BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

Cómputo de la anchura arbórea de un grafo

por

Oscar Rafael Arachi Merced

Tesis presentada como requisito para obtener el título de
Licenciatura en Ciencias de la Computación

Facultad de Ciencias de la Computación
Asesor: Dr. Carlos Guillén Galván

Enero 2016

Resumen

En esta tesis se presentan los conceptos y resultados necesarios para computar la anchura arbórea de un grafo a través de uno de sus árboles de descomposición. Puesto que el concepto de anchura arbórea inicialmente es dado sobre la familia de todos los árboles de descomposición del grafo, se presentan resultados de la posibilidad de realizar este cómputo sobre cualquier elemento de dicha familia. También, se muestra un algoritmo de parámetro fijo tratable (PFT) junto con su implementación en Java. Finalmente, se muestran relaciones de tratabilidad con la posibilidad de expresar una propiedad gráfica en lenguaje de lógica monádica de segundo orden y la condición adicional de que la anchura arbórea del grafo sea acotada.

Introducción

Un problema muy importante en las ciencias e ingeniería es la determinación de algoritmos eficientes en el tiempo que resuelvan una tarea computacional planteada. Desde los trabajos de Robertson y Seymour los conceptos de árbol de descomposición y anchura arbórea (clique, banda, camino y rama, entre otros) han demostrado ser de gran importancia en la teoría de la complejidad computacional [Wei10, RS84, Bod93b]. Sin embargo, como se menciona en [Wei10], algunos ejemplos de aplicaciones se encuentran en áreas como grafo de datos (graph data), GIS (Geographic Information System), bases de datos XML, bioinformática, redes sociales y ontologías. La utilidad práctica de los métodos basados en el árbol de descomposición ha sido limitada por los dos problemas siguientes:

(1) Calcular la anchura arbórea de un grafo es díficil. Determinar si la anchura arbórea de un grafo dado es a lo más un entero dado p es **NP-completo** [ACP87, Bod93a]. Existen muchas heurísticas y algoritmos de aproximación para determinar la anchura arbórea, el inconveniente es que pocos de éstos pueden tratar con grafos que contienen más de 1000 nodos [KBvH01].

(2) Aún si la anchura arbórea es determinada, no se puede garantizar un buen desempeño, ya que la complejidad en tiempo de la mayor parte de los algoritmos es exponencial en la anchura arbórea.

En consecuencia, para resolver de manera eficiente problemas díficiles por métodos basados en el árbol de descomposición se requiere exigir que el grafo tenga anchura arbórea acotada. De ahí la importancia de los métodos basados en el concepto de árbol de descomposición y de parámetro fijo para que muchos problemas intratables puedan ser resueltos en tiempo polinomial e incluso en tiempo lineal para grafos con anchura arbórea acotada por una constante.

En esta dirección existe una cantidad considerable de trabajos orientados a caracterizar diversos problemas que puedan ser modelados a través de grafos y así obtener medidas que nos conduzcan a la determinación de algoritmos eficientes en el tiempo para su solución [MZ06, BELL05, SA91]. En esta misma dirección podemos mencionar los trabajos relacionados con los ancho conceptos [Cou92, Cou97, BC12]. En términos intuitivos, un ancho concepto es una medida que se obtiene de un grafo, el cual a su vez es obtenido de otro grafo que modela un determinado problema. Esta medida junto con la posibilidad de expresar

el problema en términos de una estructura lógica nos conduce a saber sobre la existencia de un algoritmo computacionalmente eficiente en el tiempo.

En el presente trabajo mostramos los conceptos y resultados para obtener el árbol de descomposición de un grafo, se presentan también algunos ejemplos de su obtención. Se puede afirmar que un árbol de descomposición del grafo es parte fundamental para obtener la medida, que en nuestro caso de estudio estará enfocada a la anchura arbórea de un grafo. Tomando en cuenta que existen muchos algoritmos para la obtención del árbol de descomposición, algunos muy complicados, mostraremos un algoritmo simple del estado del arte que es de parámetro fijo tratable (PFT), así como su implementación en lenguaje Java.

El contenido de la tesis es el siguiente. En el primer capítulo se establecen los conceptos básicos de la teoría de grafos y del cálculo proposicional. En el capítulo 2 se presenta la teoría de los lenguajes lógicos, en el capítulo 3 se establece la teoría necesaria para obtener el árbol de descomposición de un grafo y su anchura arbórea y se establece la relación de tratabilidad con la expresabilidad en lenguaje monádico de segundo orden y la acotación de la anchura arbórea del grafo. En el capítulo 4, se presenta el algoritmo que obtiene el árbol de descomposición de un grafo y su anchura arbórea. Finalmente en el capítulo 5 se presentan las conclusiones y trabajo futuro. En el apéndice se presenta la implementación del algoritmo que construye un árbol de descomposición de un grafo dado y que realiza el cómputo de su anchura arbórea.

Reconocimientos

Me gustaría agradecer a todas las personas que formaron parte de la realización de este proyecto.

En primer lugar a la Vicerrectoría de Investigación y Estudios de Posgrado (VIEP), por su apoyo al proyecto **Unimodalidad de los polinomios de independencia de sumas de Zyckov (ID: 00311)**, aprobado en marzo de 2015, que hizo posible la elaboración de esta tesis.

Al Dr. Carlos Guillén Galván, asesor de tesis, por apoyarme durante la elección y desarrollo del tema, por el tiempo dedicado a la elaboración y revisión, sin su dirección esto no habría sido posible.

A mis sinodales: Dr. César Bautista Ramos, Dr. Rafael Lemuz López y Dra. Irene Ayaquica, por sus valiosas observaciones y correcciones al trabajo.

Finalmente a mis Padres que estuvieron desde el inicio y que nunca me escatimaron su apoyo durante todos mis estudios y en el proceso de desarrollo de esta tesis.

Contenido

Lista de Figuras

Símbolos

R	Relación.
$\sim$	Relación de equivalencia.
$\preccurlyeq$	Relación de orden parcial.
$\mathbb{R}$	Conjunto de los números reales.
G	Grafo.
V	Conjunto de vértices.
E	Conjunto de aristas.
K_n	Grafo completo de n nodos.
$\overline{G}$	Complemento de G.
$\|G\|$	Número de vértices de G.
g	Género de un grafo.
$\mathcal{S}$	Teoría formal.
Σ	Alfabeto.
Σ^*	Conjunto de cadenas finitas formadas en el alfabeto Σ.
$\mathfrak{P}$	Sistema Post Canónico.
$\mathbf{L}$	Teoría axiomática **L**.
L	Lenguaje en $\Sigma^* \times \Sigma^*$.
L_k	$k-$ésima porción de L.
$\Phi(\langle x, k\rangle)$	Máquina de Turing que reconoce el lenguaje L_k.
Φ_D	Máquina de Turing determinista.
Φ_N	Máquina de Turing no determinista.
$O(f(x))$	O grande de f.
PFT	Parámetro fijo tratable.
$\mathbf{P}$	Clase de complejidad **P**.
$\mathbf{NP}$	Clase de complejidad **NP**.
$O^*(f(k))$	Un algoritmo que corre en tiempo $f(k)\|x\|^c$.
$\mathcal{T}$	Árbol de descomposición de un grafo.
T_x	Nodo x de $\mathcal{T}$.

Capítulo 1

Preliminares

En este capítulo se presenta la terminología básica que es útil para el desarrollo de la tesis. Se inicia describiendo algunos conceptos de relaciones y posteriormente se dan los elementos necesarios de la teoría de grafos. Para el desarrollo de este capítulo nos basamos en [Die10, Ros12].

1.1 Relaciones

Definición 1.1. Una *relación binaria* entre los conjuntos A y B, es cualquier subconjunto R de $A \times B$. Si $(a, b) \in R$, se denota aRb.

Definición 1.2. Una relación $R \subseteq A \times A$ es *reflexiva* si para todo $x \in A$, xRx.

Por ejemplo sea $R = \{(a, b)|a = b \ o \ a = -b\}$ la relación sobre el conjunto de los enteros. R es reflexiva porque cumple que $a = a$.

Definición 1.3. Es *simétrica* si para cada $x, y \in A$, xRy implica yRx.

Por ejemplo sea $R = \{(a, b)|a + b \leq 4\}$ la relación sobre el conjunto de los enteros. R es simétrica porque si $a + b \leq 4$ entonces $b + a \leq 4$.

Definición 1.4. Es *antisimétrica* si para todo $x, y \in A$, xRy y yRx implica $x = y$.

Por ejemplo sea $R = \{(a, b)|a = b + 1\}$ la relación sobre el conjunto de los enteros. R es antisimétrica porque no hay un par de elementos de la forma (b, a). En este caso $a = b + 1$ no es lo mismo que $b = a + 1$.

Definición 1.5. Es *transitiva* si para todo $x, y, z \in A$, xRy y yRz implica xRz y es *total* si todo par de elementos $x, y \in A$, son comparables, es decir xRy o yRx.

Por ejemplo sea R la relación de divisibilidad sobre el conjunto de los enteros positivos, suponiendo que $a|b$ y $b|c$, existen x en $\mathbb{Z}$ y y en $\mathbb{Z}$ tales que $a = bx$ y $b = cy$ entonces $c = a(xy)$ y así $a|c$. Por lo tanto R es transitiva.

Definición 1.6. Una relación R es *relación de equivalencia* en A, si es reflexiva, simétrica y transitiva, en tal caso denotamos $R = \sim$.

Por ejemplo, sea R la relación sobre el conjunto de los números reales tal que aRb si y solo si $a - b$ es un número entero. R es reflexiva porque $a - a = 0$ para todos los números reales a, entonces aRa. Es simétrica porque $a - b$ es un número entero y se cumple aRb, de este modo $b - a$ también lo es y se cumple que bRa. Es transitiva porque si aRb y bRc, entonces $a - b$ y $b - c$ son números enteros. Por ello $a - c = (a - b) + (b - c)$ también lo es, entonces aRc. Por lo tanto R es una relación de equivalencia.

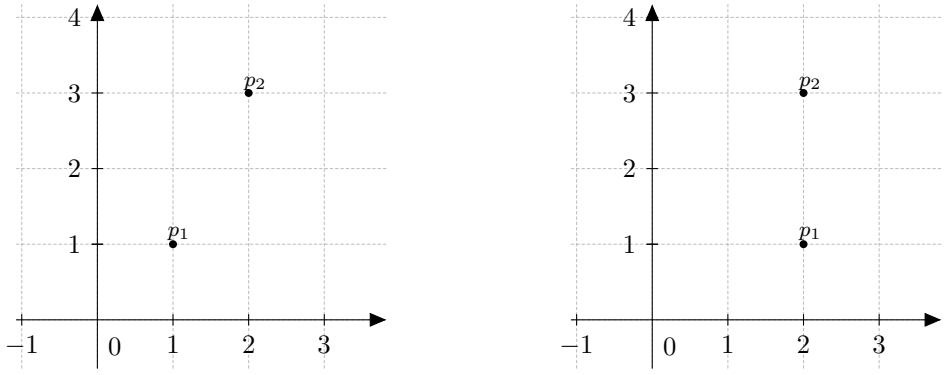
Definición 1.7. R una *relación de orden parcial* en A, si R es reflexiva, antisimétrica y transitiva, aquí denotamos $R = \preccurlyeq$.

Por ejemplo, sea S la clase de conjuntos con la relación contención $\subseteq$, es reflexiva porque $A \subseteq A$ cuando A es un subconjunto de S, es transitiva porque $A \subseteq B$ y $B \subseteq C$ implica que $A \subseteq C$. Finalmente, $\subseteq$ es antisimétrica porque $A \subseteq B$ y $B \subseteq A$ implica que $A = B$.

Definición 1.8. Si $\preccurlyeq$ es un orden parcial en A, un subconjunto X de A es una *cadena* sii para todo $x, y \in X$, se cumple $x \preccurlyeq y$ o $y \preccurlyeq x$. Un orden parcial $\preccurlyeq$ en A, es un **orden total** (lineal o completo) sii A es una cadena.

Por ejemplo sea $(\mathbb{R}^2, \preccurlyeq)$, consideremos el orden lexicográfico. Si p_1 y p_2 son dos puntos cualesquiera en $\mathbb{R}^2$ con las coordenadas $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, entonces $(x_1, y_1) \preccurlyeq (x_2, y_2)$ si y sólo si $x_1 < x_2$ o $(x_1 = x_2$ y $y_1 \leq y_2)$. En la figura 1.1 se muestra el ordenamiento lexicográfico de los puntos p_1 y p_2 en $\mathbb{R}^2$, que representa a $p_1 \preccurlyeq p_2$.

Definición 1.9. Dados n conjuntos $A_1, ..., A_n$, una *relación n-aria* entre estos conjuntos es un subconjunto de $A_1 \times A_2 \times ... \times A_n$.



(A) $x_1 < x_2$ (B) $x_1 = x_2$ y $y_1 \leq y_2$

FIGURA 1.1: Orden lexicográfico en $\mathbb{R}^2$.

Por ejemplo, en $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ la relación R definida por $(x, y, z) \in R$ si y sólo si $x < y < z$ es una relación de aridad 3, donde la relación $<$ es el orden estricto convencional en el conjunto de los números reales $\mathbb{R}$. Cualquier subconjunto de A es una relación unaria.

1.2 Grafos

Definición 1.10. Un *grafo* G es una estructura matemática que consta de un par ordenado de conjuntos (V, E), siendo $V \neq \emptyset$. Los elementos de V se llaman *vértices* o *nodos* y los elementos de E se llaman *aristas*. Una arista es un par $\{x, y\}$ no ordenado de vértices diferentes, esto es, E es un subconjunto del conjunto de subconjuntos binarios de V.

Consideremos por ejemplo el grafo G de la figura 1.2, el par $G = (V, E)$ viene dado por los conjuntos $V = \{1, 2, 3, 4\}$ y $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$.

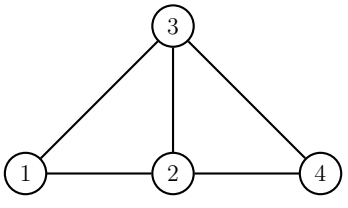


FIGURA 1.2: Grafo $G = (V, E)$

Definición 1.11. Sea V un conjunto no vacío (conjunto de *nodos*) y $E \subseteq V \times V$ (conjunto de *aristas*). Entonces, el par (V, E) se denomina *grafo dirigido* (en V) o *digrafo* (en V).

Por ejemplo, sea $G = (V, E)$ el grafo de la figura 1.3 donde $V = \{1, 2, 3, 4\}$ es el conjunto de nodos, mientras que $E = \{(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (3, 2), (3, 4), (4, 1), (4, 3)\}$ es el conjunto de aristas.

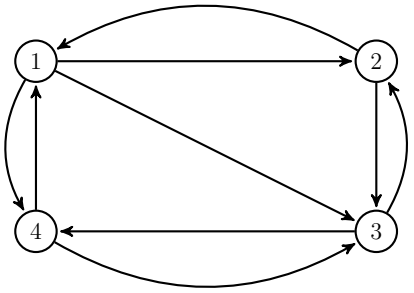


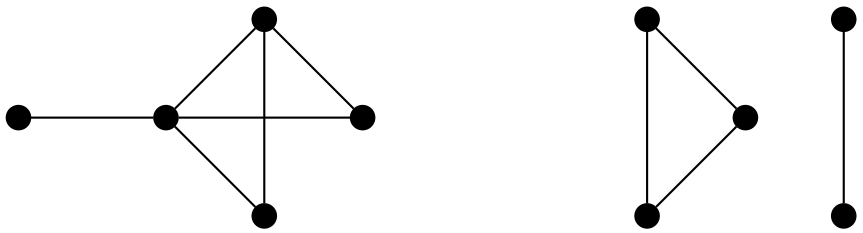
FIGURA 1.3: Ejemplo de un digrafo.

Definición 1.12. Dado $G = (V, E)$ un grafo, para $x, y \in V$, se dice que hay un *camino* en G de x a y, si existe una sucesión no vacía finita de aristas distintas de E, como $\{x, x_1\}$, $\{x_1, x_2\}$, $\cdots$, $\{x_{i-1}, x_i\}$, $\{x_{n-1}, x_n\}$, $\{x_n, y\}$. Cuando $x = y$, el camino se denomina *ciclo*. El número de aristas de un camino (ciclo) se denomina *longitud* del camino (ciclo).

Por ejemplo, en la figura 1.2, $\{1, 2\}$, $\{2, 4\}$, $\{4, 3\}$, es un camino del nodo 1 al nodo 3 de longitud 3.

Definición 1.13. Un grafo G se denomina *conexo* si existe un camino entre cualesquiera dos vértices distintos de G. Dado un grafo dirigido G, su grafo no dirigido asociado es el obtenido de G, ignorando la dirección de las aristas. G se considera conexo, si este grafo asociado es conexo. Un grafo que no sea conexo se denomina *no conexo*.

En la figura 1.4(A) se muestra un grafo conexo y en la figura 1.4(B) otro grafo que no es conexo.



