UNIVERSITAT D'ALACANT
Facultat de Ciències
Facultad de Ciencias

FACULTAD DE CIENCIAS

GRADO EN MATEMÁTICAS

TRABAJO FIN DE GRADO

CURSO ACADÉMICO [2018-2019]

TÍTULO:

Estudio empírico de la complejidad del problema del subgrafo isomofo

AUTOR:

Héctor Masip Ardévol

Resumen

El problema del isomorfismo de grafos es de gran interés tanto en reconocimiento de patrones como en aprendizaje automatizado. De hecho, los grafos se utilizan para apoyar y mejorar algunos tipos de descripciones estructurales y para la representación de imágenes.

Sabemos que, entre los distintos tipos emparejamientos de grafos, el problema del subgrafo isomorfo es un problema NP-Completo; mientras que sigue siendo una pregunta abierta si el problema del isomorfismo de grafos también lo es. Como consecuencia de esto, el tiempo requerido para resolver estos problemas por fuerza bruta crece exponencialmente con el tamaño de los grafos que intentemos procesas, restringiendo la aplicación de técnicas basadas en grafos a problemas que impliquen grafos con un número pequeño de aristas y nodos.

Sin embargo, los algoritmos que funcionan bien para resolver estos problemas en grafos suficientemente grandes, han sido objeto de estudio en las últimas décadas. Algunos de ellos reducen la complejidad computacional del problema imponiendo restricciones topológicas a los grafos. Otros, se centran más en usar una representación adecuada del proceso de búsqueda; eliminando caminos que no se pueden aprovechar en el espacio de búsqueda. En este último sentido, no se imponen restricciones a los grafos y, por tanto, los algoritmos obtenidos se pueden usar de forma general.

En 2017 László Babai, catedrático de la Universidad de Chicago, consiguió demostrar que el problema del isomorfismo de grafos era cuasipolinómico, es decir, encontró un algoritmo para solucionar el problema en tiempo cuasipolinómico. Esto ha supuesto un punto de inflexión en ciencias de la computación, ya que un problema que, intuitivamente, se pensaba que podía ser NP-Completo, ha resultado ser que está más cerca de ser un problema P.

En este trabajo introduciremos formalmente todo esto. Las dos primeras secciones tienen como objetivo dar una base de definiciones y demostraciones necesarias para entender el porqué de la NP-Completitud del problema del subgrafo isomorfo así como una breve introducción a la teoría de grafos. En las secciones 4 y 5 se presentarán las clases de grafos que trataremos y un modelo que las utiliza y que automatiza un problema de clasificación basado en grafos. Las secciones 6 y 7 son unos resultados experimentales que se llevan a cabo con el modelo y su complejidad temporal.

Abstract

The exact graph matching problem is of interest in a variety of different pattern recognition contexts. In fact, graphs are used to support structural descriptions as well as for low level image representations.

As it is well known, among the different types of graph matching subgraph isomorphism is a NP-complete problem, while it is still an open question if also graph isomorphism is a NP-complete problem too. As a consequence, time requirements of brute force matching algorithms increase exponentially with the size of the input graphs, restricting the applicability of graph based techniques to problems implying graphs with a small number of nodes and edges.

Therefore, algorithms having time requirements suited for matching large graphs, have been a subject of research during the last three decades. Some of them reduce the computational complexity of the matching process by imposing topological restrictions on the graphs. Others are focused in using an adequate representation of the searching process and pruning unprofitable paths in the search space. In this way, no restrictions must be imposed on the structure of the input graphs and the obtained algorithms can be generally applied.

In 2017 László Babai, University of Chicago professor, managed to prove that the graph matching problem is Quasi-polynomial, meaning that he found an algorithm to solve it in quasi-polynomial time. This has supposed an inflecition point in computer science since it has been intuitively believed that this problem was NP-Complete, Babai showed that that is closer to be a P problem.

In this work we will formaly introduce all of this. The two first sections have the objective of giving some necessary definitions and proves to understand the subgraph isomorphism NP-Completeness as well as a brief introduction about graph theory. In sections 4 and 5 it will be introduced the graph classes that we will treat and a model that use them to compute a classification problem based on graphs. The last sections 6 and 7 show some experimental results that are carried out with the model and his time complexity.

Índice

1. Introducción **6**

2. Teoría de la complejidad computacional **8**

2.1. Definiciones y conceptos básicos 8

2.2. Problemas de decisión 9

2.3. Máquinas de Turing deterministas y la clase P 10

2.4. Máquinas de Turing no-deterministas y la clase NP 13

2.5. Relación entre las clases P y NP y la clase NP-Completa 17

2.6. NP-Completitud del problema del subgrafo isomorfo 22

3. Teoría de grafos **25**

3.1. Introducción a Grafos 25

3.2. Grafo extendido y lenguaje de grafos 28

3.3. Multiconjuntos 29

4. K-testabilidad de grafos **30**

5. Modelo de automátas de grafos y algoritmo **33**

5.1. Autómata de grafos 33

5.2. Definición del algoritmo 35

5.3. Eficiencia del algoritmo 38

6. Resultados del algoritmo **40**

6.1. Bases de datos 40

6.2. Resultados experimentales 42

7. Comparativa de algoritmos **43**

7.1. Algoritmos 44

7.2. Comparativa 47

Referencias **49**

A. Apéndice **50**

A.1. NP-Completitud del problema del subgrafo isomorfo 50

A.2. Relaciones binarias 52

A.3. Matriz de vecindades e isomorfismo 53

1. Introducción

El problema del subgrafo isomorfo es una generalización del problema del isomorfismo de grafos en el cual se tienen dos grafos y en el que responder a la pregunta de si un grafo contiene un subgrafo isomorfo al otro.

En general, no es fácil responder a esta pregunta; pero disponemos de una rama de la teoría de la computación, llamada teoría de la complejidad computacional, que se ocupa de evaluar el coste computacional de la pregunta anterior.

Más concretamente, la teoría de la complejidad computacional se centra en la clasificación de los problemas computacionales de acuerdo con su dificultad inherente, y en la relación entre dichas clases de complejidad.

Un problema se cataloga como "inherentemente difícil" si su solución requiere de una cantidad significativa de recursos computacionales, sin importar el algoritmo utilizado. La teoría de la complejidad computacional formaliza dicha aseveración, introduciendo modelos de computación matemáticos para el estudio de estos problemas y la cuantificación de la cantidad de recursos necesarios para resolverlos, como tiempo y memoria.

Veamos un ejemplo que nos ayude a entender el objetivo de esta materia.

Supongamos que eres un empleado en una empresa de producción de equipos electrónicos y software. Un día tu jefe te llama y te dice que la empresa quiere mejorar los algoritmos de los que disponéis para poder acceder a un mercado más competitivo en ese sector. Para esto, necesitáis de un buen método para determinar si un conjunto de especificaciones funcionan bien para un nuevo tipo de componente que va a salir al mercado. Como eres el jefe del departamento de diseño de algoritmos, tu tarea es encontrar un algoritmo eficiente que haga lo anterior.

Después de consultar con tus compañeros de departamento para determinar a que tipo de problema os estáis enfrentando, te vas corriendo a tu despacho, colocas varios libros sobre tu mesa y te sumerges en la tarea con un gran entusiasmo. Algunas semanas más tarde, tu entusiasmo con el algoritmo ha decrecido de manera exponencial. Después de todo este tiempo, no has sido capaz de encontrar un algoritmo mejor que pasar por buscar entre todas las posibles combinaciones de especificaciones. Esto no le gustará mucho a tu jefe, ya que tomaría años y años de computación intentar asociar un simple puñado de especificaciones y el departamento tiene que realizarlo para unas cuantas componentes.

La idea no es ir a tu jefe y decirle: "No puedo encontrar un algoritmo eficiente. La verdad es que no soy tan brillante como pensaba". Para evitar daños a tu posición en la

empresa, sería mucho mejor si pudieras probar que el problema es intractable, es decir, que ningún algoritmo pueda resolverlo de forma eficiente. En ese caso, podrías ir a la oficina de tu jefe y decirle con total confianza: "No puedo encontrar un algoritmo eficiente, !pero es que imposible encontrar uno!".

Desafortunadamente, probar la intractabilidad de un algoritmo puede ser tan difícil como encontrar algoritmos eficientes. Incluso los mejores teóricos han fallado al intentar encontrar demostraciones para problemas muy conocidos. Sin embargo, la teoría de la complejidad trata de dar técnicas para probar que un problema dado es tan difícil como un gran lista de problemas muy conocidos. Armado con estas técnicas, tu podrías demostrar que el problema es NP-Completo y, por lo tanto, que es equivalente a esta lista de problemas conocidos. Así, podrías ir a la oficina de tu jefe y decirle: "Yo no puedo encontrar un algoritmo eficiente para nuestro problema, pero tampoco lo puede hacer ningún experto en el mundo".

Gracias a esto asegurarías tu puesto en la empresa, ya que si tu jefe decidiera reemplazarte para encontrar a alguien que sepa más que tú del tema; llegaría a los mismos resultados.

Descubrir que un problema es NP-Completo es solamente el punto de partida a la hora de solucionar el problema. Los problemas de tu departamento no desaparecerán de la noche a la mañana simplemente por saber que tu problema es NP-Completo. Sin embargo, saber que un problema es NP-Completo nos da una información muy valiosa sobre que caminos no tomar a la hora de intentar resolverlo. De hecho, la búsqueda de un algoritmo eficiente para un problema de este estilo se suele considerar de una prioridad baja.

Después de tener clara la situación del algoritmo en el mundo de la teoría de la complejidad, podemos centrarnos en unas aproximaciones menos ambiciosas. Por ejemplo, podríamos buscar un algoritmos eficiente que resuelva casos especiales o particulares del problema. O también, podríamos relajar el problema buscando un algoritmo que funcione para la gran mayoría de las especificaciones del componente.

A continuación, introduciremos todos los conceptos de los que hemos estado comentando a lo largo de este ejemplo. Así como las ideas fundamentales de lo que compone la teoría de la complejidad computacional.

2. Teoría de la complejidad computacional

El objetivo de esta sección es hacer entender al lector lo que se entiende por NP-Completitud, pasando por las máquinas de Turing y las definiciones de los lenguajes P y NP. Así, conoceremos la clase de los lenguajes que constituyen esta teoría y estaremos a disposición de probar que el problema del subgrafo isomorfo es NP-Completo.

2.1. Definiciones y conceptos básicos

En general, estamos interesados en encontrar el algoritmo más "eficiente" para resolver un problema. En un sentido amplio, la noción de eficiencia involucra a todos los recursos que la computación necesita para ejecutar un algoritmo. Sin embargo, cuando hablamos de el "más eficiente", normalmente nos referiremos al más rápido. Como el tiempo es siempre un factor determinante para saber si un algoritmo es suficientemente bueno, vamos a centrarnos en él.

El tiempo requerido por una algoritmo se suele expresar en términos de una única variable, el "tamaño" de una instancia de un problema (un caso particular del mismo). Esto es conveniente porque se espera que la dificultad de una instancia de un problema varíe con su tamaño.

Cabe observar que la descripción de una instancia de un problema que damos como input al ordenador puede ser visto como una única cadena finita de símbolos elegidos de un alfabeto finito. Como hay muchas formas (de hecho infinitas) de hacer esto, asumiremos que una manera particular ha sido elegida y que cada problema tiene asociado un **esquema de codificación**, que básicamente asocia cada instancia de un problema con las cadenas que los describen. La **longitud de entrada** de una instancia I de un problema Π se define como el número de símbolos en la descripción de I obtenida por el esquema de codificación para Π. La longitud de entrada es lo que usaremos para medir el tamaño de una instancia.

Ejemplo 2.1 *Un problema con el que vamos a estar permanentemente trabajando en esta introducción es el problema del viajero: Los parámetros de este problema son un conjunto finito* $C = \{c_1, c_2, \ldots, c_m\}$ *de ciudades y para cada par de ciudades* c_i, c_j *en* C*, la distancia entre ellas* $d(c_i, c_j)$*. La idea es encontrar la mínima distancia de tal forma que pasemos por todas ellas y volvamos a la primera visitada.*

Este problema puede ser descrito usando el alfabeto $\{c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$*. Así, por ejemplo, "*$c[1]c[2]c[3]c[4]//10/5/9//6/9//3$*", describiría el conjunto de ciudades*

y la distancia entre ellas. La longitud del mismo sería de 32.

Un concepto muy importante es la función de complejidad temporal:

Definición 2.2 *Dado un algoritmo y una longitud de entrada para una determinada instancia de un problema, la* ***función de complejidad temporal*** *expresa la mayor cantidad de tiempo que el algoritmo necesita para resolver la instancia del problema.*

Veamos ahora la definición que nos ayudará a formalizar lo que entendemos por un problema difícil.

Definición 2.3 *Diremos que una función $f(n)$ es $O(g(n))$ cuando exista una constante c tal que $|f(n)| \leq c|g(n)|$ para todo $n \geq 0$. Un* ***algoritmo de tiempo polinómico*** *es aquel que su función de complejidad temporal es $O(p(n))$ para algún polinomio p, donde n denota la longitud de entrada. En otro caso diremos que tenemos un* ***algoritmo de tiempo exponencial****.*

Se suele decir que un problema no ha sido "resuelto" hasta que un algoritmo de tiempo polinómico lo resuelve. Por lo tanto, nos referiremos a un problema como "intractable" si es tan difícil que ningún algoritmo pueda resolverlo en tiempo polinómico.

2.2. Problemas de decisión

Toda la teoría de la NP-Completitud esta diseña para ser aplicada a **problemas de decisión**. Estos problemas solo tienen dos posibles soluciones, la primera es "sí" y la segunda es "no". Los problemas se suelen enunciar en el siguiente formato:

a) Especificando una instancia genérica del problema.

b) Formulando una pregunta de sí y no acerca del mismo.

Por ejemplo, lo siguiente describe el problema de decisión que nos concierne en este trabajo:

PROBLEMA DEL SUBGRAFO ISOMORFO

INSTANCIA: Dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$.

PREGUNTA: G_1 contiene un subgrafo isomorfo a G_2?

Formalizaremos estos conceptos en la siguiente sección.

La razón por la cual nos restringimos a problemas de decisión es porque tienen asociada una estructura, que funciona muy bien en un sentido matemático y formal. Dicha estructura se denomina lenguaje y se define como sigue:

Definición 2.4 *Dado un conjunto finito Σ de símbolos, denotaremos por Σ^* el conjunto de todas las cadenas finitas que podemos formar con símbolos en Σ. Un* ***lenguaje*** *se define como cualquier subconjunto $L \subseteq \Sigma^*$ sobre el alfabeto Σ.*

La relación entre los problemas de decisión y los lenguaje viene determinada por el esquema de codificación. Observemos que para un problema Π, el esquema de codificación e particiona Π en tres posibles clases de cadenas:

I) Cadenas que no son codificaciones de instancias de Π.

II) Cadenas que codifican instancias de Π para las cuales la respuesta es "no".

III) Cadenas que codifican instancias de Π para las cuales la respuesta es "si".

Esta tercera clase es la que definimos como el lenguaje asociado a Π y a e y escribiremos como:

$L[\Pi, e] = \{x \in \Sigma^* : \text{x es la codificación por e de una instancia para la cual la respuesta es "sí"}\}$

A partir de ahora, cada vez que introduzcamos un concepto nuevo en términos de lenguaje, observaremos que el concepto en sí en independiente del esquema de codificación que usemos. Esto significa que si e y e' son dos esquemas de codificación para Π, entonces el concepto se sostiene para $L[\Pi, e]$ y para $L[\Pi, e']$ o para ninguno.

Notemos que si funcionamos de esta manera, de alguna forma estamos perdiendo la noción de longitud de entrada. Por eso es conveniente asumir que cada problema de decisión Π tiene una función Longitud:$D_\Pi \longrightarrow \mathbb{Z}^+$ (donde D_Π denota el conjunto de instancias de un problema) que no dependa del esquema de codificación y que esté "polinómicamente relacionada" a las distintas longitudes de entrada que obtenemos para esquemas de codificación. Por "polinómicamente relacionada" no referimos a lo siguiente: para cualquier esquema de codificación e para Π, existen dos polinomios p y p' tales que si $I \in D_\Pi$ y x es una cadena que codifica I bajo e, entonces $Longitud[I] \leq p(|x|)$ y $|x| \leq p'(Longitud(I))$; donde $|x|$ denota la longitud de la cadena x.

