Universidad Nacional Autónoma de México

Facultad de Ciencias

HERRAMIENTAS PARA EL MANEJO COMPUTACIONAL DE GRÁFICAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

MAURICIO CARRASCO RUIZ

DIRECTOR DE TESIS:

CÉSAR HERNÁNDEZ CRUZ

CD. MX. 2023

1.Datos del alumno

Apellido paterno	Carrasco
Apellido materno	Ruiz
Nombre(s)	Mauricio
Teléfono	+52 56 11 03 04 21
Universidad	Universidad Nacional Autónoma de México
Facultad o escuela	Facultad de Ciencias
Carrera	Ciencias de la Computación
Número de cuenta	315191318

2. Datos del tutor

Grado	Dr.
Nombre(s)	César
Apellido paterno	Hernández
Apellido materno	Cruz

3. Datos del sinodal 1

Grado	Dr.
Nombre(s)	Canek
Apellido paterno	Peláez
Apellido materno	Valdés

4. Datos del sinodal 2

Grado	Dr.
Nombre(s)	José de Jesús
Apellido paterno	Galaviz
Apellido materno	Casas

5. Datos del sinodal 3

Grado	Dra.
Nombre(s)	Adriana
Apellido paterno	Ramírez
Apellido materno	Vigueras

6. Datos del sinodal 4

Grado	M. en C.
Nombre(s)	Fernando Esteban
Apellido paterno	Contreras
Apellido materno	Mendoza

7.Datos del trabajo escrito

Título	Herramientas para el manejo computacional de gráficas
Número de páginas	136 p.
Año	2021

Agradecimientos

Para empezar, me gustaría agradecer a la Universidad Nacional Autónoma de México (UNAM), en especial a la Facultad de Ciencias, donde me brindaron los espacios y recursos necesarios para completar y expandir mis estudios; al Dr. César Hernández Cruz por ofrecerme la oportunidad de trabajar en este tema, así como la paciencia y cuidado que tuvo al momento de revisar mi trabajo. Esta investigación fue realizada gracias al Programa de Apoyo a Proyectos de Investigación e Innovación Tecnológica (PAPIIT) de la UNAM IA104521. Agradezco a la DGAPA-UNAM por la beca recibida durante la realización de este trabajo.

Una mención especial al Dr. Ignacio Arroyo Fernández por darme la oportunidad de conocer y trabajar en el mundo de la investigación académica; a la Mtra. Angela Eugenia Villanueva Vilchis a quien le debo la pasión que tengo por las Ciencias de la Computación; y al Dr. Ludovic Stephan por el tiempo que dedicó a revisar la implementación de uno de los algoritmos presentados por parte de alguien que no tenía ninguna obligación de hacerlo.

De igual forma, quiero agradecer a las personas que me han acompañado durante todo este proceso. A mi madre y mis tías, quienes me han acompañado en las buenas, malas y feas, y han apoyado incondicionalmente con amor mis sueños y aspiraciones; a mi padre y mi abuela quienes siempre intentaron ver por mi bien; a aquellos a quienes considero mis hermanos aunque no tengamos ninún lazo de sangre, José, Miguel, y Álvaro; con quienes pude compartir mi amor por la ciencia, Diana, Kevin, Denisse, Bruce, y Alma; a mi *meilleur ami* y confidente Nathan; a quienes me acompañaron durante la carrera y pude encontrar una hermosa amistad, Alan, Emiliano, Jerónimo, Martha, Miguel, Mirén, y Sandra; a mi psicóloga, Claudia, quien me ayudó a ser más objetivo y asertivo en esta vida; a Gisselle, quien me ha escuchado, apoyado, y sostenido en los momentos difíciles, y con quien he podido compartir muchos de los momentos más bellos de esta vida; por último, pero no por eso menos importantes, a mis perritos, Romeo, Shadow, y mis niñas preciosas Zelda y Niebla, una de mis principales motivaciones en esta vida y a quienes amo con todo mi corazón.

Prefacio

Como lo mencionaron Aho, Hopcroft, y Ullman en su libro, *El Diseño y Análisis de Algoritmos de Computación* (1974), el estudio de los algoritmos es la esencia de las Ciencias de la Computación. En particular, en los últimos años se ha estudiado mucho la relación que existe entre los algoritmos y la Teoría de Gráficas, en donde por lo general suele haber un avance mutuo; es decir, el avance de una rama beneficia a la otra, y viceversa. El propósito de este trabajo es presentar una serie de algoritmos cuya finalidad es facilitar el estudio de las estructuras y propiedades que ciertas gráficas pueden tener.

Requisitos

La mayor parte de este trabajo es autocontenido, no es necesario tener conocimientos previos sobre la Teoría de Gráficas o el Análisis de Algoritmos. Sin embargo, sí es deseable que el lector tenga cierta madurez matemática para poder comprender con mayor facilidad los conceptos con los que se trabajan. De igual manera, es deseable que el lector tenga algo de experiencia con la programación para facilitar la lectura de los algoritmos. Todas las proposiciones, teoremas, y algoritmos presentados (a excepción de la conversión a formatos) son demostrados en su respectiva sección. En el caso de los algoritmos, se intenta dar una explicación intuitiva sobre su funcionamiento antes de presentar formalmente al algoritmo y su demostración.

Contenidos

El primer capítulo es una introducción tanto a la Teoría de Gráficas como al Análisis de Algoritmos. Se define lo que es una gráfica, los diferentes tipos de gráficas que podemos tener, como los árboles, y además algunas propiedades que estas cumplen que nos servirán a lo largo del trabajo. Finalmente, definimos lo que es un algoritmo y cómo los podemos analizar por medio de la notación O grandota, incluyendo un

ejemplo sobre el diseño y análisis de un algoritmo para ordenar lexicográficamente una sucesión de cadenas.

En el segundo capítulo presentamos diversas maneras en las que podemos guardar gráficas de forma persistente en memoria. Se estudia el formato `Graph6`, el cual trabaja con gráficas simples. También presentamos el formato `Loop6`, una variante del formato anterior la cual nos permite trabajar con gráficas con lazos. El formato `Digraph6` para digráficas. Por último, el formato `Sparse6` que nos permite trabajar tanto con gráficas simples como con gráficas con multiaristas y lazos.

En el tercer capítulo discutimos el algoritmo lineal para detectar el isomorfismo entre dos árboles arbitrarios, presentado por primera vez por Aho, Hopcroft, y Ullman en 1974. Se presenta primero una variente del problema, la cual trabaja con árboles ordenados enraizados. Después se generaliza el problema para trabajar con el isomorfismo de árboles enraizados. Y por último, trabajamos con el problema original de detectar el isomorfismo de árboles arbitrarios. Para cada problema se diseñó un algoritmo eficiente tanto en espacio como en tiempo.

En el cuarto capítulo discutimos sobre el ordenamiento doblemente lexicográfico de una matriz, presentado por primera vez por Lubiw, Paige y Tarjan en 1987. El propósito de este capítulo es diseñar e implementar un algoritmo eficiente el cual nos permita obtener un ordenamiento de una matriz, tal que sus renglones y columnas, como vectores, están ordenados lexicográficamente de mayor a menor.

Finalmente, en el último capítulo presentamos las conclusiones de este trabajo, las cuales abarcan desde la importancia que tienen los algoritmos presentados aquí como las dificultades y optimizaciones que podemos encontrar al momento de implementar estos algoritmos en diversos lenguajes de programación.

Índice general

Capítulo 1

Introducción

En este capítulo presentamos una corta introducción tanto a la Teoría de Gráficas como al Análisis de Algoritmos. Todas las definiciones, conceptos, y proposiciones que presentamos aquí serán de utilidad durante el resto del trabajo. Las definiciones relacionadas con la Teoría de Gráficas las podemos encontrar en los libros **Graph Theory** por Bondy y Murty [1], y **Graph Theory, Fifth Edition** por Diestel [2]. Por último, las definiciones relacionadas con el Análisis Algoritmos las podemos encontrar en el libro **Introduction to Algorithms, Third Edition** por Cormen, Leiserson, Rivest y Stein [3].

1.1. Definiciones básicas

Una **gráfica** G es una terna ordenada (V_G, E_G, ψ_G). El conjunto no vacío V_G contiene a los elementos de la gráfica que denominamos **vértices**. Los elementos del conjunto E_G, en donde $E_G \cap V_G = \varnothing$, son llamados **aristas**. Finalmente, la función ψ_G es la **función de incidencia** de G que relaciona a cada arista con una pareja de vértices (no necesariamente distintos) de G. Formalmente, ψ_G se define de la siguiente forma:

$$\psi_G \colon E_G \to \binom{V_G}{1} \cup \binom{V_G}{2}.$$

Consideramos a la gráfica con un sólo vértice y sin aristas como **trivial**, y a cualquier otra como **no trivial**.

Además, podemos dar una representación visual de una gráfica mediante un dibujo o diagrama. Por cada vértice dibujamos un círculo pequeño o punto, y una línea por cada arista, de tal forma que en los extremos de la línea se encuentren los vértices

en los que es incidente la arista. A continuación en las figuras 1.1 y 1.2 mostramos ejemplos de la representación visual de algunas gráficas.

Figura 1.1: Dibujo de una gráfica trivial.

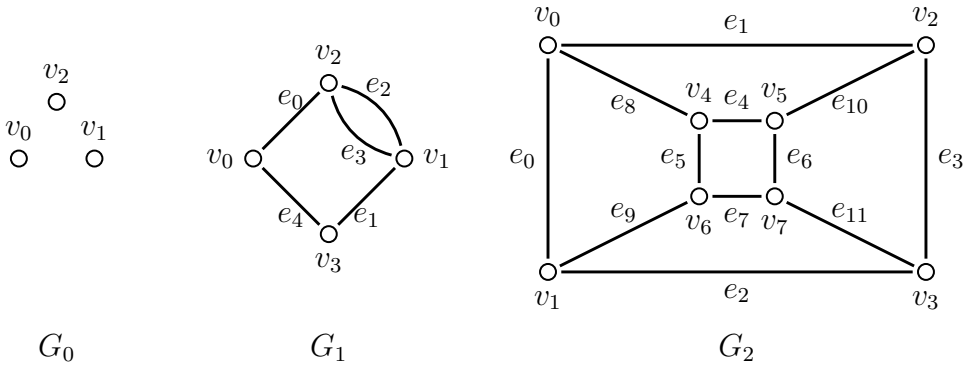


Figura 1.2: Dibujos de gráficas no triviales.

Por ejemplo, para definir a la gráfica G_1 de la figura 1.2, primero definimos al conjunto de vértices $V_G = \{v_0, v_1, v_2, v_3\}$, después al conjunto de aristas $E_G = \{e_0, e_1, e_2, e_3, e_4\}$, y por último a la función de incidencia:

$$\begin{aligned} &\psi_G(e_0) = \{v_0, v_2\}, \quad \psi_G(e_3) = \{v_1, v_2\}, \\ &\psi_G(e_1) = \{v_1, v_3\}, \quad \psi_G(e_4) = \{v_0, v_3\}. \\ &\psi_G(e_2) = \{v_1, v_2\}, \end{aligned}$$

Si tenemos $e \in E_G$ y $u, v \in V_G$ tal que $\psi_G(e) = \{u, v\}$, entonces decimos que u es **adyacente** a v y viceversa. También decimos que e es **incidente** en u y en v. Definimos a los **extremos** de e como los vértices con los que incide. Dos vértices son **vecinos** si son adyacentes. De esta forma, definimos a la **vecindad** de un vértice v como el conjunto $N(v) = \{u\colon\ \psi_G(e) = \{u, v\}, e \in E_G\}$, es decir, su conjunto de vecinos. Al número de aristas incidentes en un vértice v lo llamamos **grado**, y lo denotamos por $d_G(v)$. Por ejemplo, en la gráfica G_2 de la figura 1.2, el vértice v_0 es adyacente al vértice v_1, la vecindad del vértice v_4 es $N(v_4) = \{v_0, v_5, v_6\}$, y nótese que el grado de todos los vértices es 3.

Denominamos a una arista con extremos iguales como un **lazo**, estos cuentan dos veces al momento de determinar el grado de un vértice. Si tenemos dos o más aristas que comparten los mismos extremos, entonces decimos que son **aristas paralelas** o **multiaristas**.