(A) G_1 (B) G_2

FIGURA 1.4: El Grafo G_1 es conexo, y G_2 no lo es.

Definición 1.14. Un grafo $G = (V, E)$ se denomina *multigrafo*, si para $a, b \in V$, $a \neq b$, hay dos o más aristas de la forma a) (a, b) (para un digrafo); o b) $\{a, b\}$ (para un grafo no dirigido).

Por ejemplo, para el grafo G de la figura 1.5: $V = \{a, b, c, d\}$, $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, donde $e_1 = \{a, b\}$, $e_2 = \{a, c\}$, $e_3 = \{b, d\}$, $e_4 = \{a, d\}$, $e_5 = \{c, d\}$, $e_6 = \{a, b\}$, $e_7 = \{a, c\}$.

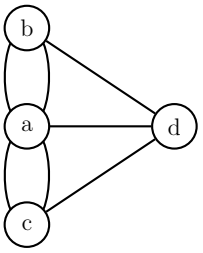
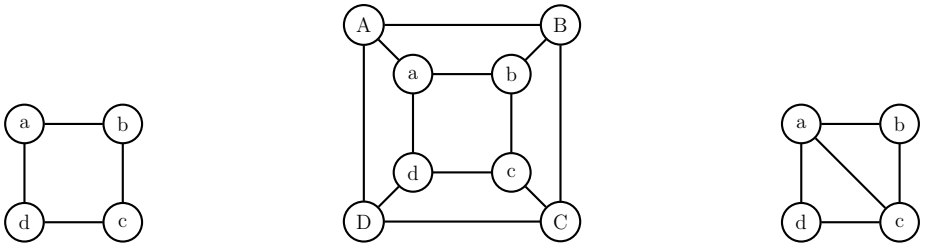


FIGURA 1.5: Multigrafo

Definición 1.15. Si $G = (V, E)$ es un grafo (dirigido o no dirigido), $G' = (V', E')$ se denomina *subgrafo* de G si $\varnothing \neq V' \subseteq V$ y $E' \subseteq E$, donde cada arista de E' es incidente con vértices de V'.

Un ejemplo de subgrafo se muestra en la figura 1.6.

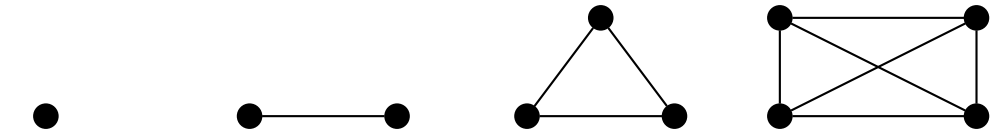


(A) G_1 (B) G_2 (C) G_3

FIGURA 1.6: G_1 es subgrafo de G_2 y también de G_3

Definición 1.16. El *grafo completo* en V, denotado por K_n es un grafo no dirigido, sin lazos, en el que para cualquier $a, b \in V$, $a \neq b$, hay una arista $\{a, b\}$.

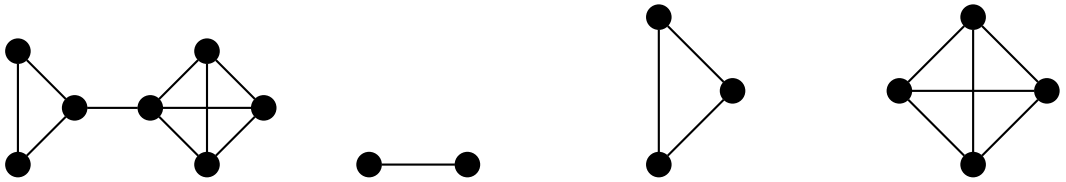
En la figura 1.7 se muestran algunos ejemplos de este tipo de grafo.



(A) K_1 (B) K_2 (C) K_3 (D) K_4

FIGURA 1.7: Grafos completos K_1, K_2, K_3 y K_4.

Definición 1.17. Un $m-$*clique* de un grafo G es el conjunto de vértices de un subgrafo completo K_m de G.



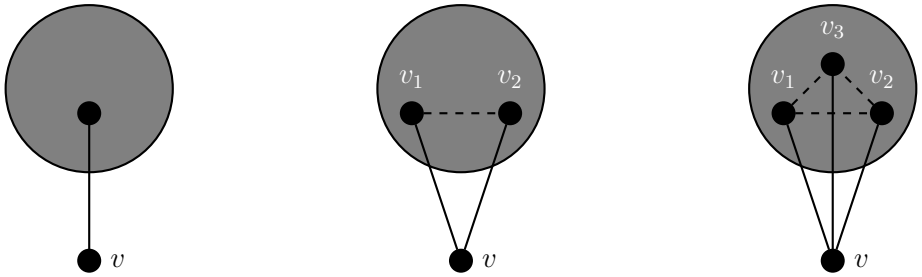
(A) G (B) $2-$*clique* (C) $3-$*clique* (D) $4-$*clique*

FIGURA 1.8: (A) Grafo G con un $2-$*clique*(B), $3-$*clique*(C) y un $4-$*clique*(D)

Definición 1.18. El *grado* de un vértice en un *grafo no dirigido* es el número de aristas incidentes con él, excepto que un ciclo en un vértice contribuye dos veces al grado de ese vértice. El grado de un vértice v se denota por $grado(v)$.

Definición 1.19. Un vértice v es *simplicial* en un grafo no dirigido G, si el conjunto de los vecinos de v forman un clique en G.

En la figura 1.9 se muestran algunos ejemplos de un vértice simplicial.



(A) (B) (C)

FIGURA 1.9: Grafo no dirigido que contiene un vértice v con grado 1 para el grafo (A), 2 para (B), y 3 para (C).

Definición 1.20. Sea G un grafo no dirigido sin lazos de n-vértices, el *complemento* de G, denotado por $\overline{G}$, es el subgrafo de K_n formado por los n vértices de G y las aristas que no están en G. (Si $G = K_n$, $\overline{G}$ es un grafo formado por n-vértices y sin aristas. Este se denomina grafo *nulo*.)

En la figura 1.10 se muestra un ejemplo del complemento de un grafo.

(A) G (B) G_1

FIGURA 1.10: Un grafo G y su grafo complemento.

Definición 1.21. Sea $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ dos grafos no dirigidos. Una función $\varphi : V_1 \to V_2$ se denomina *isomorfismo de grafos* si se cumplen las condiciones:

(a) φ es uno a uno y suprayectiva.

(b) Para todo $a, b \in V_1$, $\{a, b\} \in E_1$ si, y sólo si, $\{\varphi(a), \varphi(b)\} \in E_2$.

Cuando tal función existe, se dice que G_1 y G_2 son *grafos isomorfos*. Observe que la correspondencia de vértices de un isomorfismo de grafos mantiene las adyacencias. De esta manera, se mantiene la estructura de los grafos.

Se considera el siguiente ejemplo:

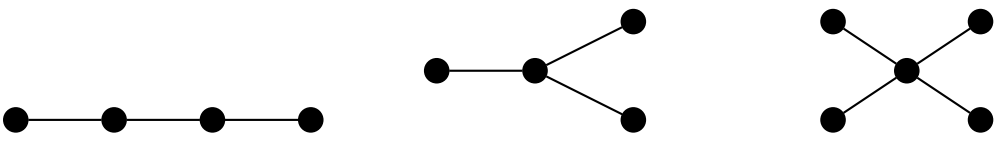
$V(G) = \{a, b, c, d\}$, $V(G') = \{s, t, u, v\}$

$ab \in E(G) \Rightarrow \varphi(a)\varphi(b) = vt \in E(G')$ $bd \in E(G) \Rightarrow \varphi(b)\varphi(d) = tu \in E(G')$ $bc \in E(G) \Rightarrow \varphi(b)\varphi(c) = ts \in E(G')$

Definición 1.22. Si φ es un isomorfismo del grafo G, en sí mismo es decir $\varphi : G \to G$ entonces φ se llama *automorfismo*

Definición 1.23. Sea $G = (V, E)$ un grafo no dirigido. G se denomina *árbol* si es conexo y no contiene ciclos.

En la siguiente figura 1.11 se muestran ejemplos de árboles.



(A) G (B) G_1 (C) G_2

FIGURA 1.11: Árboles.

Definición 1.24. Una *cubierta de vértices* de un grafo no dirigido $G = (V, E)$ es un subconjunto $V' \subseteq V$ tal que si $(u, v) \in E$, entonces $u \in V'$ o $v \in V'$ (o ambos). Esto es, cada vértice "cubre" sus aristas incidentes, y cada cubierta de vértices para G es un conjunto de vértices que cubren todas las aristas en E. El tamaño de una cubierta de vértices es el número de vértices en el.

Por ejemplo, el grafo de la figura 1.12(A) tiene una cubierta $\{c, f\}$ de tamaño 2. Mientras que el grafo de la figura 1.12(B) tiene una cubierta $\{a, b, d, e, g\}$ de tamaño 5.

(A) G (B) G_2

FIGURA 1.12: El grafo (A) con cubierta $\{c, f\}$ y el grafo (B) con cubierta $\{a, b, d, e, g\}$.

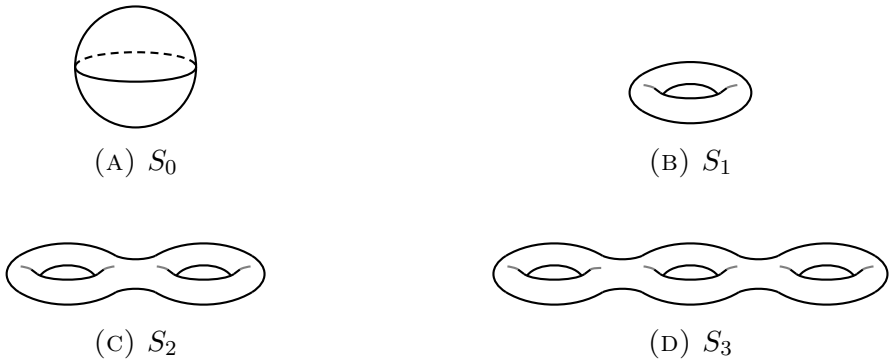
Definición 1.25. Un grafo es llamado *plano* si puede ser dibujado en el plano sin ningún cruzamiento de aristas (donde un cruzamiento de aristas es la intersección de las líneas que representan un punto que no es el punto final común).

Por ejemplo el grafo K_4 de la figura 1.13(a) es plano porque se puede dibujar sin que sus aristas se crucen.

(A) G (B) G_2

FIGURA 1.13: El grafo K_4 (A) y K_4(B) sin cruce de aristas .

Definición 1.26. El *género* de un grafo, denotado g, es el subíndice de la primera superficie entre la familia $S_0, S_1, S_2, \ldots$, sobre la cual el grafo se puede dibujar sin que ninguna de sus aristas se crucen.



(A) S_0 (B) S_1

(C) S_2 (D) S_3

FIGURA 1.14: Esfera(A), toro de un agujero(B), toro de dos agujeros(C) y toro de tres agujeros(D)

En la figura 1.14 se muestran los cuatro primeros miembros de una familia infinita de superficies.

Por ejemplo el grafo $K_{3,3}$ tiene género $g = 1$ porque no es un grafo plano y no puede ser dibujado sin que sus aristas se crucen en S_0, pero se puede dibujar en S_1 como se muestra en la figura 1.15. Entonces 1 es el subíndice de la primera superficie en la familia $S_0, S_1, S_2, \ldots$, en la cual se puede dibujar sin que se crucen sus aristas.

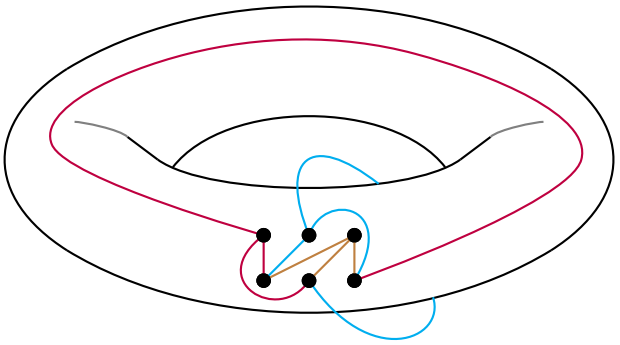
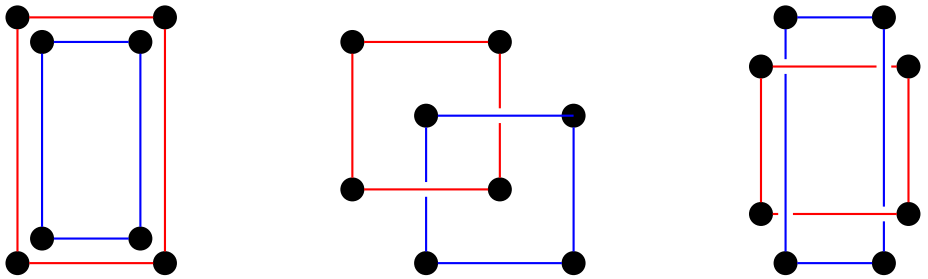


FIGURA 1.15: $K_{3,3}$ dibujado en S_1

Definición 1.27. El *número de ligaduras* de un grafo G, visto como un subconjunto de un espacio tridimensional, es el tamaño máximo de una colección de ciclos disjuntos ligados topológicamente.



(A) (B) (C)

FIGURA 1.16: El número de ligadura del grafo (A) = 0, el de (B)=1 y el de (C)=2.

Capítulo 2

Lenguajes Lógicos

En esta sección presentamos los conceptos que nos llevan a la representación en el lenguaje de la lógica monádica de segundo orden (MS) de una propiedad gráfica. También se presentan algunos ejemplos que ilustran esta representación. Al final de la sección 2.4 presentamos una extensa lista de propiedades gráficas expresables en MS. Para el desarrollo de este material nos basamos en [Cou97, BC12, Men15].

2.1 Teoría formal

Nos interesa determinar si cierta estructura de un grafo puede ser modelada a través de un sistema formal, en particular si puede ser expresada en términos de la lógica monádica de segundo orden. Para este fin comenzamos describiendo en términos generales en que consiste una teoría formal, también recordamos las definiciones formales de los conceptos de *prueba, teorema, decidible* e *indecidible*.

Una teoría formal $\mathcal{S}$ consiste de una tupla

$$\mathcal{S} = \langle \Sigma, F, A, R \rangle$$

donde

Σ: Es un conjunto contable de *símbolos*.

F: Es un subconjunto de Σ^*(expresiones) llamado conjunto de *fórmulas bien formadas* (*fbfs*).

A: Es un subconjunto de F llamado *conjunto de axiomas*.

R: Es un conjunto finito de relaciones entre *fbfs* llamadas *reglas de inferencia.*

Una teoría formal es llamada *teoría axiomática* si existe un procedimiento efectivo para determinar si una expresión dada es un axioma.

Una *prueba* es una secuencia de fórmulas bien formadas $F_1, F_2, ..., F_n$, donde cada fórmula F_i o es un axioma o es una consecuencia directa de algunas de las fórmulas predecesoras mediante la aplicación de una de las reglas de inferencia del sistema $\mathcal{S}$. Un *teorema* en la teoría $\mathcal{S}$ es una fórmula T bien formada tal que T es la última *fbf* de una prueba en $\mathcal{S}$.

Aun cuando exista un procedimiento efectivo para revisar que cualquier fbf dada es un axioma, el concepto de 'teorema' no es necesariamente efectivo ya que, en general, no existe un procedimiento para determinar si dada cualquier fbf $\mathcal{B}$ existe una prueba para $\mathcal{B}$. Una teoría para la cual existe un procedimiento efectivo para determinar si una fórmula es un teorema, se dice que es *decidible*; en otro caso, se dice que la teoría es *indecidible.*

De manera informal, una teoría es decidible si se puede diseñar una máquina para saber qué fbfs son teoremas, mientras que, para una teoría indecidible, se requiere de ingenio para determinar qué fbfs son teoremas.

Una fbf $\mathcal{C}$ se dice que es una consecuencia en $\mathcal{S}$ de un conjunto Γ de fbfs si y sólo si existe una secuencia $\mathcal{B}_1$, ...,$\mathcal{B}_k$ de fbfs tales que $\mathcal{C}$ es $\mathcal{B}_k$ y, para cada i,o $\mathcal{B}_i$ es un axioma o $\mathcal{B}_i$ está en Γ o $\mathcal{B}_i$ es una consecuencia directa por alguna regla de inferencia de algunas de las anteriores fbfs en la secuencia. Esa secuencia es llamada una *prueba* (o *deducción*) *de* $\mathcal{C}$ *a partir de* Γ. Los miembros de Γ son llamados *hipótesis* o *premisas* de la prueba. Usamos $\Gamma \vdash \mathcal{C}$ como una abreviación para "$\mathcal{C}$ es una consecuencia de Γ". Para evitar confusión cuando tratemos con más de una teoría escribimos $\Gamma \vdash_{\mathcal{S}} \mathcal{C}$ agregando el subíndice $\mathcal{S}$ para indicar la teoría en cuestión.