2.3. Máquinas de Turing deterministas y la clase P

Para poder formalizar la noción de algoritmo, necesitamos fijar un modelo particular de ordenador. El modelo será la máquina de Turing determinista de una cinta (abreviadamente DTM). El modelo consiste en un registro de estado, un cabezal y una cinta constituida por una secuencia infinita de celdas bidireccional, que etiquetaremos por $\ldots, -2, -1, 0, 1, 2, 3, \ldots$

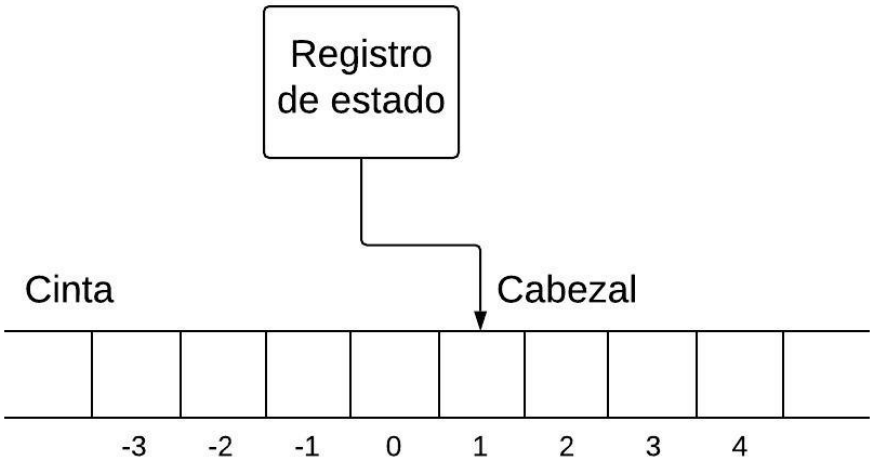


Figura 1: *Representación esquemática de una máquina de Turing determinista de una cinta.*

Un programa para una DTM especifica la siguiente información:

(1) Un conjunto finito Γ de símbolos, incluyendo un subconjunto $\Sigma \subset \Gamma$ de símbolos de entrada y un símbolo vacío $b \in \Gamma - \Sigma$.

(2) Un conjunto finito de estados Q, incluyendo el estado inicial q_0 y dos estados de detención q_Y y q_N.

(3) Una función de transición $\delta : (Q - \{q_Y, q_N\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 1\}$.

La entrada a la DTM es una cadena $x \in \Sigma^*$, que se coloca desde la celda 1 hasta la celda $|x|$; un símbolo por celda. Inicialmente, todas las demás celdas contienen el símbolo vacío. El programa comienza su funcionamiento en el estado q_0, con el cabezal escaneando la celda 1. Todo el funcionamiento se produce paso a paso. Si el estado actual q es q_Y o q_N, entonces el proceso termina, con la respuesta "sí" si $q = q_Y$ y "no" si $q = q_N$. En otro caso, $q \in (Q - \{q_Y, q_N\})$, algún símbolo $s \in \Gamma$ está en la celda siendo escaneada y se define el valor $\delta(q, s) = (q', s', \Delta)$. Ahora el cabezal borra el símbolo s, escribe s' en su lugar, y se mueve una celda a la izquierda si $\Delta = -1$ o una celda a la derecha si $\Delta = 1$. Al mismo tiempo, el registro de entrada cambia su estado de q a q'. Esto completa un paso del proceso y estamos listos para el siguiente paso, si es que lo hay.

Definición 2.5 *Diremos que un programa M de una DTM con el alfabeto de entrada Σ* ***acepta*** *$x \in \Sigma^*$ si, y solo si, M para en el estado q_Y al aplicarlo sobre x. El lenguaje L_M reconocido por el programa M se definide por el conjunto:*

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

Observemos que no hemos dicho nada sobre si las cadenas que no pertenezcan a L_M tengan que parar en algún momento dado. Sin embargo, para una mejor aproximación a nuestra definición de algoritmo, forzaremos a que la DTM siempre pare en algún momento.

La relación que hay entre reconocer lenguajes y resolver problemas de decisión es sencilla. Diremos que un programa M de una DTM resuelve un problema de decisión Π bajo un esquema de codificación e si M se para para cualquier cadena de entrada y $L_M = L[\Pi, e]$.

Ejemplo 2.6 *Consideremos el siguiente problema de decisión de teoría de números:*

DIVISIBILIDAD DE UN ENTERO POR CUATRO

INSTANCIA: Un entero positivo N.

PREGUNTA: Existe algun entero positivo m tal que $N = 4m$?

Para resolver el problema, recurriremos al siguiente programa de una DTM:

$$\Gamma = \{0, 1, b\},\ \Sigma = \{0, 1\}$$

$$Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$$

	0	1	b
q_0	$(q_0, 1, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

La tabla describe los valores de la función de transición δ, donde la entrada en la fila q y en la columna s corresponde con el valor $\delta(q, s)$. No es difícil comprobar que este programa reconoce el lenguaje

$$\{x \in \{0, 1\}^* : \textit{los dos últimos símbolos de } x \textit{ son } 0\}$$

Debido a esto último y como sabemos que un entero positivo escrito en binario es divisible por cuatro si, y solo si, los dos últimos dígitos en su representación son los dos 0; tenemos que el programa resuelve nuestro problema de decisión, ya que el programa para para cualquier cadena que le introduzcamos.

Ahora estamos a disposición de poder definir el concepto de *complejidad temporal.*

Definición 2.7 *El **tiempo** usado en la computación de un programa M de una DTM con una entrada x se define como el número de pasos que hay en la computación hasta que llegamos a un estado de detención. Definimos la **función de complejidad temporal** $T_M : \mathbb{Z}^+ \longrightarrow \mathbb{Z}^+$ como*

$T_M(n) = \max\{m : \exists x \in \Sigma^*,\ |x| = n,\ \textit{tal que la computación de } M \textit{ en } x \textit{ toma tiempo } m\}$

Por lo tanto, diremos que un programa M es un **programa DTM con tiempo polinómico** si existe un polinomio p tal que $T_M(n) \leq p(n), \forall n \in \mathbb{Z}^+$

Ahora podemos definir la primera clase importante de lenguajes que vamos a considerar en todo el trabajo, la clase P.

Definición 2.8 *La clase P es el conjunto de lenguajes para los cuales existe un programa M de una DTM con tiempo polinómico, i.e.*

$$P = \{L : \exists\ M \textit{ de una DTM con tiempo polinómico tal que } L = L_M\}$$

Diremos que un problema de decisión Π pertenece a P bajo el esquema de codificación e si $L[\Pi, e] \in P$, esto es, que exista un programa M de una DTM con tiempo polinómico que resuelva Π bajo el esquema e.

2.4. Máquinas de Turing no-deterministas y la clase NP

En esta sección introduciremos la segunda clase de lenguajes, la clase NP. Antes de hacerlo formalmente, veremos una idea intuitiva de una noción informal de lo que esta clase significa.

Consideremos el problema del viajero con el que iniciamos este trabajo pero con un añadido: Dado un conjunto de ciudades, las distancias entre ellas y una cota B, existe algún recorrido de todas las ciudades con longitud menor o igual a B? Por ahora no se conoce ningún algoritmo en tiempo polinómico que lo resuelva. Sin embargo, supongamos que alguien afirma, para una instancia particular del problema, que la respuesta para esa instancia es "sí". Sería muy fácil probar o desmentir lo que esa persona afirma; simplemente comprobando que lo que nos indica es un recorrido y viendo que la longitud es menor o igual a B. Por lo tanto, sería muy fácil verificar, en tiempo polinómico, si ha resuelto la instancia del problema o no.

Observar que la verificación en tiempo polinómico no implica necesariamente la resolución en tiempo polinómico. Esta noción de verificabilidad es la que la clase NP intenta aislar. Informalmente podríamos definir NP en base a términos de lo que vamos a llamar algoritmo no-determinista. Este algoritmo está compuesto por dos etapas; una primera,

que será una etapa de suposición, y una segunda, que será una etapa de comprobación. Dada una instancia de un problema I, la primera etapa lo que hará es adivinar alguna estructura S. Entonces introducimos tanto a I como a S como entradas en la etapa de comprobación, donde se procede de una manera determinista hasta que se pare con la respuesta "sí",la respuesta "no" o nunca se pare.

Un algoritmo no-determinista resuelve un problema de decisión Π si las siguientes propiedades se sostienen para cualquier $I \in D_\Pi$:

1. Si $I \in Y_\Pi$, entonces existe alguna estructura S tal que, cuando es adivinada por la etapa de adivinación, la etapa de comprobación se parará con un "sí" para I y para S; siendo Y_Π el conjunto de instancias de Π para las cuales la respuesta es "sí".

2. Si $I \notin Y_\Pi$, entonces no existe ninguna estructura S tal que, cuando es adivinada por la etapa de adivinación, la etapa de comprobación se parará con un "sí" para I y para S.

Definición 2.9 *Un algoritmo no-determinista que resuelve un problema* Π *se dice que opera en tiempo polinómico si existe algún polinomio* p *tal que, para toda* $I \in Y_\Pi$*, existe alguna estructura* S *que haga a la etapa de comprobación pararse en "sí" para* I *y para* S *en tiempo* $p(Longitud(I))$

Así, la clase NP está definida informalmente como la clase de todos los problemas de decisión Π que puedan ser resueltos en tiempo polinómico por un algoritmo no-determinista.

Una observación muy importante es que, como un algoritmo determinista para para cualquier entrada, entonces si un problema Π pertenece a P; el problema complementario también lo hace (bastaría cambiar los síes por noes). Esto no pasa para una máquina no-determinista, y por lo tanto si un problema pertenece a NP, su complementario no tiene por que hacerlo. Esta observación nos ayudará a entender, más adelante, la relación entre estas clases.

Una máquina de Turing no-determinista de una cinta (abreviadamente NDTM) es un modelo que está estructurado como una DTM, a excepción de que ahora tenemos un módulo de adivinación (que también podrá escribir en la cinta, pero no leer).

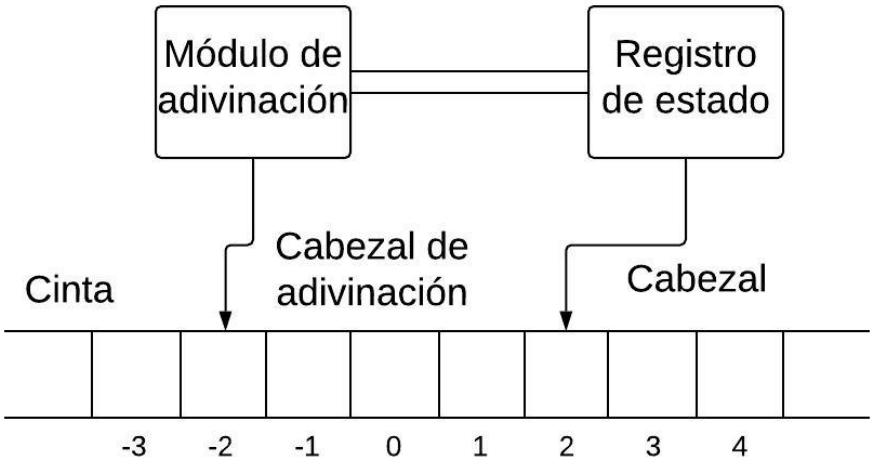


Figura 2: *Representación esquemática de una máquina de Turing no-determinista de una cinta.*

Un programa para la $NDTM$ especifica lo mismo que un programa para una DTM:

(1) Un conjunto finito Γ de símbolos, incluyendo un subconjunto $\Sigma \subset \Gamma$ de símbolos de entrada y un símbolo vacío $b \in \Gamma - \Sigma$.

(2) Un conjunto finito de estados Q, incluyendo el estado inicial q_0 y dos estados de detención q_Y y q_N.

(3) Una función de transición $\delta : (Q - \{q_Y, q_N\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 1\}$.

La computación de un programa de una $NDTM$ difiere de una DTM debido a que tiene lugar en dos etapas. La primera etapa, como hemos mencionado anteriormente, es la etapa de adivinación. Inicialmente, la cadena de entrada x se escribe desde la celda 1 hasta la celda $|x|$ (todas las demás celdas contienen en símbolo vacío b). El cabezal está escaneando la celda 1, el módulo de adivinación está escaneando la celda -1 y el registro de estado está inactivo. Ahora, el módulo de adivinación escribe algún símbolo de Γ en la celda -1 y se mueve una celda hacia la izquierda, o se para, en cuyo caso el módulo de adivinación se vuelve inactivo y el registro de estado se activa en el estado q_0. La elección del símbolo de Γ que se escoge y cuando se para el módulo de adivinación se hace de una manera totalmente aleatoria. Por lo tanto, el módulo puede escribir cualquier símbolo de Γ^* antes de pararse, si alguna vez lo hace.

La etapa de comprobación empieza cuando el registro de estado se activa en el estado q_0. A partir de aquí, la computación procede exactamente de la misma manera que se hacía

en una DTM. La computación termina cuando el registro de estado introduce alguno de los estado de detención y se dice que se acepta la computación si para en el estado q_Y. En cualquier otro caso, deteniéndose o no, se dice que se rechaza la computación.

Observar que cualquier programa M de una $NDTM$ tendrá un número infinito de posibles computaciones, una por cada posible cadena que se escriba de Γ^* por el módulo de adivinación. Por lo tanto,

Definición 2.10 *Diremos que un programa M de una $NDTM$ con el alfabeto de entrada Σ* ***acepta*** *$x \in \Sigma^*$ si, y solo si, M para en el estado q_Y al aplicarlo sobre x para alguna de las posibles computaciones en la primera etapa. El lenguaje L_M reconocido por el programa M es definido por el conjunto*

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

Análogamente a como lo hicimos en el caso de una DTM:

Definición 2.11 *El* ***tiempo*** *usado en la computación de un programa M de una $NDTM$ en una entrada x se define como el mínimo del número de pasos (para todas las posibles computaciones que aceptamos para x) que hay hasta que llegamos a un estado de detención; contando las dos etapas. Definimos la* ***función de complejidad temporal*** *$T_M : \mathbb{Z}^+ \longrightarrow \mathbb{Z}^+$ como*

$$T_M(n) = \max\{\{1\} \cup \{m : \exists x \in \Sigma^*,\ |x| = n,\ \text{tal que la computación de } M \text{ en } x \text{ toma tiempo } m\}\}$$

Por lo tanto, diremos que un programa M es un **programa DTM con tiempo polinómico** si existe un polinomio p tal que $T_M(n) \leq p(n)$, $\forall n \in \mathbb{Z}^+$.

Finalmente, podemos definir la segunda clase importante de lenguajes que vamos a considerar en todo el trabajo, la clase NP.

Definición 2.12 *La clase NP es el conjunto de lenguajes para los cuales existe un programa M de una $NDTM$ con tiempo polinómico, i.e.*

$$NP = \{L : \exists\ M \text{ de una } NDTM \text{ con tiempo polinómico tal que } L = L_M\}$$

Diremos que un problema de decisión Π pertenece a NP bajo el esquema de codificación e si $L[\Pi, e] \in NP$, esto es, que exista un programa M de una $NDTM$ con tiempo polinómico que resuelva Π bajo el esquema e.

En la siguiente sección discutiremos la relación que existe entre las clase P y NP y que nos servirá de antesala a la introducción de la clase más importante, la clase de problemas NP-Completos.

2.5. Relación entre las clases P y NP y la clase NP-Completa

Empezaremos esta sección probando el siguiente lemma, que nos indica la ya obvia relación entre las clases P y NP.

Lema 2.13 $P \subseteq NP$.

Demostración.