En algunos casos nos vamos a encontrar con gráficas que no cuentan ni con lazos ni con multiaristas, como es el ejemplo de la gráfica G_2 de la figura 1.2, a estas gráficas las denominamos **gráficas simples**. Para este tipo de gráficas resulta redundante definir una función de incidencia, pues ninguna arista tiene extremos iguales y tampoco comparten los mismos extremos con alguna otra, por lo tanto definimos al conjunto de aristas de una gráfica simple G como $E_G \subseteq \binom{V_G}{2}$. Así, a una gráfica simple G la podemos representar únicamente como la pareja ordenada (V_G, E_G). Para facilitar la lectura, en el caso de las gráficas simples escribir $uv \in E$ es equivalente a tener $e = \{u, v\} \in E_G$. De igual manera, cuando trabajemos con una gráfica G vamos a utilizar de manera implícita su representación como gráfica simple, $G = (V_G, E_G)$, a menos de que se especifique lo contrario.

1.2. Digráficas

Las aristas de una gráfica se pueden considerar como de *doble-sentido*, es decir, si un vértice v es adyacente a otro vértice u, entonces también se cumple que u es adyacente a v. Por este motivo, introducimos a las **gráficas dirigidas** o digráficas, en donde si v es adyacente a u, entonces no necesariamente se cumple que u sea adyacente a v.

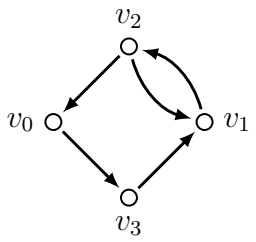
Una **digráfica** D es una terna ordenada $D = (V_D, \mathcal{A}_D, \psi_D)$. El conjunto no vacío V_D contiene a los vértices de la digráfica. Los elementos del conjunto $\mathcal{A}_D$, en donde $\mathcal{A}_D \cap V_D = \varnothing$, son llamados **flechas**. Y la función ψ_D es la función de incidencia de D que relaciona a cada flecha con una pareja ordenada de vértices (no necesariamente distintos) de D. Formalmente ψ_D se define como:

$$\psi_D \colon \mathcal{A}_D \to (V_D \times V_D).$$

Sea $a \in \mathcal{A}_D$, tal que $\psi_D(a) = (u, v)$, decimos que u y v son la **cola** y **cabeza** de a respectivamente. Así, al igual que en las gráficas, una digráfica puede tener varias flechas que comparten la misma cola y cabeza, al igual que tener flechas en donde la cola y la cabeza sean el mismo vértice. Por ende, a aquellas digráficas que no presenten los casos anteriores las conocemos como **digráficas simples**, y las definimos como $D = (V_D, \mathcal{A}_D)$, en donde $\mathcal{A}_D \subseteq (V_D \times V_D) - \{(v, v) \colon \ v \in V_D\}$.

También podemos representar a una digráfica por medio de un dibujo o diagrama,

con la única diferencia de que le agregamos una punta a las líneas que representan las flechas para indicar quién es la cola y la cabeza; en este caso, la punta indica quién es la cabeza de la flecha.



D

Figura 1.3: Dibujo de una digráfica D.

La **gráfica subyacente** U_D de una digráfica D, es una gráfica que asociamos a D, la cual definimos como $V_{U_D} = V_D$ y $uv \in E_{U_D}$ si y sólo si $(u, v) \in \mathcal{A}_D$ o $(v, u) \in \mathcal{A}_D$.

1.3. Subgráficas

Sean G y H dos gráficas. Si $V_H \subseteq V_G$, $E_H \subseteq E_G$, y $\psi_H(e) = \psi_G(e)$ para toda $e \in E_H$, entonces decimos que H es una **subgráfica** de G, y lo denotamos por $H \subseteq G$. En caso de que $V_G = V_H$, entonces H es una **subgráfica generadora**.

Sea $V' \subseteq V_G$, definimos a $G - V'$ como la subgráfica de G obtenida al borrar los vértices de V' junto con las aristas que inciden en ellos, en el caso de que $V' = \{v\}$ entonces sólo escribimos $G - v$. De igual manera, sea $E' \subseteq E_G$, definimos a $G - E'$ como la subgráfica de G cuyo conjunto de aristas es $E_G - E'$, en caso de que $E' = \{e\}$ entonces sólo escribimos $G - e$.

Si $E' \subseteq E_G$, la subgráfica de G cuyo conjunto de vértices son los extremos de las aristas en E', y cuyo conjunto de aristas es E', es la subgráfica de G **inducida** por E', denotada por $G[E']$.

Proposición 1.3.1. *Sea G una gráfica, entonces se cumple que*

$$\sum_{v \in V_G} d(v) = 2|E_G|.$$

Demostración. Procedemos por inducción sobre $|E_G|$. El caso base es cuando $|E_G| = 0$. Al no tener aristas entonces el grado de todos los vértices es cero, por ende

$\sum_{v\in V_G} d(v) = 0 = 2\cdot 0 = 2|E_G|$. Para el paso inductivo supongamos que $|E_G| = k > 0$. Sea e una de las aristas de G, observamos que $|E_{G-e}| = k-1$, así por hipótesis inductiva tenemos que $\sum_{v\in V_{G-e}} d(v) = 2|E_{G-e}|$. Sean u y v los extremos de e, observemos que al agregar de nuevo e a $G-e$ estamos aumentando exactamente en uno el grado u y v, por ende:

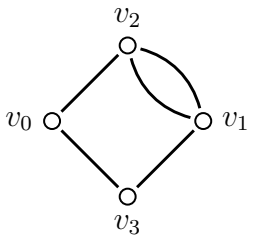
$$2|E_G| = 2|\{e\}| + 2|E_{G-e}| = 2 + 2|E_{G-e}| = 2 + \sum_{v\in V_{G-e}} d(v) = \sum_{v\in V_G} d(v).$$

□

1.4. Matriz de adyacencia

A toda gráfica G con n vértices la podemos representar por medio de una matriz de tamaño $n \times n$, usualmente llamada A_G, a la cual conocemos como **matriz de adyacencia**. Definimos $A_G = (a_{ij})$, donde a_{ij} es el número de aristas cuyos extremos son v_i y v_j. Si G es simple, lo anterior lo podemos definir simplemente como

$$a_{ij} = \begin{cases} 1 & \text{si } v_iv_j \in E, \\ 0 & \text{si } v_iv_j \notin E. \end{cases}$$



G

	v_0	v_1	v_2	v_3
v_0	0	0	1	1
v_1	0	0	2	1
v_2	1	2	0	0
v_3	1	1	0	0

A_G

Figura 1.4: Gráfica G junto con su matriz de adyacencia A_G.

Por ejemplo, en la figura 1.4 podemos observar que G no es una gráfica simple pues existen dos aristas con extremos v_1 y v_2. Así, siguiendo la definición de A_G tenemos que $a_{12} = a_{21} = 2$, las demás aristas no comparten extremos con alguna otra por lo que $a_{02} = a_{03} = a_{13} = a_{20} = a_{30} = a_{31} = 1$.

De igual manera, a toda digráfica D con n vértices le corresponde una matriz de $n \times n$, $A(D)$, llamada matriz de adyacencia de D, con $A(D) = (a_{ij})$, donde a_{ij} es el número de flechas cuya cabeza y cola son v_i y v_j respectivamente.

1.5. Caminos, trayectorias y ciclos

Definimos a un **camino** W en G como una sucesión alternante de vértices y aristas de la forma

$$W = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k),$$

donde para cada $i \in \{0, \ldots, k\}$ tenemos que $v_i \in V_G$ y para cada $j \in \{1, \ldots, k\}$ tenemos que $e_j \in E_G$ tal que $\psi_G(e_j) = \{v_{j-1}, v_j\}$. La **longitud** de W, denotada por $\ell(W)$, es k. Observemos que si W no repite vértices, entonces la longitud de W es la cantidad de aristas que éste contiene. Sean u y v el vértice inicial y final de W, respectivamente, decimos que W es un uv-camino. De esta forma, decimos que u y v son los **extremos** de W, y a cualquier otro vértice de W distinto de u y v lo llamamos **vertice intermedio** de W. Adicionalmente, definimos a la **reversa de un camino** W, denotado como W^{-1}, como el camino W pero con el orden de sus vértices y aristas invertido,

$$W^{-1} = (v_k, e_k, \ldots, e_2, v_1, e_1, v_0).$$

Si G es una gráfica simple, entonces entre cualesquiera dos vértices existe a lo más una arista, por lo que resulta redudante especificar a las aristas que forman parte de un camino, por este motivo los caminos dentro de las gráficas simples pueden ser descritas como únicamente una sucesión de vértices de la forma

$$W = (v_0, v_1, v_2, \ldots, v_k),$$

donde para cada $i \in \{1, \ldots, k\}$ se tiene que $v_{i-1}v_i \in E_G$.

Sea W un camino tal que $W = (u_0, e_1, u_1, e_2 \ldots, e_k, u_k)$, y sean i, j enteros que cumplen $0 \leq i < j \leq k$, definimos al **subcamino** u_iWu_j como

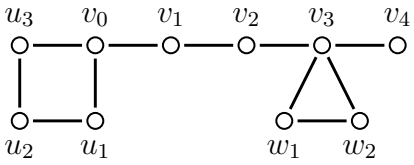
$$u_iWu_j = (u_i, e_{i+1}, u_{i+1}, \ldots, e_j, u_j);$$

cabe mencionar que si $i = 0$, entonces escribimos únicamente Wu_j, y si $j = k$, entonces escribimos únicamente u_iW.

Sean $W_1 = W$ y $W_2 = (v_0, \ldots, v_l)$ dos caminos tal que $u_k = v_0$, definimos al camino W_1W_2 como

$$W_1W_2 = (u_0, \ldots, u_k = v_0, v_1, \ldots, v_l).$$

Un camino que no repite ni aristas ni vértices lo llamamos **trayectoria**. Si el vértice inicial y final de un camino son el mismo, entonces decimos que es un **camino cerrado**, y si además no se repiten vértices salvo el inicial y final entonces es un **ciclo**.



G

Figura 1.5: Una gráfica simple con múltiples caminos, trayectorias y ciclos.

Por ejemplo, en la figura 1.5, en la gráfica G tenemos que $W = (v_0, v_1, v_2, v_1, v_0)$ es un camino, $P = (v_0, v_1, v_2, v_3, v_4)$ es una trayectoria, y $C_1 = (v_0, u_3, u_2, u_1, v_0)$ y $C_2 = (v_3, w_1, w_2, v_3)$ son dos ciclos.

Proposición 1.5.1. *Sea G una gráfica. Si el grado de todos los vértices es mayor o igual a 2 entonces G contiene un ciclo.*

Demostración. Si G tiene lazos, sea v el vértice con al menos un lazo ℓ_1, entonces el camino $C = (v, \ell_1, v)$ es el ciclo que buscábamos en G. Por otro lado si G tiene multiaristas, sean e_1 y e_2 las aristas que comparten extremos u y v, entonces el camino $C = (u, e_1, v, e_2, u)$ es el ciclo que buscábamos en G. Así, si G es simple, sea $P = (u_1, u_2, \ldots, u_k)$ la trayectoria de longitud máxima en G. Observamos que todos los vecinos de u_1 se encuentran dentro de P, ya que de lo contrario podríamos extender a P agregando al vecino no se encuentra dentro de la trayectoria. Al ser u_1 de grado mayor o igual a 2 entonces existe otro vecino distinto de u_2 en P. Sea u_i este vecino, entonces podemos construir el siguiente ciclo $C = u_1 P u_i u_1$. Obteniendo así el ciclo que buscábamos en G. □

1.6. Conexidad, distancia y excentricidad

Decimos que una gráfica es **conexa** si entre cualesquiera dos vértices existe un camino que los conecta. A partir de esta definición, también decimos que un vértice u **alcanza** a un vértice v en G si existe un uv-camino en G.

Por ejemplo, en la figura 1.2, las gráficas G_1 y G_2 son conexas, mientras que la gráfica G_0 no lo es pues v_0 no alcanza ni a v_1 ni a v_2 y viceversa.

A las distintas subgráficas de una gráfica G, máximas por contención con la propiedad de ser conexas, las conocemos como las **componentes conexas** de G. Así, una gráfica es conexa si y sólo si tiene una única componente conexa. Si existe una arista e tal que $G - e$ tiene más componentes conexas que G, entonces decimos que e es un **puente**.

Proposición 1.6.1. *Sean G una gráfica conexa y C un ciclo contenido en G. Si e es una arista que esta contenida en C, entonces $G - e$ sigue siendo conexa.*

Demostración. Sean u y v los extremos de e, expresamos al ciclo C como $C = (u = u_0, u_1, \ldots, u_k = v, u_0)$, donde $P' = (u_0, u_1, \ldots, u_k)$ es la parte de C que no contiene a e; es decir, una uv-trayectoria que no contiene a e.