Si Γ es un conjunto finito $\{\mathcal{H}_1, \ldots, \mathcal{H}_m\}$, escribimos $\mathcal{H}_1, \ldots, \mathcal{H}_m \vdash \mathcal{C}$ en lugar de $\{\mathcal{H}_1, \ldots, \mathcal{H}_m\} \vdash \mathcal{C}$. Si Γ es el conjunto vacío $\varnothing$ entonces $\varnothing \vdash \mathcal{C}$ si y sólo si $\mathcal{C}$ es un teorema. Se acostumbra omitir el signo "$\varnothing$" y sólo escribir $\vdash \mathcal{C}$. Entonces $\vdash \mathcal{C}$ es otra manera de afirmar que $\mathcal{C}$ es un teorema.

Ejemplos simples de sistemas formales que resultan ser teorías axiomáticas son los *sistemas Post* (mediante estos sistemas se puede determinar la computabilidad de una función en el sentido clásico).

Sistemas Post. Consideremos un conjunto finito no vacío de símbolos Σ, una *producción* es una operación de la forma

$$g_0 S_1 g_1 S_2 \dots g_{m-1} S_m g_m \to h_0 S_{i_1} h_1 S_{i_2} \dots S_{i_n} h_n \tag{2.1}$$

donde las S_i son cadenas sobre el alfabeto Σ para todo $i = 1, 2, ..., m$, las g_i y h_j son cadenas fijas que pueden ser nulas, esto para cada $i = 1, 2, ..., m$ y $j = 1, 2, ..., n$. Los subíndices $i_1, i_2, ..., i_n$ son elegidos del conjunto $\{1, 2, ..., m\}$.

Un *sistema Post canónico* $\mathfrak{P}$ consiste de:

1. Un alfabeto Σ,
2. Las fórmulas bien formadas de cualquier elemento de Σ^*,
3. Un subconjunto A de Σ^*, los axiomas de $\mathfrak{P}$ y
4. Las reglas de inferencia, un conjunto finito Q de producciones de la forma 2.1 cuyas cadenas fijas pertenecen a Σ^*.

Un ejemplo particular de un sistema Post canónico es el siguiente:

1. $\Sigma = \{a, b\}$,
2. Las fórmulas bien formadas son cadenas finitas formadas con las letras a y b,
3. Los axiomas son: la cadena vacía Λ, la letra a y la letra b y
4. Las reglas de inferencia son las producciones: $S \to aSa$ y $S \to bSb$

Observe que en este ejemplo se tienen todos los requisitos para que $\mathfrak{P}$ sea un sistema formal. En este caso, es un sistema axiomático, porque las cadenas de cardinalidad cero y uno de Σ^* se eligen de forma conveniente como axiomas. También podemos comprobar que la teoría es decidible, en realidad cualquier palíndromo construido desde el alfabeto Σ es un teorema. Por ejemplo, la siguiente lista es una demostración para el palíndromo *ababa*:

1. a (axioma)
2. bab (regla de inferencia $S \to bSb$ con $S = a$)
3. $ababa$ (regla de inferencia $S \to aSa$ con $S = bab$)

Teoría axiomática L. Ahora introducimos una teoría formal axiomática **L** para el cálculo proposicional, éste es otro ejemplo de una teoría decidible. A continuación se enlistan los elementos necesarios para una teoría axiomática.

1. Los símbolos de **L** son $\neg$, $\Rightarrow$, (,), y las letras A_i con enteros i positivos como subíndices: $A_1, A_2, A_3, \ldots$. Los símbolos $\neg$ y $\Rightarrow$ son llamados *conectivos primitivos*, y las letras A_i son llamadas *letras proposicionales*.
2. (a) Todas las letras proposicionales son fbfs.
   (b) Si $\mathcal{B}$ y $\mathcal{C}$ son fbfs, entonces también lo son $(\neg\mathcal{B})$ y $(\mathcal{B} \Rightarrow \mathcal{C})$.
   Así, una fbf de **L** sólo es una forma proposicional construida a partir de las letras proposicionales A_i mediante los conectivos $\neg$ y $\Rightarrow$.
3. Si $\mathcal{B}$, $\mathcal{C}$ y $\mathcal{D}$ son fbfs de **L**, entonces los siguientes son axiomas de **L**:

   (A1) $(\mathcal{B} \Rightarrow (\mathcal{C} \Rightarrow \mathcal{B}))$

   (A2) $((\mathcal{B} \Rightarrow (\mathcal{C} \Rightarrow \mathcal{D})) \Rightarrow ((\mathcal{B} \Rightarrow \mathcal{C}) \Rightarrow (\mathcal{B} \Rightarrow \mathcal{D})))$

   (A3) $(((\neg\mathcal{C}) \Rightarrow (\neg\mathcal{B})) \Rightarrow (((\neg\mathcal{C}) \Rightarrow \mathcal{B}) \Rightarrow \mathcal{C}))$
4. La única regla de inferencia de **L** es *modus ponens*: $\mathcal{C}$ es una consecuencia directa de $\mathcal{B}$ y $(\mathcal{B} \Rightarrow \mathcal{C})$. Abreviamos la aplicación de esta regla mediante **MP**.

Observe que cada uno de los tres esquemas de axiomas (A1)−(A3) generan un conjunto infinito de axiomas de **L**. Se puede comprobar mediante algoritmos de búsqueda en árboles si una fbf dada es o no un axioma; por lo tanto la teoría **L** es axiomática. Al definir el sistema **L**, la intención es obtener como teoremas precisamente la clase de todas las tautologías.

Se introducen otros conectivos como la conjunción, disjunción y la bicondicional como a continuación se enlistan de manera respectiva:

(D1) $(\mathcal{B} \wedge \mathcal{C})$ por $\neg(\mathcal{B} \Rightarrow \neg\mathcal{C})$

(D1) $(\mathcal{B} \vee \mathcal{C})$ por $(\neg\mathcal{B}) \Rightarrow \mathcal{C}$

(D3) $(\mathcal{B} \Leftrightarrow \mathcal{C})$ por $(\mathcal{B} \Rightarrow \mathcal{C}) \wedge (\mathcal{C} \Rightarrow \mathcal{B})$

El significado de (D1), por ejemplo, es que, para cualesquiera fbfs $\mathcal{B}$ y $\mathcal{C}$, "$(\mathcal{B} \wedge \mathcal{C})$" es una abreviación para "$\neg(\mathcal{B} \Rightarrow \neg\mathcal{C})$".

2.2 El lenguaje de la lógica de primer orden

El cálculo proposicional se queda muy limitado cuando necesitamos expresar que todos o algún elemento de un conjunto satisface una propiedad determinada. Por este motivo se hace necesario extender el sistema **L** a un sistema lógico que incluya este tipo de situaciones. Para esto se introducen los cuantificadores que se utilizan para expresar propiedades que comparten un conjunto de elementos sin nombrar a los mismos.

Si $P(x)$ afirma que x tiene la propiedad P, entonces $(\forall x)P(x)$ significa que todo elemento x cumple con la propiedad P. Por otra parte, $(\exists x)P(x)$ significa que por lo menos existe un x que tiene la propiedad P. En $(\forall x)P(x)$, a '$(\forall x)$' se le llama *cuantificador universal*; en $(\exists x)P(x)$, '$(\exists x)$' se le llama *cuantificador existencial*.

Utilizaremos comas, paréntesis, los símbolos $\neg$ y $\Rightarrow$ del cálculo proposicional, el símbolo del cuantificador universal $\forall$, y la siguiente tupla de símbolos:

$$\mathcal{S}_1 = \langle \Sigma_V, \Sigma_C, \Sigma_P, \Sigma_F \rangle$$

donde

Σ_V: Es el conjunto de *variables individuales* $(x_1, x_2, \ldots, x_n, \ldots)$.

Σ_C: Es el conjunto de *constantes individuales* $(a_1, a_2, \ldots, a_n, \ldots)$.

Σ_P: Es el conjunto de *símbolos predicado* (A_k^n donde n y k son algunos enteros positivos).

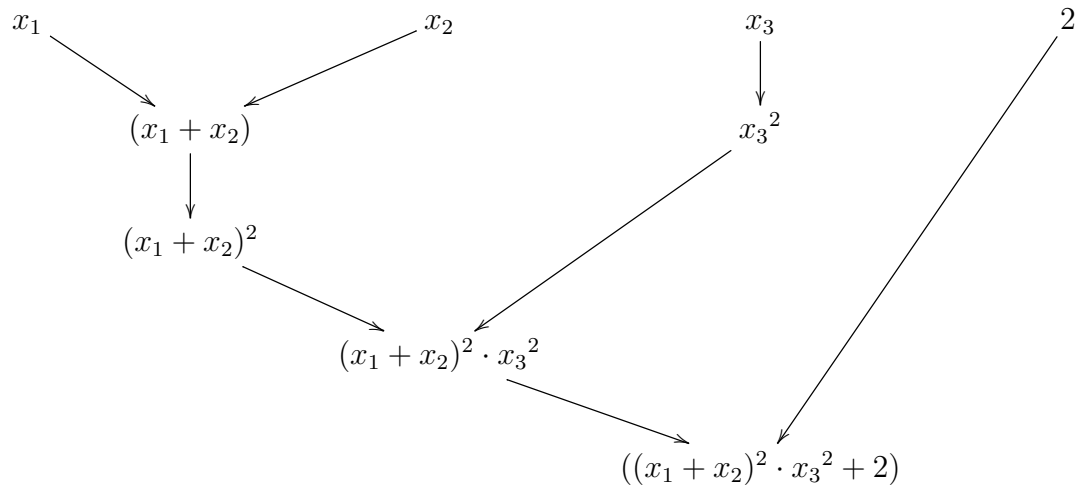
Σ_F: Es el conjunto de *símbolos función* (f_k^n donde n y k son algunos enteros positivos).

El entero positivo n es un superíndice del símbolo de predicado A_k^n o de un símbolo de función f_k^n que indica el número de argumentos, mientras que el subíndice k es solo un número índice para distinguir diferentes símbolos de predicados o de funciones con un mismo número de argumentos.

El Σ_F aplicado a Σ_V y Σ_C generan los términos:

1. Las variables y las constantes individuales son términos.
2. Si f_k^n es un símbolo de función y $t_1, t_2, \ldots, t_n$ son términos, entonces la expresión $f_k^n(t_1, t_2, \ldots, t_n)$ es un término.
3. Una expresión es un término sólo si se puede demostrar que es un término en función de las condiciones 1 y 2.

El siguiente árbol ilustra paso a paso la construcción del término $(x_1+x_2)^2 \cdot {x_3}^2+2$.



En este ejemplo x_1, x_2 y x_3 son variables individuales, el número 2 es una constante individual y hay tres símbolos función:

1. $f_1^1(x) = x^2$
2. $f_2^2(x, y) = (x + y)$
3. $f_3^2(x, y) = x \cdot y$

En el nivel cero (nivel superior) se encuentran los términos más simples que son las variables y constantes individuales, en el primer nivel se encuentran los términos

$(x_1 + x_2)$ y ${x_3}^2$ que se forman a partir de $f_2^2(x_1, x_2)$ y $f_1^1(x_3)$ respectivamente. El segundo nivel se obtiene de $f_1^1(f_2^2(x_1, x_2))$, el tercer nivel es una consecuencia de $f_3^2(f_1^1(f_2^2(x_1, x_2)), f_1^1(x_3))$ y por último en el cuarto nivel tenemos el término $f_2^2(f_3^2(f_1^1(f_2^2(x_1, x_2)), f_1^1(x_3)), 2)$.

Los símbolos de predicado aplicados a los términos producen las *fórmulas atómicas*, es decir, si A_k^n es un símbolo de predicado y $t_1, t_2, \ldots, t_n$ son términos, entonces $A_k^n(t_1, t_2, \ldots, t_n)$ es una fórmula atómica.

Por ejemplo, la desigualdad del triángulo es una fórmula atómica, donde tenemos el predicado $A_1^2(x, y) : x \leq y$ y los términos $t_1 = f_1^1(a + b) = |a + b|$ y $t_2 = f_2^2(|a|, |b|) = |a| + |b|$. Entonces la desigualdad del triángulo queda representada por

$$A_1^2(t_1, t_2) : |a + b| \leq |a| + |b|$$

Las *fórmulas bien formadas* (*fbf*) de la teoría de la cuantificación se definen de la siguiente manera:

1. Toda fórmula atómica es una fbf.
2. Si $\mathcal{B}$ y $\mathcal{C}$ son fbfs y si y es una variable, entonces $(\neg\mathcal{B})$, $(\mathcal{B} \Rightarrow \mathcal{C})$, y $((\forall y)(\mathcal{B}))$ son fbfs.
3. Una expresión es una fbf sólo si se puede demostrar que es una fbf en función de las condiciones 1 y 2.

La expresión $(\exists y)(\mathcal{B})$ denota la fórmula $\neg((\forall y)(\neg\mathcal{B}))$. En $((\forall y)\mathcal{B})$ y $((\exists y)(\mathcal{B}))$, la fórmula '$\mathcal{B}$' se llama el *alcance* del cuantificador '$\forall$' y $(\exists)$ respectivamente. La fórmula $\mathcal{B}$ no necesita contener la variable y. En tal caso, entendemos que $((\forall y)\mathcal{B})$ significa lo mismo que $\mathcal{B}$. Como ejemplo se tiene que la definición de límite de una función es una fórmula en un lenguaje de primer orden. Para esto se describen a continuación cada uno de los elementos que integran la fórmula.

- Variables individuales: x, ϵ, δ
- Constantes individuales: a, L, 0
- Símbolos predicado: $A_1^2(t_1, t_2) : t_1 < t_2$
- Símbolos función $f_1^1 = f$, $f_2^2(x, y) = (x - y)$, $f_3^1(x) = |x|$

- Términos: x, ϵ, δ, $f_1^1(x) = f(x)$, $f_3^1(f_2^2(f(x), L)) = |f(x) - L|$, $f_3^1(f_2^2(x, a)) = |x - a|$
- Fórmulas átomicas: $A_1^2(0, |x - a|) : 0 < |x - a|$, $A_1^2(|x - a|, \delta) : |x - a| < \delta$, $A_1^2(|f(x) - L|, \epsilon) : |f(x) - L| < \epsilon$

Entonces la siguiente fórmula está expresada en un lenguaje de primer orden.

$$(\forall\epsilon)(\exists\delta)\forall x\ 0 < |x - a| \wedge |x - a| < \delta \Rightarrow |f(x) - L| < \epsilon$$

2.3 El lenguaje de la lógica de segundo orden

El lenguaje de la lógica de segundo orden es una extensión del lenguaje de la lógica de primer orden, su sintáxis se obtiene del lenguaje de primer orden al añadir variables predicado y variables función así como la cuantificación sobre esas nuevas variables.

Como convención se utilizan las letras $X, Y, \ldots$ para variables predicado y $u, v, \ldots$ para variables función. Como en la lógica de primer orden usamos $x, y, \ldots$ para variables individuales.

A continuación se describen todos los elementos necesarios en un lenguaje de segundo orden, el cual se denotará con la letra L.

Para un lenguaje L, el conjunto de $L-$términos se da a continuación:

1. Las variables individuales son $L-$términos.
2. Los símbolos constantes en L son $L-$términos.
3. Si $t_1, \ldots, t_n$ son $L-$términos y f es un símbolo función $n-$ario de L, entonces $f(t_1, \ldots, t_n)$ es un $L-$término.
4. Si $t_1, \ldots, t_n$ son $L-$términos y u es una variable función de aridad n, entonces $u(t_1, \ldots, t_n)$ es un $L-$término.

Para un lenguaje L, el conjunto de $L-$fórmulas se define de la siguiente manera:

1. Si $t_1, \ldots, t_n$ son $L-$términos y P es un símbolo predicado $n-$ario de L, entonces $P(t_1, \ldots, t_n)$ es una $L-$fórmula.

2. Si $t_1, \ldots, t_n$ son $L-$términos y X es una variable relación $n-$aria, entonces $X(t_1, \ldots, t_n)$ es una $L-$fórmula.

3. Si t_1 y t_2 son $L-$términos, entonces $t_1 = t_2$ es una $L-$fórmula.

4. Si A es una $L-$fórmula, entonces $\neg A$, $(\forall x)A$, $(\exists x)A$, $(\forall X)A$, $(\exists X)A$, $(\forall u)A$, y $(\exists u)A$ son $L-$fórmulas.

5. Si A y B son $L-$fórmulas, entonces $(A \wedge B), (A \vee B)$ y $(A \rightarrow B)$ son $L-$fórmulas.

Un ejemplo de una fórmula en L es

$$(\exists P)(\psi_1 \wedge \psi_2 \wedge \psi_3)$$

donde $P(x, y)$ es el predicado "existe un camino de x a y", $G(x, y)$ es el predicado "el camino de x a y es una arista de G" y

$$\begin{aligned}
\psi_1 &= (\forall x)(\forall y)(P(x, y) \vee P(y, x) \vee (x = y)) \\
\psi_2 &= (\forall x)(\forall y)(\forall z)((\neg P(x, x)) \wedge ((P(x, y) \wedge P(y, z)) \rightarrow P(x, z)) \\
\psi_3 &= (\forall x)(\forall y)(((P(x, y) \wedge (\forall z)(\neg P(x, z) \vee \neg P(z, y))) \rightarrow G(x, y))
\end{aligned}$$

ψ_1 especifica que o hay un camino de x a y o un camino de y a x o $x = y$. Esto establece un orden en los vértices.

