Björner and Michelle L. Wachs. Shellable nonpure complexes and posets. I. *Trans. Amer. Math. Soc.*, 348(4):1299–1327, 1996.
- [CH96] Nadia Creignou and Miki Hermann. Complexity of generalized satisfiability counting problems. *Inform. and Comput.*, 125(1):1–12, 1996.
- [DP20] Samuel Dittmer and Igor Pak. Counting linear extensions of restricted posets. *Electron. J. Combin.*, 27(4):Paper No. 4.48, 13, 2020.
- [ES04] Alan Edelman and Gilbert Strang. Pascal matrices. *Amer. Math. Monthly*, 111(3):189–197, 2004.
- [GP21] Yibo Gao and Junyao Peng. Counting shellings of complete bipartite graphs and trees. *J. Algebraic Combin.*, 54(1):17–37, 2021.
- [GPP⁺19] Xavier Goaoc, Pavel Paták, Zuzana Patáková, Martin Tancer, and Uli Wagner. Shellability is NP-complete. *J. ACM*, 66(3):Art. 21, 18, 2019.
- [Hac08] Masahiro Hachimori. Decompositions of two-dimensional simplicial complexes. *Discrete Math.*, 308(11):2307–2312, 2008.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, The IBM Research Symposia Series, pages 85–103. Plenum, New York-London, 1972.
- [Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley Publishing Company, Reading, MA, 1994.
- [SGW21] Andrés Santamaría-Galvis and Russ Woodroffe. Shellings from relative shellings, with an application to NP-completeness. *Discrete Comput. Geom.*, 66(2):792–807, 2021.
- [Sta18a] Richard P. Stanley. *Algebraic combinatorics*. Undergraduate Texts in Mathematics. Springer, Cham, second edition, 2018. Walks, trees, tableaux, and more.
- [Sta18b] Richard P. Stanley. Counting “connected” edge orderings (shellings) of the complete graph. MathOverflow, 2018. <https://mathoverflow.net/q/297416> (version: 2018-04-11).
- [Val79a] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.
- [Val79b] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8:410–421, 1979.