Observamos que cada problema de decisión que se puede resolver mediante un algoritmo determinista de tiempo polinómico también se puede resolver mediante un algoritmo no-determinista de tiempo polinómico. Para ver esto, basta con observar que todo algoritmo determinista puede ser usado como etapa de comprobación de uno no-determinista. Así, si $\Pi \in P$ y A es un algoritmo determinista de tiempo polinómico; podemos obtener un algoritmo no-determinista de tiempo polinomico simplemente usando a A como la etapa de comprobación e ignorando la etapa de comprobación. Por lo tanto $\Pi \in NP$.

■

Aunque aún no esté demostrado, hay muchas razones para pensar que la inclusión anterior es estricta ya que los algoritmos no-deterministas parecen un tipo de algoritmos mucho más poderosos; como indica la demostración anterior. Además que no hay métodos que nos transformen unos en otros. De hecho, el mejor resultado que uno puede probar es el siguiente:

Teorema 2.14 *Si* $\Pi \in NP$*, entonces existe un polinomio* p *tal que* Π *puede ser resuelto por un algoritmo determinista de tiempo exponencial* $O(2^{p(n)})$.

Demostración.

Sea A un algoritmo no-determinista con tiempo polinómico que resuelve Π, y sea $q(n)$ el polinomio que acota la función de complejidad temporal de A. Por lo tanto, sabemos que para cualquier entrada aceptada de longitud n, existirá alguna cadena adivinada (sobre el alfabeto Γ) de longitud a lo sumo $q(n)$ que haya hecho que la etapa de comprobación responda "sí", en no más de $q(n)$ pasos. De esta manera, el número total de adivinaciones que tenemos que considerar es $k^{q(n)}$, con $k = |\Gamma|$ (rellenando con el símbolo de vació si es necesario). Ahora, podemos descubrir de forma determinista si A acepta una computación para una longitud de entrada n simplemente aplicando le etapa de comprobación determinista de A; hasta que se pare o realice $q(n)$ pasos para cada uno de los casos que tenemos. Esto, claramente produce un algoritmo determinista para resolver Π. Además, su función de complejidad temporal es $q(n)k^{q(n)}$ y, por lo tanto, existirá algún polinomio

p tal que esta función sea $O(2^{p(n)})$.

■

Debido al teorema anterior, de forma general se suele pensar que $P \neq NP$, incluso estando muy alejados de una demostración de dicha afirmación. Los problemas se suelen ver como intratables hasta que encontremos un algoritmo eficiente que los resuelva. De todas formas, trabajaremos bajo la sospecha de que la región $NP-P$ no está deshabitada completamente.

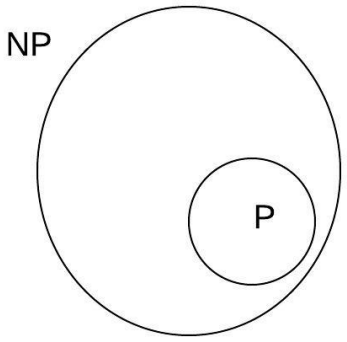


Figura 3: *La supuesta esquematización que se tiene del mundo P/NP.*

Si P difiere mucho de NP, entonces la distinción entre P y $NP-P$ es muy importante. De todas formas, hasta que no se pruebe que $P \neq NP$, no hay ninguna esperanza de que podamos probar que algún problema pertenezca a $NP-P$ (solo haría falta encontrar uno de estas condiciones).

Aunque parezca complicado probar que un problema pertenece a NP, existen algunas técnicas (que nos llevará a la definición de la última clase de este trabajo) que nos reducirán el problema drásticamente. Pare ello, necesitamos ver cómo un programa de una DTM puede generar funciones; que veremos como funciones asociadas al programa.

Definición 2.15 *Sea un programa M de una DTM con alfabeto de entrada Σ y alfabeto de escritura Γ que se para para cualquier cadena $x \in \Sigma^*$. M genera la función $f_M : \Sigma^* \longrightarrow \Gamma^*$, que se define para cada $x \in \Sigma^*$ como la cadena obtenida empezando en la celda 1 y acabando en la celda de más a la derecha que sea distinta del vacío.*

Ahora estamos en condiciones de poder definir una transformación polinómica, clave en esta teoría.

Definición 2.16 *Una **transformación polinómica** de un lenguaje $L_1 \subseteq \Sigma_1^*$ a otro*

$L_2 \subseteq \Sigma_2^*$ es una función $f : \Sigma_1^* \longrightarrow \Sigma_2^*$ tal que cumple las condiciones:

1. *Existe un programa DTM de tiempo polinómico que genera f.*

2. *$\forall x \in \Sigma_1^*$, $x \in L_1$ si y solo si $f(x) \in L_2$.*

Si existe una transformación polinómica de L_1 a L_2, escribiremos $L_1 \propto L_2$. El símbolo está escogido adrede ya que la propiedad simétrica de esta relación no se cumple en general.

La importancia de esta definición es debida, entre otras cosas, al siguiente lema:

Lema 2.17 *Si $L_1 \propto L_2$, entonces $L_2 \in P$ implica que $L_1 \in P$. Equivalentemente, $L_1 \notin P$ implica que $L_2 \notin P$.*

Demostración.

Sean Σ_1 y Σ_2 los alfabetos de L_1 y L_2 respectivamente, sea $f : \Sigma_1^* \longrightarrow \Sigma_2^*$ una transformación polinómica de L_1 a L_2, sea también M_f un programa de una DTM de tiempo polinómico que general f y sea M_2 un programa DTM de tiempo polinómico que reconoce L_2.

Un programa DTM de tiempo polinómico que reconoce L_1 se puede construir mediante M_f y M_2 de la siguiente manera. Dada una entrada $x \in \Sigma_1^*$, primero aplicamos la porción que corresponde al programa M_f para construir $f(x) \in \Sigma_2^*$. Después aplicamos la porción que corresponde al programa L_2 para determinar si $f(x) \in L_2$. Como $x \in L_1$ si y solo si $f(x) \in L_2$, esto nos lleva a determinar el programa que reconoce L_1.

Como M_f y M_2 son dos programas de tiempo polinómico, entonces el que sale de la composición de ambos también lo será. Formalmente, si p_f y p_2 son los dos polinomios que acotan los tiempos de ejecución de M_f y M_2, entonces $|f(x)| \leq p_f(|x|)$, y el tiempo de ejecución del programa construido será $O(p_f(|x|) + p_2(p_f(|x|)))$. Para acabar bastaría tomar un polinomio como la suma de ambos anteriores.

■

En términos de problemas de decisión, una transformación polinómica de Π_1 a Π_2 es una función $f : D_{\Pi_1} \longrightarrow D_{\Pi_2}$ que verifica:

1. f es computable por un algoritmo en tiempo polinómico.

2. $\forall I \in D_{\Pi_1}$, $I \in Y_{\Pi_1}$ si y solo si$f(I) \in Y_{\Pi_2}$.

Antes de pasar al siguiente resultado veamos un ejemplo de esto. Dado un grafo $G = (V, E)$ siendo V el conjunto de vértices y E el conjunto de aristas, llamaremos **circuito simple** en G a una secuencia $< v_1, \ldots, v_k >$ de vértices distintos de V tales que $\{v_i, v_{i+1}\} \in E$ para $1 \leq i < k$ y tal que $\{v_k, v_1\} \in E$. Un **circuito hamiltoniano** es un circuito simple que contiene todos y cada uno de los vértices de G. Esto nos impulsa a definir el siguiente problema:

CIRCUITO HAMILTONIANO

INSTANCIA: Un grafo $G = (V, E)$

PREGUNTA: ¿Contiene G un circuito Hamiltoniano?

Vamos a ver que el problema **CIRCUITO HAMILTONIANO (HC)** puede ser transformado al **PROBLEMA DEL VIAJERO (TS)**. Para ello necesitamos una función f que relacione cada instancia de HC a la correspondiente instancia de TS y probar que está función cumple las dos propiedades correspondientes a transformaciones polinómicas.

La función f se forma de la siguiente manera. Sea $G = (V, E)$ con $|V| = m$ una instancia de HC. La correspondiente instancia de TS tiene un conjunto de ciudades C que es idéntico a V. Para dos ciudades cualesquiera $v_i, v_j \in C$ la distancia entre ellas $d(v_i, v_j)$ se define como 1 si $\{v_i, v_j\} \in E$ y 2 en otro caso. La cota B del tamaño máximo del recorrido la establecemos en m.

Para cada uno de las $m(m-1)/2$ distancias que hay que especificar, sólo es necesario examinar G para ver si la arista está en E. Por lo tanto, f es computable por un algoritmo en tiempo polinómico. Para la segunda condición, es suficiente demostrar que G contiene un circuito Hamiltoniano si y solo si existe un recorrido de todas las ciudades en $f(G)$ tal que su longitud no sea mayor a B.

Primero supongamos que $< v_1, \ldots, v_k >$ es un circuito Hamiltoniano de G. Entonces $< v_1, \ldots, v_k >$ es también un recorrido en $f(G)$ con longitud total igual a $m = B$ (por como hemos definido las distancias). Recíprocamente, supongamos que $< v_1, \ldots, v_k >$ es un recorrido en $f(G)$ con longitud total no superior a B. Por la propia definición de $f(G)$ se observa que $\{v_i, v_{i+1}\}$ para $1 \leq i < k$ y $\{v_k, v_1\}$ son todo aristas de G, y por lo tanto, $< v_1, \ldots, v_k >$ es un camino Hamiltoniano en G. Esto probaría que $HC \propto TS$.

También es de vital importancia observar que las transformaciones polinómicas cumplen la relación transitiva:

Lema 2.18 *Si $L_1 \propto L_2$ y $L_2 \propto L_3$, entonces $L_1 \propto L_3$ (propiedad transitiva).*

Demostración.

Sean Σ_1, Σ_2 y Σ_3 los alfabetos de L_1, L_2 y L_3 respectivamente, sea $f_1 : \Sigma_1^* \longrightarrow \Sigma_2^*$ una transformación polinómica de L_1 a L_2 y sea $f_2 : \Sigma_2^* \longrightarrow \Sigma_3^*$ una transformación polinómica de L_2 a L_3. Bastaría tomar la función $f : \Sigma_1^* \longrightarrow \Sigma_3^*$ definida por $f(x) = f_2(f_1(x))$ para terminar la prueba. Claramente, $f(x) \in L_3$ si y solo si $x \in L_1$ y el hecho de que exista un programa DTM de tiempo polinómico que genera f viene dado mediante una demostración análoga al lema anterior.

■

Podemos decir que dos lenguajes L_1 y L_2 (o dos problemas de decisión Π_1 y Π_2) son **polinómicamente equivalentes** cuando $L_1 \propto L_2$ y $L_2 \propto L_1$ (o cuando $\Pi_1 \propto \Pi_2$ y $\Pi_2 \propto \Pi_1$).

Ahora estamos a disposición de poder definir la clase de lenguajes más importante de todo este trabajo, la clase de problemas $NP - Completos$.

Definición 2.19 *Un lenguaje L se llama **NP-Completo** si $L \in NP$ y para cualquier otro $L' \in NP$ se cumple que $L' \propto L$. De forma similar, un problema de decisión Π se llama NP-Completo si $\Pi \in NP$ y para cualquier otro $\Pi' \in NP$ se cumple que $\Pi' \propto \Pi$.*

El **lema 2.16** nos dice que los problemas $NP - Completos$ son los problemas más difíciles en NP. Si algún problema $NP - Completo$ pudiese ser resuelto en tiempo polinómico, entonces todos los problemas en NP también serían resueltos. También, si algún problema en NP fuera intratable, entonces todos los problemas $NP-Completos$ también lo serían. Un problema $NP-Completo$ Π cumple pues la siguiente propiedad: Si $P \neq NP$ entonces $\Pi \in NP - P$. O lo que es lo mismo, $\Pi \in P$ si y solo si $P = NP$.

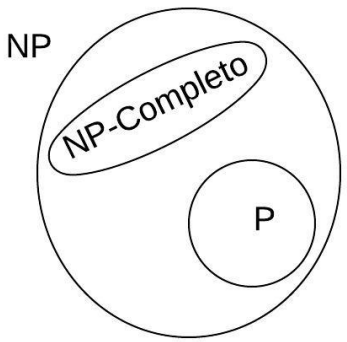


Figura 4: *Una esquematización más completa que se tiene del mundo P/NP.*

Debido a que no es fácil probar para que cualquier problema en NP es posible crear

una transformación polinómica a un problema $NP-Completo$, necesitaremos usar alguna cosa más para exprimir al máximo toda la información que tenemos.

El siguiente lema, que es una consecuencia inmediata de la transitiva, simplifica enormemente toda esta teoría si consiguiéramos demostrar solamente un problema $NP - Completo$.

Lema 2.20 *Si L_1 y L_2 pertenecen a NP tales que L_1 es $NP - Completo$ y $L_1 \propto L_2$ entonces L_2 es también $NP - Completo$.*

Demostración.

Como $L_2 \in NP$, lo único que hay que probar es que $\forall L' \in NP$, $L' \propto L_2$. Sea $L' \in NP$. Como L_1 es $NP - Completo$, entonces se cumple que $L' \propto L_1$. Pero por la propiedad transitiva, $L' \propto L_2$, que es lo que se pedía demostrar.

■

Así, para probar que un problema de decisión Π es $NP - Completo$, simplemente tendríamos que probar que $\Pi \in NP$ y que existe algún problema conocido $NP-Completo$ Π' el cual podemos transformar a Π.

2.6. NP-Completitud del problema del subgrafo isomorfo

Esta sección tiene como objetivo principal demostrar que el problema del subgrafo isomorfo es un problema $NP - Completo$. Para ello nos apoyaremos principalmente en el primer problema $NP - Completo$ que se consiguió demostrar (de hecho es el único que se prueba por definición).

El honor de ser el primer problema $NP - Completo$ es un problema de decisión de lógico Booleana, al que nos referiremos por el problema de **SATISFACIBILIDAD**. En el apéndice se puede consultar una definición formalizada del mismo.

SATISFACIBILIDAD

INSTANCIA: Un conjunto U de variables y una colección C de cláusulas sobre U.

PREGUNTA: ¿Existe algún asignamiento verdadero y satisfactorio para C?

Teorema 2.21 *(Teorema de Cook) SATISFACIBILIDAD es un problema $NP-Completo$.*

La demostración se deja como referencia con ánimo de no extender en exceso este trabajo.

Un problema inmediato que surge al enunciar el anterior es el siguiente. Que es en el

que nos apoyaremos directamente para demostrar el objeto de esta sección.

3-SATISFACIBILIDAD (3SAT)

INSTANCIA: Una colección $C = \{c_1, c_2, \ldots, c_m\}$ de cláusulas de un conjunto finito U de variables tales que $|c_i| = 3$ para $1 \leq i \leq m$.

PREGUNTA: ¿Existe algún asignamiento verdadero para U que satisfaga todas las cláusulas en C?

Teorema 2.22 *3-SATISFACIBILIDAD es un problema $NP - Completo$.*

Demostración.

La prueba, en la que no vamos entrar en detalles, consiste en transformar el problema de SATISFACIBILIDAD a este problema. Por lo tanto por el **lema 2.19** acabaríamos.

■

La demostración de que el problema del subgrafo isomorfo es $NP - Completo$ se basa principalmente en la demostración de la $NP - Completitud$ de los siguientes dos problemas. Que ya nos otorgarán unas pistas de cómo podemos reducirlos hasta nuestro problema objetivo.

CUBIERTA DE VÉRTICES (VC)

INSTANCIA: Un grafo $G = (V, E)$ y un entero positivo $K \leq |V|$.

PREGUNTA: ¿Existe alguna cubierta de vértices de tamaño como mucho K de G, esto es, un subconjunto $V' \subseteq V$ tal que $|V'| \leq K$ y, para cada arista $\{u, v\} \in E$, al menos u o v pertenecen a V' ?

CLIQUE (C)

INSTANCIA: Un grafo $G = (V, E)$ y un entero positivo $J \leq |V|$.

PREGUNTA: ¿Contiene G algún clique de tamaño J o mayor, esto es, un subconjunto $V' \subseteq V$ tal que $|V'| \geq J$ y cada par de vértices en V' están unidos por una arista en E?

A pesar de que VC y C sean resultados independientemente útiles para probar resultados de la NP-Completitud, los dos problemas son formas distintas de ver el mismo problema. Para ver esto, se conviene verlos en disyunción con un tercer problema, llamado CONJUNTO INDEPENDIENTE.