ψ_2 especifica que P es transitivo pero no reflexivo.

ψ_3 especifica que si hay un camino de x a y y no hay un vértice z, tal que hay un camino de x a z y un camino de z a y, entonces debe ser el caso que el camino de x a y es una arista de G.

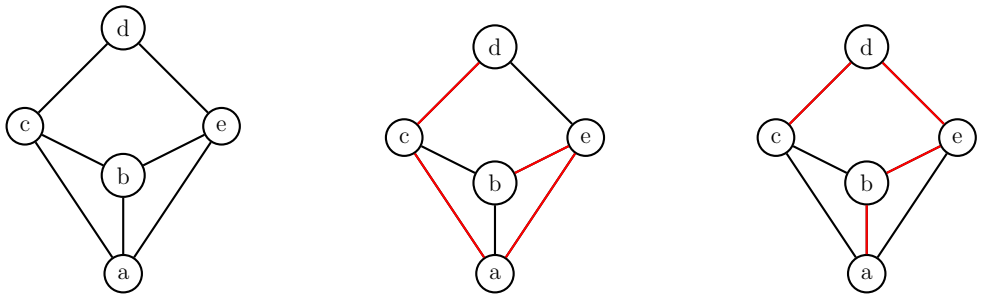


FIGURA 2.1: Camino hamiltoniano (B) y (C) del grafo (A).

2.4 El lenguaje de la lógica monádica de segundo orden

Como se verá a continuación, el lenguaje de la lógica monádica de segundo orden tiene gran poder expresivo a pesar de ser una restricción del lenguaje L, de hecho existen problemas NP-completos expresables en este lenguaje.

El lenguaje monádico de segundo orden es un lenguaje de segundo orden en el que todas las variables de segundo orden son monádicas, es decir las variables relación son de aridad uno o equivalentemente son conjuntos.

Veamos una definición precisa de esta lógica. Primero se definen los términos. Sea τ un vocabulario. Aquí distinguimos dos tipos de variables, las *variables de primer orden* que se denotan con las letras $x, y, z, ...$ y las llamadas *variables de segundo orden* (*conjuntos*) que se representan mediante $X, Y, Z, ...$ Los $\tau-$términos los definimos recursivamente de la siguiente manera:

1. Si x es una variable de primer orden, x es un término de primer orden.
2. Si X es un conjunto variable de segundo orden, X es un término de segundo orden.
3. Si f es una función $n-$aria en el vocabulario τ y $t_1, \ldots, t_n$ son términos de primer orden, entonces $f(t_1, \ldots, t_n)$ es un término de primer orden.

Definidos los términos podemos pasar a definir las fórmulas atómicas:

1. Si t y t^* son términos de primer orden, $t = t^*$ es una fórmula atómica.

2. Si T y T^* son términos de segundo orden, $T = T^*$ es una fórmula atómica.

3. Si T es un término de segundo orden y t es un término de primer orden, entonces $T(t)$ es una fórmula atómica.

Definidas las fórmulas atómicas, podemos pasar a definir las fórmulas:

1. Si α es una fórmula atómica, α es una fórmula.

2. Si α y β son fórmulas, entonces $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$ y $\neg\alpha$ son fórmulas.

3. Si $\alpha(x, X)$ es una fórmula, entonces $(\exists x)\alpha(x, X)$, $(\exists X)\alpha(x, X)$, $(\forall x)\alpha(x, X)$ y $(\forall X)\alpha(x, X)$ son fórmulas.

La notación $\alpha(x, X)$ significa que las variables x y X aparecen en la fórmula α. Si X es un conjunto o una propiedad se denota con $X(t)$ el hecho de que t es un elemento de X o satisface la propiedad X.

Veamos primero algunos ejemplos simples, sea δ_1 la sentencia

"dada cualquier propiedad, un individuo puede o no cumplirla"

La sentencia δ_1 puede ser expresada en lenguaje monádico de segundo orden como sigue:

- Términos de primer orden: x
- Términos de segundo orden: P
- Fórmulas atómicas: $\alpha(x, P) = P(x)$ y $\neg\alpha(x, P) = \neg P(x)$
- Subfórmulas: $(P(x) \vee \neg P(x))$, $(\forall x)(P(x) \vee \neg P(x))$ y δ_1

Entonces la fórmula δ_1 tiene la forma

$$(\forall P)(\forall x)(P(x) \vee \neg P(x))$$

Consideramos ahora un ejemplo sobre grafos. Sea γ_2 la sentencia

"el grafo G es 3-coloreable'

Se describen a continuación los elementos que integran la fórmula monádica de segundo orden.

- Términos de primer orden: x, y, z
- Términos de segundo orden: X, Y, Z
- Fórmulas atómicas: $edg(x, y)$ que denota el predicado

  "x e y son los extremos de una arista",

  $t \in H$, donde $t \in \{x, y, z\}$ y $H \in \{X, Y, Z\}$, $x = y$
- Fórmula δ_2:

$$(\exists X)(\exists Y)(\exists Z)(Part(X, Y, Z) \\ \wedge(\forall x)(\forall y)(edg(x, y) \wedge x \neq y \Rightarrow \neg(x \in X \wedge y \in X) \\ \wedge \neg(x \in Y \wedge y \in Y) \wedge \neg(x \in Z \wedge y \in Z)))$$

Donde $Part(X, Y, Z)$ indica que (X, Y, Z) es una partición del dominio. La fórmula $Part(X, Y, Z)$ se escribe como:

$$(\forall x)((x \in X \vee x \in Y \vee x \in Z) \wedge (\neg(x \in X \wedge x \in Y) \\ \wedge\neg(x \in Y \wedge x \in Z) \wedge \neg(x \in X \wedge x \in Z))).$$

Es decir que los extremos de una arista están en diferentes conjuntos X, Y, Z lo que significa que tienen diferentes colores. Por cada entero k, se puede construir

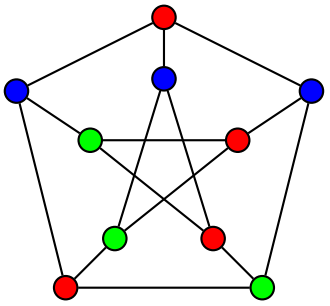


FIGURA 2.2: Ejemplo de 3-coloreo

una sentencia similar γ_k que represente el predicado *"G es k-coloreable"*.

En [SA91] se puede encontrar la siguiente lista de propiedades expresables en lenguaje monádico de segundo orden.

Teorema 2.1. *Las siguientes propiedades son expresables en lenguaje monádico de segundo orden:*

1. *Número domático (domatic) para k fijo.*
2. *k-colorabilidad de un grafo.*
3. *Número acromático para k fijo.*
4. *Triángulo cromático.*
5. *Partición en triángulos.*
6. *Partición en subgrafos isomorfos para H fijo conectado.*
7. *Partición en subgrafos Hamiltonianos.*
8. *Partición en bosques para k fijo.*
9. *Partición en cliques para k fijo.*
10. *Partición en acoplamientos (machings) perfectos para k fijo.*
11. *Cubiertas por cliques para k fijo.*
12. *Cubiertas por subgrafos bipartitos completos para k fijo.*
13. *Subgrafos inducidos con propiedad P (para propiedades monádicas de segundo orden P y k fijo).*
14. *Subgrafos conectados inducidos con propiedad P (para propiedades monádicas de segundo orden P y k fijo).*
15. *Caminos inducidos para k fijo.*
16. *Subgrafos cúbicos.*
17. *Completado Hamiltoniano para k fijo.*
18. *Circuito Hamiltoniano.*
19. *Circuito Hamiltoniano dirigido.*

20. *Camino Hamiltoniano (y camino Hamiltoniano dirigido).*
21. *Subgrafo isomorfismo para H fijo.*
22. *Grafo contractabilidad para H fijo.*
23. *Grafo homomorfismo para H fijo.*
24. *Camino con pares prohibidos para n fijo.*
25. *Kernel.*
26. *Grado de árbol abarcador restringido con k fijo.*
27. *Caminos conectados disjuntos para k fijo.*
28. *Compleción grafo cordal para k fijo.*
29. *Índice cromático para k fijo.*
30. *Problema de paridad árbol abarcador.*
31. *Distancia d número cromático para d y k fijos.*
32. *Espesor $\leq k$ para k fijo.*
33. *Membresía para cada clase C de grafos que son cerrados bajo la toma de menores.*

Capítulo 3

Anchura arbórea

Antes de introducir los resultados principales alrededor del concepto de anchura arbórea es necesario mostrar el significado de un problema tratable con parámetro fijo. Para este objetivo es conveniente recordar primero algunas nociones básicas de la teoría de la complejidad que nos interesan.

3.1 Problemas de decisión

Toda tarea computacional se puede clasificar principalmente dentro de tres grandes grupos de problemas:

- Problemas de decisión
- Problemas de conteo
- Problemas de optimización

En términos informales en un problema de decisión nos interesa determinar si una propiedad o característica se cumple o no. En un problema de conteo se requiere saber cúantos objetos cumplen con una característica dada, esto es, se requiere del cómputo de un número que expresa la cardinalidad del conjunto de los elementos que cumplen un predicado. Finalmente en un problema de optimización se requiere saber del óptimo sobre un conjunto con una propiedad determinada.

Por ejemplo, el problema de caminos Hamiltonianos puede plantearse como sigue.

Problema *Caminos Hamiltonianos*
Entrada: G un grafo.
Salida: *sí/no* (*sí*, cuando existe un camino Hamiltoniano en G,
no, en otro caso).

La versión de conteo de este problema puede ser un ejemplo de un problema de conteo.
Problema *Conteo de Caminos Hamiltonianos*
Entrada: G un grafo.
Salida: k (donde k es el número de caminos Hamiltonianos en G).

Como ejemplo de un problema de decisión podemos presentar el problema de determinar si un grafo tiene un clique.

Problema *Clique*
Entrada: G un grafo.
Salida: *sí/no* (es "*sí*" si G tiene un clique y es "*no*" en caso contrario).

En este trabajo todos los problemas que se plantean se ubican dentro de la clase de problemas de decisión, por lo que nos enfocaremos sólo a recordar algunas nociones relacionadas con esta clase de problemas.

Una *máquina de Turing determinista de k-cintas* T es una tupla $\langle Q, \Sigma, I, q_0, F\rangle$ donde

1. Q es un conjunto finito llamado el *conjunto de estados*.
2. Σ es un conjunto finito llamado *cinta de alfabeto* el cual siempre se asume que contiene un símbolo especial llamado *blanco* y es denotado por $\square$.
3. I es un estado finito de quintuplas $\{q, s, s', m, q'\}$ con $q, q' \in Q$, $s \in \Sigma^k$, $s' \in (\Sigma - \{\square\})^k$, y $m \in \{L, R, S\}^k$ de tal manera que no hay pares de quintuplas en el conjunto con los mismos dos primeros elementos(I incluye a lo más $|Q||\Sigma|^k$ elementos).
4. $q_0 \in Q$ es un estado especial llamado el *estado inicial*.
5. $F \subseteq Q$ es un conjunto de estados especiales llamado *estados finales*.

Una *máquina de Turing no determinista de k-cintas* se define de manera similar a la máquina de Turing determinista, solo que en el punto 3 si se permiten pares de quintuplas con los mismos primeros dos elementos.

Un *problema de decisión* se puede plantear en el contexto de lenguajes, formalmente un problema de decisión es una función

$$\Pi : \Sigma^* \to \{si, no\}$$

El *lenguaje derivado* del problema Π es el conjunto de cadenas en Σ^* que son aceptadas por Π, esto es

$$L_\Pi = \{s \in \Sigma^* : \Pi(s) = si\}$$

Dada la función $f : \mathbb{N} \to \mathbb{N}$ la clase de complejidad $DTIME(f(n))$ es la clase de lenguajes aceptados por una máquina de Turing determinista Φ_D en tiempo $f(n)$. La clase $NTIME(f(n))$ es la clase de lenguajes aceptados por una máquina de Turing no determinista Φ_N en tiempo $f(n)$. La clase **P** es la clase de lenguajes aceptados por una máquina de Turing determinista Φ_D en tiempo polinomial, esto es

$$\mathbf{P} = \bigcup_k DTIME(n^k)$$

La clase **NP** es la clase de lenguajes aceptados por una máquina de Turing no determinista Φ_D en tiempo polinomial, es decir

$$\mathbf{NP} = \bigcup_k NTIME(n^k)$$

Un lenguaje $L_1 \subseteq \Sigma_1^*$ es *reducible* al lenguaje $L_2 \subseteq \Sigma_2^*$ si existe una función $f : \Sigma_1^* \to \Sigma_2^*$ tal que

$$\forall x \in \Sigma_1^* : x \in L_1 \Leftrightarrow f(x) \in L_2$$

Decimos que L_1 *es reducible en tiempo polinomial a* L_2 si f es computable en tiempo polinomial por una Φ_D.

Un lenguaje $L \subseteq \Sigma^*$ es **NP-duro** si todo lenguaje $L' \in \mathbf{NP}$ es reducible en tiempo polinomial a L. Un lenguaje L es **NP-completo** si es **NP-duro** y $L \in \mathbf{NP}$.

3.2 Tratabilidad de parámetro-fijo

Dado un alfabeto Σ se consideran lenguajes L en el producto cartesiano $\Sigma^* \times \mathbb{N}$, es decir $L \subseteq \Sigma^* \times \mathbb{N}$. Tales lenguajes son llamados *lenguajes parametrizados*. Si $\langle x, k \rangle$

está en un lenguaje parametrizado L, k es llamado *parámetro*. Comúnmente el parámetro será un entero positivo, también puede ser un grafo o alguna estructura algebraica. Entonces el dominio del parámetro básicamente es el conjunto de los números naturales $\mathbb{N}$, esto es, se consideran los lenguajes en $\Sigma^* \times \mathbb{N}$. Para un k fijo llamamos a $L_k = \{\langle x, k\rangle : \langle x, k\rangle \in L\}$ la *k-ésima porción de L*.

Por ejemplo, consideremos el alfabeto $\Sigma = \{a, b\}$ y el lenguaje $L \subset \Sigma^* \times \mathbb{N}$ donde

$$\langle x, k\rangle \in L \Leftrightarrow x \in \Sigma^* \wedge |x| = k$$

entonces se tiene para $k = 0, 1, 2, 3$ que las k-ésimas porciones son:

$$L_0 = \{\lambda\},\ L_1 = \{\langle a, 1\rangle, \langle b, 1\rangle\},\ L_2 = \{\langle aa, 2\rangle, \langle ab, 2\rangle, \langle ba, 2\rangle, \langle bb, 2\rangle\}$$

y

$$L_3 = \{\langle aaa, 3\rangle, \langle aab, 3\rangle, \langle aba, 3\rangle, \langle abb, 3\rangle, \langle baa, 3\rangle, \langle bab, 3\rangle, \langle bba, 3\rangle, \langle bbb, 3\rangle\}$$

respectivamente.

Que una porción sea tratable significa que hay una constante c, independiente de k, tal que para todo k, la membresía de L_k puede ser determinada en tiempo $O(|x|^c)$. En el ejemplo anterior, $\langle x, k\rangle \in L_k$ se determina en tiempo $O(|x|)$, es decir $c = 1$. Formalmente se tiene la siguiente definición básica.

Definición 3.1. Un lenguaje parametrizado L es *parámetro fijo tratable* (PFT) si y sólo si existe un algoritmo Φ, una constante c y una función computable f tal que para todo $x, k, \Phi(\langle x, k\rangle)$ corre en tiempo a lo más $f(k)|x|^c$ y

$$\langle x, k\rangle \in L \Leftrightarrow \Phi(\langle x, k\rangle) = 1$$

Retomando el lenguaje L del ejemplo anterior, demostremos que es parámetro fijo tratable. Para esto sean $x \in \Sigma^*$ y $k \in \mathbb{N}$. Sea Φ el algoritmo:

La complejidad en tiempo se puede estimar de la siguiente manera. La linea 1 es una asignación que se realiza en una unidad de tiempo, la linea 2 consiste de $|x|$ comparaciones y $|x|$ asignaciones, por lo que se lleva $2|x|$ unidades de tiempo. Finalmente en la linea 3 se hace una comparación y una instrucción de escritura, por lo que consume un tiempo de 2 unidades. En consecuencia la complejidad del

Algorithm 3.1 Φ

Require: x, k (palabra, longitud).
Ensure: 1/0 (1 si la longitud de x es igual a k, 0 en otro caso)
1: i=0
2: Mientras la i-ésima letra de x sea distinta de blanco incrementar i en una unidad.
3: Escribir 1 si $i = k$, en otro caso escribir 0.

algoritmo Φ es del orden exacto de $3 + 2|x|$, entonces el algoritmo $\Phi(x, k)$ corre en tiempo a lo más $5|x|^1$ para palabras de longitud mayor que 1. Claramente se tiene que $c = 1$ y $f(k)$ es la constante 5.