Sean $x, y \in V_G$. Veamos que en $G - e$ sigue existiendo un xy-camino W. Si W no contiene a e, entonces W es un xy-camino en $G - e$. Si W contiene a e, sean u y v los extremos de e, supongamos sin pérdida de generalidad que W pasa primero por u. Entonces el camino $W' = xWuP'vWy$ es un xy-camino en $G - e$. Así, como entre cualesquiera dos vértices hay un camino que los conecta, entonces $G - e$ es conexa. □

Sea G una gráfica y $u, v \in V_G$. Definimos la **distancia** de u a v, denotada por $d(u, v)$, como la longitud de la uv-trayectoria más corta en G, y si u no alcanza a v entonces definimos $d(u, v) = \infty$.

La **excentricidad** de un vértice u se define como $\max_{v \in V_G} \{d(u, v)\}$, es decir, la máxima distancia que existe entre el vértice u y cualquier otro en G. Así, el **diámetro** de G se define como el máximo de las excentricidades, es decir, la distancia entre los vértices más alejados de G. Finalmente, definimos un **centro** de G como un vértice con menor excentricidad, en otras palabras, un centro de G es un vértice que más cerca se encuentra de todos los demás. Siguiendo el ejemplo de la figura 1.2, G_0 tiene diámetro ∞, G_1 tiene diámetro 2, y G_2 tiene diámetro 3. Algo curioso es que todos los vértices, tanto de G_1 como de G_2, son centros de sus respectivas gráficas.

1.7. Árboles

Un **árbol** es una grafica G acíclica (sin ciclos) y conexa. Sea T un árbol y sea v un vértice de T, si v tiene grado igual a 1 entonces decimos que v es una **hoja**, de lo contrario, decimos que v es **no hoja**. Cabe mencionar que un árbol no puede contener ni lazos ni multiaristas pues estas inducen ciclos.

Los árboles tienen múltiples propiedades interesantes que serán de mucha utilidad en las siguientes secciones del trabajo. A continuación las presentamos junto con sus respectivas demostraciones.

Proposición 1.7.1. *Sea G una gráfica. Entre cualesquiera dos vértices $u, v \in V_G$ existe una única uv-trayectoria si y sólo si G es un árbol.*

Demostración. Si entre cualesquiera dos vértices $u, v \in V_G$ existe una única uv-trayectoria entonces G es conexa por definición. Por otro lado, esto implica que no podemos tener ciclos dentro de G, ya que de existir uno, para cualquier pareja de vértices dentro del ciclo se tendrían dos trayectorias distintas, por lo que G es acíclica. Concluyendo así que G es un árbol.

Si G es un árbol, entonces es acíclica y conexa. Así, para cualesquiera dos vértices $u, v \in V_G$ existe una uv-trayectoria. Sean T_1 y T_2 dos uv-trayectorias, estas dos trayectorias son iguales. De lo contrario, podemos expresar a T_1 y T_2 de la siguiente manera:

$$T_1 = uT_1xT_1yT_1v,$$

$$T_2 = uT_1xT_2yT_1v.$$

En donde a partir del vértice x, las trayectorias T_1 y T_2 son distintas, y a partir del vértice y las trayectorias vuelven a coincidir; notemos que no necesariamente x es distinto de u y y es distinto de v. Observemos que $W = xT_1yT_2^{-1}x$ es un camino cerrado. Si consideramos a W como una subgráfica de G, es posible notar que todos los vértices tienen grado al menos 2 en W. Claramente, todo vértice interno tiene dos aristas de W incidentes, pues T_1 y T_2 son trayectorias. Además, como a partir de x las trayectorias T_1 y T_2 difieren, la primera y la última aristas de W son distintas, por lo que el grado de x en W también es al menos 2. Análogamente, el grado de y en W es al menos 2. Es posible aplicar la proposición 1.5.1 para obtener un ciclo en W, que también es un ciclo en G. Por lo tanto, la uv-trayectoria en G es única. □

Proposición 1.7.2. *Sean T un árbol y v una hoja de T. La gráfica $T - v$ es un árbol.*

Demostración. Veamos que para cualesquiera dos vértices existe una única trayectoria en $T - v$. Sean $x, y \in V_{T-v}$ y sea P la única xy-trayectoria en T. Observemos que v no se puede encontrar como vértice intermedio de P, ya que esto implicaría que el grado de v es mayor a 1, por lo que P está en $T - v$ y es única. Se concluye por la proposición 1.7.1 que $T - v$ es un árbol. □

Proposición 1.7.3. *Sean T un árbol, v un vértice que no se encuentra en T, y $T+v$ la gráfica obtenida al hacer adyacente v a un único vértice de T. La gráfica $T + v$ es un árbol.*

Demostración. Sean $x, y \in V_{T+v}$, veamos que existe una única xy-trayectoria en $T+v$. Si $x \neq v$ y $y \neq v$, sabemos que en T existe una xy-trayectoria P. Además esta trayectoria sigue siendo única, ya que para formar una nueva xy-trayectoria con v, éste tendría que ser adyacente a al menos dos vértices, pero el grado de v es 1. Ahora, supongamos sin pérdida de generalidad que $x = v$ y sea u el vértice adyacente a v. Sea y un vértice distinto de v, sabemos que existe una única uy-trayectoria en $T+v$ y hay una única forma de extenderla a v usando la arista uv en $T+v$, por lo tanto la vy-trayectoria $vuTy$ es única en $T+v$. Así, entre cualesquiera dos vértices existe una única trayectoria; concluyendo por la proposición 1.7.1 que $T+v$ es un árbol. □

Proposición 1.7.4. *Para todo árbol se cumple*

$$|E| = |V| - 1.$$

Demostración. Sea T un árbol. Procedemos por inducción sobre $|V_T|$. El caso base es cuando $|V_T| = 1$, es decir, sólo tenemos un vértice v. Como no permitimos lazos dentro de los árboles entonces el grado de v es cero, por ende T no tiene aristas. En otras palabras $|E_T| = 0 = 1 - 1 = |V_T| - 1$, lo que buscábamos. Para el paso inductivo, cuando $|V_T| = k > 1$. Dado que T es acíclica y conexa, la contrapositiva de la proposición 1.5.1 nos asegura que existe al menos un vértice de grado 1. Sea v una de las hojas de T, recordemos que $T - v$ es un árbol. Por hipótesis de inducción tenemos que $|E_{T-v}| = |V_{T-v}| - 1$, al volver a agregar v a $T - v$ lo único que hacemos es agregar un vértice y una arista más, por lo tanto:

$$|E_T| = |E_{T-v}| + 1 = (|V_{T-v}| - 1) + 1 = (|V_{T-v}| + 1) - 1 = |V_T| - 1.$$

□

Proposición 1.7.5. *Un árbol tiene exactamente un centro o dos centros adyacentes.*

Demostración. Sea T un árbol y P una trayectoria de longitud máxima de T. Sean u y v los extremos de P, notemos que $d(u, v) = \ell(P)$ es el diámetro de T pues al ser P de longitud máxima entonces u y v son los vértices que más alejados están en T.

Sea $P = (u = v_1, v_2, \ldots, v_k = v)$. Si P tiene longitud par, observemos que el vértice w en el centro de P, el de la posición $i = \lceil \frac{k}{2} \rceil$, es el que menor excentridad tiene en T. Lo anterior se debe a que la distancia entre w y cualquier otro vértice siempre será menor o igual a $\lfloor \frac{k}{2} \rfloor$, y por ende la excentricidad de w es igual a $d(w, u) = d(w, v)$.

Para verificar lo anterior, sea x un vértice diferente de w, veamos que siempre tenemos que $d(x, u) > d(w, u)$ ó $d(x, v) > d(w, v)$. Si $x \in P$, sin pérdida de generalidad supongamos que x ocurre antes que w en P, por lo tanto tenemos

$d(x,u) < d(w,u) = d(w,v) < d(x,v)$; esta última desigualdad nos asegura que x tiene mayor excentricidad que w. Si $x \notin P$, notemos que si la xu-trayectoria no contiene a w entonces la xv-trayectoria sí y viceversa, por otro lado también es posible que ambas trayectorias contengan a w. Sin pérdida de generalidad, sea P' la xu-trayectoria que contiene a w, tenemos que $d(w,u) < d(x,u)$; x tiene mayor excentricidad que w. Por lo tanto, independientemente del caso, el vértice con menor excentricidad es w, es decir, w es el centro de T.

Si P tiene longitud impar, observemos que los vértices en el centro de P, los de las posiciones $i = \frac{k}{2}$ y $j = \frac{k}{2} + 1$, son los de menor excentricidad en T. Sean w_1 y w_2 estos dos vértices, la distancia entre w_1 y cualquier otro vértice x siempre será menor o igual a $\frac{k}{2} + 1$, y lo mismo para w_2. Utilizando argumentos semejantes al caso anterior, podemos observar que cualquier otro vértice distinto a w_1 y w_2 tiene una excentricidad mayor a la de estos. Por lo tanto, w_1 y w_2 son los centros de T.

De esta manera se concluye que si el diámetro de T es par entonces T tiene sólo un centro, y si el diámetro es par entonces T tiene dos centros los cuales son adyacentes entre sí. □

Proposición 1.7.6. *Sean T un árbol y P una trayectoria de longitud máxima en T. Los extremos de P son hojas.*

Demostración. Sea $P = (u_1, u_2, \ldots, u_k)$ la trayectoria de longitud máxima. Observamos que el grado de u_1 tiene que ser igual a 1. De lo contrario, u_1 tendría otro vecino dentro de P pero esto implica la existencia de un ciclo, y de otro modo podríamos extender a P agregando a este vecino a la trayectoria pero esto implica que P no era de longitud máxima. Lo mismo aplica para u_k. Se concluye de esta forma que los extremos de P son hojas. □

1.7.1. Árboles generadores

Si una gráfica G tiene una subgráfica generadora H que es un árbol, entonces decimos que H es un **árbol generador**. Veamos ahora que toda gráfica conexa contiene al menos un árbol generador.

Proposición 1.7.7. *Si G es una gráfica conexa entonces G contiene un árbol generador.*

Demostración. Procedemos por inducción sobre el número de ciclos que tiene G. Para el caso base, cuando G no tiene ciclos, observamos que G es conexa y acíclica, por lo tanto G es el árbol generador que buscábamos.

Para el paso inductivo, cuando G tiene k ciclos. Sea C uno de los ciclos de G, y sea e una de las aristas de C. Sabemos por la proposición 1.6.1 que $G-e$ sigue siendo conexa y que el número de ciclos ha disminuido en al menos uno. De esta forma, por hipótesis inductiva tenemos que $G-e$ contiene un árbol generador, y por ende G contiene un árbol generador. □

1.7.2. BFS, búsqueda por amplitud

Como podemos observar, a partir de una gráfica conexa G podemos construir un árbol generador recorriendo cada ciclo de G y eliminando una de sus aristas. Sin embargo, lo anterior implica que tenemos que identificar primero a todos los ciclos que se encuentran contenidos en G. Esto es poco eficiente pues en el peor de los casos, supongamos que estamos trabajando con una gráfica completa K_n (en donde cualesquiera dos vértices son adyacentes), la cantidad de ciclos que tiene K_n esta definido por:

$$\sum_{i=3}^{n} \binom{n}{i},$$

lo cual resulta bastante costoso computacionalmente.

Por este motivo, proponemos el siguiente algoritmo que se basa en recorrer la gráfica y construir un árbol generador al mismo tiempo.

Existen múltiples maneras de recorrer una gráfica, una de ellas es por medio del algoritmo **BFS** (algoritmo 1), búsqueda por amplitud, o mejor conocido en inglés como *Breadth First Search*. La idea del algoritmo es la siguiente, empezamos colocándonos en un vértice v de la gráfica, después recorremos a cada uno de sus vecinos u indicando que venimos de v, repetimos el proceso para cada uno de los vecinos u sin volver a considerar a los vértices a los cuales ya recorrimos, y así sucesivamente hasta terminar de recorrer a todos los vértices de la gráfica. Al final, este procedimiento construye un árbol generador. A continuación, presentamos un ejemplo de un árbol generador construido a partir de BFS:

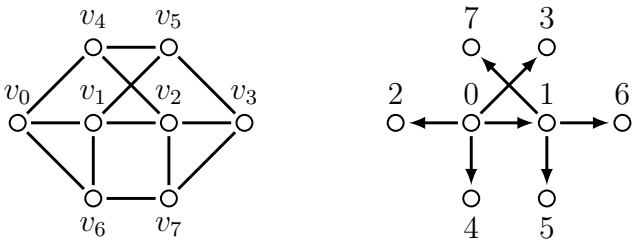


Figura 1.6: El árbol generador H obtenido de hacer BFS sobre G a partir de v_1.