Un **conjunto independiente** en un grafo $G = (V, E)$ es un subconjunto $V' \subseteq V$ tal que, para todo $u, v \in V'$; la arista $\{u, v\} \notin E$.

CONJUNTO INDEPENDIENTE

INSTANCIA: Un grafo $G = (V, E)$ y un entero positivo $H \leq |V|$.

PREGUNTA: ¿Contiene G algún conjunto independiente V' tal que $|V'| \geq H$?

El siguiente lema caracteriza la relación que existe entre conjuntos independientes, cubiertas de vértices y cliques.

Lema 2.23 *Para cualquier grafo $G = (V, E)$ y un subconjunto $V' \subseteq V$, se cumplen las siguientes equivalencias:*

(a) V' es una cubierta de vértices para G.

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

(c) $V - V'$ es un clique en el grafo complementario G^c de G, donde $G^c = (V, E^c)$ con $E^c = \{\{u, v\} : u, v \in V \ \ y \ \{u, v\} \notin E\}$.

Demostración.

Probaremos que $(a) \iff (b)$. Hacia la derecha, denotamos por $W = V - V'$. Claramente $W \subseteq V$. Sean $u, v \in W$, como $u, v \notin V'$ tenemos que $\{u, v\} \notin E$. Así, probamos la primera implicación. Recíprocamente, sea $\{u, v\} \in E$. Al ser W un conjunto independiente, $\{u, v\} \notin W$ a la vez. Por lo tanto, o $u \in V'$ o $v \in V'$, lo que prueba el recíproco.

De una forma muy parecida se pueden probar las otras dos equivalencias restantes.

■

Debido a este lema, tenemos que los tres problemas son problemas que se preguntan lo mismo pero desde un punto de vista totalmente diferente. Además, las relaciones en el lema convierte en trivial el encontrar una transformación de un problema a otro.

Esto implica que la NP-Completitud de estos tres problemas se basan en la NP-Completitud de solamente uno de ellos. Por tanto, nos bastará con probarlo para VC.

Teorema 2.24 *VC es un problema $NP - Completo$.*

La demostración puede ser consultada en el apéndice o en [1].

Así, conseguimos probar que VC es un problema $NP - Completo$, y por lo tanto por el **lema 2.22** y el **lema 2.19** tenemos que el problema CLIQUE es un problema $NP - Completo$.

SATISFACIBILIDAD $\longrightarrow$ 3SAT $\longrightarrow$ VC $\longrightarrow$ CLIQUE

Figura 5: *Diagrama de la secuencia de transformaciones para probar que CLIQUE es $NP-Completo$.*

Ahora solo nos faltaría demostrar que el problema del subgrafo isomorfo también lo es. Lo haremos mediante lo que se llama prueba por restricción. Esto es, probar que si un problema Π contiene a un problema **NP-Completo** conocido como un caso especial de Π; entonces Π es un problema $NP-Completo$.

Teorema 2.25 *El problema del subgrafo isomorfo es un problema $NP-Completo$.*

Demostración.

Bastaría restringir a CLIQUE únicamente instancias tales que el grafo para el cual tenemos que encontrar el isomorfismo es un grafo completo, es decir, que su conjunto de aristas contenga todas las posibles uniendo dos vértices de su conjunto de vértices. Por lo tanto, el problema del subgrafo isomorfo contiene un problema que es $NP-Completo$ (ya que el grafo que estamos considerando no tiene porqué ser completo) y, necesariamente, el problema del subgrafo isomorfo es un problema $NP-Completo$.

■

3. Teoría de grafos

Los grafos, los objetos principales con los que trataremos en toda esta memoria, son muy útiles a la hora de analizar una cantidad enorme de problemas muy diferentes como, por ejemplo, problemas de construcción de redes, problemas de asignación de tareas...

Informalmente, un grafo es una colección de **vértices** o **nodos** (se representan como puntos del plano) y de **aristas** (se representan como líneas del planos) que unen los vértices.

Antes de poder definir el concepto de grafo en nuestro contexto, necesitaremos comprender bien los elementos que conforman el mismo.

3.1. Introducción a Grafos

Empecemos definiendo el concepto de grafo que usaremos en toda la memoria.

Definición 3.1 *Un **grafo** es una tupla $g = (V, E, \mu)$, donde V es un conjunto finito de nodos, $E \subseteq V \times V - id_V$ es un conjunto de aristas (donde $id_V = \{(v, v) : v \in V\}$ es la relación binaria reflexiva más pequeña), y $\mu : V \to \Sigma$ es la función que etiqueta a los nodos.*

Cabe destacar que la definición no permite a un nodo ser ambos componentes de una arista, i.e., los lazos (arista que tiene como origen y final el mismo nodo) no están permitidos.

Otro de los conceptos que nos resultará esencial para entender la idea principal de la memoria es el concepto de subgrafo:

Definición 3.2 *Dado un grafo $g = (V, E, \mu)$, se llama **subgrafo** de g a la tupla $g' = (V', E', \mu')$ donde $V' \subseteq V$, $E' = E \cap (V' \times V')$ y μ' es la función μ restringida a los nodos de V'.*

Más adelante, definiremos distintos tipos de subgrafos que nos resultarán de más interés.

En lo que sigue, mientras no se diga otra cosa; vamos a considerar grafos dirigidos, es decir, grafos donde todas las aristas son dirigidas desde un vértice a otro.

Definición 3.3 *Dos grafos $g_1 = (V_1, E_1, \mu_1)$ y $g_2 = (V_2, E_2, \mu_2)$ son **isomorfos**, que denotaremos por $g_1 \cong g_2$, si existe una biyección $f : V_1 \to V_2$ que cumple que, para todo nodo $u_1, v_1 \in V$ se cumple que:*

$$(u_1, v_1) \in E_1 \Leftrightarrow (f(u_1), (v_1)) \in E_2$$

Esta definición nos ayudará a comprender el enunciado del problema origen de toda esta memoria: El problema del subgrafo isomorfo.

Conjetura 3.4 *Dados dos grafos g y h cualesquiera, ¿extiste un subgrafo g' de g tal que $g' \cong h$?*

En la siguiente sección entenderemos este problema en el marco de la teoría de la complejidad y veremos que en algunos casos (depende del tipo de grafo) este problema ya ha sido resuelto.

Definición 3.5 *Algunos otros conceptos asociados con un grafo son los siguientes:*

1. *Dado un grafo $g = (V, E, \mu)$, una secuencia de nodos $w_1, w_2, ..., w_k$ tales que $(w_i, w_{i+1}) \in E$ para $1 \leq i \leq k$ es un **camino** entre w_1 y w_2.*

2. *Definimos la **longitud** de un camino como el número de nodos en el mismo. Como puede haber varios, denotaremos por $|u \longmapsto v|$ el más corto de todos ellos.*

3. *Definimos el* ***diámetro*** *de un grafo como la longitud del camino más corto entre los dos nodos conectados más alejados del grafo, i.e.,*

$$diametro(g) = \max_{u,v \in V} \{|u \longmapsto v| : |u \longmapsto v| < \infty\}$$

4. *Un* ***ciclo*** *es un camino tal que $w_1 = w_k$ y cuyos nodos sean todos distintos.*

5. *Un grafo* ***acíclico*** *es un tipo de grafo el cual no tiene ciclos.*

Si no existe ningún camino entre dos nodos, es coherente escribir que $|u \longmapsto v| = \infty$ y claramente $|u \longmapsto u| = 1$. Con respecto a la definición dada de grafo, un grafo acíclico es aquel que la clausura reflexiva-transitiva de E es un orden parcial.

Ejemplo 3.6 *En este ejemplo, los dos grafos que vemos son claramente isomorfos: Basta tomar $f : \{1,2,3,4\} \longrightarrow \{1,2,3,4\}$ dada por $f(1) = 1$,$f(2) = 4$,$f(4) = 2$ y $f(3) = 3$ para comprobar el isomorfismo. Los dos son ejemplos de grafos acíclicos con diámetro igual a* 3.

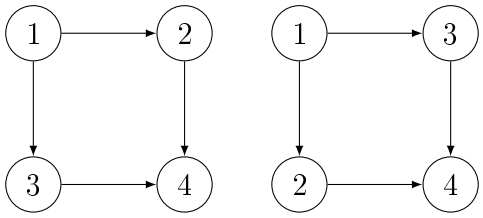


Figura 6: *Ejemplo de grafos isomorfos.*

Para poder entender el funcionamiento del algoritmo, necesitaremos definir un tipo de subgrafo que será necesario para poder definir la clase de grafos que nos interesa en todo el trabajo.

Definición 3.7 *Dado un grafo $g = (V, E, \mu)$, llamaremos* ***subgrafo de*** *g* ***anclado*** *en el nodo v con radio k a $R_g(v,k) = (W, E', \mu')$ donde:*

$W = \{u \in V : |u \longmapsto v| \leq k\}$

y donde $E' = E \cap (W \times W)$, esto es, el conjunto de aristas restringidas a los nodos en W. De la misma manera, μ' es la restricción de μ a los nodos en W. Además, denotaremos por $R_g(v) = (W, E', \mu')$ al subgrafo de g anclado en el nodo v, donde:

$W = \{u \in V : |u \longmapsto v| < \infty\}$

y E' y μ' están definidas como antes.

3.2. Grafo extendido y lenguaje de grafos

En esta sección nos centraremos el tipo de objetos con los que trabajaremos a lo largo de la memoria y formalizaremos el etiquetado de los propios grafos. Para ello, primero veremos la noción de grados de entrada y grados de salida de un nodo.

Definición 3.8 *Dado un nodo v, una arista cualquiera (u, v) se denomina* ***entrante*** *(respectivamente* ***saliente*** *para las aristas de la forma (v, u)). El* ***grado de entrada*** *de un nodo v (respectivamente* ***grado de salida****) es el número de aristas entrantes (respectivamente salientes) de v; que denotaremos por $idg(v)$ y definiremos como $idg(v) = |\{(u, v) \in E\}|$ (respectivamente el grado de salida se define como $odg(v) = |\{(v, u) \in E\}|$).*

Extrapolando la definición anterior a un grafo en su totalidad, dado un grafo $g = (V, E, \mu)$, $V_m^n(g)$ es el conjunto de nodos con grado de entrada n y grado de salida m. En lo que sigue, prestaremos especial interés en dos conjuntos de nodos: el conjunto de nodos con ningún grado de entrada y el conjunto con ningún grado de salida; que denotaremos por $V^0(g)$ y $V_0(g)$, respectivamente.

Definición 3.9 *Llamamos* ***alfabeto escrito*** *Σ_r a la asociación de un alfabeto Σ con una relación finita $r \subseteq (\Sigma \times \mathbb{N} \times \mathbb{N})$.*

Esta relación permite etiquetar los nodos de cualquier grafo mediante el grado de entrada y el grado de salida. Con el objetivo de referirnos explícitamente al subconjunto de símbolos del alfabeto con un grado de entrada y salida fijos, denotaremos por Σ_m^n al conjunto: $\{s \in \Sigma : (s, n, m) \in r\}$. Es necesario observar que, como r es una relación finita, Σ_m^n es no-vacío únicamente para un número finito de $n's$ y $m's$.

Definición 3.10 *Un* ***lenguaje de grafo*** *es cualquier subconjunto $L_G \subseteq \mathcal{G}(\Sigma_r)$, donde $\mathcal{G}(\Sigma_r)$ es el conjunto de grafos posibles sobre Σ_r. También, denotaremos por $\widehat{\Sigma}$ al* ***alfabeto extendido****, que se define por el conjunto:*

$$\widehat{\Sigma} = \{a_m^n : a \in \Sigma, (a, n, m) \in r\}$$

Esta última definición nos permite definir el **grafo extendido** como $\hat{g} = (V, E, \hat{\mu})$ donde $\hat{\mu} : V \longrightarrow \widehat{\Sigma}$, y para cada nodo v en $V_m^n(g)$, si $\mu(v) = a$, entonces $\hat{\mu}(v) = a_m^n$.

De aquí en adelante usaremos símbolos griegos cuando representemos las etiquetas de los nodos internos de un grafo (aquellos tales que $odg(v) \neq 0$) y símbolos latinos cuando representemos las etiquetas de los nodos frontera (aquellos tales que $odg(v) = 0$).

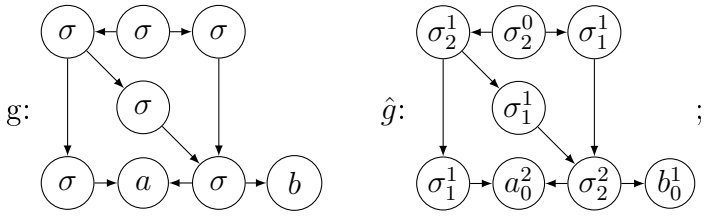


Figura 7: *Ejemplo de grafo g y su correspondiente grafo extendido $\hat{g}$.*

3.3. Multiconjuntos

El objetivo de esta sección es definir los multiconjuntos, la generalización natural de un conjunto; que nos servirá de gran ayuda a la hora de visualizar como funciona un autómata y en el algoritmo.

Definición 3.11 *Dado un conjunto D, un* ***multiconjunto*** *es el par $\langle D, f\rangle$ donde $f : D \longrightarrow \mathbb{N}$ es la función tal que para cada $a \in D$, $f(a)$ denota el número de elementos a en el multiset. El tamaño del multiset $A = \langle D, f\rangle$ se denota or $|A|$ y definido por $|A| = \Sigma_{a\in D} f(a)$.*

Como viene siendo natural, veamos como se comparan dos multisets y como efectuamos operaciones entre ellos.

Definición 3.12 *Dos multisets $A = \langle D, f\rangle$ y $B = \langle D, g\rangle$ sobre un mismo conjunto D, son iguales si y solo si, $\forall a \in D$, $f(a) = g(a)$. De la misma manera, A es un subconjunto de B si y solo si, $f(a) \leq g(a)$. También definiremos la suma de multisets $(A \oplus B)$ como el multiset $C = \langle D, h\rangle$ donde la función h cumple que $\forall a \in D$, $h(a) = f(a) + g(a)$*

Un concepto que nos ayudará a trabajar más cómodamente con los multisets es la llamada función de Parikh; que consiste en una función $\Psi : D^* \longrightarrow \mathbb{N}^n$ donde $D = \{d_1, ..., d_n\}$ es un conjunto y D^* es el conjunto de cadenas sobre D. La función está definida de la siguiente manera: dado un $x \in D^*$, $\Psi(x) = (\#_{d_1}, ..., \#_{d_n})$ donde $\#_{d_i}$ denota el número de apariciones de d_i en la cadena x.

Observar que esta función nos permite representar un multiset cualquiera aplicando la función anterior a la cadena que represente dicho multiset. Denotaremos por $\mathcal{M}_n(D)$ a la clase de los multisets que tienen tamaño n sobre D y $\mathcal{M}(D)$ a la clase de los multisets de cualquier tamaño sobre D.

Ejemplo 3.13 *Sean $A = \{a, a, b, b, b, c\}$ y $B = \{a, a, b, c\}$ dos multiconjuntos. Para A tenemos que $f(a) = 2$,$f(b) = 3$ y $f(c) = 1$ (análogamente para B). Además, claramente $B \subseteq A$ y $C = A \oplus B = \{a, a, a, a, b, b, b, b, c, c\}$. Por último, bastaría aplicar la función de*

Parikh a la cadena "aabbbc" para representar el multiset A.

4. K-testabilidad de grafos

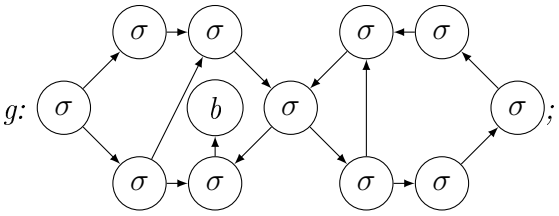
El objetivo de este trabajo, como hemos comentado en la introducción, es el de presentar un algoritmo que nos ayude a convertir una instancia particular de el problema del subgrafo isomorfo a la clase P. Para ello una herramienta fundamental será la introducción del concepto de vector k-testable, que nos ayudará a definir la clase de lenguajes de grafos que nos interesa.