Los siguientes ejemplos motivan otras variantes de tratabilidad de parámetro fijo.

Ejemplo 3.1. El problema de encontrar la menor cubierta de un grafo se conoce como *el problema de la cubierta de vértices*. Se sabe que este problema es **NP-completo** [Kar72]. Este problema conduce al problema de parámetro fijo siguiente.

Cobertura de vértices
Caso: Un grafo $G = (V, E)$.
Parámetro: Un entero positivo k.
Pregunta: ¿G tiene una cubierta de vértices de tamaño $\leq k$?

Se puede mostrar que este problema parametrizado por k puede ser resuelto en tiempo $2^k|G|$.

Teorema 3.1 ([Meh84, DF92])**.** *El problema de la cobertura de vértices es soluble en tiempo $O(2^k|G|)$, donde la constante oculta es independiente de k y $|G|$.*

Demostración. Se construye un árbol binario de altura k, de la siguiente manera. Se etiqueta la raíz del árbol con el conjunto vacío y el grafo G. Se elije una arista $uv \in E(G)$. En cualquier cubierta de vértices V' de G, se debe tener que $u \in V'$ o $v \in V'$, entonces se generan de la raíz los hijos correspondientes a estas dos posibilidades, uno con la etiqueta u y otro con la etiqueta v. El conjunto de vértices que etiquetan un nodo representan una cubierta de vértices potencial, y el grafo de etiquetas de nodos representa lo que debe ser cubierto en G. En general, para un nodo con etiquetas de vértices S, se elige una arista $wq \in E(G)$ con ni w ni q conectados con cualquier vértice en S, y crear los dos nodos hijos etiquetados,

respectivamente, $S \cup \{w\}$ y $S \cup \{q\}$. Si se crea un nodo de altura a lo más k en el árbol que cubre G, entonces una cobertura de vértices de cardinalidad k ha sido encontrada. No hay necesidad de explorar el árbol más allá de la altura k. □

Se ha demostrado que el problema de la cubierta de vértices, con parámetro k puede ser resuelto en tiempo $2^k|G|$, esta situación puede formalizarse mediante la siguiente definición de la **notación O***.

Definición 3.2. Un algoritmo parametrizado corre en tiempo $O^*(f(k))$, si éste corre en tiempo $f(k)|x|^c$.

De acuerdo a esta definición el algoritmo para la cubierta de vértices descrito en la prueba del teorema anterior corre en tiempo $O^*(2^k)$, esto es, se ignora la parte polinomial en $2^k|G|$.

Ilustremos la prueba del teorema anterior mediante el siguiente ejemplo. Dado el grafo de la figura 3.1 determinar si existe una cubierta de vértices de tamaño $k = 3$. Vamos a construir un árbol binario de altura $k = 3$ de la siguiente manera. Etiquetamos la raíz del árbol con el conjunto vacío y el grafo G (ϕ, G). Elegimos cualquier arista del grafo (en este caso $\{a, b\}$) y generamos los hijos del nodo raíz, uno con la etiqueta a y otro con la etiqueta b. Ahora elegimos otra arista de G que no este conectada con los vértices a o b (en este caso la arista $\{d, e\}$) y generamos los hijos del nodo a $(a, d$ y $a, e)$ y los hijos del nodo b $(b, d$ y $b, e)$. De manera similar elegimos otra arista que no este conectada con los vértices a, b, d, o e (en este caso la arista $\{g, h\}$). Ahora generamos los hijos de a, d $(a, d, g$ y $a, d, h)$, los hijos de a, e $(a, e, g$ y $a, e, h)$, los hijos de b, d $(b, d, g$ y $b, d, h)$ y los hijos de b, e $(b, e, g$ y $b, e, h)$. El árbol binario de altura k=3 que construimos se muestra en la figura 3.2.

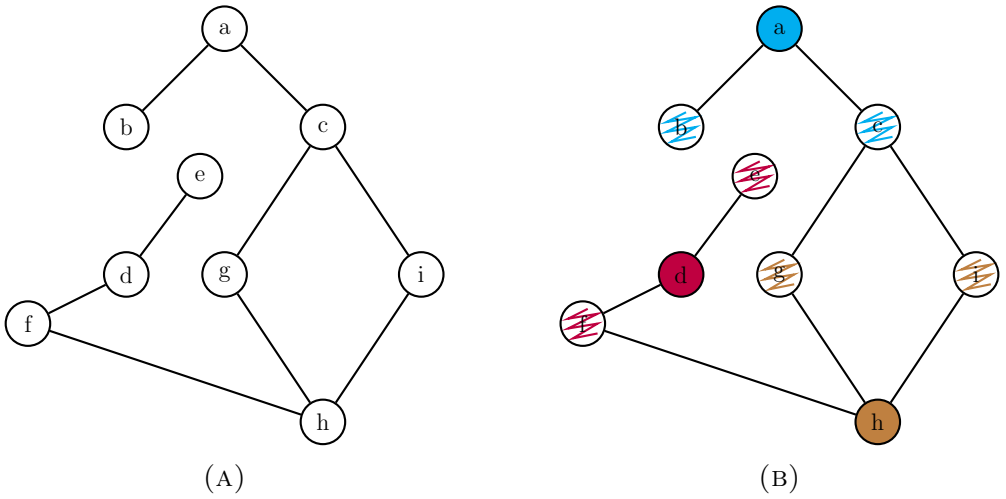


FIGURA 3.1: Grafo que ilustra la prueba del teorema 1.

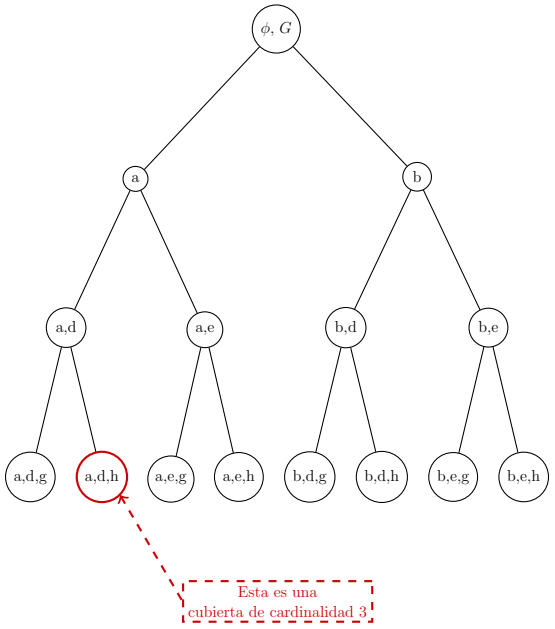


FIGURA 3.2: Árbol que prueba si hay una cubierta de cardinalidad $\leq k = 3$.

Ejemplo 3.2. El problema de determinar si un grafo dado puede ser embebido en una superficie con k agujeros es conocido como el problema del *género de un grafo*, que puede ser planteado como sigue.

Género de un grafo
Caso: Un grafo $G = (V, E)$.
Parámetro: Un entero positivo k.
Pregunta: ¿G tiene genero k?

Fellows y Langston en [FL87, FL88] introducen un método que les permite usar los resultados de Robertson y Seymour [RS95, RS04] para construir un algoritmo Φ que acepta (G, k) como instancias del algoritmo y como salida Φ determina si G tiene género k en tiempo $O(|G|^3)$. Posteriormente Mohar en [Moh96] mejora este tiempo a $O(|G|)$.

El siguiente ejemplo tiene que ver con la medida de entrelazamiento de un grafo sobre sí mismo, también ilustra otra situación de tratabilidad de parámetro fijo que posteriormente remarcaremos conjuntamente con los dos ejemplos anteriores.

Ejemplo 3.3. Número de ligadura de un grafo

Caso: Un grafo $G = (V, E)$.
Parámetro: Un entero positivo k.
Pregunta: ¿Puede ser G embebido dentro de un espacio tridimensional tal que el tamaño máximo de una colección de ciclos disjuntos ligados topológicamente es acotado por k?

Fellows y Langston en [FL87, FL88] también utilizan los métodos de Robertson y Seymour para probar que para cada parámetro k hay un algoritmo Ψ_k que corre en tiempo $O(|G|^3)$ y el cual determina si el grafo G tiene número de ligadura k.

Si consideramos las versiones clásicas de los tres ejemplos anteriores, es decir donde k no es fijo, entonces todos ellos pertenecen a la clase $NP-$difícil [Gar79].

Como se mencionó anteriormente con parámetro fijo todos estos problemas son del orden $O(|G|^c)$ para alguna *parte acertada* (*slicewise*) c. Sin embargo, cada uno de ellos muestra una variante de tratabilidad parametrizada. Para esto, se observan las siguientes diferencias de *tratabilidad de parámetro-fijo*.

En el problema de la cobertura de vértices (ejemplo 1) se conoce un algoritmo que trabaja para todo k y la constante involucrada en el orden de complejidad puede ser computada, en este caso es 2^k, y por tanto se puede calcular el tiempo exacto del algoritmo para cualquier k, recordando éste fue $2^k|G|$. En general, cuando contamos con estos dos elementos en la definición 3.1 decimos que el problema es *fuertemente uniforme tratable de parámetro fijo.*

Otra variante de tratabilidad se da en el ejemplo 2 del género de un grafo, donde se tiene un algoritmo Φ para todo k, pero no se tiene la manera de computar la constante involucrada en el orden de complejidad, lo que sabemos en este caso que para cada k, el tiempo de corrimiento del algoritmo Φ sobre una instancia $\langle G, k\rangle$ is $O(|G|^3)$. Este comportamiento es llamado *uniforme tratable de parámetro fijo.*

La última característica de tratabilidad se distingue en el problema del ejemplo 3 sobre el número de ligadura de un grafo. Aquí todo lo que se conoce es el exponente del tiempo de corrimiento para los algoritmos, esto es, para cada k existe un algoritmo diferente (y desconocido) que corre en $O(|G|^3)$ con constantes desconocidas. En tal caso, se dice que el problema es *no-uniforme tratable de parámetro fijo.*

Definición 3.3. Sea A un problema parametrizado.

(i) A es *fuertemente uniforme tratable de parámetro fijo* sii existe un algoritmo Φ, una constante c y una función computable $f : \mathbb{N} \to \mathbb{N}$ tal que, para cada x, k, el algoritmo $\Phi(\langle x, k\rangle)$ corre en tiempo a lo más $f(k)|x|^c$ y $\langle x, k\rangle \in A$ sii $\Phi(\langle x, k\rangle) = 1$.

(ii) A es *uniforme tratable de parámetro fijo* si existe un algoritmo Φ, una constante c, y una función $f : \mathbb{N} \to \mathbb{N}$ tal que el tiempo de corrimiento de $\Phi(\langle x, k\rangle)$ es a lo más de $f(k)|x|^c$.

(iii) A es *no-uniforme tratable de parámetro fijo* si existe una constante c, una función $f : \mathbb{N} \to \mathbb{N}$ y una colección de procedimientos $\{\Phi_k : k \in \mathbb{N}\}$ tal que para cada $k \in \mathbb{N}$ y el tiempo de corrimiento de $\Phi_k(\langle x, k\rangle)$ es $f(k)|x|^c$ y $\langle x, k\rangle \in A$ sii $\Phi_k(\langle x, k\rangle) = 1$.

3.3 Árbol de descomposición de un grafo

En términos informales el concepto de *anchura arbórea* es un parámetro que mide, a través de un *árbol de descomposición*, que tan parecido es un grafo a un árbol o un bosque, éste surge inicialmente en los trabajos de Robertson y Seymour [[RS85], [RS95], [RS04]]. Posteriormente aparecen conceptos equivalentes, descubiertos aparentemente de manera independiente, en otros contextos como en la inteligencia artificial.

Como se menciona en [Wei10], la utilidad práctica de los métodos basados en el árbol de descomposición ha sido limitada debido a los siguientes dos problemas:

(1) Calcular la anchura arbórea de un grafo es díficil. Determinar si la anchura arbórea de un grafo dado es a lo más un entero dado p es **NP-completo** [ACP87]. Aunque para p fijo existan algoritmos lineales para resolver el problema *anchura arbórea* $\leq p$ las constantes ocultas tienen valores tan altos que impiden que el algoritmo sea útil en la práctica [Bod93a]. Existen muchas heurísticas y algoritmos de aproximación para determinar la anchura arbórea, el inconveniente es que pocos de éstos pueden tratar con grafos que contienen más de 1000 nodos [KBvH01].

(2) Aún si la anchura arbórea es determinada, todavía no se puede garantizar que el buen desempeño será obtenido, ya que la complejidad en tiempo de la mayor parte de los algoritmos es exponencial en la anchura arbórea.

En consecuencia, para resolver de manera eficiente problemas díficiles por métodos basados en el árbol de descomposición se requiere exigir que el grafo tenga anchura arbórea acotada (digamos menor que 10).

Los métodos de parámetro fijo permiten un enfoque uniforme en tiempo lineal para los algoritmos de una amplia clase de grafos. Por otra parte, el parámetro es también la base de la evolución de algoritmos modernos como "Bidimensionalidad" y "Protuberancias". Como se verá más adelante, el problema de determinar si un grafo tiene anchura arbórea k es parámetro fijo tratable. Sin embargo, como los algoritmos conocidos para comprobar esta afirmación son imprácticos, en un principio se pensó que la anchura arbórea no impactaría en la algorítmia más que como una herramienta de clasificación. Contrario a este pensamiento, la teoría desarrollada en torno al concepto de anchura arbórea ha tenido impacto

en la teoría de grafos topológica, combinatoria, problemas de satisfacción de restricciones, problemas de inferencia y redes de creencia Bayesianas entre otras [KvHK02]. Todas las aplicaciones mencionadas arriba pueden ser encontradas en las referencias: [KvHK02, LS88, ADN05, Tho98, YAM03].

La siguiente definición, aunque desde el punto de vista práctico no es muy útil es conveniente darla para efectos teóricos.

Definición 3.4. ($k-$Árbol)

(a) La clase de $k-$árboles es la clase más pequeña que cumple con

  (i) K_{k+1}, es un $k-$*árbol* y

  (ii) Si G es un $k-$*árbol* y H es un subgrafo de G isomorfo a K_k, entonces el grafo G' construido de G al agregar primero un nuevo vértice v a G y luego añadir aristas para crear $H \cup \{v\}$ una copia de K_{k+1}, es un $k-$*árbol*.

(b) Si G_1 es un subgrafo de un $k-$*árbol*, entonces G_1 es llamado un $k-$*árbol parcial*.

Observe de (i) del inciso (a), que para cada $n \in \mathbb{N}$ el grafo completo K_n es un $(n-1)$- árbol. También si consideramos un grafo G que consiste de un sólo nodo, de acuerdo a (b), G es un $k-$*árbol parcial* para todo entero positivo k. Todos los posibles subgrafos de K_3 son ejemplos de $3-$árboles parciales (ver figura 3.3), de acuerdo a la definicion 3.1 (b), también son $k-$árboles parciales para todo $k \geq 3$ ya que también son subgrafos de todos los K_n para $n \geq 3$. También observe que dado un $k-$árbol parcial, éste también puede ser un $k'-$árbol parcial para un $k' < k$, por ejemplo, K_3 es un $3-$árbol parcial y también es un $2-$árbol parcial. Sin embargo, K_3 no puede ser un $1-$árbol parcial. Esta última observación motiva la siguiente definición de *anchura arbórea*.

La construcción dada en la definición 3.1 (a) se ilustra mediante los grafos dados en las figuras 3.4 y 3.5.

Siguiendo el procedimiento de la figura 3.4 se genera cualquier elemento de la familia de todos los 1-árboles. Observe que esta familia es la clase de todos los bosques.

Siguiendo el mismo procedimiento las siguientes figuras corresponden a la construcción de 3-árboles.

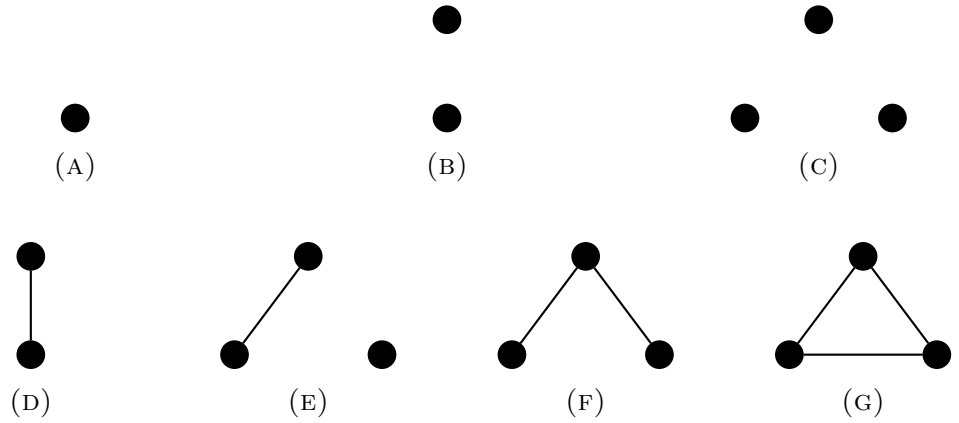


FIGURA 3.3: $k-$árboles parciales para todo $k \geq 3$

Definición 3.5. (Anchura arbórea) Decimos que un grafo G tiene una *anchura arbórea* k si k es el mínimo entero positivo tal que G es un $k-$árbol parcial.