Siguiendo el ejemplo de la figura 1.6. Construimos un árbol generador de G colocándonos en v_1, después recorremos a cada uno de sus vecinos v_2, v_0, v_5, v_6 en este orden. Nos colocamos en v_2, pues fue el primer vecino de v_1 que recorrimos, y recorremos a sus respectivos vecinos v_7, v_3 y v_4 en este orden. Observemos que en este punto ya terminamos de recorrer a todos los vértices de la gráfica, por lo tanto el conjunto de aristas que utilizamos en el recorrido inducen un árbol generador H para G. Cabe mencionar que el orden en el recorremos a los vecinos de los vértices de G es arbitrario, y el presentado aquí sólo fue un ejemplo.

Un detalle importante sobre la implementación de BFS es que ésta utiliza una estructura de datos conocida como **cola**, *queue* en inglés. La estructura anterior tiene una propiedad bastante útil computacionalmente, nos permite agregar y eliminar elementos en tiempo constante (más adelante veremos a qué nos referimos con esto). Esta estructura tiene un orden FIFO (*First In, First Out*), es decir, el primer elemento que agreguemos será el primero en salir; un ejemplo visual puede ser la fila que nosotros hacemos para comprar algún producto en el mercado, el primero en llegar a la fila es el primero en ser atendido y por ende el primero en salir de ella.

Sea Q una cola. Definimos a la **cabeza** de Q como el elemento que no tiene antecesor, en otras palabras, el que está hasta en frente de la cola. El único elemento de Q al que tenemos acceso es la cabeza, por ende es el único elemento que podemos eliminar. Sea x el último elemento de Q, agregar un elemento w a Q es simplemente ponerlo después de x en Q; si Q no tiene elemento alguno, entonces w se convierte en la cabeza de Q. Por ejemplo, sea Q_0 una cola, supongamos que agregamos a los elementos a, b y c en este orden. Entonces a es la cabeza de Q_0, y para que podemos tener acceso a b, primero tenemos que eliminar a a, y si queremos acceder a c entonces tenemos que eliminar a a y después a b. Si agregamos el elemento d a Q_0, entonces el orden que tiene Q_0 es (a, b, c, d). Si eliminamos la cabeza, entonces el nuevo orden es (b, c, d), en donde es claro que b es la nueva cabeza de Q_0.

El algoritmo BFS (algoritmo 1) que presentamos en este trabajo construye y regresa tres funciones. La función $t\colon V_G \to \mathbb{N}$, el tiempo de exploración, en donde si $t(v) = i$ entonces v es el i-ésimo vértice de G en ser recorrido. La función $d\colon V_G \to \mathbb{N}$, la profundidad, en donde $d(v)$ es la distancia entre v y el vértice distinguido sobre el cual ejecutamos BFS. Por último, la función $p\colon V_G \to V_G$, el parentesco, en donde $p(v)$ es el vértice por el que llegamos a v; cabe mencionar que esta función no esta definida para el vértice distinguido.

Algoritmo 1: BFS, búsqueda por amplitud

Input: Una gráfica conexa G con un vértice distinguido r.
Output: Funciones de parentesco p, profundidad d y tiempo de exploración t.

```
1  Q ← []; i ← 0
2  colorear a r de negro
3  agregar r al final de Q
4  t(r) ← i, p(r) ← ∅, d(r) ← 0
5  i ← i + 1
6  while Q ≠ [] do
7  |  elegir la cabeza x de Q
8  |  for y ∈ N(x) do
9  |  |  if y sin colorear then
10 |  |  |  colorear a y de negro
11 |  |  |  agregar y al final de Q
12 |  |  |  t(y) ← i, p(y) ← x, d(y) ← d(x) + 1
13 |  |  |  i ← i + 1
14 |  |  end
15 |  end
16 |  eliminar a x de Q
17 end
18 return (p, d, t)
```

Ahora veamos que la función de parentesco p que regresa BFS efectivamente construye un árbol generador.

Lema 1.7.8. *La función de parentesco que regresa el algoritmo 1 sobre una gráfica conexa G es capaz de construir un árbol generador de G.*

Demostración. Sea $E' = \{vp(v) \in E\colon v \in V_G\}$ el conjunto de aristas que induce la

función de parentesco p que regresa el algoritmo 1, veamos que $G[E']$ es un árbol generador de G. Para ello tenemos que ver que $G[E']$ es conexa y acíclica.

Para ver que $G[E']$ es conexa, sea r el vértice distinguido sobre el que aplicamos BFS, primero veamos que todo vértice v alcanza a r. Procedemos por inducción sobre $d(r, v)$.

Para el caso base, cuando $d(r, v) = 1$, observamos que esto ocurre cuando v es vecino de r. Cuando entramos al ciclo de la línea 6 por primera vez, la cabeza de Q es r, y recorremos a todos sus vecinos en la línea 8. Por lo tanto, en algún momento coloreamos a v y asignamos $p(v) = r$, asegurando de esta forma que $vp(v) = vr \in E'$.

Para el paso inductivo, cuando $d(r, v) > 1$. Sea $u = p(v)$, notamos que $d(r, u) = d(r, v) - 1$. Lo anterior se debe a que en BFS primero recorremos a todos los vértices que se encuentran a distancia 1 de r, después a los de distancia 2, y así eventualmente con todos los vértices. Por ende, si u es padre de v, entonces $d(r, u) = d(r, v) - 1$. Así, por hipótesis de inducción tenemos que u alcanza a r, y como v alcanza a u por medio de $vp(v) = vu \in E'$, entonces v alcanza a r.

De esta forma, sean $x, y \in V_G$, veamos que existe un xy-camino en $G[E']$. Si P y Q son el xr-camino y el ry-camino en $G[E']$, respectivamente, entonces el camino PQ es un xy-camino en $G[E']$. Concluyendo así que $G[E']$ es conexa pues entre cualesquiera dos vértices existe un camino que los conecta.

Cabe mencionar que, durante la ejecución del algoritmo, colorear a un vértice v implica que existe un camino desde v hasta el vértice distinguido r en $G[E']$. Por lo tanto, si se cumple la condición de la línea 9 al recorrer un vértice x, es porque hasta ese punto el vértice x no alcanza a r en $G[E']$. En otras palabras, nunca agregamos una arista desde un vértice que ya alcanza a r a otro que también ya lo alcanza, y como todos los vértices alcanzan a r, entonces para todos los vértices existe una única xr-trayectoria. Esto además nos garantiza que entre cualesquiera dos vértices $u, v \in V_G$ va a existir una única uv-trayectoria en $G[E']$. Concluyendo por la proposición 1.7.1 que $G[E']$ es acíclica y conexa; un árbol generador. □

1.7.3. DFS, búsqueda por profundidad

Otro algoritmo para recorrer árboles es DFS, búsqueda por profundidad, o mejor conocido en inglés como *Depth First Search*. La idea del recorrido es la siguiente, empezamos colocándonos en un vértice v de la gráfica, despúes recorremos a uno de sus vecinos u indicando que venimos de v, repetimos este proceso hasta que lleguemos a un vértice z al que todos sus vecinos han sido explorados. Después, vamos de regreso, nos colocamos en el vértice y por el que llegamos a z, y verificamos si aún quedan vecinos de y que no hayamos explorado. En caso de que sí, nos movemos a

este vecino no explorado y repetimos el proceso anterior. En caso de que no, entonces nos colocamos en el vértice x por el que llegamos a x y repetimos el proceso anterior. Al finalizar este proceso habrémos construido un árbol generador. Presentamos un árbol generador construido a partir de DFS:

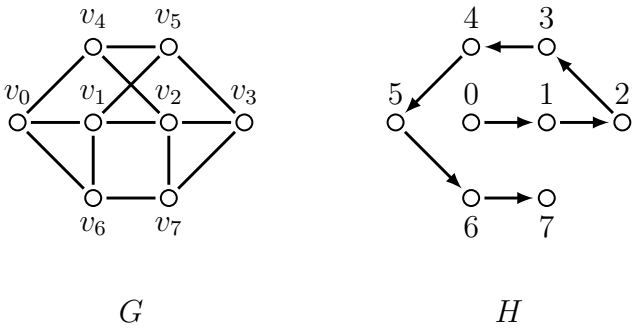


Figura 1.7: El árbol generador H obtenido de hacer DFS sobre G a partir de v_1.

Siguiendo el ejemplo de la figura 1.7, construimos un árbol generador de G colocándonos en v_1, y recorremos a $v_2, v_3, v_5, v_4, v_0, v_6$ y v_7 en este orden. Nos detenemos en v_7 pues todos sus vecinos han sido explorados, y entonces vamos de regreso. Sin embargo, para todos los vértices en los que nos coloquemos de regreso, todos sus vecinos ya han sido explorados. Concluyendo que ya hemos recorrido a todos los vértices de la gráfica, y por ende el conjunto de aristas que utilizamos en el recorrido inducen un árbol generador H de G. Cabe mencionar que el orden en el recorremos a los vecinos de los vértices de G es arbitrario, y el presentado aquí sólo fue un ejemplo.

En este trabajo presentamos **DFS** (algoritmo 2) como un algoritmo recursivo que únicamente regresa la función de parentesco antes definida.

Algoritmo 2: DFS, búsqueda por profundidad

Input: Una gráfica conexa G con un vértice distinguido r.
Output: Función de parentesco p.

```
1 colorear a r de negro
2 p(r) ← ∅
3 G, p ← DFSAuxiliar(G, r, p)
4 return p
```

Veamos que la función de parentesco que regresa DFS efectivamente construye un árbol generador.

Algoritmo 3: DFS Auxiliar

Input: Una gráfica conexa G, el vértice en el que nos encontramos actualmente v, y la función de parentesco p.

Output: La gráfica conexa G con v coloreado y la función de parentesco p.

```
1 for u ∈ N(v) do
2     if u sin colorear then
3         colorear a u de negro
4         p(u) ← v
5         G, p ← DFSAuxiliar(G, u, p)
6     end
7 end
8 return G, p
```

Lema 1.7.9. *La función de parentesco que regresa el algoritmo 2 sobre una gráfica conexa es capaz de construir un árbol generador de G.*

Demostración. Sea $E_p = \{vp(v) \in E \colon v \in V_G\}$ el conjunto de aristas que induce la función de parentesco p que regresa el algoritmo 2. Veamos que $G[E_p]$ es un árbol generador de G, es decir que $G[E_p]$ es conexa y acíclica. Procedemos por inducción sobre $|V|$.

El caso base, cuando $|V| = 1$. La gráfica G es la gráfica trivial con un único vértice v. Al ejecutar el algoritmo 2 sobre G, dado que v es el vértice distinguido entonces $p(v) \leftarrow \varnothing$, por lo que $E_p = \varnothing$. Así, $G[E_p]$ incluye a v y no podemos tener lazos ya que no incluimos ninguna arista; por lo tanto $G[E_p]$ es conexa y acíclica.

El paso inductivo es cuando $|V| = k > 1$. Sea v el vértice distinguido y $P = (v = u_0, u_1, u_2, \ldots, u_m)$ la primera trayectoria más larga que obtenemos por medio de 2 y sus llamadas recursivas, en donde $p(u_i) = u_{i-1}$ para toda $1 \leq i \leq m$. Dado que ya no podemos extender a P por medio de más llamadas recursivas, lo anterior implica que al momento de llamar a la subrutina 3 con u_m, para todos los vecinos de u_m la condición de la línea 2 no se cumple; es decir, todos los vecinos de u_m ya se encuentran dentro de P, y por ende pueden ser alcanzados desde v. De esta forma concluimos que $G - u_m$ es conexa, y como $|V_{G-u_m}| = k - 1$ entonces por hipótesis inductiva tenemos que la función de parentesco p' que regresa el algoritmo 2 sobre $G - u_m$ induce un árbol generador de $G - u_m$.

Sin pérdida de generalidad supongamos que la función de parentesco obtenida en $G - u_m$ preserva el orden en el que exploramos a los vértices de P. De esta forma, agregar u_m por medio de la arista $u_m u_{m-1}$ a $G[E_{p'}]$ construye un árbol generador de

G; como sólo estamos agregando una arista entonces no inducimos ciclos. Además, como p' preserva el orden obtenido en P, al realizar 2 sobre G, al momento de explorar u_{m-1} entonces recorremos a u_m, para después regresar a u_{m-1} y así recorrer el resto de los vértices por el orden descrito en p'. Concluyendo de este modo $G[E_{p'} \cup \{u_m u_{m-1}\}]$ es un árbol generador que podemos obtener por medio del algoritmo 2. □

1.8. Algoritmos

Ahora que ya mencionamos a BFS y DFS, tenemos que hablar sobre algoritmos.