Definición 4.1 *Dado un alfabeto escrito Σ_r y su correspondiente conjunto de grafos $\mathcal{G}_r$; para cualquier $g = (V, E, \mu) \in \mathcal{G}_r$ y cualquier $k \geq 2$, definimos el* ***vector k-testable*** $T_k(g) = (I_{k-1}(g), P_k(g), F_{k-1}(g))$ *donde:*

1. $I_{k-1}(g) = \{R_{\hat{g}}(v, k-1) : v \in V^0(g)\}$
2. $P_k(g) = \{R_{\hat{g}}(v, k) : v \in V,\ \ diametro(R_g(v)) \geq k\}$
3. $F_{k-1}(g) = \{R_{\hat{g}}(v, k-1) : v \in V,\ \ diametro(R_g(v)) \leq k-1\}$

Claramente, si $diametro(g) < k$ entonces $P_k(g) = \emptyset$. Observar que los grafos en cada componente del vector k-testable están etiquetados con la función del grafo extendido.

Ejemplo 4.2 *Esta definición no está restringida a grafos acíclicos. Dado el grafo (cíclico)*



su vector 2-testable se compone por:

$$I_1(g) = \emptyset; \qquad F_1(g) = \left\{ \left(b_0^1\right) \right\};$$

$$P_2(g) = \left\{ \begin{array}{c} \left(\sigma_1^1\right) \\ \downarrow \\ \left(\sigma_1^2\right) \end{array} ; \begin{array}{c} \left(\sigma_1^2\right) \\ \downarrow \\ \left(\sigma_2^2\right) \end{array} ; \begin{array}{c} \left(\sigma_2^2\right) \\ \swarrow \quad \searrow \\ \left(\sigma_2^1\right) \quad \left(\sigma_2^1\right) \end{array} ; \begin{array}{c} \left(\sigma_2^1\right) \\ \swarrow \quad \searrow \\ \left(b_0^1\right) \quad \left(\sigma_2^1\right) \end{array} ; \begin{array}{c} \left(\sigma_2^1\right) \\ \swarrow \quad \searrow \\ \left(\sigma_1^2\right) \quad \left(\sigma_1^1\right) \end{array} ; \begin{array}{c} \left(\sigma_1^1\right) \\ \downarrow \\ \left(\sigma_1^1\right) \end{array} \right\}$$

Las componentes del vector k-testable pueden ser extendidas a un conjunto de grafos G de la forma más natural:

- $I_k(G) = \bigcup_{g \in G} I_k(g)$

- $P_k(G) = \bigcup_{g \in G} P_k(g)$
- $F_k(G) = \bigcup_{g \in G} F_k(g)$

Para cualquier par de grafos g y g' es posible definir una relación de equivalencia (que denotaremos por $\equiv_k$) sobre $\mathcal{G}_r$ teniendo en cuenta el vector k-testable. Diremos que $g \equiv_k g'$ si y solo si $T_k(g) = T_k(g')$. Esta relación nos abre el paso a las siguientes definiciones, que son una parte fundamental de este trabajo.

Definición 4.3 *Un lenguaje de grafos $G \subseteq \mathcal{G}_r$ es **k-testable** ($k \geq 2$) si es el resultado de la unión de un número finito de clases de equivalencia de la relación $\equiv_k$.*

Definición 4.4 *Para cualquier $k \geq 2$, un lenguaje de grafos $G \subseteq \mathcal{G}_r$ es **k-testable en el sentido estricto** (k-TSS) si existen tres conjuntos de grafos finitos (B, S, E) tales que, $g \in G$ si y solo si $I_{k-1}(g) \subseteq B$, $P_k(g) \subseteq E$ y $R_{k-1}(g) \subseteq S$.*

Por un lado tenemos que, de acuerdo a la definición, un grafo con diámetro menor que k pertenezca a un lenguaje de grafos k-TSS depende únicamente de $F_{k-1}(g)$. Esto es porque para estos grafos $P_k(g) = \emptyset$ y $I_{k-1}(g) \subseteq F_{k-1}(g)$.

Por otro lado, dado un conjunto de grafos G, el vector k-testable define un lenguaje k-TSS cuando los conjuntos de grafos B, S, y E son $I_{k-1}(G)$, $P_k(G)$ y $I_{k-1}(G)$ respectivamente. A este lenguaje de grafos lo denotaremos por $L_k(G)$.

Veamos ahora algunos resultados respecto a $L_k(G)$.

Lema 4.5 *Dado G un conjunto de grafos finito y $k \geq 2$, se satisface que $G \subsetneq L_k(G)$.*

Demostración.

Sea $g \in G$. Claramente tenemos que $I_{k-1}(g) \subseteq I_{k-1}(G)$, $P_k(g) \subseteq P_k(G)$ y $F_{k-1}(g) \subseteq F_{k-1}(G)$. Por lo tanto, $g \in L_k(G)$.

Para probar la inclusión estricta, bastaría observar que cualquier grafo desconectado obtenido por la unión de grafos en G pertenecerá a $L_k(G)$ (por la extensión hecha para las componentes del vector k-testable). Con esto también probamos que cualquier lenguaje de grafo k-TSS, excepto el conjunto vacío, es infinito.

■

Lema 4.6 *Dado G un conjunto de grafos y $k \geq 2$, $L_k(G)$ es el lenguaje de grafos k-TSS más pequeño que contiene a G.*

Demostración.

Sea H un lenguaje de grafos k-TSS que contiene a G. Veamos que $T \not\subset L_k(G)$.

Primero observamos que, como $G \subset T$, todas y cada una de las estructuras que hay en el vector k-testable $T_k(G)$ también están dentro del vector que caracteriza a T, por lo tanto T y $L_k(G)$ no pueden ser incomparables.

Sean (B_T, S_T, E_T) los conjuntos que definen el lenguaje T. Supongamos que $T \subset L_k(G)$; entonces existe un grafo $g \in L_k(G) - T$. Por un lado, $g \in L_k(G)$, entonces $I_{k-1}(g) \subseteq I_{k-1}(G)$, $P_k(g) \subseteq P_k(G)$ y $F_{k-1}(g) \subseteq F_{k-1}(G)$. Por otro lado, $g \notin T$, entonces $I_{k-1}(g) \not\subseteq B_T$, $P_k(g) \not\subseteq S_T$ y $F_{k-1}(g) \not\subseteq E_T$. Esto quiere decir que, habrán algunas estructuras en el vector k-testable de $L_k(G)$ que no estarán presentes en el vector de T. Por lo tanto, existirá algún grafo g' tal que $g' \in G$ y $g' \notin T$; entonces $G \not\subseteq T$ lo que contradice la suposición.

■

Lema 4.7 *Sean G y G' dos conjuntos de grafos y $k \geq 2$. Si $G \subseteq G'$, entonces $L_k(G) \subseteq L_k(G')$.*

Demostración.

Como $G \subseteq G'$, claramente tenemos que $I_{k-1}(G) \subseteq I_{k-1}(G')$, $P_k(G) \subseteq P_k(G')$ y $F_{k-1}(G) \subseteq F_{k-1}(G')$. Por lo tanto $L_k(G) \subseteq L_k(G')$.

■

Lema 4.8 *Para cualquier conjunto de grafos G y $k \geq 2$ se cumple que $L_{k+1}(G) \subseteq L_k(G)$.*

Demostración.

Tenemos que probar que dado $g \in L_{k+1}(G)$, $I_{k-1}(g) \subseteq I_{k-1}(G)$, $P_k(g) \subseteq P_k(G)$ y $F_{k-1}(g) \subseteq F_{k-1}(G)$. Observar que los conjuntos $I_{k-1}(G)$ y $F_{k-1}(G)$ pueden ser obtenidos de los conjuntos $I_k(G)$ y $F_k(G)$:

$I_{k-1}(G) = I_{k-1}(I_k(G))$

$F_{k-1}(G) = F_{k-1}(F_k(G))$

Así, tenemos dos de las tres inclusiones. Con respecto a $P_k(G)$ y $P_{k+1}(G)$, para cualquier $g \in L_{k+1}(G)$, distinguimos dos casos:

- Si $diametro(g) \leq k$, entonces $P_{k+1}(G) = \emptyset$; que es un subconjunto de $P_k(G)$.
- Si $diametro(g) > k$, entonces $P_k(g) = P_k(P_{k+1}(g)) \subseteq P_k(P_{k+1}(G)) = P_k(G)$

Por lo tanto concluimos que $L_{k+1}(G) \subseteq L_k(G)$.

■

En la siguiente sección veremos un modelo de automátas de grafos para grafos acíclicos dirigidos y lo usaremos para proponer el algoritmo.

5. Modelo de automátas de grafos y algoritmo

5.1. Autómata de grafos

En lo que sigue, los multiconjuntos serán representados por la función de Parikh aplicada a la cadena correspondiente.

Definición 5.1 *Sea Σ_r el alfabeto escrito donde w denota el mayor grado de salida en $\widehat{\Sigma}$. Un autómata de grafos no-determinista para un lenguaje sobre Σ_r se define como la tupla $GA = (Q, \widehat{\Sigma}, \delta, F)$ donde Q es un conjunto finito de estados, $\widehat{\Sigma}$ es el alfabeto extendido de Σ_r, el conjunto $F \subseteq Q$ es un conjunto de estados finales y δ es la función de transición definida de esta manera:*

$$\delta = \bigcup_{\substack{0 \leq j \leq w \\ j:\exists n>0, \Sigma_j^n \neq \emptyset}} \delta_j$$

donde cada una de los δ_j se definen como:

$$\delta_j : \widehat{\Sigma}_j^n \times \mathcal{M}_j(Q) \longrightarrow Q, \qquad 0 \leq j \leq w$$

Cabe destacar que el dominio de la función de transición considera el alfabeto extendido en vez del alfabeto base. Esto permite procesar el grafo teniendo en cuenta tanto el grado de salida (y también el tamaño del multiconjunto) y el grado de entrada del nodo.

Con la idea de extender la definición de la función de transición para operar en grafos, la idead intuitiva consiste en un análisis recursivo sobre cada uno de los nodos con grado de entrada igual a cero. Formalmente, dado un grafo cualquiera g la función de transición δ se define:

$$\delta : \mathcal{G}(\Sigma) \cup \mathcal{G}(\widehat{\Sigma}) \longrightarrow \mathcal{M}(Q)$$

tal que $\delta(g) = \delta(\hat{g}) = \bigoplus_{v_i \in V^0(\hat{g})} \delta(R_{\hat{g}}(v_i))$

donde, suponiendo que el grado de salida del nodo v_i es m:

$$\delta(R_{\hat{g}}(v_i)) = \delta_m(\mu_{\hat{g}}(v_i), M_{i1} \oplus \ldots \oplus M_{im}), \qquad M_{ij} = \delta(R_{\hat{g}}(w_j)), (v_i, w_j) \in E$$

Para cualquier grafo g, el grafo extendido $\hat{g}$ es isomorfo a él, por lo tanto no hay

problema en realizar el análisis de un grafo a su versión extendida.

Definición 5.2 *El **lenguaje aceptado** por el autómata es el conjunto de grafos:*

$$L(GA) = \{g \in \mathcal{G}(\Sigma_r) : \forall q \in \delta(\hat{g}), q \in F\}$$

Así, cualquier grafo g es aceptado por el autómata si y solo si la función de transición devuelve un multiconjunto que solo contiene estados finales.

El modelo de autómatas de grafos anterior no puede procesar grafos que contengan ciclos debido a no poder establecer un orden; ya que estos no tienen porqué incluir nodos con grado de entrada igual a cero. Además, tampoco está muy claro el hecho de crear un criterio de aceptación para este tipo de grafos por la misma razón anterior.

Ejemplo 5.3 *Considerando el grafo en la Figura 7 y el siguiente autómata:*

$$\delta(a_0^2, \lambda) = q_1$$
$$\delta(b_0^1, \lambda) = q_2$$
$$\delta(\sigma_2^2, q_1q_2) = q_1$$
$$\delta(\sigma_1^1, q_1) = q_2$$
$$\delta(\sigma_2^1, q_2q_2) = q_1$$
$$\delta(\sigma_2^0, q_1q_2) = q_1, \text{ donde } q_1 \in F$$

El análisis del grafo se realiza de la siguiente manera:

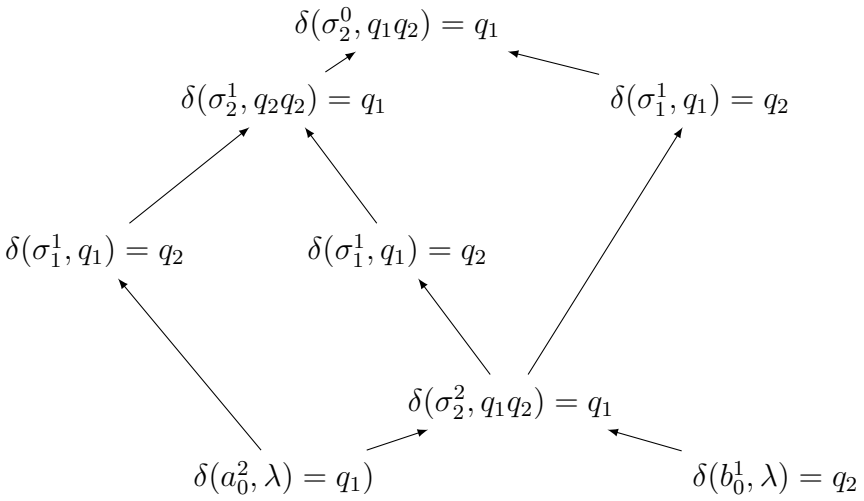


Figura 8: *Análisis del grafo de la Figura 7.*

El grafo es atravesado recursivamente hasta alcanzar los nodos con grado de salida igual a cero, por lo tanto, los primeros nodos que se relacionan con un estado son aquellos

que no tienen una llamada recursiva. La figura anterior muestra el orden del análisis cuando se alcanzan estos nodos. Como sólo hay un nodo con grado de entrada igual a cero y el análisis nos da un estado final, el grafo es aceptado por el autómata.

5.2. Definición del algoritmo

Primeramente, veamos un pseudocódigo del algoritmo para el posterior análisis de la complejidad. El algoritmo se basa en el aprendizaje inductivo a partir de una muestra que le proporcionemos, i.e., es un algoritmo de inferencia gramatical.

INPUT: Un conjunto G de grafos. Un valor $K \geq 2$.

OUTPUT: Un autómata de grafos que reconoce el lenguaje $L_k(G)$.

Método:

1: Calcular $(I_{k-1}(G), P_k(G), F_{k-1}(G))$

2: Sea Σ_r el alfabeto escrito de G y $\widehat{\Sigma}_r$ el alfabeto extendido.

3: **for** $g \in \{I_{k-1}(G) \cup I_{k-1}(P_k(G)) \cup F_{k-1}(G)\}$ **do**

4: Sea $Q[g]$ un nuevo estado relacionado con g

5: **end for**

6: $F = \{Q[g] : g \in I_{k-1}(G)\}$

7: **for all** $g \in F_{k-1}(G), v \in V_m^0(g)$, donde $(v, w_i) \in E_g$, $1 \leq i \leq m$ **do**

8: $\delta_m(\mu(v), Q[R_g(w_1]) \ldots Q[R_g(w_m]) = Q[g]$

9: **end for**

10: **for all** $g \in P_k(G), v \in V_m^0(g)$, donde $(v, w_i) \in E_g$, $1 \leq i \leq m$ **do**

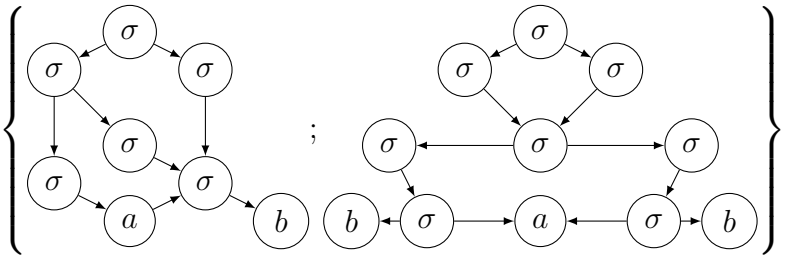
11: $\delta_m(\mu(v), Q[R_g(w_1, k-1]) \ldots Q[R_g(w_m, k-1]) = Q[R_g(v, k-1)]$

12: **end for**

13: **return** $(Q, \widehat{\Sigma}_r, F, \delta)$

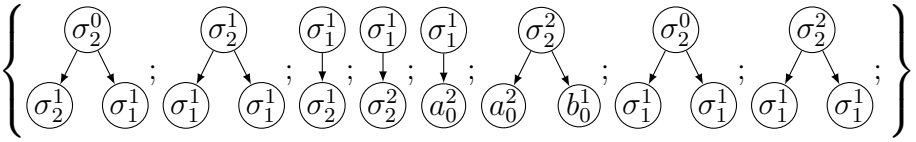
Primero, el algoritmo establece el conjunto de estados tomando en consideración las estructuras de los grafos de diámetro $k-1$ en el vector k-testable del conjunto de grafos inicial (la muestra). El conjunto de estados finales también es construido. Después, el algoritmo crea las transiciones usando los grafos en $Fk-1(G)$ y $P_k(G)$. Veamos un ejemplo de como funciona esto.