Es inmediato que la anchura arbórea de un grafo completo de n nodos es $n-1$, sin embargo, para determinar la anchura arbórea con esta definición en general resulta complicado, además para propósitos algorítmicos no es de mucha ayuda, por lo que una definición equivalente de la anchura arbórea, dada a través del *árbol de descomposición*, se ofrece a continuación.

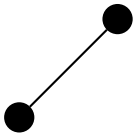
Definición 3.6. (a) Un *árbol de descomposición* de un grafo $G = (V, E)$ es un árbol $\mathcal{T}$ junto con una colección de subconjuntos T_x (llamados *bolsas*) de V etiquetadas por los vértices x de $\mathcal{T}$ tal que $\cup_{x\in\mathcal{T}} T_x = V$ y (1) y (2) se cumplen:

(1) Por cada arista uv de G, hay algún x tal que $\{u, v\} \subseteq T_x$.

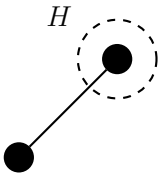
(2) (Propiedad de interpolación) Si y es un vértice en el único camino en $\mathcal{T}$ de x a z entonces $T_x \cap T_z \subseteq T_y$.

(b) El *ancho* de un árbol de descomposición es el máximo valor de $|T_x| - 1$ tomado sobre todos los vértices x del arbol $\mathcal{T}$ de la descomposición.

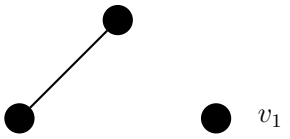
Según esta definición un grafo puede tener más de un árbol de descomposición, sin embargo como se verá en el próximo teorema, para determinar la anchura arbórea de un grafo, basta con determinar el ancho de uno de sus árboles de descomposición. También esta definición puede generalizarse fácilmente a multigrafos, hipergrafos, matroides y similares. Si el árbol de descomposición de un grafo es un



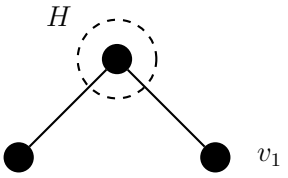
(A) $G = K_2$ es un 1-árbol



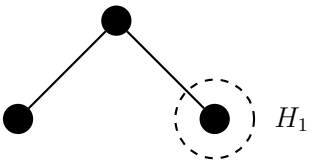
(B) H es un subgrafo de G isomorfo a K_1



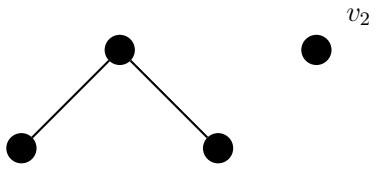
(C) Primero se agrega un nuevo vértice v_1 a G



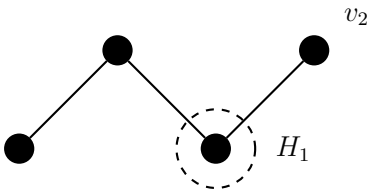
(D) Luego se agregan aristas (en este caso una) de v_1 con H para crear una copia de K_2, el grafo resultante G_1 es un 1-árbol.



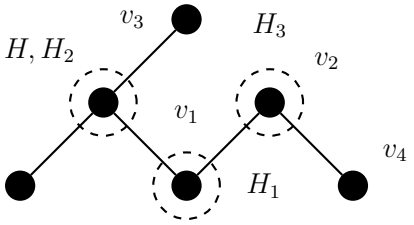
(E) Se elige un subgrafo H_1 de G_1 isomorfo a K_1



(F) Se agrega un nuevo vértice v_2 a G_1



(G) Se añaden aristas entre los vértices de H_1 y v_2 para crear una copia de K_2, el grafo resultante G_2 es un 1-árbol.



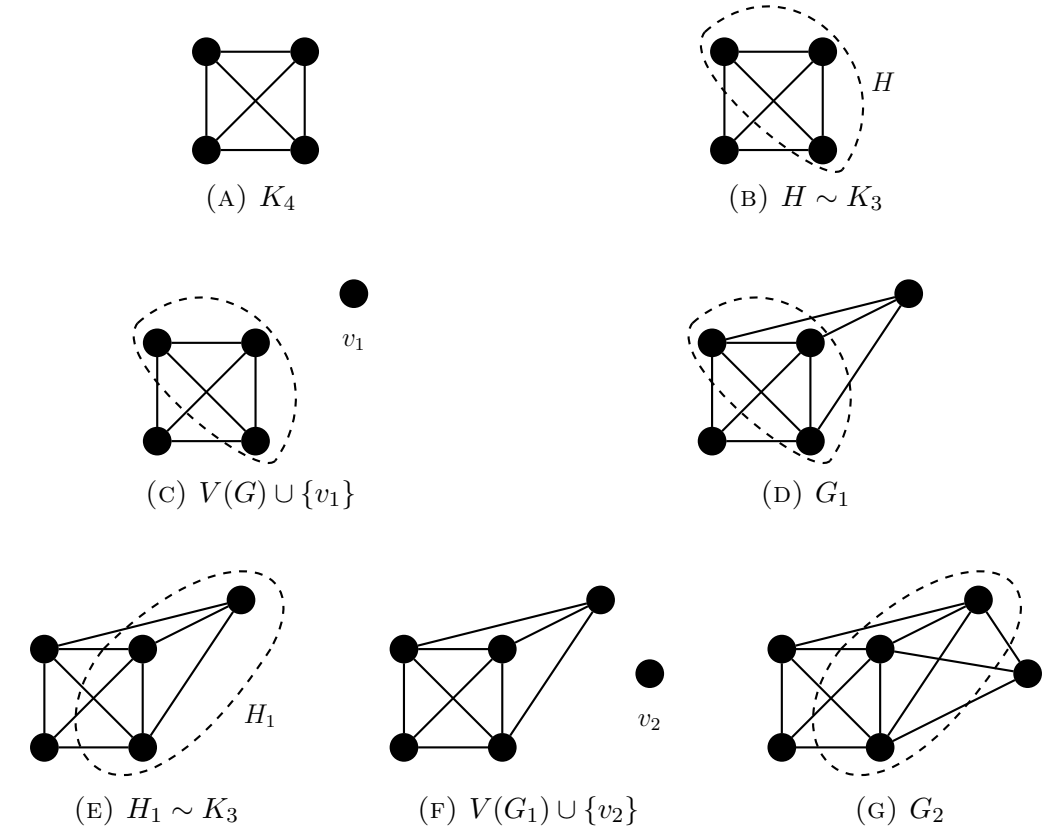
(H) Grafo resultante de aplicar la definición X.X eligiendo H, H_1, H_2 y H_3 isomorfos a K_1 y agregando nuevos vértices v_1, v_2, v_3 y v_4 a los grafos resultantes.

FIGURA 3.4: Construcción basada en la definición 3.1

camino decimos que el árbol de descomposición es un *camino de descomposición* y el término anchura arbórea es sustituido por *ancho camino*.

Ilustremos primero la definición mediante algunos ejemplos simples donde se exhiben los árboles de descomposición, posteriormente en el capítulo 4 se mostrará un algoritmo para construirlos.

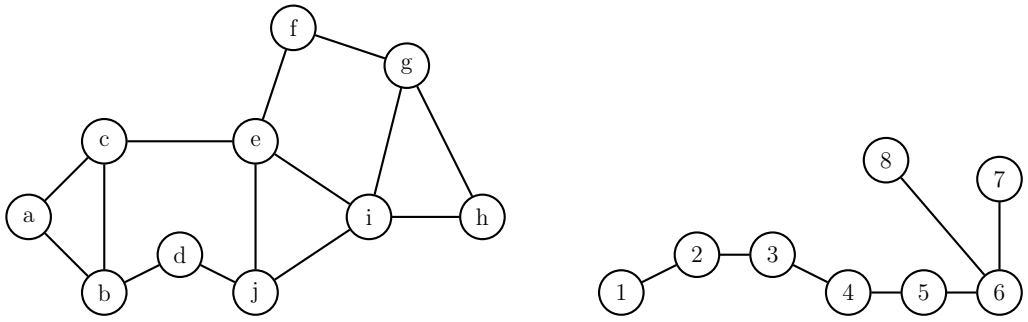
Ejemplo 3.4. Dado el grafo G mostrado por la figura 3.6(A) y si elegimos: $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $T_1 = \{a, b, c\}$, $T_2 = \{b, c, d\}$, $T_3 = \{c, d, e\}$, $T_4 = \{d, e, j\}$,



(A) K_4 (B) $H \sim K_3$ (C) $V(G) \cup \{v_1\}$ (D) G_1 (E) $H_1 \sim K_3$ (F) $V(G_1) \cup \{v_2\}$ (G) G_2

FIGURA 3.5: Otra construcción basada en la definición 3.1

$T_5 = \{e, i, j\}$, $T_6 = \{e, g, i\}$, $T_7 = \{g, h, i\}$, $T_8 = \{g, f, g\}$.

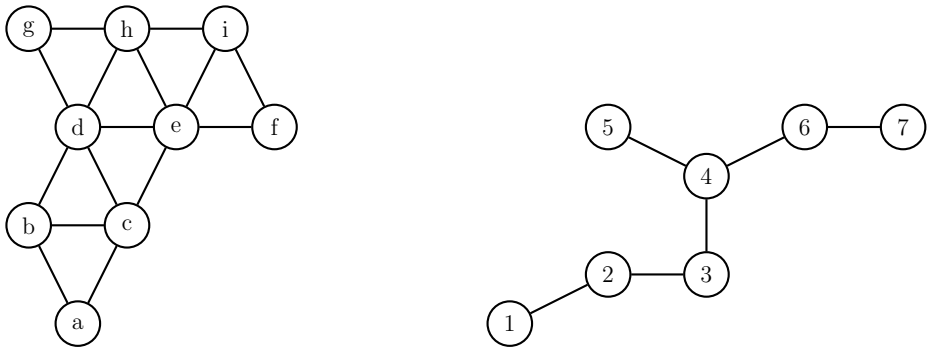


(A) (B)

FIGURA 3.6: Árbol decomposición del ejemplo 3.4.

Si $\mathcal{T}$ es el árbol que se muestra en la figura 3.6(B), entonces $\langle (T_x | x \in X), \mathcal{T} \rangle$ es un árbol de descomposición de G.

Ejemplo 3.5. Sea G el grafo mostrado por la figura 3.7(A) y hagamos: $X = \{1, 2, 3, 4, 5, 6, 7\}$, $T_1 = \{a, b, c\}$, $T_2 = \{b, c, d\}$, $T_3 = \{c, d, e\}$, $T_4 = \{d, e, h\}$, $T_5 = \{g, h, d\}$, $T_6 = \{h, i, e\}$, $T_7 = \{e, i, f\}$. El árbol $\mathcal{T}$ que se muestra en la figura 3.7(B) con las bolsas $\{T_x | x \in X\}$ es un árbol de descomposición de G.



(A) (B)

FIGURA 3.7: Árbol decomposición del ejemplo 3.5.

El siguiente lema establece que la colección de todas las etiquetas de las bolsas que contienen un nodo fijo de un grafo forman un subárbol de su árbol de descomposición y que todo clique del grafo debe estar contenido en una bolsa.

Lema 3.1. *Sea* $\{T_x : x \in \mathcal{T}\}$ *un árbol de descomposición de* G.
i) Sea $x \in V(G)$, *entonces la colección* $\{y \in \mathcal{T} : x \in T_y\}$ *es un subárbol de* $\mathcal{T}$.
ii) Si C *es un clique de* G, *entonces existe* $x \in \mathcal{T}$ *tal que* $C \subseteq T_x$.

Prueba (i). Dado $x \in V(G)$, veamos que la colección $\mathcal{C} = \{y \in \mathcal{T} : x \in T_y\}$ es un subgrafo conectado de $\mathcal{T}$. Para esto sean y' y y'' elementos de la colección $\mathcal{T}$ y $z \in \mathcal{T}$ tal que z está en el único camino en $\mathcal{T}$ que une a y' con y''. Entonces de la definición 3.3 tenemos $T_{y'} \cap T_{y''} \subseteq T_z$ y como $x \in T_{y'} \cap T_{y''}$, entonces $x \in T_z$ y así $z \in \mathcal{C}$. □

Prueba (ii). Sea C un clique en G y $v \in V(C)$, de la definición 3.3, existe $x \in \mathcal{T}$ tal que $v \in T_x$. Veamos que $C \subseteq T_x$, sea $w \in V(C)$ tal que $w \neq v$, entonces $vw \in E(C)$ y de la definición 3.3, $\{v, w\} \subseteq T_x$. □

El siguiente teorema establece la equivalencia entre la definición 3.2 y la definición 3.3 (b).

Teorema 3.2 (Arnborg, Gavril [Gav74], Rose, Tarjan y Lueker [RTL76])**.** *Un grafo tiene anchura arbórea* k *sii* G *tiene un árbol de descomposición de ancho* k.

Prueba$(\Leftarrow)$. Sea G un grafo con un árbol de descomposición $\{T_x : x \in \mathcal{T}\}$ de ancho k, entonces toda bolsa T_x no puede tener más de $k+1$ elementos. Veamos que k es el mínimo entero tal que G es un subgrafo de un $k-$árbol. Tenemos

que demostrar que G es un subgrafo de un $k-$árbol y que si $k > k'$ entonces G no puede ser subgrafo de ningún $k'-$árbol. Para esto, basta demostrar que G es un subgrafo de un $k-$árbol $K(G)$ en el cual todo subconjunto de una bolsa T_x (de $K(G)$), con no más de k elementos, es parte de un $k-$clique. Con lo anterior garantizamos que G no puede contener un k''-clique con $k'' > k$ y un grafo con un k-clique no puede ser subgrafo de un $k'-$árbol con $k' < k$.

Usamos inducción sobre los árboles $\mathcal{T}$. Para el caso base, el resultado se sigue trivialmente ya que si $\mathcal{T}$ consiste de una sola bolsa T_x de tamaño $k+1$, por lo que G es un subgrafo del $k-$árbol $K(G) = K_{k+1}$.

Ahora sea x una hoja de $\mathcal{T}$. Sea $\mathcal{T}'$ el resultado de eliminar x de $\mathcal{T}$, y sea G' un grafo correspondiente a $\mathcal{T}'$. Por hipótesis, G' es un subgrafo de un $k-$árbol $K(G')$ con las propiedades deseadas. Sea y el nodo adyacente a x en $\mathcal{T}$. Sea $\mathcal{T}_x \cap T_y = T'_x$ y $T_x - T'_x = T''_x$. Si $\mathcal{T}'' = \phi$, entonces

$$T_x \subseteq T'_x = T_x \cap T_y \subseteq T_y \rightarrow T_x \subseteq T_y \rightarrow \ G = G'$$

Suponemos entonces que $T''_x \neq \phi$. Por la propiedad de interpolación de la definición 3.3 para toda $z \neq x$, $T''_x \cap T_z = \phi$. En efecto, si $z = y$ entonces

$$T''_x \cap T_y = (T_x - T'_x) \cap T_y = T_x \cap T_y \cap T'^c_x = T_x \cap T_y \cap (T_x \cap T_y)^c = \phi,$$

si $y \neq z \neq x$ entonces por la propiedad 3.3 (2), $T_z \cap T_x \subseteq T_y$ entonces

$$T''_x \cap T_z = (T_x - T'_x) \cap T_z = T_x \cap T_z \cap T'^c_x = T_x \cap T_z \cap (T^c_y \cup T^c_x) =$$

$$= T_x \cap T_z \cap T^c_y \subseteq T_y \cap T^c_y = \phi$$

Esto es T''_x no comparte elementos con ninguna otra bolsa T_z para $z \neq x$ y T'_x es un subconjunto propio de T_x. Como el ancho es k, T'_x es un subconjunto de a lo más k elementos de T_y. Por hipótesis T'_x es un subconjunto de parte de un $k-$clique C de $K(G')$. Ahora la idea es adicionar a $K(G')$ el conjunto T''_x de nodo en nodo, uno a la vez mediante el algoritmo siguiente.

1. Inicialmente, sea $K = K(G')$. Mientras $T''_x \neq \phi$ repetimos los pasos 2-5:

2. Se elije un vértice v de T''_x.
3. Se agrega v a K y también se agregan todas las aristas entre los vértices de C y v.
4. Sea $T''_x \leftarrow T''_x - \{v\}$.
5. Si hay algún vértice $c \in C - T_x$, sea $C \leftarrow [C - \{c\}] \cup \{v\}$.