Informalmente, un **algoritmo** es un procedimiento computacional *bien definido* que recibe un elemento (o conjunto de elementos) al que llamamos **entrada** (*input*) y produce otro elemento (o conjunto de elementos) al que llamamos **salida** (*output*).

Podemos entender a un algoritmo como una herramienta que nos permite resolver un problema en específico. Por ejemplo, BFS es un algoritmo diseñado para poder construir un árbol generador de una gráfica conexa G a partir de un vértice distinguido. A un elemento que cumple con las condiciones impuestas por el problema a resolver lo conocemos como la **instancia de un problema**. Siguiendo con el ejemplo, toda gráfica conexa es una instancia de un problema de BFS. Además, decimos que un álgoritmo es **correcto** si dada cualquier instancia, el algoritmo produce la salida esperada. Finalizando con el ejemplo, ya demostramos que BFS es correcto.

1.8.1. Análisis de Algoritmos

En la sección anterior hicimos una breve mención sobre la eficiencia que puede tener un algoritmo, sin embargo aún no hemos formalizado este concepto.

Para poder determinar qué tan eficiente es un algoritmo, primero tenemos que analizarlo, es decir, determinar cuántos recursos necesita para funcionar adecuadamente. Por ejemplo, un recurso podría ser el espacio que ocupa un algoritmo A para poderse ejecutar, una instancia de este ejemplo es la cantidad de memoria RAM que necesita una computadora para ejecutar A. Otro ejemplo podría ser el tiempo que necesitamos para que el algoritmo complete su ejecución.

Para analizar un algoritmo, primero tenemos que ver cúal es el **peor caso** que podemos tener, es decir, la instancia con la cual el algoritmo ocupa la máxima cantidad de recursos posibles. Retomando el ejemplo anterior, podemos considerar el peor caso en espacio como la instancia que más memoria ocupa o el peor caso en tiempo como la instancia que más tarda en ser resuelta por el algoritmo. La razón por la que nos enfocamos en el peor de los casos es porque nos da una cota superior en cuanto a la cantidad de recursos que requiere un algoritmo, conocer esta cota nos

garantiza que el algoritmo nunca va a necesitar más de estos recursos para funcionar correctamente.

Así, definimos a $c_i \geq 0$ como la cantidad de recursos que ocupa el algoritmo en la instrucción i, dependiendo del análisis será lo que representa este elemento. Por ejemplo si es sobre espacio, entonces c_i puede hacer referencia a la cantidad de bytes que ocupa una variable. De esta forma, podemos definir a la función $T(n)$ como la cantidad de recursos que ocupa el algoritmo, en donde n es el tamaño de la entrada.

Algo importante a considerar es que el valor de c_i puede o no depender del tamaño de la entrada. Existen instrucciones que sin importar el tamaño de la entrada, siempre van a consumir la misma cantidad de recursos. Por ejemplo, las instrucciones de asignación de constantes a variables. Para este tipo de instrucciones podemos considerar a c_i como un valor constante. En este trabajo, al momento de hacer un análisis se especificarán a las instrucciones que no consideramos que tomen recursos constantes y explicarémos cómo estas dependen de la entrada. Por último, si una instruccion toma recursos constantes y estamos haciendo un análisis sobre tiempo entonces decimos que toma **tiempo constante**; si el análisis es sobre espacio, entonces toma **espacio constante**.

Veamos un ejemplo haciendo un análisis de tiempo sobre BFS (algoritmo 1). Recordemos que el algoritmo recibe como entrada una gráfica, por lo tanto podemos considerar como entradas a su conjunto de vértices V y aristas E. Así, a menos de que se especifique lo contrario, siempre vamos a considerar a n como el número de vértices que hay en la gráfica; de igual manera, consideramos a m como el número de aristas que hay en la gráfica.

Proposición 1.8.1. *El análisis sobre tiempo del algoritmo 1 cumple con*

$$T(n,m) \leq \sum_{i=1}^{5} c_i + |V|(c_6 + c_7 + c_{16}) + 2|E|(c_9 + c_{10} + c_{11} + c_{12} + c_{13}) + c_{18}.$$

Demostración. Como las primeras cinco instrucciones sólo se ejecutan una vez, al igual que la última, entonces en $T(n,m)$ consideramos a c_1, c_2, c_3, c_4, c_5 y c_{18} como constantes.

La cantidad de iteraciones del ciclo de la línea 6 coincide con el número de vértices que hay en la gráfica. Lo anterior se debe a que todos los vértices de la gráfica son agregados en algún momento a la cola, y esto ocurre porque la función de parentesco que regresa el algoritmo induce una gráfica conexa. Además observamos que sólo agregamos una única vez a cada vértice de la gráfica pues para ser agregado este necesita no estar coloreado, y se colorea al ser agregado. Por lo tanto, las instrucciones c_6, c_7 y c_{16} se repiten $|V|$ veces.

Las instrucciones 8-14 también se repiten $|V|$ veces, pero en la línea 8 recorremos a cada vecino del vértice en el que nos encontremos actualmente, y para cada vecino verificamos si este esta coloreado o no. Sea $v \in V$, observemos que la condición de la línea 9 para v se cumple sólo una vez durante toda la ejecución (con la excepción de r, pues nunca se cumple esta condición para el vértice distinguido). Por lo tanto, para cada vértice que recorremos en la línea 8 no necesariamente se va a cumplir la condición de la línea 9. Sin embargo, supongamos que la condición de la línea 9 siempre se cumple para ver cuál sería la máxima cantidad de recursos que podríamos ocupar; en todo caso, el valor real siempre será menor al que presentamos a continuación. Así, la máxima cantidad de recursos que podemos usar en las línea 8-14 es:

$$\sum_{v \in V_G} d(v)(c_9 + c_{10} + c_{11} + c_{12} + c_{13}).$$

Recordemos que $\sum_{v \in V_G} d(v) = 2|E|$, por lo tanto, podemos expresar lo anterior como:

$$2|E|(c_9 + c_{10} + c_{11} + c_{12} + c_{13}).$$

Concluyendo de esta forma, que la cantidad de recursos que necesita el algoritmo al ser analizado sobre tiempo es:

$$T(n, m) \leq \sum_{i=1}^{5} c_i + |V|(c_6 + c_7 + c_{16}) + 2|E|(c_9 + c_{10} + c_{11} + c_{12} + c_{13}) + c_{18}.$$

□

Todas las instrucciones que se encuentran en la desigualdad anterior toman tiempo constante. Así, para facilitar la lectura, podemos definir a la función $T(n, m)$ obtenida anteriormente como $T(n, m) \leq a + b|V| + c|E|$. En donde las constantes a, b y c dependen de los recursos c_i de las instrucciones del algoritmo.

1.8.2. Tasa de crecimiento y la notación O grande

Como podemos observar, en un principio simplificamos el costo real de las instrucciones del algoritmo a sólo las constantes c_i, pero en ocasiones estas nos pueden dar más información de la que realmente necesitamos. Por ello, en el caso de BFS, decidimos expresar a todas estas constantes por medio de a, b y c.

Continuando con BFS, observemos que para valores muy grandes de $|V|$ y $|E|$, el valor de a se vuelve relativamente insignificante, por lo tanto podemos expresar

a $T(n)$ sólo por sus terminos dominantes, es decir, $b|V| + c|E|$. Nótese que también podemos ignorar a las constantes b y c, pues conforme más crecen los valores de $|V|$ y $|E|$ menos relevantes son las constantes que acompañan a estos factores. Por lo tanto, dada nuestra función $T(n)$, extraer los términos dominantes nos permite determinar cuáles son los factores que terminan afectando el rendimiento de nuestro algoritmo y por cuánto. A esta abstracción la conocemos como la **tasa de crecimiento** de una función, la cual comúnmente esta asociada con un algoritmo.

De esta forma, la tasa de crecimiento nos permite entender cómo funciona nuestro algoritmo conforme el tamaño de la entrada crece. Lo anterior nos permite determinar si la tasa de crecimiento se encuentra acotada superiormente por una función $g(n)$, tal que no importa de qué tamaño sea n, siempre vamos a tener que $T(n) \leq g(n)$. Cuando trabajamos sobre la cota superior asíntotica de un algoritmo, entonces utilizamos la **notación O grande**. En donde dada una función $g(n)$, denotamos a $O(g(n))$ como el conjunto de funciones:

$$\begin{aligned} O(g(n)) = \ & \{f(n) \colon \text{existen } c_0, n_0 \in \mathbb{N}^+ \text{ tal que} \\ & 0 \leq f(n) \leq c_0 g(n) \text{ para todo } n \geq n_0\}. \end{aligned}$$

Así, mostramos la cota superior asintótica para el tiempo de ejecución de BFS.

Proposición 1.8.2. *Sea $T(n, m)$ la función obtenida al analizar sobre tiempo al algoritmo 1, tenemos que $T(n, m) \in O(|V| + |E|)$.*

Demostración. Sea $a = \sum_{i=1}^{5} c_i + c_{18}, b = c_6 + c_7 + c_{16}$ y $c = c_9 + c_{10} + c_{11} + c_{12} + c_{13}$, entonces la función $T(n)$ se puede acotar de la siguiente forma:

$$T(n, m) \leq a + b|V| + c|E|.$$

Sea $g(n, m) = |V| + |E|, c_0 = (a + b + c)$, entonces para toda $n_0 \geq 1$ y $m_0 \geq 1$ se cumple que

$$a + b|V| + c|E| \leq (a + b + c)(|V| + |E|) \text{ para todo } |V| \geq n_0, |E| \geq m_0.$$

La desigualdad anterior se cumple, pues para toda $|V| \geq 1, |E| \geq 1$, tenemos que $a < a(|V| + |E|), b|V| < b(|V| + |E|)$, y $c|E| < c(|V| + |E|)$. Por lo tanto, tenemos $c_0, n_0, m_0 \in \mathbb{N}^+$ tal que $0 \leq T(n, m) \leq c_0(|V| + |E|)$ para toda $|V| \geq n_0, |E| \geq m_0$. Concluyendo de esta forma que $T(n, m) \in O(|V| + |E|)$. □

Por lo general, en el caso del análisis de tiempo, no solemos escribir $T(n) \in O(g(n))$, simplemente decimos que el algoritmo **corre en tiempo** $O(g(n))$. De igual manera, en el análisis de espacio, decimos que el algoritmo **utiliza espacio** $O(g(n))$.

De esta forma, utilizando la notación O grande, no es necesario dar la función $T(n)$ en específico de un algoritmo, pues basta con ver la estructura que tiene el algoritmo para poder determinar el orden que tiene. Por ejemplo, en el caso de BFS, las instrucciones 1-5 y 18 se ejecutan una sola vez, por lo que estas instrucciones se ejecutan en tiempo $O(1)$; cuando una instrucción o algoritmo se ejecuta en tiempo $O(1)$, decimos que se ejecuta en **tiempo constante**. Por otro lado, las instrucciones 6-7 y 16 se ejecutan $|V|$ veces, es decir, se ejecutan en tiempo $O(|V|)$; cuando una instrucción o algoritmo se ejecuta en tiempo $O(n)$, decimos que se ejecuta en **tiempo lineal**. Por último, las instrucciones 9-13 se ejecutan $2|E|$ veces, se ejecutan en tiempo $O(|E|)$. Concluyendo así, que el conjunto de instrucciones de BFS se ejecutan en tiempo $O(1), O(|V|), O(|E|)$, por lo que BFS se ejecuta en tiempo $O(|V| + |E|)$.

De ahora en adelante demostraremos el orden al que pertenece un algoritmo utilizando únicamente la notación O grande. Para finalizar, veamos el análisis sobre espacio de BFS, y el análisis sobre espacio y tiempo de DFS.