Ejemplo 5.4 *Tomemos $k = 2$ y el conjunto de grafos siguiente:*

$G =$ 

Los elementos del vector 2-testable son:

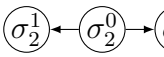
$$I_1(G) = \left\{\sigma_2^0\right\}; \quad F_1(G) = \left\{a_0^2; b_0^1\right\};$$

$P_2(G) =$ 

Primero, el algoritmo construye el conjunto de estados teniendo en cuenta $I_1(G)$, $F_1(G)$ *y* $I_1(P_2(G))$*:*

$$Q[a_0^2] = q_1; \quad Q[b_0^1] = q_2; \quad Q[\sigma_2^2] = q_3;$$
$$Q[\sigma_1^1] = q_4; \quad Q[\sigma_2^1] = q_5; \quad Q[\sigma_2^0] = q_6$$

Segundo, el algoritmo define el conjunto de estados finales, que en este caso es $F = \{q_6\}$*. Ahora, el algoritmo considera los grafos en* $F_1(G)$*. Como los grafos de* $F_1(G)$ *tienen diámetro uno, las transiciones* $\delta(a_0^2, \lambda) = q_1$ *y* $\delta(b_0^1, \lambda) = q_2$ *son añadidas al autómata. Tercero, el algoritmo procesa los grafos en* $P_2(G)$*. Veamos que pasa con uno de ellos:*



El algoritmo considera los subgrafos de diámetro uno por debajo del nodo σ_2^0 *y el grafo de diámetro uno anclado en el nodo* σ_2^0*. Por lo tanto, el algoritmo añade la transición* $\delta(\sigma_2^0, q_5q_4) = q_6$*. Después de procesar todos los grafos en* $P_2(G)$*, obtenemos el automáta:*

$$\delta(a_0^2, \lambda) = q_1$$
$$\delta(b_0^1, \lambda) = q_2$$
$$\delta(\sigma_2^0, q_5q_4) = q_6, \text{ donde } q_6 \in F$$
$$\delta(\sigma_2^1, q_4q_4) = q_5$$
$$\delta(\sigma_1^1, q_3) = q_4$$
$$\delta(\sigma_1^1, q_5) = q_4$$
$$\delta(\sigma_1^1, q_1) = q_4$$
$$\delta(\sigma_2^2, q_1q_2) = q_3$$
$$\delta(\sigma_2^0, q_4q_4) = q_6, \text{ donde } q_6 \in F$$
$$\delta(\sigma_2^2, q_4q_4) = q_3$$

Observamos que sería posible dar varios ejemplos de grafos que no estaban en el conjunto G pero que pertenecen al lenguaje de grafos 2-TSS. Veamos ahora que el algoritmo identifica la clase de lenguajes de grafos k-TSS a partir de una muestra.

Teorema 5.5 *El algoritmo anterior la clase de lenguajes de grafos k-TSS a partir de una muestra.*

Demostración.

Por un lado, para cualquier grafo g, pertenecer al lenguaje del autómata del algoritmo implica analizar cada uno de los p nodos con grado de entrada 0. Para aceptar al grafo g, todos estos análisis deben devolver un multiconjunto final (un multiconjunto que solo contiene estados finales).Por otro lado, para cualquier grafo g, pertenecer a $L_k(G)$ implica que, para cualquier nodo $v \in V^0(g)$, $R_{\hat{g}}(v, k-1) \in I_{k-1}(G)$. Por lo tanto, sin pérdida de generalidad, consideraremos grafos que solo tienen un nodo con grado de entrada igual a cero.

Veamos $L_k(G) \subseteq L(GA)$:

Probaremos por inducción en el diámetro de los grafos, que, si $g \in L_k(G)$, entonces $\delta(g)$ devuelve un estado final. Primero, si $diametro(g) < k$ y $v \in V^0(g)$, entonces g es isomorfo a $\hat{g} = R_{\hat{g}}(v) \in I_{k-1} \cap F_{k-1}$, y el algoritmo establece $\delta(R_{\hat{g}}(v)) = Q[R_{\hat{g}}(v, k-1)] = Q[R_{\hat{g}}(v)]$; que es un estado final. Supongamos que, para cualquier grafo g tal que $diametro(g) = n$, se satisface que $\delta(g) = Q[R_{\hat{g}}(v, k-1)]$ donde $v \in V^0(g)$. Ahora sea g un grafo con $v \in V_m^0(g)$, tal que, para $1 \leq i \leq m$, existe $(v, w_i) \in E_g$ y $diametro(R_g(w_i)) \leq n$, y donde al menos uno de los grafos $R_g(w_i)$ tiene diámetro n. Por lo tanto, $\delta(R_{\hat{g}}(w_i)) = Q[R_{\hat{g}}(w_i, k-1)]$ para cada i (por la hipótesis de inducción), y entonces:

$$\delta(g) = \delta(\hat{g}) = \delta_m(\mu_{\hat{g}}, \delta(R_{\hat{g}}(w_1)) \oplus \delta(R_{\hat{g}}(w_2)) \oplus \ldots \oplus \delta(R_{\hat{g}}(w_m))) =$$

$$= \delta_m(\mu_{\hat{g}}, Q[R_{\hat{g}}(w_1, k-1)]Q[R_{\hat{g}}(w_2, k-1)] \ldots Q[R_{\hat{g}}(w_m, k-1)])$$

El grafo que conseguimos de la unión de todos los $R_{\hat{g}}(w_i, k-1)$ es tal que $\delta(g) = Q[R_{\hat{g}}(v, k-1)]$, y como este estado es un estado final acabamos.

Veamos ahora $L(GA) \subseteq L_k(G)$:

Probaremos que, para cualquier grafo $g \in L(GA)$, se satisface que $F_{k-1}(g) \subseteq F_{k-1}(G)$, $P_k(g) \subseteq P_k(G)$ y que existe un grafo $q \in I_{k-1}(g)$ tal que $\delta(g) = Q[q]$ (un estado final). Lo volveremos a hacer por inducción en el diámetro del grafo.

Nuevamente, si $diametro(g) < k$ con $v \in V^0(g)$, entonces $\hat{g} \in F_{k-1}(g) \subseteq F_{k-1}(G)$, $P_k(g) = \emptyset$ y $R_{\hat{g}}(v, k-1) \in I_{k-1}(G)$. Supongamos, por hipótesis de inducción que, para cualquier grafo $g \in L(GA)$ tal que $diametro(g) = n \geq k$, se cumple que $F_{k-1}(g) \subseteq$

$F_{k-1}(G)$, $P_k(g) \subseteq P_k(G)$ y $\delta(g) = Q[R_{\hat{g}}(v, k-1)]$, donde $v \in V^0(g)$.

Sea ahora g un grafo tal que $v \in V_m^0(g)$, con $(v, w_i) \in E_g$ y donde $diametro(R_g(w_i)) \leq n$ para todo $1 \leq i \leq m$, con al menos uno de los grafos $R_g(w_i)$ de diámetro n. Así tenemos:

$\delta(g) = \delta(\hat{g}) = \delta_m(\mu_{\hat{g}}, \delta(R_{\hat{g}}(w_1)) \oplus \delta(R_{\hat{g}}(w_2)) \oplus \ldots \oplus \delta(R_{\hat{g}}(w_m))) =$

$= \delta_m(\mu_{\hat{g}}, Q[R_{\hat{g}}(w_1, k-1)]Q[R_{\hat{g}}(w_2, k-1)] \ldots Q[R_{\hat{g}}(w_m, k-1)]) = Q[R_{\hat{g}}(v, k-1)]$

donde $R_{\hat{g}}(v, k) \in P_k(g)$ debido a que $(v, w_i) \in E_g$ para todo $1 \leq i \leq m$. Además, $Q[R_{\hat{g}}(v, k-1)] = q$. Y para concluir:

$$F_{k-1}(g) = \bigcup_{1 \leq i \leq m} F_{k-1}(R_{\hat{g}}(w_i)) \subseteq F_{k-1}(G)$$

$$P_k(g) = \left(\{R_{\hat{g}}(v, k)\} \cup \bigcup_{1 \leq i \leq m} P_k(R_{\hat{g}}(w_i)) \} \right) \subseteq P_k(G)$$

Por lo tanto, y como $R_{\hat{g}}(v, k-1) \in I_{k-1}(G)$ tenemos que $g \in L_k(G)$.

■

5.3. Eficiencia del algoritmo

En esta sección se centra en demostrar que nuestro algoritmo es eficiente en tiempo polinómico y, por lo tanto, puede ser usado en la práctica sin ningún tipo de ataduras. Veamos unas primeras definiciones.

Definición 5.6 *Una clase de lenguajes es* ***identificable en el límite*** *con respecto a un algoritmo si, para cualquier lenguaje en la clase existe un conjunto de entrenamiento S tal que, cuando se introduce al algoritmo, produce un hipótesis correcta que no cambia cuando el algoritmo considera superconjuntos de S.*

Cada lenguaje de grafo k-TSS está relacionado con un vector k-testable, y, por lo tanto, para cada lenguaje en la clase, siempre existe un conjunto de grafos que pueden ser procesados para obtener el vector k-testable. Por esta razón tenemos que la clase de los lenguajes de grafos k-TSS son identificables en el límite.

Desde esta última definición, se han propuesto muchas condiciones de eficiencia en cuanto al tiempo se refiere. Normalmente el criterio de eficiencia considera un punto de vista más amplio, es decir, se suele asumir que los algoritmos de inferencia iteran en el conjunto de entrada, y modifican una hipótesis para adaptarse a una nueva muestra.

Nuestro algoritmo procesa el conjunto de grafos de entrada y da como salida el autómata que acepta el lenguaje de grafos k-TSS más pequeño que contiene al conjunto de

grafos entrante; dado un k en particular. Sin embargo, hay que pensar en un algoritmo que olvide el conjunto de grafos de entrada pero mantenga la hipótesis actual. En lo que sigue, consideraremos esta versión del algoritmo para estudiar el comportamiento del algoritmo desde este punto de vista.

El criterio que se suele considerar es lo que se llama **errores de cambio mentales**. Dada x una presentación que se admite y M el método de identificación, definimos $DH(M, x)$ como el número de hipótesis distintas que da como salida M, y sea $MC(M, x)$ el número de cambios "mentales", esto es, el número de veces que da como salida un cambio la hipótesis.

Definición 5.7 *Una clase de lenguajes $\mathcal{L}$ es **identificable en el límite en tiempo polinómico** si y solo si existe un algoritmo M tal que, dado $l \in \mathcal{L}$, identifica $l' \in \mathcal{L}$ equivalente a l en el límite , con la propiedad de que existen polinomios p y q tales que para cualquier n, y para cualquier l de tamaño n, el número de veces que M realiza un conjetura errónea es, a lo sumo, $p(n)$, y el tiempo para actualizar una conjetura, a lo sumo, $q(n, N)$ donde N es la suma de las longitudes de todos los datos dados.*

En nuestro caso, el algoritmo modifica el autómata solo cuando la muestra que se está procesando no es aceptada. Por lo tanto, podemos identificar fácilmente una conjetura incorrecta y un cambio mental. También, nuestro algoritmo solo cambiará como mucho una regla de transición del autómata para cada muestra. Entonces, podemos asegurar que el número de cambios mentales de nuestro algoritmo estará acotado por el tamaño del autómata que representa el lenguaje objetivo. Así, para probar que nuestro algoritmo es eficiente en el sentido anterior, sólo necesitamos probar que el tiempo para actualizar una conjetura es polinómico.

Lema 5.8 *El tiempo que tarda nuestro algoritmo en actualizar una conjetura es polinómico.*

Demostración.

Sea G el conjunto de grafos de entrada. Sea también w el grado de salida más grande de todos los grafos en G. Finalmente, sea Σ_r el alfabeto escrito de los grafos en G.

Dado un valor de k, la complejidad temporal para obtener cada transición está acotada por $O(w \cdot |\widehat{\Sigma}_r| \cdot w^{k-1})$. Está cota representa el grado de salida más grande por el tamaño del alfabeto por el tamaño del subgrafo más grand que puede ser reducido a un estado.

Cada paso en la inferencia toma como mucho la creación de tantas transiciones como el número de nodos que hay en el grafo que se está evaluando. Sea n esa cantidad de pasos. Por lo tanto, el tiempo para actualizar una conjetura está acotado por $O(n \cdot |\widehat{\Sigma}_r| \cdot w^k)$.

■

Teorema 5.9 *Nuestro algoritmo identifica en el límite, en tiempo polinómico, la clase de lenguaje de grafos k-TSS.*

6. Resultados del algoritmo

El objetivo de esta sección es mostrar algunos resultados experimentales para ilustrar como de bien funciona la clase de los lenguajes de grafos k-TSS para modelizar datos. Para llevarlo a cabo, se ha probado el algoritmo en algunos conjuntos de datos que veremos en esta sección.

Primero describiremos los conjuntos de datos y los parámetros usados para evaluar el comportamiento del algoritmo. Después, realizaremos un experimento para estudiar las habilidades de discriminación entre clases del autómata k-TSS.

6.1. Bases de datos

En este trabajo hemos tratado conjuntos de datos de IAM-Graph DB. IAM-Graph DB es un repositorio de acceso público que esta disponible de forma gratuita para fines únicamente académicos. De los conjuntos de datos disponibles en este repositorio hemos elegido tres: AIDS, GREC y Mutagenicity; que describimos a continuación.

AIDS AIDS consiste en grafos que representan componentes moleculares. Este conjunto de datos consiste en dos clases (activa o inactiva), que representan moléculas que tienen algún tipo de actividad contra el VIH o no. Las moléculas se pasan a grafos representando los átomos como nodos y los enlaces covalentes como aristas. Los nodos son etiquetados con el número del símbolo químico correspondiente. La elección del nodo con grado de entrada igual a cero se ha hecho de forma aleatoria. Este conjunto de datos contiene 2000 elementos (1600 inactivos y 400 activos).

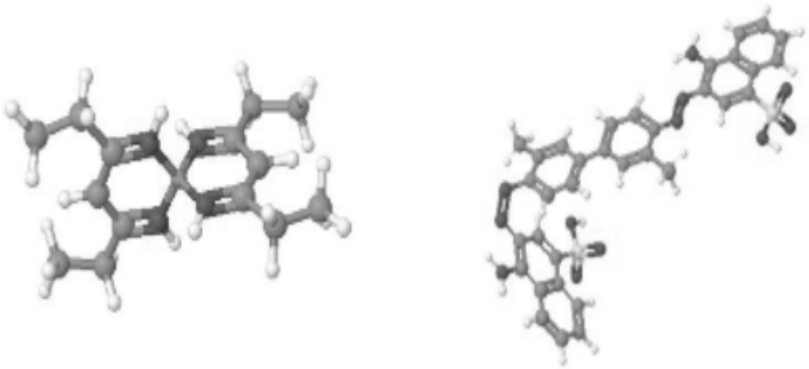


Figura 9: *Ejemplo de componente molecular activo e inactivo.*

GREC GREC consiste en grafos que representan símbolos de dibujos electrónicos y arquitectónicos. Las imágenes ocurren en diferentes niveles de distorsión. Dependiendo del nivel de distorsión, se aplica erosión, dilatación o otras operaciones morfológicas. El resultado se reduce has obtener lineas de un píxel de grosor. Finalmente, se extraen los grafos del resultado de insonorizar las imágenes trazado las líneas de un lado a otro y detectando tanto intersecciones como esquinas. Los nodos han sido etiquetados para representar tres posibles clases: intersecciones, esquinas y puntos finales. La elección del nodo con grado de entrada igual a cero se ha hecho de forma aleatoria. Este conjunto de datos contiene 1110 grafos distribuidos uniformemente en 22 clases.