Por inducción en el algoritmo, podemos ver que después de cada iteración , C es un $k-$clique de K. Además, cada paso emula la definición de un $k-$árbol, y entonces como $K(G')$ es un $k-$árbol, también lo es K_{fin} que es el resultado del algoritmo anterior aplicado a $K(G')$. Por inducción en el algoritmo de arriba, se puede ver que todo T''_x es agregado junto con todas las posibles aristas que pueden existir entre los vértices de $T_x = T'_x \cup T''_x$. Entonces, K_{fin} contiene G y tiene las propiedades deseadas.

Prueba($\Rightarrow$) Supongamos que G es un $k-$árbol parcial. Dado que es suficiente considerar $k-$árboles, vamos a suponer que G es un $k-$árbol. Probemos por inducción sobre $|V(G)|$ que G tiene un árbol de descomposición de ancho k. Para esto, usamos la hipótesis fuerte de que si G' es un $k-$árbol con menos que $|V(G)|$ vértices, entonces G' tiene un árbol de descomposición $\{T_x : x \in \mathcal{T}\}$ donde $|T_x| \leqslant k+1$ para todo $x \in \mathcal{T}$, y cada $k-$clique de G' ocurre en algún T_x. El resultado es claro si $G = K_{k+1}$. Entonces suponemos que G tiene más de $k+1$ nodos y sea v el último nodo agregado en la formación de G (esto es, v es un nodo *simplicial*). Sea $N(v)$ los vecinos de v en G. Sea G' el resultado de remover v y las k aristas vu con $u \in N(v)$. Entonces, podemos aplicar la hipótesis de inducción a G' y obtener el árbol de descomposición $\{T'_x : x \in \mathcal{T}'\}$ de G' con las propiedades deseadas. Ahora, $N(v)$ es un $k-$clique y entonces, por hipótesis, hay algún $y \in T'$ con $N(v) \subseteq T'_y$. Se crea un nuevo árbol $\mathcal{T}$ de $\mathcal{T}'$ para adjuntar una nueva hoja $l = l(v)$ a y. Sea $T_l = N(v) \cup \{v\}$. Por hipótesis, $\mathcal{T}$ será el árbol de descomposición relevante de G.

La siguiente definición también es útil para la discusión de la próxima sección.

Definición 3.7. (*Anchura arbórea acotada*). Decimos que la clase de grafos $\mathcal{C}$ tiene *anchura arbórea acotada* si existe un k tal que para todo $G \in \mathcal{C}$, G tiene anchura arbórea $\leq k$.

Una nota importante dada en [rMRF13] es que el árbol de descomposición establece métodos teórico automáticos para demostrar la tratabilidad de una amplia

clase de propiedades para grafos con anchura arbórea acotada. Los métodos de anchura arbórea pueden ser utilizados para reemplazar los métodos combinatorios ad hoc previamente empleados por muchos autores y que por lo general conllevan un análisis complicado de casos. Un ejemplo de este fenómeno es el teorema de Courcelle.

Teorema 3.3 (Courcelle [Cou92], [ALS88, ALS91])**.** *Cada problema expresable en un lenguaje monádico de segundo orden puede ser resuelto en tiempo lineal para una clase arbitraria de grafos con universalmente anchura arbórea acotada.*

Este teorema es un meta-teorema basado en lógica para establecer junto con el teorema de Bodlaender que varias propiedades teóricas de grafos son decidibles en tiempo lineal PFT, cuando el parámetro de entrada es la anchura arbórea del grafo.

Una prueba de este teorema en una versión más fuerte y de la del teorema de Bodlaender pueden ser encontradas en [RGD13]. Todos los argumentos empleados en este tipo de meta-teoremas se basan en el hecho de que el problema es parámetro fijo tratable. En este caso, el problema denominado ANCHURA ARBÓREA es de tal tipo.

ANCHURA ARBÓREA
Entrada: Un grafo G.
Parámetro: Un entero positivo k.
Pregunta: ¿G tiene anchura arbórea k?

Teorema 3.4 (Bodlaender [Bod96])**.** *El problema ANCHURA ARBÓREA es fuertemente PFT. De hecho, por cada k, hay un algoritmo en tiempo lineal que reconoce grafos con anchura arbórea k. Además, dado un grafo G con anchura arbórea k, el algoritmo producirá un árbol de descomposición de G de ancho k en tiempo lineal.*

Para demostrar la primera parte de este teorema, específicamente se demuestra que existe un algoritmo con parámetro fijo tratable de $O(|G|)$ el cual tiene parámetros k y p con $k < p$, y el cual dado un $p-$árbol de descomposición de G determina si G tiene anchura arbórea k.

En el capítulo 4 presentamos de manera detallada un algoritmo simple con las caractéristicas del teorema de Bodlaender.

Capítulo 4

Algoritmos

4.1 Introducción

La idea principal del algoritmo de descomposición es reducir el grafo removiendo uno a uno sus vértices, y al mismo tiempo poner los vértices removidos en una pila (stack), y después el árbol puede ser construido con la información de la pila. Primero, un vértice v con un grado en específico es identificado. Luego comprobamos que todos sus vecinos forman un clique, si no, agregamos las aristas que faltan para construir un clique. Entonces, v junto con sus vecinos se colocan en la pila, después se elimina v y sus aristas del grafo. El algoritmo presentado aquí está basado en el artículo de Fang Wei [Wei10].

4.2 Algoritmo para obtener el árbol de descomposición

Algorithm 4.1 *tree_decomp*(G)

Input: $G = (V, E)$ is a directed graph.
Output: return the tree decomposition T_G
1: Transform G into an undirected graph UG;
2: *graph_reduction*(UG); {output the vertex stack S}
3: *tree_construction*(S, G); {output the tree decomposition }

El programa comienza removiendo los vértices aislados y vértices con grado 1. Entonces el proceso de reducción procede con los vértices con grado $2, 3, \ldots$. Denotamos tal proceso de remover todos los vértices con grado x como *degree-x reduction*.

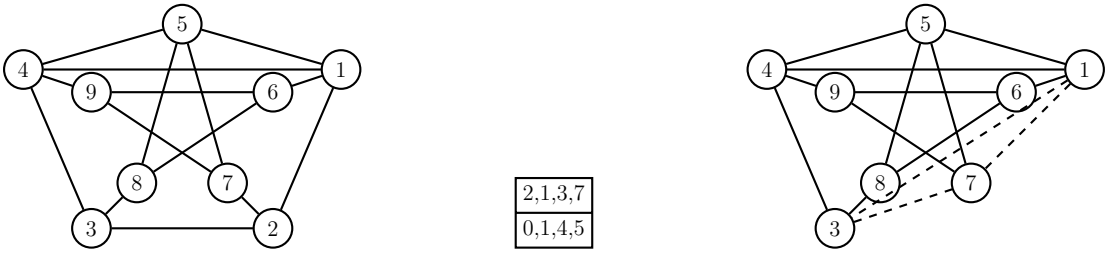
Ejemplo.

Considerando el grafo no dirigido de la figura 4.1(A) comenzamos eligiendo un vértice con grado tres, en este caso el vértice 0 (osea *degree-3 reduction*). Metemos en el stack al vértice elegido junto con sus vecinos que son los vértices 1,4 y 5. Después formamos un clique entre sus vecinos y borramos el vértice elegido del grafo.

(A) G (B) G (C) G_2

FIGURA 4.1: Descomposición de árbol (paso 1)

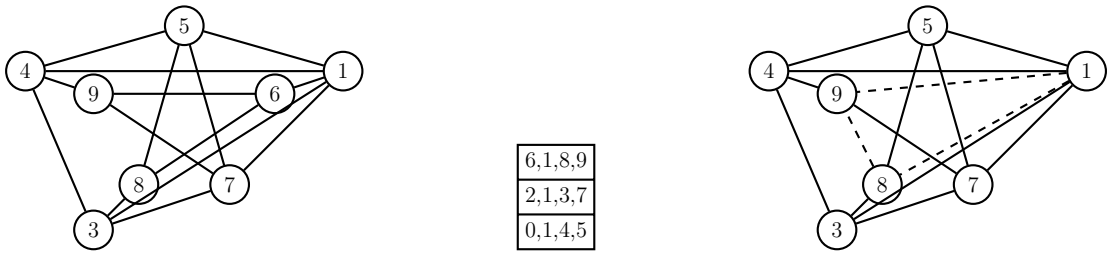
Ahora tomamos el vértice 2 que tiene grado tres y lo metemos junto con sus vecinos que son los vértices 1, 3 y 7 en el stack. Formamos un clique entre los vecinos del vértice 2 y borramos este vértice del grafo como se mustra en la figura 4.2.



(A) Grafo resultante (B) Pila (C) $degree-3$ *reduction*

FIGURA 4.2: Descomposición de árbol (paso 2)

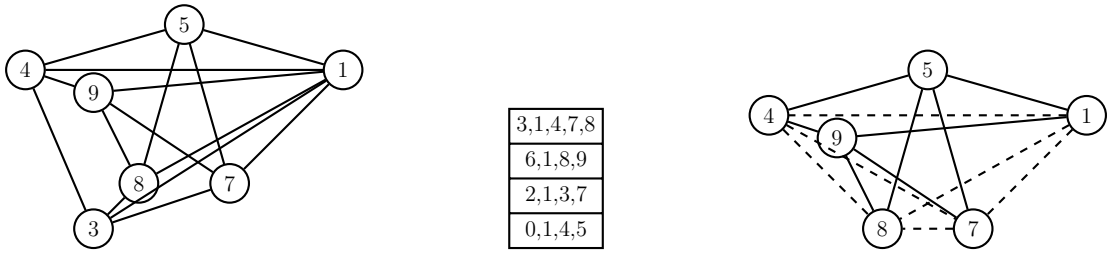
Ahora tomamos el vértice 6 que es de grado tres y lo metemos junto con sus vecinos al stack. Formamos un clique entre sus vecinos y se elimina del grafo el vértice 6 como se mustra en la figura 4.3.



(A) Grafo resultante (B) Pila (C) *degree* − 3 *reduction*

FIGURA 4.3: Descomposición de árbol (paso 3)

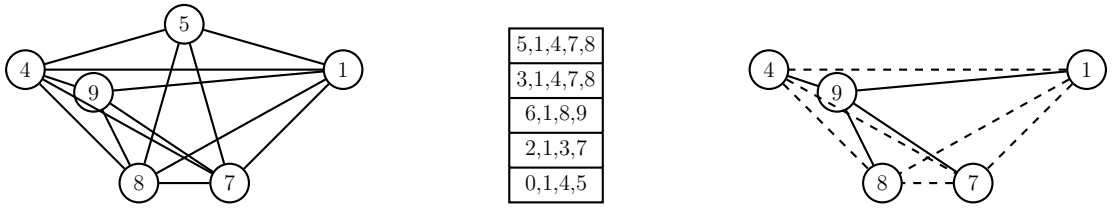
Ahora, como ya no hay vértices con grado tres elegimos un vértice con grado cuatro, en este caso el vértice 3. Metemos al vértice junto con sus vecinos que son los vértices 1, 4, 7 y 8 al stack. Después, formamos un clique entre los vecinos del vértice y borramos el vértice 4 del grafo como se mustra en la figura 4.4.



(A) Grafo resultante (B) Pila (C) *degree* − 3 *reduction*

FIGURA 4.4: Descomposición de árbol (paso 4)

Ahora tomamos el vértice 5 que tiene grado cuatro y lo metemos junto con sus vecinos que son los vértices 1, 4, 7 y 8 en el stack. Entre sus vecinos formamos un clique y después eliminamos el vértice 5 del grafo como se mustra en la figura 4.5.

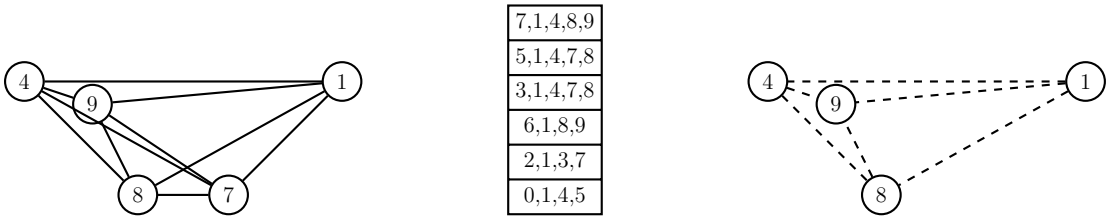


(A) Grafo resultante (B) Pila (C) *degree-3 reduction*

FIGURA 4.5: Descomposición de árbol (paso 5)

Ahora tomamos el vértice 7 que tiene grado cuatro, y lo metemos a la pila junto con sus vecinos que en este caso son los vértices 1, 4, 8 y 9. Después, formamos

un clique entre sus vecinos y eliminamos el vértice 7 del grafo como se mustra en la figura 4.6.



(A) Grafo resultante (B) Pila (C) *degree-3 reduction*

FIGURA 4.6: Descomposición de árbol (paso 6)

Ahora como sólo hay vértices con grado menor a cuatro (en este caso solo quedan vértices con grado tres) termina el proceso.

El procedimiento *graph_reduction* terminará cuando una de las siguientes condiciones se cumpla.

(a) El grafo se ha reducido a un conjunto vacío. Por ejemplo, si el grafo sólo contiene ciclos simples, será reducido a un conjunto vacío después de *degree-2 reductions*. Usualmente es el caso para grafos esparcidos extremadamente.

(b) Para grafos que no son esparcidos, uno tiene que definir una cota superior l para la reducción, para que el programa se detenga después de *degree-l reduction*. Observe que como el grado incrementa, la efectividad de la reducción decrementará, porque en el peor de los casos, necesitamos agregar $x(x-1)/2$ aristas así como se van eliminando x aristas.

Después del proceso de reducción, el árbol de descomposición puede ser construido de la siguiente manera:

(a) Primero recolectamos todos los vértices que no fueron removidos por el proceso de reducción y este conjunto se asigna como la bolsa de la raíz del árbol. El tamaño de la raíz depende de la estructura del grafo (i.e. cuántos vértices se dejaron después de la reducción).

(b) El resto del árbol es generado desde la información almacenada en la pila S. Sea X_c el conjunto de los vértices $\{v, v_1, \ldots, v_x\}$ el cual es sacado del top de S. Aquí, v es el vértice removido y $\{v_1, \ldots, v_x\}$ son los vecinos de v los

Algorithm 4.2 *graph_reduction*(UG)

Input: UG is the undirected graph of G, l is the upper bound for the reduction.
Output: stack S and the reduced graph UG'

```
1: initialize stack S;
2: for i = 1 to l do
3:     remove_upto(i);
4: end for
5: return S, UG;
6: procedure remove_upto(x)
7: while TRUE do
8:     if there exists a vertex v with degree less than x then
9:         {v_1, ..., v_x} = neighbors of v;
10:        build a clique for {v_1, ..., v_x}
11:        push v, v_1, ..., v_x into S;
12:        delete v and all its edges from UG;
13:    else
14:        break;
15:    end if
16: end while
```

cuales forman un clique. Después de que la bolsa padre X_p la cual contiene $\{v_1, \ldots, v_x\}$ es colocada en el árbol, X_c es agregada como una bolsa hijo de X_p. Este proceso continúa hasta que S está vacía. El algoritmo 3 ilustra el proceso.

Algorithm 4.3 *tree_construction*(S, G, UG')

Input: S is the stack storing the removed vertices and their neighbors, G is the directed graph, UG' is the reduced graph of T_G
Output: return tree decomposition T_G

```
1: construct the root of T_G containing all the vertices of UG';
2: while S is not empty do
3:     pop up a bag X_c = {v, v_1, ..., v_x} from S;
4:     find the bag X_p containing {v_1, ..., v_x};
5:     add X_c into T as the child node of X_p;
6: end while
7: generate transitive closure in all bags;
```

EL último paso del proceso del árbol de descomposición es generar la cerradura transitiva de cada bolsa.

La correctez del algoritmo del árbol de descomposición puede ser mostrada por la inducción de los pasos de reducción. Observe que durante el proceso de reducción, las aristas son agregadas al grafo original. Por lo consiguiente, el árbol

de descomposición que obtenemos de acuerdo al algoritmo está basado en un grafo que contiene aristas extras. Sin embargo, esto no afecta la correctez de la prueba porque dados $G = (V, E)$ y $G' = (V, E')$ dos grafos donde $E \subseteq E'$, entonces de la definición 3.3 se sigue que cualquier árbol de descomposición de G' es un árbol de descomposición de G.

Una vez que tenemos el algoritmo que construye el árbol de descomposición de un grafo el teorema de Arnbrog nos garatiza que podemos computar su anchura arbórea mediate un simple algoritmo que tiene como entrada las bolsas de un árbol de descomposición del grafo. El algoritmo consiste prácticamente de una sola línea como se muestra a continuación.