Proposición 1.8.3. *El algoritmo 1 utiliza espacio $O(|V|)$.*

Demostración. Observamos que la variable i es un entero, por lo tanto i tiene tamaño constante. Por otro lado, la cola Q, en el peor de los casos, almacena en algún punto a todos los vértices de la gráfica, por ende ocupa espacio $O(|V|)$. De igual manera, las funciones t, p y d se definen para todos los vértices de la gráfica, por lo que tienen tamaño $O(|V|)$. Así, el conjunto de variables del algoritmo tienen espacio $O(1)$ y $O(|V|)$, concluyendo de esta forma que el algoritmo utiliza espacio $O(|V|)$. □

Proposición 1.8.4. *El algoritmo 2 utiliza tiempo $O(|V| + |E|)$ y espacio $O(|V|)$.*

Demostración. Las instrucciones 1-2 se ejecutan en tiempo $O(1)$. Después llamamos a la subrutina 3 pasando como entrada a G, r y p, en donde G es una gráfica conexa, r el vértice distinguido, y p la función de parentesco. A partir de este punto empezamos el recorrido de la gráfica, como recorremos a todos los vértices, entonces la subrutina 3 es llamada un total de $|V|$ veces. Además, sea v el vértice que le pasamos como entrada, en la línea 1 recorremos a todos los vecinos de v, por lo que esta llamada toma tiempo $O(d(v))$. La condición de la línea 2 únicamente se cumple para aquellos vértices que no hemos explorado, por lo que nunca exploramos más de una vez a todos los vértices de la gráfica. Recordando que $\sum_{v \in V} d(v) = 2|E|$, entonces el conjunto de instrucciones del algoritmo toman tiempo $O(1), O(|V|)$ y $O(|E|)$. Concluyendo de esta forma que el algoritmo toma tiempo $O(|V| + |E|)$.

Para el análisis de espacio, observemos que al final la función de parentesco p induce un árbol generador de G, por lo que el tamaño de p es $O(|V|)$. Además, observemos que cada llamada a la subrutina 3 corresponde a un vértice de la gráfica

que estamos explorando, y como no podemos explorar más de una vez a cada vértice, entonces el tamaño de la pila recursiva es a lo más $|V|$. Concluyendo así que el algoritmo 2 utiliza espacio $O(|V|)$. □

1.8.3. Sobre las funciones en los algoritmos

En este trabajo, vamos a considerar a una función como una estructura de datos que nos permite guardar relaciones entre elementos, con la principal ventaja de que el agregar y acceder a las relaciones nos toma tiempo constante. Por ejemplo, en la función de parentesco p de BFS, la instrucción $p(v)$ implica que estamos accediendo al elemento con el que relacionamos a v bajo p; en este caso, $p(v)$ es el vértice *padre* de v. De igual forma, la instrucción $p(v) \leftarrow u$ implica que estamos asociando a v con u bajo p; dado el contexto, lo anterior indica que u es el *padre* de v.

Dentro de la programación, a estas estructuras de datos las conocemos como **diccionarios**. En nuestro contexto, no hay diferencia alguna entre un diccionario y una función. Sin embargo, a lo largo del trabajo las vamos a estar llamando como funciones. La razón principal es para facilitar la lectura. Dado que son funciones, el definir el dominio y contradominio de éstas nos ayuda a especificar de una mejor manera cuál es la relación que está representando y cómo la vamos a utilizar.

1.8.4. Algoritmo para ordenar multiconjuntos

Definimos a un **multiconjunto** M como un conjunto en donde permitimos la repetición de elementos. Formalmente, un multiconjunto M es una pareja ordenada (X, m) donde X es un conjunto y $m\colon X \to \mathbb{N}$, de tal forma que dado $x \in X$, $m(x)$ es el número de ocurrencias de x en M. De igual forma, definimos a la cardinalidad de un multiconjunto M como $|M| = \sum_{x \in X} m(x)$. Para facilitar la escritura, presentaremos a los multiconjuntos de la misma forma que a los conjuntos. Por ejemplo, $M = \{0, 0, 3, 1, 0\}$ es el equivalente a:

$$M = (\{0, 3, 1\}, m) \text{ donde } m(0) = 3,\ m(3) = 1,\ m(1) = 1.$$

Más adelante trabajaremos con un algoritmo que utiliza sucesiones de multiconjuntos, en donde necesitamos determinar si dos sucesiones S_1 y S_2 tienen los mismos elementos. Una forma práctica de hacerlo es ordenando ambas sucesiones y verificando que sean iguales. Sin embargo, primero necesitamos definir un orden entre multiconjuntos. Lo anterior se logra definiendo una representación en cadena para un multiconjunto, de tal forma que un multiconjunto A es menor a otro multiconjunto B si y sólo si la representación en cadena de A es lexicográficamente menor a

la representación en cadena de B, donde el orden lexicográfico es el orden que encontramos en un diccionario. La representación es una cadena que contiene a todos los elementos del multiconjunto en un orden de menor a mayor. Por ejemplo, la representación en cadena del multiconjunto $\{0, 1, 3, 3, 2, 0, 0\}$ es 0001233. Adicionalmente, dada una cadena c_i, definimos a $c_{i,\ell}$ como el carácter en la posición ℓ en la cadena c_i.

Presentamos el algoritmo 4, en donde dado un multiconjunto M, el algoritmo regresa la representación en cadena de M.

Algoritmo 4: Multiconjunto a Cadena

Input: Un multiconjunto $M = (X, m)$ cuyos elementos son naturales.
Output: La representación en cadena c de M.

```
1  max ← 0
2  c ← ε (la cadena vacía)
3  for x ∈ X do
4  |  if x > max then
5  |  |  max ← x
6  |  end
7  end
8  S ← un arreglo de longitud max
9  for x ∈ X do
10 |  S[x] ← m(x)
11 end
12 for i ∈ {0, ..., max − 1} do
13 |  Concatena S[i] veces i a c
14 end
15 return c
```

Proposición 1.8.5. *Sean M_1 y M_2 dos multiconjuntos y sean c_1 y c_2 las cadenas obtenidas al ejecutar el algoritmo 4 con los multiconjuntos anteriores, respectivamente. Los multiconjuntos M_1 y M_2 son iguales si y sólo si c_1 y c_2 son iguales.*

Demostración. Sean $M_1 = (X_1, m_1)$ y $M_2 = (X_2, m_2)$. Sin pérdida de generalidad, veamos qué ocurre cuando le pasamos M_1 como entrada al algoritmo 4. Primero, definimos a nuestra cadena c como la cadena vacía y en las líneas 3-7 recorremos a todos los elementos de X_1 en busca del elemento máximo k. Una vez que lo encontramos, creamos un arreglo S de tamaño k. Volvemos a recorrer a X, y por cada $x \in X$ guardamos el valor $m_1(x)$ en la localidad x del arreglo S. Después, recorremos

el arreglo S en orden, y por cada posición i del arreglo, concatenamos $S[i]$ veces el elemento i a la cadena c. Dado que recorremos el arreglo en orden, entonces la cadena c se ve de la siguiente forma:

$$c = c_1 c_2 c_3 \dots c_n.$$

En donde $c_i \in X$ y se cumple que $c_{i-1} \leq c_i$ para todo $i \in \{1, \dots, n\}$, además $n = |M_1|$. En otras palabras, la cadena c contiene a todos los elementos de M_1 en orden de menor a mayor.

Si M_1 y M_2 son iguales, por el argumento anterior, podemos observar que c_1 contiene a todos los elementos de M_1 en orden, de menor a mayor, y para cada $x \in X_1$, el elemento x se repite $m_1(x)$ veces en c_1. Lo mismo sucede para c_2 con X_2 y m_2. Dado que $X_1 = X_2$ y $m_1(x) = m_2(x)$ para todo $x \in X_1$, entonces c_1 y c_2 tienen los mismos elementos, la misma cantidad de veces, en el mismo orden, es decir, son iguales.

Ahora veamos qué ocurre cuando M_1 y M_2 no son iguales. Si $X_1 \neq X_2$, entonces c_1 tendrá al menos un carácter que no se encuentra en c_2, ya que c_1 contiene a todos los elementos de M_1; por lo tanto c_1 y c_2 son distintos. Por otro lado, si $X_1 = X_2$ entonces existe $x \in X_1$ tal que $m_1(x) \neq m_2(x)$, por lo tanto el carácter x se encuentra $m_1(x)$ veces en c_1 mientras que en c_2 se encuentra $m_2(x)$ veces; es decir, las cadenas no son iguales.

Concluyendo de esta forma que M_1 y M_2 son iguales si y sólo si c_1 y c_2 son iguales. □

De igual forma, presentamos el algoritmo 5, en donde dada una cadena c la cual es la representación de un multiconjunto M, el algoritmo regresa M.

Proposición 1.8.6. *Sean c_1 y c_2 representaciones en cadena de multiconjuntos, y sean M_1 y M_2 los multiconjuntos obtenidos al ejecutar el algoritmo 5 con las cadenas anteriores respectivamente. Las cadenas c_1 y c_2 son iguales si y sólo si M_1 y M_2 son iguales.*

Demostración. Sean c_1 y c_2 representaciones en cadena de multiconjuntos. Sin pérdida de generalidad, veamos qué ocurre cuando le pasamos c_1 como entrada al algoritmo 5. Primero definimos a los conjuntos vacíos X y m en las líneas 2-3. Después, en las líneas 4-11, recorremos a todos los caracteres de c_1 en orden. Sea $c_{1,i}$ el carácter que estamos recorriendo. Si $c_{1,i} \notin X$, entonces lo agregamos a X y definimos a $m(c_{1,i}) = 1$. Si $c_{1,i} \in X$, entonces aumentamos en uno el valor de $m(c_{1,i})$. Observemos que de esta manera, X es el conjunto de caracteres de c_1 y m es la función que nos

Algoritmo 5: Cadena a Multiconjunto

Input: Una cadena c la cual es la representación de un multiconjunto M.
Output: El multiconjunto M.

```
1  ℓ ← |c|
2  X ← ∅
3  m ← ∅
4  for i ∈ {0, ..., ℓ − 1} do
5  |  if c[i] ∉ X then
6  |  |  X ← X ∪ {c[i]}
7  |  |  m(c[i]) ← 1
8  |  else
9  |  |  m(c[i]) ← m(c[i]) + 1
10 |  end
11 end
12 return (X, m)
```

indica cuántas veces se repite cada carácter de X; en otras palabras, $M = (X, m)$ es el multiconjunto correspondiente a c_1.

Si c_1 y c_2 son iguales, por el argumento anterior podemos observar que los multiconjuntos $M_1 = (X_1, m_1)$ y $M_2 = (X_2, m_2)$ obtenidos por el algoritmo son iguales. Lo anterior se debe a que como c_1 y c_2 son iguales, entonces tienen al mismo conjunto de caracteres la misma cantidad de veces en el mismo orden, por ende $X_1 = X_2$ y $m_1 = m_2$.

Veamos ahora qué pasa si c_1 y c_2 no son iguales. Si c_1 y c_2 no tienen el mismo conjunto de caracteres, entonces $X_1 \neq X_2$. Por otro lado, si tienen al mismo conjunto de caracteres pero no la misma cantidad de veces, entonces para algún $c \in X$ tenemos que $m_1(c) \neq m_2(c)$. Por último, observemos que no es posible que c_1 y c_2 tengan el mismo conjunto de caracteres la misma cantidad de veces pero en un orden distinto, pues las cadenas c_1 y c_2 son representaciones de los elementos de un multiconjunto en un orden de mayor a menor.

Concluyendo de esta forma, que c_1 y c_2 son iguales si y sólo si M_1 y M_2 son iguales. □

De esta forma, presentamos una manera en la que podemos convertir multiconjuntos a cadenas y viceversa. Veamos qué tan eficientes son los algoritmos anteriores.

Proposición 1.8.7. *Dado $M = (X, m)$ un multiconjunto, el algoritmo 4 regresa la*

representación en cadena de M en tiempo $O(|X| + k)$ y espacio $O(k)$ donde $k = \max\{x : x \in X\}$.

Demostración. Para el análisis de tiempo. En las líneas 3-7 buscamos al elemento máximo k de X, lo cual lo hacemos en tiempo $O(|X|)$. Después creamos un arreglo de tamaño k, esto lo podemos hacer en tiempo $O(1)$. En las líneas 9-11 volvemos a recorrer a todos los elementos de X para guardar en la localidad x de S al elemento $m(x)$, esto toma tiempo $O(|X|)$. Finalmente, recorremos al arreglo S en orden para construir a la representación en cadena de M, lo cual toma tiempo $O(k)$. Concluyendo de esta forma que el algoritmo toma tiempo $O(|X| + k)$.

Para el análisis en espacio. La variable S es un arreglo de tamaño k, por lo que ocupa espacio $O(k)$. Todas las demás variables que se usan durante el algoritmo son constantes, por lo que usan espacio $O(1)$. Así, el algoritmo utiliza espacio $O(k)$. □

Proposición 1.8.8. *Dada c, la representación en cadena de un multiconjunto $M = (X, m)$, el algoritmo 5 regresa a M en tiempo $O(|c|)$ y espacio $O(|X|)$.*

Demostración. Para el análisis en tiempo. Primero, creamos dos conjuntos vacíos X y m en tiempo $O(1)$. Después, recorremos a toda la cadena c una sola vez. En donde por cada carácter c_i que recorremos, si c_i no está en X, lo agregamos e indicamos que $m(c_i) = 1$, de lo contrario sólo aumentamos en uno el valor de $m(c_i)$. Las operaciones de asignación y comparación toman tiempo $O(1)$, por lo tanto, todo lo anterior toma en total tiempo $O(|c|)$, dado que en un inicio la longitud de la cadena es de $|c|$.