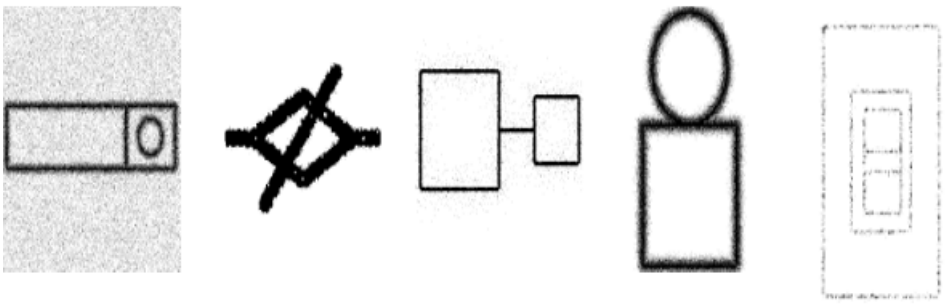


Figura 10: *Ejemplos de unos dibujos de la base de datos para cada uno de los niveles de distorsión.*

Mutagenicity Mutagenicity es un conjunto de datos de estructuras moleculares con diversos compuestos químicos clasificados como mutagénico y no mutagénico. Las moléculas se convierten a grafos representado los átomos como nodos y los enlaces covalentes

como aristas. Los nodos son etiquetados por su correspondiente símbolo químico. La elección del nodo con grado de entrada igual a cero se ha hecho de forma aleatoria. Este conjunto de datos contiene 4337 moléculas divididas en dos clases: 2401 son elementos mutagénicos y 1936 son elementos no-mutagénicos.

6.2. Resultados experimentales

Acabaremos la sección haciendo un resumen de los resultados obtenidos en el experimento. Estos experimentos demuestran el buen comportamiento de nuestro algoritmo. Para cada conjunto de datos, hemos usado el algoritmo para obtener un autómata de grafos que modelice cada clase. Los modelos son usados para clasificar un conjunto de muestras que no se solapan usando la estrategia "one-vs.-rest". Esto es, cada autómata tiene que clasificar correctamente las muestras de la clase que el autómata modeliza (las llamaremos clases positivas) y también tiene que rechazar las muestras de otras clases (a estas las llamaremos clases negativas).

Para cuantificar el comportamiento del algoritmo se ha usado la medida-F (F-m). Esta es una métrica usada ampliamente para problemas de clasificación. F-m se puede definir en términos de Precisión y Exhaustividad como sigue:

$$\text{Precisión} = \frac{\text{TP}}{\text{TP}+\text{FP}}; \qquad \text{Exhaustividad} = \frac{\text{TP}}{\text{TP}+\text{FN}}$$

$$\text{medida-F} = 2 \cdot \frac{\text{Precisión} \cdot \text{Exhaustividad}}{\text{Precisión}+\text{Exhaustividad}}$$

donde TP (positivos verdaderos) denota el número de muestras clasificadas correctamente de la clase positiva, FN (falsas negativas) denota el número de muestras clasificadas incorrectamente de la clase positiva y FP (falsas positivas) denota el número de muestras negativas clasificadas como positivas.

Se ha utilizado un esquema de validación cruzada triple a los conjuntos de datos que hemos descrito. Cada conjunto de datos se ha divido en dos subconjuntos que no se solapan, considerando el 80 % de las muestras de cada clase como un conjunto de entrenamiento y el 20 % restante como un conjunto de testeo. Hemos generado un autómata de grafos k-TSS por cada clase tomando valores de k que van desde 2 hasta 4.

La tabla siguiente muestra los resultados en términos de Precisión, Exhaustividad y F-m en porcentaje; obtenidos para cada valor de k previamente dichos junto con el número de reglas que aprende el autómata.

Se puede observar que en todos los casos el autómata se comporta de la misma forma:

		AIDS	GREC	Mutagenicity
$k = 2$	Reglas	902	34'77	840'58
	Prec.	70'24	87'64	81'4
	Exh.	60'94	91'94	85'52
	F-m	88'87	89'45	83'41
$k = 3$	Reglas	2487'5	41'59	3016'24
	Prec.	93'04	89'47	84'29
	Exh.	23'91	88'59	85'29
	F-m	37'88	90'02	84'79
$k = 4$	Reglas	3427'5	44'05	4873'35
	Prec.	99'97	89'48	86'79
	Exh.	13'62	88'59	84'83
	F-m	23'57	90'05	85'8

Tabla 1: *Tabla que muestra los resultados en porcentajes en términos de Precisión, Exhaustividad y medida-F. También se ha añadido la parámetro reglas que aprende el autómata para cada conjunto de datos y cada k.*

cuando se incrementa el valor de k, el autómata tiene que aprender un número de reglas que se dispara. Esto se debe a que los elementos del vector k-testable que representa el lenguaje de grafos tiene una variabilidad mucho más grande. Sin embargo, esto hace que el autómata sea mejor a la hora de discriminar entre clases.

Este experimento demuestra como el algoritmo es capaz de modelar, a un nivel de clases, los distintos tipos de grafos considerados y distinguirlos de el resto de muestras en el mismo conjunto de datos.

7. Comparativa de algoritmos

Nuestro modelo, a la hora de comparar grafos lo hace mediante un algoritmo llamado VF2 (lo explicaremos más adelante). La idea fundamental de está sección es comprobar como afecta el algoritmo utilizado para la comparación entre grafos que se realizan múltiples veces en el modelo a la hora de aplicarlo a las distintas bases de datos utilizadas. Lo haremos en términos de tiempo de ejecución del algoritmo al fijar una base de datos y un valor para k.

Primero definiremos los algoritmos y veremos como funcionan. Después veremos varias

gráficas donde se nos reflejará el tiempo que tarda el procesamiento del modelo para cada algoritmo en función de la base de datos y el k prefijado. El tiempo será el tiempo global de todo el proceso, sin incluir la carga de datos (que es un invariante al prefijar la base de datos).

7.1. Algoritmos

El primero de ellos es el **algoritmo de Ullmann**. Este algoritmo es una extensión de lo que se denomina un algoritmo de búsqueda en profundidad (DFS). La idea de la búsqueda en profundidad es partir de un nodo con grado de entrada igual a cero (si fuera un grafo en general se partiría de un nodo aleatoriamente), para después expandirse por uno de sus caminos. Cuando ya no quedan nodos en ese camino es cuando empieza el backtraking, i.e., volver hacia atrás desde el último nodo examinado y realizar la exploración en cada uno de los caminos que se va encontrando.

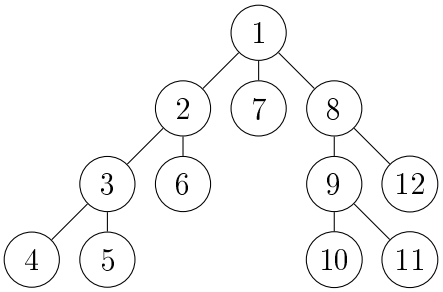


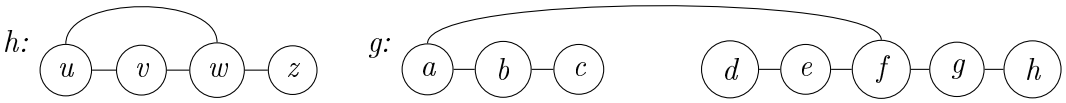
Figura 11: *Ejemplo de una búsqueda en profundidad. Orden en que se buscan los nodos.*

Dados los grafos g y h queremos saber si existe un subgrafo g' contenido en g que sea isomorfo a h. Podemos codificar cualquier isomorfismo de grafos como una $|V_g| \times |V_h|$ matriz M la cual contenga exactamente un 1 en cada fila y como mucho un 1 en cada columna. Así, establecemos que $m_{ij} = 1$ si $v_j \in g$ corresponde a $v_i \in h$ en el isomorfismo. Así, tenemos que $P = M(MH)^t$ donde P y H son las matrices de vecindades de los grafos respectivamente (más información en el anexo).

El funcionamiento del algoritmo se basa en enumerar sistemáticamente posibles matrices M y comprobar si codifican un isomorfismo. Se puede encontrar más información en [9].

Veamos un ejemplo de como funciona.

Ejemplo 7.1 *Vamos a ver, mediante el algoritmo, si el grafo g tiene un subgrafo isomorfo a h:*



La primera observación que hacemos es que un nodo de h solo puede ser asociado a un nodo de g si como mínimo tiene grado igual o mayor. Denotaremos por candidatos(u) al conjunto de vértices de g a los cuales tiene sentido asociar el nodo u. Así, en un inicio tenemos candidatos(u) = candidatos(v) = {a, b, e, f, g}, candidatos(w) = {f} y los candidatos para z son todos los nodos de g.

Ahora, el algoritmo comprueba que para cualquier nodo x en g que esté entre algún candidato de algún vértice y en h, entonces cada nodo vecino de y tiene al menos un candidato entre los nodos vecinos de x. Si la comprobación falla, entonces eliminamos x de los candidatos de y. Hacemos esto hasta que no podamos eliminar ninguno más.

Por ejemplo, h está entre los candidatos para z. Sin embargo, w es un nodo vecino de z, pero g no está entre los candidatos de w. Por lo tanto, nunca podremos asociar z a h. Haciendo está comprobación para los demás, obtenemos que candidatos(u) = candidatos(v) = candidatos(z) = {a, e, g} y candidatos(w) = {f}.

Ahora empezamos el backtracking: Primero intentamos asociar u con a. Es decir, hacemos candidatos(u) = {a} y eliminamos a de los otros conjuntos de candidatos. Pero ahora tenemos que e y g no son vecinos de a, por lo tanto tenemos que eliminarlos de los candidatos de v (que es vecino de u). Esto hace que la lista de candidatos para v sea vacía, por lo tanto, intentamos asociar a u con otro nodo.

Análogamente llegaremos a conjuntos vacíos si intentamos asociar los nodos restantes a u. Por lo tanto podemos asegurar que g no es isomorfo a h.

Los siguientes son los **algoritmos VF y VF2**. El algoritmo VF no es más que una versión antigua del VF2; que contaba con algunos pasos extra a la hora de proceder. Nos vamos a centrar a explicar el VF2, en particular, la primera versión del mismo.

Supongamos que queremos ver si existe un isomorfismo entre g y h. El algoritmo procesa un grafo de arriba a abajo (si estamos hablando de grafos dirigidos) o toma como origen un nodo aleatorio. A cada paso que da intenta asociar un nodo del primer grafo con el segundo grafo y para cuando ha pasado por todos los nodos de h.

Los pasos del algoritmo serían los siguientes:

1. Primero asocia el nodo vacío de g con el nodo vacío de h. (Esto siempre funciona).

2. Intenta asociar un nodo de g con un nodo de h.

...

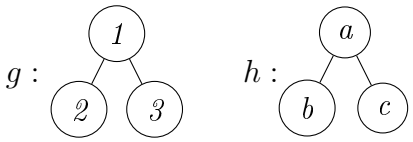
i. Intenta asociar un nodo de g con un nodo de h. Si no puede, empieza el backtracking y prueba otra asociación en el paso anterior. En el caso de que tampoco funcione la nueva asociación, va un paso hacia atrás y repite lo mismo.

...

n. Termina cuando encuentra el último nodo o cuando a pasado por todos ellos (en el caso de que sea cíclico, por ejemplo).

Veamos un ejemplo de esto:

Ejemplo 7.2 *Sean los grafos isomorfos:*



Veamos como funcionaría:

1. *Se asocia el nodo vacío con el nodo vacío.*

2. *Podemos asociar* 3 *con a, b o c. Asociamos* 3 *con a.*

3. *Podemos asociar* 1 *con b o c. Asociamos* 1 *con c. Esto funciona porque los dos asociaciones que hemos hecho forman un subgrafo isomorfo.*

4. *Al intentar asociar* 2 *con b observamos que es el resultado no es isomorfo, ya que debería haber una arista que uniera b y c y ninguna que uniera* 1 *y* 2.

   Volvemos al paso anterior.

5. *Asociamos* 1 *con b.*

6. *El resultado aquí será análogo al paso* 4. *Por lo tanto, volvemos al paso 2.*

7. *Asociamos* 3 *con c.*

8. *Asociamos* 1 *con a.*

9. *Asociamos* 2 *con b. Los subgrafos resultantes son isomorfos; y como coinciden con los grafos de partida, acabamos.*

El último algoritmo con el que trabajaremos será un algoritmo de fuerza bruta. El algoritmo se va a basar en comprobar todas las permutaciones que va a tener alguna de las dos matrices de vecindades asociadas a los grafos de partida. Si al comprobarlo, alguna permutación de alguna de ellas da como resultado la otra; tenemos un isomorfismo (condición necesaria y suficiente de isomorfía).

Ejemplo 7.3 *Trabajando con el ejemplo anterior, tenemos las siguientes matrices de vecindades para cada grafo.*

G :

	1	2	3
1	0	1	1
2	1	0	0
3	1	0	0

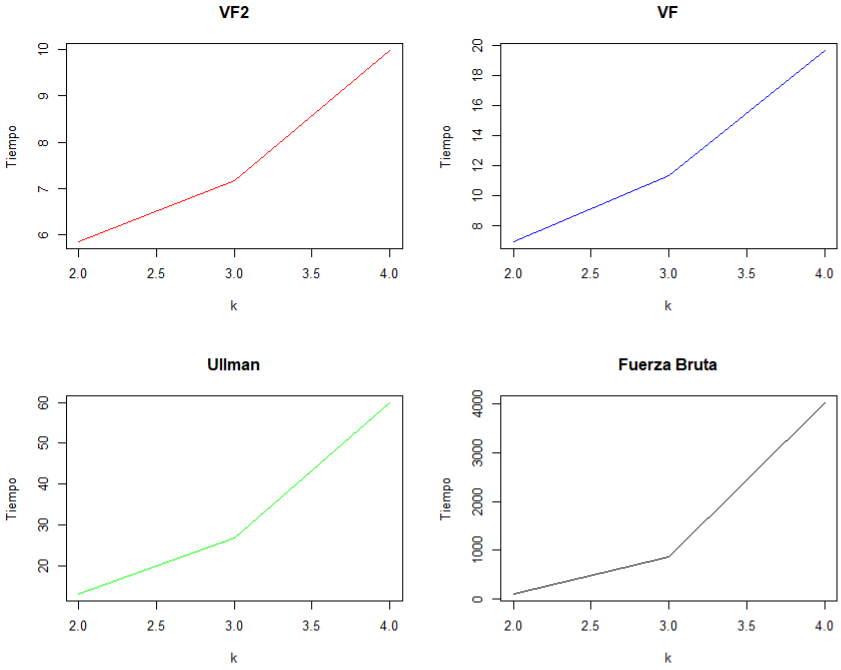
H :

	b	a	c
b	0	1	0
a	1	0	1
c	0	1	0

Por lo tanto, en este caso es fácil ver que al intercambiar la primera y segunda fila y la primera y segunda columna de H obtenemos G. Así, podemos afirmar que g y h son isomorfos.

7.2. Comparativa

Pues ya ves


VF2
Tiempo
6
7
8
9
10
2.0
2.5
3.0
3.5
4.0
k
VF
Tiempo
8
10
12
14
16
18
20
2.0
2.5
3.0
3.5
4.0
k
Ullman
Tiempo
20
30
40
50
60
2.0
2.5
3.0
3.5
4.0
k
Fuerza Bruta
Tiempo
0
1000
2000
3000
4000
2.0
2.5
3.0
3.5
4.0
k

Referencias

[1] Garey, Michael R. and Johnson, David S. (1979). Computers and Intractability. A guide to the theory of NP-Completeness. W. H. Freeman and company, New York.