Algorithm 4.4 *tree_width*

Input: $tree\_construction(S, G, UG')$
Output: return integer $width$
1: $width = maximum\{|T_x| : x \in \mathcal{T}\} - 1$;

Capítulo 5

Conclusiones

5.1 Discusión

Como se ha mencionado en capítulos previos la anchura arbórea de un grafo es una medida que combinada con la posibilidad de expresar una propiedad gráfica en lenguaje de la lógica monádica de segundo orden nos permite saber de la existencia de algoritmos parámetro fijo tratables (Teorema de Courcelle). Existen otros resultados como los que se muestran en el libro de Marek [MC15] que muestran la existencia de un algoritmo para un grafo G de n vértices y un entero positivo k, que corre en tiempo $k^{O(k^3)}n$ y tiene como posibles salidas la construcción del árbol de descomposición de G con ancho a lo más k, o la conclusión de que la anchura arbórea de G es estrictamente mayor que k. Otro resultado que también se presenta en REF es el de la existencia de un algoritmo para un grafo G de n vértices y un entero positivo k, que corre en tiempo $O(8^k k^2 n^2)$ y tiene como posibles salidas la construcción del árbol de descomposición de G con ancho a lo más $4k + 4$, o concluye que la anchura arbórea de G es estrictamente mayor que k.

Cabe mencionar que la anchura arbórea no es la única medida que nos conduce a este tipo de resultados, esto es, existen otras medidas como: *ancho clique*, *ancho camino*, *ancho banda* y *ancho cincho* entre otros, que nos conducen a resultados similares en REF podemos encontrar relaciones entre algunas de estas medidas.

5.2 Conclusiones y Trabajo Futuro

De las tres variantes de tratabilidad de parámetro fijo se ha presentado un algoritmo simple para calcular la anchura arbórea de un grafo que es fuertemente PFT y de orden $O(|G|)$. Como se ha visto en el capítulo 4, el peso (mayor tiempo de ejecución) del algoritmo se encuentra en la construcción del árbol de descomposición del grafo. A través del árbol de descomposición de un grafo se pueden establecer métodos teórico automáticos para demostrar la tratabilidad de una cantidad considerable de propiedades para grafos de anchura arbórea acotada, un ejemplo de esto es el que se da en el teorema de Courcelle presentado en el capítulo 3, el cual hace evidente,para la obtención de la tratabilidad, la importancia de la combinación de la acotación de la anchura arbórea con la expresibidad en el lenguaje de la lógica monádica de segundo orden.

Para un trabajo futuro, se presentan varias opciones que involucran tanto estudios teóricos, como aplicaciones relacionadas con los temas presentados en esta tesis. En lo que corresponde a la parte teórica se pueden estudiar algoritmos y resultados análogos al teorema de Courcelle para los casos de ancho clique, ancho banda, ancho camino y ancho cincho, entre otros. Por la parte lógica se puede estudiar la expresibidad en $L-$estructuras y estudiar también la combinación con la ancho acotación establecida.

En la línea de las aplicaciones se pueden abordar áreas como Grafo de datos (data Graph), GIS, bases de datos XML, bioinformática, redes sociales y ontologías.

De las opciones mencionadas la que resulta de nuestro interés para un estudio futuro es la que se refiere a la combinación de las $L-$ estructuras con la ancho acotación.

Appendix A

Código Algoritmo tree_width

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.io.IOException;

public class LoadMatrix{
  public int[][] loadMatrix()throws FileNotFoundException, IOException{
    String pathFile = "aMEc_Pk4.txt";
    FileReader fr = new FileReader(pathFile);
    BufferedReader br = new BufferedReader(fr);

    ArrayList<String> aux = new ArrayList<String>();
    String line = br.readLine();
    while(line != null){
      aux.add(line);
      line = br.readLine();
    }

    int row = aux.size();
    int col = aux.get(0).split(" ").length;

    int[][] matrix = new int[row][col];
    int i = 0;
```

```
    for(String str : aux){
      String[] auxS = str.split(" ");
      int j = 0;
      for(String str2 : auxS){
        matrix [i][j] = new Integer(str2).intValue();
        j++;
      }
      i++;
    }
    return matrix;
  }
}
```

```
public class Matrix{
  public void printMatrix(int[][] matrix){
    int size = matrix.length;
    for(int i = 0; i < size; i++){
      for(int j = 0; j < size; j++){
        System.out.print(matrix[i][j]);
      }
      System.out.println();
    }
  }
}
```

```
import java.util.List;
import java.util.ArrayList;

public class Neighbor{
  TD td = new TD();

  public void getNeighbor(int index, int[][] matrix, List<Integer>
    array){
    int x = index;
    array.add(x);
    System.out.println(td.bag.get(0));
    int size = matrix.length;
    for(int i = 0; i < size; i++){
```

```
      if(matrix[index][i] != 0){
        matrix[index][i] = 0;
        matrix[i][index] = 0;
      }
    }
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class StackBag{
  TD td = new TD();
  public void addBag(){
    int size = td.bag.size();
    for(int i = 0; i < size; i++){
      int value = td.bag.get(i);
      td.stackBag.get(td.index).add(value);
    }
    td.index ++;
    td.bag.clear();
  }

  public void printStackBag(){
    System.out.println("action for printStackBag...");
    for(int i = 0; i < td.stackBag.size(); i++){
      for(int j = 0; j < td.stackBag.get(i).size(); j++){

   System.out.println("StackBag["+i+"]["+j+"]="+td.stackBag.get(i).get(j));
      }
    }
    System.out.println("End for printStackBag");
  }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.io.FileNotFoundException;
```

```
import java.io.IOException;

public class TD{
  List<List<Integer>> stackBag = new ArrayList<List<Integer>>();
  List<List<Integer>> buildTree = new ArrayList<List<Integer>>();
  List<Integer> bag = new ArrayList<Integer>();
  int index = 0;

  public void vectorZero(int[] vector){
    for(int i = 0; i < vector.length; i++){
      vector[i] = 0;
    }
  }

  public void printVector(int[] vector){
    for(int i = 0; i < vector.length; i++){
      System.out.println("Vecinos.Vector[" + i + "]=" + vector[i]);
    }
  }

  public int[] getDegree(int[] vector, int[][] matrix){
    vectorZero(vector);
    for(int i = 0; i < matrix.length; i++){
      for(int j = 0; j < matrix.length; j++){
        vector[i] += matrix[i][j];
      }
    }
    return vector;
  }

  public boolean existsDegree(int upperBound, int[] vector){
    boolean exists = false;
    for(int i = 0; i < vector.length; i++){
      if(vector[i] == upperBound){
        exists = true;
        //return exists;
      }
    }
    return exists;
```

```
}

public void createRow(){
  stackBag.add(new ArrayList<Integer>());
}

public void addBag(){
  createRow();
  for(int i = 0; i < bag.size(); i++){
    int x = bag.get(i);
    stackBag.get(index).add(x);
  }
  bag.clear();
  index ++;
}

public void printBags(){
  for(int i = 0; i < stackBag.size(); i++){
    for(int j = 0; j < stackBag.get(i).size(); j++){
      System.out.println("td.stackBag[" + i + "][" + j + "] = " +
 stackBag.get(i).get(j));
    }
    System.out.println("++++++++++++++++++++++");
  }
}

public void getNeighbor(int index, int[][] matrix){
  int x = index;
  bag.add(x);
  int size = matrix.length;
  for(int i = 0; i < size; i++){
    if(matrix[index][i] != 0){
      bag.add(i);
      matrix[index][i] = 0;
      matrix[i][index] = 0;
    }
  }
  addBag();
}
```

```
public void buildClique(int[][] matrix, List<List<Integer>> vecinos){
  for(int i = 1; i < vecinos.get(index-1).size(); i++){
    for(int j = 1; j < vecinos.get(index-1).size(); j++){
      if(vecinos.get(index-1).get(i) != vecinos.get(index-1).get(j)){

 matrix[vecinos.get(index-1).get(i)][vecinos.get(index-1).get(j)] = 1;
      }
    }
  }
}

//
//

public void createRow2(){
  buildTree.add(new ArrayList<Integer>());
}

public void addBag2(){
  createRow2();
  for(int i = 0; i < bag.size(); i++){
    int x = bag.get(i);
    buildTree.get(index).add(x);
  }
  bag.clear();
  index ++;
}

public void printBags2(){
  for(int i = 0; i < buildTree.size(); i++){
    for(int j = 0; j < buildTree.get(i).size(); j++){
      System.out.println("td.stackBag[" + i + "][" + j + "] = " +
 buildTree.get(i).get(j));
    }
    System.out.println("++++++++++++++++++++++");
    }
}
```

```
//
//

public static void main(String[] args)throws FileNotFoundException,
  IOException{
   TD td = new TD();
   LoadMatrix lm = new LoadMatrix();
   Matrix m = new Matrix();
   Neighbor n = new Neighbor();
   int[][] myMatrix;
   int[] degree;

   System.out.println("Comenzando... again :)");
   myMatrix = lm.loadMatrix();
   degree = new int[myMatrix.length];

   int indice = 1;
   td.getDegree(degree, myMatrix);
   System.out.println(td.existsDegree(2, degree));
   System.out.println(degree[0]);

   do{
     do{
       for(int i = 0; i < myMatrix.length; i++){
         if(td.existsDegree(indice, degree) && degree[i] == indice){
           td.getNeighbor(i, myMatrix);
           m.printMatrix(myMatrix);
           td.buildClique(myMatrix, td.stackBag);
           td.getDegree(degree, myMatrix);
           td.printVector(degree);
           System.out.println("+++++++++++++++++");
         }
       }
       //obj.getDegree(myMatrix, degree);
     }while(td.existsDegree(indice,degree));
     indice++;
   }while(indice <= 10);

   System.out.println("action for printStackBag...");
```

```
for(int i = td.stackBag.size()-1; i >= 0; i--){
  System.out.println("[" + i + "]");
  for(int j = 0; j < td.stackBag.get(i).size(); j++){
    //System.out.println("StackBag["+i+"]["+j+"]="+
td.stackBag.get(i).get(j));
    System.out.print(td.stackBag.get(i).get(j) + " ");
  }
  System.out.println();
}
System.out.println("End for printStackBag");

//
//
int x = td.stackBag.size();
System.out.println(x);
System.out.println(td.buildTree.isEmpty());
int[][] resultMatrix = new int[x][x];

for(int i = 0; i < resultMatrix.length; i++){
  for(int j= 0; j < resultMatrix.length; j++){
    resultMatrix[i][j] = 0;
  }
}



td.index = 0;
int auxiliar = 0;
int contador = 0;
boolean isEqual = false;
for(int i = td.stackBag.size() - 1; i >= 0; i--){
  for(int j = 0; j < td.stackBag.get(i).size(); j++){
    auxiliar = td.stackBag.get(i).get(j);
    td.bag.add(auxiliar);
  }
  if(td.buildTree.isEmpty()){
    td.addBag2();
    System.out.println("Estaba vacia...");
  }
  else{
```

```
        isEqual = false;
        td.addBag2();
        System.out.println("Se agrego bolsa: " + (td.index - 1));
        for(int ii = 0; ii < td.buildTree.size()-1; ii++){
          if(isEqual == false){
            contador = 0;
            for(int k = 1; k < td.buildTree.get(td.index - 1).size();
k++){
              for(int j = 0; j < td.buildTree.get(ii).size(); j++){
                System.out.println("[" + ii + "][" + j + "]");
                if(td.buildTree.get(td.index - 1).get(k) ==
td.buildTree.get(ii).get(j)){
                  System.out.println("Fue igual...");
                  contador ++;
                  System.out.println("Contador: "+contador+ " ==
tamanio: " + (td.buildTree.get(td.index -1).size()-1));
                  if(contador == td.buildTree.get(td.index
-1).size()-1){
                    System.out.println("isEqual sera true...");
                    isEqual = true;
                    resultMatrix[td.index - 1][ii] = 1;
                    resultMatrix[ii][td.index - 1] = 1;
                  }
                  break;
                }
                else{
                  System.out.println("No");
                }
              }
            }
          }
          else{
            break;
          }
        }
      }
    }
    td.printBags2();
    m.printMatrix(resultMatrix);
```

```
    int width = 0;
    for(int inicio=0; inicio < td.buildTree.size(); inicio++){
      if(td.buildTree.get(inicio).size() > width){
        width = td.buildTree.get(inicio).size();
      }
    }
    System.out.println("Treewidth = " + (width-1));
    //
    //
  }
}
```

Bibliografía

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987. v, 34

[ADN05] Jochen Alber, Frederic Dorn, and Rolf Niedermeier. Experimental evaluation of a tree decomposition-based algorithm for vertex cover on planar graphs. *Discrete Applied Mathematics*, 145(2):219–231, 2005. 35

[ALS88] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Problems easy for tree-decomposable graphs extended abstract. In *Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 38–51. Springer Berlin Heidelberg, 1988. 42

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991. 42

[BC12] Joost Engelfriet Bruno Courcelle. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, 1st edition, 2012. v, 11

[BELL05] Andreas Brandstädt, Joost Engelfriet, Hoàng-Oanh Le, and Vadim V. Lozin. Clique-width for four-vertex forbidden subgraphs. In *Fundamentals of Computation Theory, 15th International Symposium, FCT 2005, Lübeck, Germany, August 17-20, 2005, Proceedings*, pages 185–196, 2005. v

[Bod93a] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC, pages 226–234, 1993. v, 34

[Bod93b] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993. v

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 6:1305–1317, 1996. 42

[Cou92] Bruno Courcelle. The monadic second-order logic of graphs iii: Tree-decompositions, minors and complexity issues. *RAIRO - Informatique Théorique et Applications*, 26(3):257–286, 1992. v, 42

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of Graph Grammars and Computing by Graph Transformation*, 1997. v, 11

[DF92] Rod Downey and Michael Fellows. Fixed-parameter tractability and completeness. *Congressus Numerantium*, 87:161–178, 1992. 29

[Die10] Reihard Diestel. *Graph Theory*. Springer, 4 edition, 2010. 1

[FL87] Michael R. Fellows and Michael A. Langston. Nonconstructive advances in polynomial-time complexity. *Information Processing Letters*, 26(3):157–162, 1987. 32

[FL88] Michael R. Fellows and Michael A. Langston. Nonconstructive tools for proving polynomial-time complexity. *Journal of the ACM*, 35:727–739, 1988. 32

[Gar79] David S. Garey, Michael R.; Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1 edition, 1979. 32

[Gav74] Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *JCTB*, 16:47–56, 1974. 39

[Kar72] Richard M. Karp. Reductibility combinatorial problems. *The IBM Research Symposia Series*, pages 85–103, 1972. 29

[KBvH01] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. Treewidth: Computational experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57, 2001. v, 34

[KvHK02] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving partial constraint satisfaction problems with tree decompositions. *Networks*, 40(3):170–180, 2002. 35

[LS88] Steffen Lilholt Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society series B*, 50(2):157–224, 1988. 35

[MC15] Fedor V.Fomin Marek C. *Parameterized Algorithms*. Springer International Publishing Switzerland, 1 edition, 2015. 49

[Meh84] Kurt Mehlhorn. Data structures and algorithms 2. In *Graph Algorithms and NP-Completeness*, volume 2 of *EATCS Monographs on Theoretical Computer Science*. Springer Berlin Heidelberg, 1984. 29

[Men15] Elliott Mendelson. *Introduction to Mathematical Logic*. CRC Press, 6 edition, 2015. 11

[Moh96] Bojan Mohar. Embedding graphs in an arbitrary surface in linear time. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, STOC '96, pages 392–397. ACM, 1996. 32

[MZ06] Anil Maheshwari and Norbert Zeh. I/o-efficient algorithms for graphs of bounded treewidth. *Algorithmica*, pages 1–57, 2006. v

[RGD13] Michael R. Fellows Rodney G. Downey. *Fundamentals of Parameterized Complexity*. Text in Computer Science. Springer, 2013. 42

[rMRF13] Rodney G. Downey r Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 1 edition, 2013. 41

[Ros12] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 7 edition, 2012. 1

[RS84] Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36:49 – 64, 1984. v

[RS85] Neil Robertson and Paul Seymour. Graph minors - a survey. *Surveys in Combinatorics*, 103:153–171, 1985. 34

[RS95] Neil Robertson and Paul Seymour. Graph minors. xiii. the disjoint paths problem. *J. Comb. Theory Ser. B*, 63(1):65–110, 1995. 32, 34

[RS04] Neil Robertson and Paul Seymour. Graph minors. xx. wagner's conjecture. *J. Comb. Theory Ser. B*, 92(2):325–357, 2004. 32, 34

[RTL76] Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976. 39

[SA91] D. Seese S. Arnborg, J. Lagergren. Easy problems for tree decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991. v, 23

[Tho98] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998. 35

[Tru94] Richard J. Trudeau. *Introduction to Graph Theory*. Dover Books on Mathematics. Dover Publications, 2 edition, 1994.

[Wei10] Fang Wei. Efficient graph reachability query answering using tree decomposition. In *Reachability Problems*, pages 183–197. Springer, 2010. v, 34, 43

[YAM03] Atsuko Yamaguchi, Kiyoko F. Aoki, and Hiroshi Mamitsuka. Graph complexity of chemical compounds in biological pathways. *Genome Informatics*, 14:376–377, 2003. 35