Para el análisis en espacio. El conjunto X va a guardar a todos los elementos del multiconjunto. De igual manera, por cada $x \in X$ tenemos un valor $m(x)$. Por lo tanto, las variables que utilizamos en el algoritmo utilizan espacio $O(|X|)$. □

Ahora tenemos que ver de qué forma podemos ordenar a estas cadenas de multiconjuntos. Presentamos el algoritmo 8, el cual ordena cadenas lexicográficamente. Cabe mencionar que el algoritmo utiliza dos subrutinas, el algoritmo 6 y el algoritmo 7.

Proposición 1.8.9. *Sea S una sucesión de cadenas cuyos caracteres son enteros entre 0 y $m - 1$, y sea ℓ_{max} la longitud de la cadena más larga en S. El algoritmo 6 devuelve una función $cars$ tal que, dado $i \in \{1, \ldots, \ell_{max}\}$, $cars(i)$ es la sucesión ordenada del conjunto de caracteres que se encuentran en la posición i de las cadenas de S. El algoritmo 6 se ejecuta en tiempo $O(\ell_{total} + m)$ y usando espacio $O(\ell_{total})$, donde $\ell_{total} = \sum_{c \in S} |c|$.*

Demostración. Veamos que efectivamente la función $cars$ que regresa el algoritmo 6 esta bien definida. En otras palabras, sea $j \in \{1, \ldots, \ell_{max}\}$, tenemos que $cars(j)$ es

Algoritmo 6: Categorizar Caracteres

Input: Una sucesión S de cadenas, cuyos caracteres están entre 0 y $m-1$.
Output: La función $cars\colon \mathbb{N} \to \mathcal{P}(\mathbb{N})$, donde $cars(i)$ es la sucesión ordenada del conjunto de caracteres que se encuentran en la posición i de las cadenas de S.

```
1  cars ← ∅
2  ℓ_max ← máx {|c|: c ∈ S}
3  for ℓ ∈ {1, ..., ℓ_max} do
4  |  X ← ∅, m ← ∅
5  |  for c_i ∈ S do
6  |  |  if |c_i| < ℓ o c_{i,ℓ} ∈ X then
7  |  |  |  continue
8  |  |  end
9  |  |  else
10 |  |  |  X ← X ∪ {c_{i,ℓ}}
11 |  |  |  m(c_{i,ℓ}) = 1
12 |  |  end
13 |  end
14 |  c' ← MulticonjuntoACadena((X, m))
15 |  S' ← ()
16 |  for c_i ∈ c' do
17 |  |  agrega c_i al final de S'
18 |  end
19 |  cars(ℓ) ← S'
20 end
21 return cars
```

la sucesión ordenada del conjunto de caracteres que se encuentran en la posición j de las cadenas de S.

Primero, en la línea 2, obtenemos la longitud de la cadena más larga en S y guardamos este valor en ℓ_{max}. Después entramos en el ciclo de la línea 3, en donde recorremos a cada $\ell \in \{1, \ldots, \ell_{max}\}$ en orden, por lo que eventualmente vamos a recorrer al valor j.

Consideremos la j-ésima iteración del ciclo de la línea 3. En la línea 4 definimos a los conjuntos vacíos X y m. Recorremos S en orden y por cada $c_i \in S$ primero verificamos que su longitud sea mayor o igual a j, dado que queremos acceder al

j-ésimo carácter de c_i. Si lo anterior no se cumple, entonces continuamos con la siguiente cadena en S. De lo contrario, verificamos que el carácter en la j-ésima posición de c_i aún no se encuentre en X, en el caso de que sí se encuentra entonces continuamos con la siguiente cadena en S, si no agregamos $c_{i,j}$ a X e indicamos que $m(c_{i,j}) = 1$. Al finalizar el ciclo de las líneas 5-13, el conjunto X se encuentra definido de la siguiente forma:

$$X = \{c_{i,j} \colon c_i \in S, |c_i| \geq j\}.$$

Es decir, X es el conjunto de caracteres presentes en la j-ésima posición de las cadenas de S cuya longitud es mayor o igual a j. Observemos que $M = (X, m)$ es un multiconjunto que se comporta como conjunto, dado que no tenemos repetición de elementos. Al aplicarle el algoritmo 4 a M, lo que obtenemos es una cadena c' que contiene a los elementos de X en orden, lo cual sabemos que es cierto por la proposición 1.8.5. De esta forma, al recorrer c' carácter por carácter, y agregarlos a una sucesión S, al final S contendrá a todos los elementos de X en orden. En la línea 19 asignamos S a $cars(j)$, y de esta forma concluimos que $cars(j)$ es la sucesión ordenada de los conjuntos de caracteres presentes en la j-ésima posición de las cadenas de S; la función $cars$ ésta bien definida.

Para el análisis de tiempo. El valor ℓ_{max} lo podemos obtener en tiempo $O(|S|)$. Veamos qué ocurre en cada iteración i del ciclo de las líneas 3-20. En las líneas 5-13, por cada $c \in S$, recorremos únicamente a los caracteres que se encuentran en la posición i de c. Es decir, recorremos una sola vez a cada carácter de las cadenas de S. Por lo que al finalizar el ciclo, la operación anterior en total toma tiempo $O(\ell_{total})$ donde $\ell_{total} = \sum_{c \in S} |c|$. En la línea 14, donde ejecutamos el algoritmo 4, en el peor de los casos cada cadena $c \in S$ tiene un carácter distinto en la posición i, por lo que volvemos a recorrer a cada carácter de las cadenas de S una sola vez, por ende en total la operación anterior toma tiempo $O(\ell_{total} + m)$. De manera similar, dado el peor de los casos, recorrer a todas las cadenas c' que obtengamos será el equivalente a recorrer una vez más a todos los caracteres de las cadenas de S, por lo tanto, construir a las sucesiones S' tomará en total tiempo $O(\ell_{total})$. Concluyendo de esta forma que el algoritmo en total toma tiempo $O(\ell_{total} + m)$.

Para el análisis de espacio. En el peor de los casos, para toda posición $i \in \{1, \ldots, \ell_{max}\}$, todas las cadenas de S tienen caracteres distintos. Dado que $cars$ guarda a estos caracteres de forma ordenada, entonces el tamaño de $cars$ es de $O(\ell_{total})$.

□

Algoritmo 7: Categorizar Cadenas

Input: Una sucesión S de cadenas de cuyos caracteres están entre 0 y $m-1$.
Output: La función *long*, donde $long(i)$ es la sucesión de cadenas de S cuya longitud es i.

```
1  long ← ∅
2  ℓ_max ← máx {|c| : c ∈ S}
3  A ← un arreglo de longitud ℓ_max cuyos elementos son conjuntos vacíos
4  for c ∈ S do
5  |  agregar c a A[|c|]
6  end
7  for ℓ ∈ {1, ..., ℓ_max} do
8  |  long(ℓ) ← A[ℓ]
9  end
10 return long
```

Proposición 1.8.10. *Sea S una sucesión de cadenas cuyos caracteres son enteros entre 0 y $m-1$, y sea ℓ_{max} la longitud de la cadena más larga en S. El algoritmo 7 devuelve una función long, donde dado $\ell \in \{1, \ldots, \ell_{max}\}$, $long(\ell)$ es el conjunto de cadenas en S cuya longitud es ℓ. El algoritmo 7 toma tiempo $O(|S| + \ell_{max})$ y usa espacio $O(|S|)$.*

Demostración. Veamos que efectivamente la función *long* que regresa el algoritmo 7 esta bien definida. En otras palabras, sea $\ell \in \{1, \ldots, \ell_{max}\}$, buscamos que $long(\ell)$ sea el conjunto de cadenas de S cuya longitud es ℓ.

Primero, en la línea 2 obtenemos el valor ℓ_{max} el cual es la longitud de la cadena más larga en S. Después, en la línea 3, definimos un arreglo de tamaño ℓ_{max} cuyos elementos son conjuntos vacíos. En la línea 4-6 recorremos a todas las cadenas de S, y por cada cadena $c \in S$, guardamos a c en el conjunto $A[|c|]$. Observemos que al finalizar el ciclo anterior, para cada $i \in \{1, \ldots, \ell_{max}\}$ tenemos que $A[\ell]$ es el conjunto de cadenas cuya longitud es ℓ. Por último en el ciclo de las líneas 7-9, para cada $i \in \{1, \ldots, \ell_{max}\}$ asignamos $long(i) = A[i]$; por ende, $long(\ell) = A[\ell]$ es el conjunto que contiene a todas las cadenas de S cuya longitud es ℓ.

Para el análisis de tiempo. Obtener el valor ℓ_{max} lo hacemos en tiempo $O(|S|)$. Crear un arreglo de tamaño ℓ_{max} cuyos elementos sean conjuntos vacíos lo podemos hacer en tiempo $O(1)$. Sea $c \in S$, agregar c a $A[|c|]$ lo podemos hacer en tiempo $O(1)$. Por ende, las operaciones de las líneas 4-6 toman tiempo $O(|S|)$. Finalmente, recorremos a cada $\ell \in \{1, \ldots, \ell_{max}\}$ y asignamos $long(\ell) = A[\ell]$, en total la operación

anterior toma $O(\ell_{max})$. Concluyendo así que el algoritmo toma tiempo $O(|S|+\ell_{max})$.

Para el análisis en espacio. Observemos que al finalizar la ejecución del algoritmo, la función *long* contendrá únicamente a todos los elementos de S, por ende el tamaño de *long* es $O(|S|)$; el algoritmo utiliza espacio $O(|S|)$. □

Proposición 1.8.11. *Dada una sucesión de cadenas S cuyos caracteres son enteros que están entre 0 y $m-1$, el algoritmo 8 produce un ordenamiento lexicográfico de las cadenas de S en tiempo $O(\ell_{total}+m)$ y usando espacio $O(\ell_{total})$, donde $\ell_{total} = \sum_{c\in S}|c|$.*

Demostración. Veamos que el algoritmo 8 efectivamente produce un ordenamiento lexicográfico de las cadenas de S. En otras palabras, sea $S' = (c'_1, c'_2, \ldots, c'_n)$ la sucesión que regresa el algoritmo, veamos que S' es una pemutación de los elementos de S tal que $c'_{i-1} \leq c'_i$ para todo $i \in \{2, \ldots, n\}$.

Recordemos que las funciones *cars* y *long* definidas en las líneas 4-5 funcionan correctamente, es decir, dados $i \in \{1, \ldots, \ell_{max}\}$, $cars(i)$ es la sucesión ordenada del conjunto de caracteres en la i-ésima posición de las cadenas de S, y $long(i)$ es el conjunto de cadenas de S cuya longitud es i.

Las líneas 7-24 del algoritmo se encargan de colocar a las cadenas de S en un orden lexicográfico. Veamos que lo anterior es cierto, proponemos a la siguiente invariante del ciclo. Después de la i-ésima iteración del ciclo en Q se encuentran las cadenas de S de longitud mayor a $\ell - i$ ordenadas con respecto a sus últimos $\ell - i$ caracteres.

Si el número de iteraciones empieza en cero, antes de entrar a la iteración, en Q se encuentran todas las cadenas de longitud mayor a ℓ_{max}, dado que Q esta vacía en este punto entonces lo anterior es cierto. Así, suponiendo que después de la i-ésima iteración en Q se encuentran las cadenas de S de longitud mayor a $\ell - i$ ordenadas lexicográficamente con respecto a sus últimos $\ell - i$ caracteres, veamos qué ocurre en la $(i+1)$-ésima iteración. En las líneas 8-10, recorremos a todas las cadenas c_i de longitud $j = \ell - (i+1)$ y las agregamos al final de cola $A[c_{i,j}]$. Después recorremos a Q en orden y por cada $c_i \in Q$ colocamos a c_i en la cola $A[c_{i,j}]$. Así, para todo $k \in \{0, \ldots, m-1\}$, en $A[k]$ se encuentran las cadenas de S cuya longitud es mayor o igual a j tal que su j-ésimo carácter es k.