[2] Gallego, Antonio-Javier and López, Damián and Calera-Rubio, Jorge. (2017). Grammatical inference of directed acyclic graph languages with polynomial time complexity.

[3] Rosales, Albert. (2013). Grafos isomorfos y árboles. SIN 501. Maracay.

[4] Whitney, Hassler. (1932). Congruent Graphs and the Connectivity of Graphs. American Journal of Mathematics. **54**, 150–168.

[5] Babai, László. (2019). Graph Isomorphism Update. Personal Webpage.

[6] Fernández, Pablo. El discreto encanto de la matemática.

[7] P. Foggia and C.Sansone and M. Vento. A Performance Comparison of Five Algorithms for Graph Isomorphism. Napoli.

[8] P. Foggia and C.Sansone and M. Vento and Luigi P. Cordella. (2004) A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs.

[9] J. R. Ullmann. (1976) An Algorithm for Subgraph Isomorphism. Journal of the ACM. New York.

A. Anexo

A.1. NP-Completitud del problema del subgrafo isomorfo

Aquí se incluirán algunas demostraciones y definiciones que eran necesarias en la sección teoría de la complejidad pero que no seguían la estructura del trabajo. Cualquiera de ellas puede ser consultada más detalladamente en [1].

Para el problema de **SATISFACIBILIDAD** necesitábamos las siguientes definiciones:

Sea $U = \{u_1, u_2, \ldots, u_m\}$ un conjunto de variables Booleanas. Una **asignación verdadera** para U es una función $t : U \longrightarrow \{T, F\}$. Si $t(u) = T$ entonces decimos que u es verdad bajo t; si $t(u) = F$ entonces decimos que u es falsa. Si u es una variable en U, entonces u y $\overline{u}$se llaman **literales** sobre U. El literal u es verdadero bajo t si y solo si la variable u es verdadera bajo t; el literal $\overline{u}$ es verdadero si y solo la variable u es falsa.

Una **cláusula** sobre U es un conjunto de literales sobre U, como por ejemplo $\{\overline{u_1}, u_2, u_8\}$. Representa la disyunción entre esos literales y es satisfecha por una asignación verdadera si y solo si al menos uno de sus miembros es verdadero bajo la asignación. La cláusula de arriba, por ejemplo, será satisfecha por t a no ser que $t(u_1) = T$, $t(u_2) = F$ y $t(u_8) = F$. Una colección C de cláusulas sobre U se dice que son **satisfactorias** si y solo si existe alguna asignación verdadera para U tal que satisfaga simultáneamente todas las cláusulas en C. A este asignamiento se le denomina **asignamiento verdadero y satisfactorio** para C.

También es de gran ayuda poder consultar la demostración de que VC es un problema $NP - Completo$.

VC es un problema NP-Completo.

Demostración.

Es fácil ver que $VC \in NP$ debido a que un algoritmo no-determinista solo necesita adivinar un subconjunto de vértices y comprobar en tiempo polinómico si el subconjunto contiene al menos uno de los dos vértices de cada arista y si tiene el tamaño apropiado.

La idea será transformar el problema 3SAT a VC. Sea $U = \{u_1, \ldots, u_n\}$ y $C = \{c_1, c_2, \ldots, c_m\}$ cualquier instancia de 3SAT. Tenemos que construir un grafo $G = (V, E)$ y un entero positivo $K \leq |V|$ tales que G tiene una cubierta de vértices de tamaño K o menor si y solo si C es satisfactorio.

A cada variable $u_i \in U$ asociaremos una componente $T_i = (V_i, E - i)$, con $V_i = \{u_i, \overline{u_i}\}$

y $E_i = \{\{u_i, \overline{u_i}\}\}$; esto es, dos vértices unidos por una única arista.

A cada cláusula $c_j \in C$, asociaremos una componente $S_j = (V'_j, E'_j)$, que consistirá en tres vértices unidos dos a dos con forma de triángulo:

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

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

Para cada cláusula $c_j \in C$, denotaremos por x_j, y_j, z_j a los tres literales en c_j. Así construimos el conjunto de aristas:

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

La construcción de nuestras instancia de VC se completa cuando elegimos $K = n+2m$ y $G = (V, E)$, donde

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

y

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

Es directo ver que esta construcción se puede hacer en tiempo polinómico (debido a la facilidad de la misma). Entonces, lo único que nos queda por probar es que C es satisfactorio si y solo si G tiene una cubierta de vértices de tamaño K o menor.

Por un lado, supongamos que $V' \subseteq V$ es una cubierta de vértices de G con $|V'| \leq K$. Por su propia definición, V' tiene que contener al menos un vértice de cada T_i y al menos dos de cada S_j. Como la suma de ambas nos da un total de $n + 2m = K$, necesariamente V' contendrá exactamente un vértice de cada T_j y exactamente un vértice de cada S_j. Por lo tanto, podemos usar el hecho de que V' intersecta a cada T_j para obtener una asignación verdadera $t : U \longrightarrow \{T, F\}$. Simplemente denotamos $t(u_i) = T$ si $u_i \in V'$ y $t(u_i) = F$ si $\overline{u_i} \in V'$. Para ver que esta asignación verdadera satisface cada una de las cláusulas, consideramos las tres aristas en cada E''_j. Solo dos de estas aristas puede ser cubierta por los vértices en $V'_j \cap V'$. Pero esto implica que el literal correspondiente, u_i o $\overline{u_i}$, de la cláusula c_j is verdadero bajo t, y por lo tanto cada cláusula c_j se satisface bajo t. Por lo tanto, t es un asignamiento verdadero y satisfactorio.

Por otro lado, supongamos que $t : U \longrightarrow \{T, F\}$ es un asignamiento verdadero y satisfactorio para C. La correspondiente cubierta de vértices V' incluye un vértice de cada T_i y dos vértices de cada S_j. El vértice de T_i en V' será u_i si $t(u_i) = T$ y será $\overline{u_i}$

si $t(u_i) = F$. Esto asegura que al menos una de las tres aristas en cada conjunto E''_j está cubierta, porque t satisface cada cláusula c_j. Así, solo necesitamos incluir en V' los vértices finales de cada S_j de las otras dos aristas en E''_j, y así obtendríamos la cubierta de vértices deseada.

■

A.2. Relaciones binarias

Definición 7.4 *Sea A un conjunto no vacío. Decimos que el conjunto $\mathcal{R}$ es una* ***relación binaria*** *en A si $\mathcal{R} \subseteq A \times A$.*

Los grafos son una manera de representación de cualquier relación binaria. Entendiendo los nodos como elementos del conjunto A y las aristas entre nodos indicándonos que cada par de nodos pertenece a nuestro conjunto $\mathcal{R}$.

Ejemplo 7.5 *Consideremos el conjunto $A = \{1, 2, 3\}$. Relaciones binarias podría ser $\mathcal{R}_1 = \{(1, 2), (2, 3)\}$,$\mathcal{R}_2 = \{(1, 2)\}$ y los grafos asociados serían:*

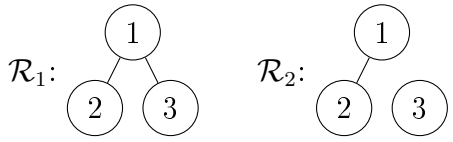


Figura 12: *Ejemplo de relación binaria.*

Definición 7.6 *Sea $\mathcal{R}$ una relación binaria en A. Se llama* ***clausura reflexiva-transitiva*** *de $\mathcal{R}$ a la relación binaria reflexiva y transitiva más pequeña $\mathcal{S}$ que contiene a $\mathcal{R}$, i.e, $\mathcal{S}$ satisface las propiedades:*

1. $\mathcal{R} \subseteq \mathcal{S}$
2. $\forall a, b, c \in A$: $(a, b) \in \mathcal{S} \wedge (b, c) \in \mathcal{S} \rightarrow (a, c) \in \mathcal{S}$ *(propiedad transitiva)*
3. $\forall a \in A$: $(a, a) \in \mathcal{S}$ *(propiedad reflexiva)*
4. *Si $\mathcal{S}'$ es otra relación binaria reflexiva y transitiva que contiene a $\mathcal{R}$, entonces $\mathcal{S} \subseteq \mathcal{S}'$*

Pasemos ahora el tipo de relación binaria que nos ayudará a contextualizar nuestra definición de grafo.

Definición 7.7 *Sea $\mathcal{R}$ una relación binaria en A. Se llama* ***orden parcial*** *a una relación binaria reflexiva, transitiva y que cumple la propiedad antisimétrica, i.e., $\mathcal{R}$ satisface:*

1. $\forall a, b, c \in A$: $(a, b) \in \mathcal{R} \wedge (b, c) \in \mathcal{R} \rightarrow (a, c) \in \mathcal{R}$
2. $\forall a \in A$: $(a, a) \in \mathcal{R}$
3. $(a, b) \in \mathcal{R} \wedge (b, a) \in \mathcal{R} \rightarrow a = b$ *(propiedad antisimétrica)*

Si se cumple la propiedad simétrica ($\forall a, b \in A$: $(a, b) \in \mathcal{R} \rightarrow (b, a) \in \mathcal{R}$) en vez de la antisimétrica en la definición anterior, a la relación binaria la llamaremos **relación de equivalencia**. Una relación de equivalencia $\mathcal{R}$ define subconjuntos disjuntos en A llamados **clases de equivalencia**:

Dado un elemento $a \in A$, el conjunto dado por todos los elementos relacionados con a definen la clase:

$$[a] = \{b \in A : (a, b) \in \mathcal{R}\}$$

Ejemplo 7.8 *Consideremos el conjunto $A = \{1, 2, 3\}$ y $\mathcal{R} = \{(1, 2), (2, 3)\}$ del ejemplo anterior. La clausura reflexiva-transitiva de $\mathcal{R}$ sería $R^* = R \cup \{(1, 1), (2, 2), (3, 3), (1, 3)\}$; que, además, es un orden parcial (la propiedad antisimétrica se cumple de forma directa).*

A.3. Matriz de vecindades e isomorfismo

Una forma muy útil de representar un grafo $g = (V, E)$ es **mediante su matriz de vecindades** (o matriz de adyacencia). La idea es formar una matriz de ceros y unos. Si el conjunto de vértices es $V = \{v_1, \ldots, v_n\}$, el grafo se puede escribir mediante una matriz $n \times n$:

	v_1	v_2	v_3	$\ldots$	v_n
v_1	δ_{11}	δ_{12}	δ_{13}	$\ldots$	δ_{1n}
v_2	δ_{21}	δ_{22}	δ_{23}	$\ldots$	δ_{2n}
v_3	δ_{31}	δ_{32}	δ_{33}	$\ldots$	δ_{3n}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
v_n	δ_{n1}	δ_{n2}	δ_{n3}	$\ldots$	δ_{nn}

donde:

$$\delta_{ij} = \begin{cases} 1, & \text{si } (v_i, v_j) \in E \\ 0, & \text{si } (v_i, v_j) \notin E \end{cases}$$

En el contexto de este trabajo, la matriz tendrá ceros en la diagonal por no permitir los lazos y no será necesariamente simétrica al contar con grados dirigidos (si el grafo no fuera dirigido la matriz sería simétrica).

Esta identificación es muy importante. Para muchas de las cuestiones que rodean a los grafos, bastará considerar el dibujo asociado a él. Pero el lenguaje de los grafos es un lenguaje especialmente diseñado para un uso algorítmico, que requerirán de un ordenador. Y para un ordenador, un grafo no es más que una matriz de ceros y unos.

Otro concepto fundamental en un grafo cualquiera en un grafo es el grado de un vértice (que no es lo mismo que el grado de entrada o salida):

Definición 7.9 *Dado un grafo $g = (V, E)$, diremos que dos vértices $v, w \in V$ son vecinos si $(v, w) \in E$. El* ***grado de un vértice*** *es el número de vecinos que tiene el grafo, i.e. :*

$$grado\ de\ v = gr(v) = |\{w \in V : (v, w) \in E\}|$$

Si el grado de un vértice es 0, diremos que es un vértice **aislado**. En la matriz de vecindades, para determinar el grado de un vértice v, basta con sumar los unos que aparezcan en su fila.

Sabemos que no es fácil, en general, decidir si dos grafos son isomorfos o no. Una manera de comprobar si dos grafos son isomorfos (que, por supuesto, habrán de tener el mismo número de vértices, digamos n), sería comprobar si alguna de las n! aplicaciones biyectivas entre los conjuntos de vértices respectivos cumple las propiedades necesarias para ser un isomorfismo entre los dos grafos. Pero este no es un procedimiento razonable si n es grande.

Sin embargo, para decidir que dos grafos no son ismorfos contamos con ciertas propiedades de un grafo que se han de conservar por isomorfismos:

1. Ambos grafos han de tener el mismo número de vértices.
2. Cada vértice ha de mantener sus relaciones de vecindad.
3. Si dos grafos son isomorfos, han de tener la misma sucesión de grados (todos los grados del grafo escritos de menor a mayor en un vector). Sin embargo, que dos grafos tengan la misma sucesión de grados no garantiza la isomorfía.
4. Los grafos isomorfos han de tener el mismo número de aristas.

Estas propiedades son necesarias a la hora de determinar un ismorfismo, pero no son suficientes. En general, no existe una caracterización para la isomorfía de dos grafos. De

todas formas, se puede interpretar también en términos de sus matrices correspondientes. Dados dos grafos g y h, con matrices de vecindades G y H, respectivamente:

g y h son isomorfos $\iff$ existe una permutación que, al aplicarla sobre las filas y columnas de G, obtenemos H (o viceversa)

El esquema de compartir secretos de Shamir, umbral (t, n), donde n es el número total de participantes y t, entre 1 y n, es el umbral: el número de participantes necesario para recuperar el secreto. La idea de Shamir para repartir un secreto $s \in F_q$ (finite field, q es primo que cumpla q >n), es escoger aleatoriamente un polinomio $f(x) \in F_q[X]$ de grado <= t-1, que cumpla $f(0) = s mod q$, y dar a cada participante P_i su fragmento $s_i = f(\alpha_i) mod q$.

Esos puntos donde se evalúan los polinomios, los $\alpha_i \in F_q$, tienen que ser DIFERENTES, para que luego la reconstrucción del secreto (a partir de los coeficientes de Lagrange) vaya bien.

Si tenemos un subconjunto de participantes A con cardinal t, entonces pueden recuperar el secreto vía interpolación polinómica:

$s = \lambda_i^A s_i \mod q$

donde $\lambda_i^A = \prod_{j \in A, j \neq i} -\alpha_j/(\alpha_i - \alpha_j) \mod q$

Esto funciona bien, en F_q todas esas diferencias $(\alpha_i - \alpha_j)$ son inveribles, bla, bla...

Pero para aplicar este esquema en la distribución de algunos sistemas de descifrado, basados en retículos, nos gustaría que esos coeficientes de recuperación λ_i^A fuesen, TODOS, lo más pequeño posibles !! Idealmente que fuesen todos 0 o 1, pero con Shamir eso sólo se puede hacer cuando t=1 y cuando t=n... muy limitado...

en vez de intentar que sean todos 0,1, pues otra pregunta sería: fijados t y n, hay alguna estrategia para escoger q, y luego para escoger los puntos de evaluación α_i, para asegurarse que todos los λ_i^A, vistos como elementos en F_q, son LO MAS PEQUEÑOS POSIBLES !! o sea que es como un problema de optimización, escoger q , $\alpha_1, ..., \alpha_n$ para minimizar el máximo de los $\lambda_i^A \mod q$, donde el máximo se toma entre todos los subconjuntos A con cardinal exactamente t, y todos los $i \in A$.

No sé si el problema así en general puede ser muy difícil que salga algún resultado genérico chulo, pero a lo mejor para valores concretos / pequeños de n y/o de t, se puede conseguir algo... no sé si para t= 2, o para (t,n)= (3,4) , etc. (por mencionar casos que suelen usarse en situaciones prácticas).