Observemos que si para alguna $k \in \{0, \ldots, m-1\}$, dos elementos a y b son colocados en la misma cola $A[k]$ de tal forma que a fue agregado antes que b, entonces tenemos los siguientes casos. Si $a, b \in long(j)$, entonces tanto a como b son lexicográficamente iguales con respecto a su último carácter. Si $a \in long(j)$ y $b \in Q$, entonces a es lexicográficamente menor a b dado que $|a| < |b|$. Por último si $a, b \in Q$, entonces nuestra invariante nos asegura que a es lexicográficamente menor o igual a b con respecto a los últimos $j-1$ caracteres, por lo tanto a sigue siendo lexico-

gráficamente menor o igual a b en $A[k]$. De esta manera concluimos que para todo $k \in \{0, \ldots, m-1\}$, los elementos de $A[k]$ se encuentran ordenados lexicográficamente. Por lo tanto, en las líneas 16-22, al recorrer $cars(j)$ en orden y colocar los elementos en Q, vamos a respetar el orden que se definió para los caracteres anteriores además de ahora considerar el orden impuesto por el j-ésimo carácter. Por ende, al finalizar la iteración, los elementos de Q están ordenados lexicográficamente con respecto a los últimos $j = \ell - (i+1)$ caracteres. Así, al finalizar el ciclo anterior, en Q se encuentran las cadenas de S pero ordenadas lexicográficamente.

Finalmente, en las líneas 25-30 colocamos en orden a todos los elementos de Q en una nueva sucesión S', de tal forma que S' ahora contiene a todos los elementos de S pero ordenados lexicográficamente.

Para el análisis en tiempo. Podemos obtener el valor ℓ_{max} en tiempo $O(|S|)$. Por otro lado, gracias a las proposiciones 1.8.9 y 1.8.10, sabemos que los algoritmos 6 y 7 toman tiempo $O(\ell_{total} + m)$ y $O(|S| + \ell_{max})$ respectivamente, donde $\ell_{total} = \sum_{c \in S} |c|$. Veamos entonces cuánto tiempo toman las líneas 7-24. Observemos que en cada iteración ℓ del ciclo, en las líneas 8-15, recorremos únicamente a las cadenas de S cuya longitud es mayor o igual a ℓ y para cada una nos enfocamos únicamente en el carácter que se encuentra en la ℓ-ésima posición. Cabe mencionar que estas cadenas se pueden encontrar en $long(\ell)$ si es que su longitud es exactamente ℓ, de otro modo aquellas cadenas cuya longitud sea mayor a ℓ se encuentran en Q. Por otro lado, en las líneas 16-22, recorremos al arreglo A pero sólo en las posiciones en donde sabemos que habrá cadenas, este último lo sabemos por el valor que guarda $cars(\ell)$. Así, recorremos una vez más a aquellas cadenas que tienen un carácter en la ℓ-ésima posición. De esta forma, en las líneas 7-24, recorremos a todos los caracteres de las cadenas de S un total de dos veces. En otras palabras, esta parte del algoritmo toma tiempo $O(2\ell_{total}) = O(\ell_{total})$.

Por último, para construir S' únicamente recorremos a la cola ordenada Q, y dado que Q en este punto contiene a todos los elementos de S en orden lexicográfico entonces esta operación toma tiempo $O(|S|)$.

Dado que $\ell_{max} \leq \ell_{total}$ entonces $O(\ell_{max} + m) \subseteq O(\ell_{total} + m)$. De igual manera $|S| \leq \ell_{total}$, por lo que $O(|S| + \ell_{max}) \subseteq O(\ell_{total}) \subseteq O(\ell_{total} + m)$. Concluyendo de esta forma que el algoritmo toma tiempo $O(\ell_{total} + m)$.

Para el análisis en espacio. La función $cars$ utiliza espacio $O(\ell_{total})$, y la función $long$ utilza espacio $O(|S|)$. En el algoritmo, la cola Q y el arreglo A guardan únicamente a las cadenas de S sin repetirlas en algún momento, por lo tanto el tamaño máximo que alcanzan estas variables es de $O(|S|)$. La sucesión S' al final contiene a todas las cadenas de S pero en orden lexicográfico, por lo que su tamaño es de $O(|S|)$. Recordando que $|S| \leq \ell_{total}$, entonces nuestro algoritmo utiliza espacio $O(\ell_{total})$. □

Proposición 1.8.12. *Sea S una sucesión de multiconjuntos, el algoritmo 9 regresa un ordenamiento de S en tiempo $O(\mathcal{M}+k)$ y espacio $O(\mathcal{M})$, donde $\mathcal{M} = \sum_{M\in S} |M|$ y $k = \max\{\max\{X\} : (X,m) \in S\}$.*

Demostración. Veamos que efectivamente la sucesión S' que regresa el algoritmo es un ordenamiento de S.

Primero, definimos una sucesión vacía S_c. Después recorremos a todos los elementos de S en orden, los convertimos a su respectiva representación en cadena, y los agregamos a S_c. De esta forma, al finalizar el recorrido S_c contiene a la representación en cadena de todos los multiconjuntos de S. Después obtenemos a la sucesión S'_c, la cual es un ordenamiento lexicográfico de las cadenas de S_c utilizando el algoritmo 8. Sabemos que este ordenamiento es correcto por la proposición 1.8.11.

Definimos a la sucesión vacía S'. Recorremos a todos los elementos de S'_c en orden, los convertimos de nuevo a multiconjuntos y los agregamos a S'. De esta forma, al finalizar el recorrido, S'_m contiene a los elementos originales de S pero un orden definido por S'_c; es decir, el orden lexicográfico de sus representaciones en cadena. Así, S' es un ordenamiento de S.

Para el análisis de tiempo. Recordemos que convertir multiconjuntos a cadenas (algoritmo 4) toma tiempo $O(|X|+k)$ donde $k = \max\{X\}$ para cada $(X,m) \in S$. Por ende, al finalizar el recorrido de las líneas 2-4, hemos recorrido a todos los elementos contenidos en los multiconjuntos de S. Además, de recorrer un número constante de veces a un arreglo de a lo más longitud $k' = \max\{\max\{X\} : (X,m) \in S\}$ para construir a las cadenas. Así, las líneas 2-4 del algoritmo toman tiempo $O(\mathcal{M}+k')$, donde $\mathcal{M} = \sum_{M\in S} |M|$. Observemos entonces que todos los multiconjuntos de S tienen valores entre 0 y k', de esta forma, ordenar a las cadenas obtenidas en la sucesión S'_c toma tiempo $O(\ell_{total}+k')$ donde $\ell_{total} = \sum_{c\in S'_c} |c|$. Además, sabemos que $|c| = |M|$ para cada $M \in S$. Por ende, $\ell_{total} = \mathcal{M}$. En otras palabras, ordenar a las representaciones en cadena de S toma tiempo $O(\mathcal{M}+k')$. De manera similar al algoritmo 4, para convertir a todas las cadenas de nuevo a multiconjuntos (algoritmo 5), tenemos que recorrer una vez a todos los caracteres contenidos en las cadenas de S'_c. Es decir, las líneas 8-11 del algoritmo toman tiempo $O(\mathcal{M})$. Concluyendo de esta forma que nuestro algoritmo tiene un conjunto de operaciones que toman tiempo $O(\mathcal{M}+k')$ y $O(\mathcal{M})$; nuestro algoritmo toma tiempo $O(\mathcal{M}+k')$.

Para el análisis de espacio. Al final en S'_c vamos a guardar las representaciones en cadena de cadena de cada multiconjunto en S, por lo que la suma de las longitudes de todas estas cadenas es $\mathcal{M}$. Al ordenar las cadenas (algoritmo 8) utilizamos espacio $O(\mathcal{M})$. Finalmente, para convertir cadenas a multiconjuntos tenemos un argumento similar al inicio, la sucesión S' contiene a todos los multiconjuntos de S pero en orden, por lo que el tamaño de S' es $O(\mathcal{M})$. □

Algoritmo 8: Ordenar Cadenas

Input: Una sucesión S de cadenas de cuyos caracteres están entre 0 y $m-1$.
Output: Un ordenamiento lexicográfico S' de S.

```
1  Q ← []
2  A ← un arreglo de longitud m cuyos elementos son colas
3  ℓ_max ← máx {|c| : c ∈ S}
4  cars ← CategorizarCaracteres(S)
5  long ← CategorizarCadenas(S)
6  ℓ ← ℓ_max
7  while ℓ ≥ 1 do
8  |  for c_i ∈ long(ℓ) do
9  |  |  agrega c_i al final de A[c_{i,ℓ}]
10 |  end
11 |  while Q no esté vacía do
12 |  |  elegir la cabeza c_i de Q
13 |  |  agrega c_i al final de A[c_{i,ℓ}]
14 |  |  elimina c_i de Q
15 |  end
16 |  for j ∈ cars(ℓ) do
17 |  |  while A[j] no esté vacía do
18 |  |  |  elegir la cabeza c_i de A[j]
19 |  |  |  agrega c_i al final de Q
20 |  |  |  elimina c_i de A[j]
21 |  |  end
22 |  end
23 |  ℓ ← ℓ − 1
24 end
25 S' ← ()
26 while Q no esté vacía do
27 |  elegir la cabeza c_i de Q
28 |  agregar c_i al final de S
29 |  eliminar c_i de Q
30 end
31 return S'
```

Algoritmo 9: Ordenar Multiconjuntos

Input: Una sucesión S de multiconjuntos de cuyos valores están entre 0 y $m-1$.

Output: Un ordenamiento de S.

```
1  S_c ← ()
2  for m ∈ S do
3  |   c ← MulticonjuntoACadena(m)
4  |   agregar c al final de S_c
5  end
6  S'_c ← OrdenarCadenas(S_c)
7  S' ← ()
8  for c_i ∈ S'_c do
9  |   m' ← CadenaAMulticonjunto(c_i)
10 |   agregar m' al final de S'_m
11 end
12 return S'
```

Capítulo 2

Formatos

Al momento de trabajar con gráficas, puede que estemos interesados en ser capaces de guardarlas de forma persistente en memoria, ya sea en un archivo binario o de texto. Por ejemplo, podría ser de utilidad tener a todos los árboles no isomorfos de cierta cantidad de vértices en un archivo. En este tipo de situaciones, nos interesa encontrar un *buen* algoritmo de conversión, de tal manera que nuestras gráficas ocupen la menor cantidad de espacio posible sin llegar a perder información.

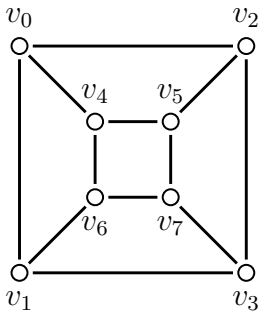
Los formatos que presentamos a continuación son los que generalmente se usan para guardar gráficas en memoria. Estos convierten una gráfica a una representación en cadena de texto en donde todos los caracteres son ASCII (*American Standard Code for Information Interchange*). Estos formatos fueron descritos por Brendan McKay, el cual define algoritmos para la conversión de cadenas de texto a gráficas [4]. Sin embargo, no hay una definición formal para el proceso inverso, en este capítulo describimos el proceso necesario para poder convertir gráficas y digráficas a texto ASCII.

2.1. Graph6

El formato `graph6` nos permite guardar una gráfica simple G dentro de un archivo de texto (de forma persistente), esto lo logramos guardando el número de vértices de la gráfica y los elementos del triángulo superior de la matriz de adyacencia correspondiente a la gráfica; a continuación explicamos con detalle este proceso. Cabe mencionar que por limitaciones prácticas, el máximo número de vértices que podemos representar para una gráfica es de $2^{36} - 1$.

Sea $G = (V_G, E_G)$ la gráfica simple que deseamos guardar, tal que $|V_G| = n$ y $A_G = (a_{ij})$ es la matriz de adyacencia correspondiente a G. Hay dos cosas que notar

sobre A_G. La primera es que la diagonal siempre va a ser cero, pues los vértices no pueden ser adyacentes consigo mismos. La segunda es que la matriz es simétrica, es decir, para $v_i, v_j \in V_G$, si v_i es adyacente a v_j entonces también viceversa, por lo tanto $a_{ij} = a_{ji}$. Es por este motivo que toda la información que necesitamos sobre las adyacencias de la gráfica se pueden encontrar en el triángulo superior de A_G.



G

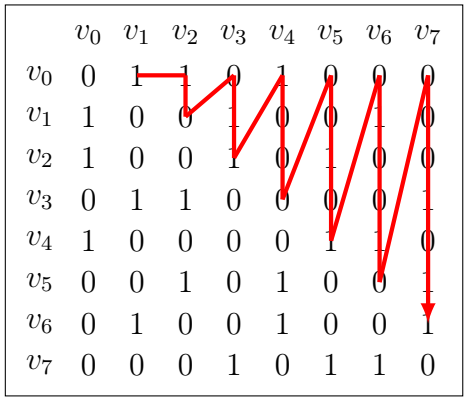
	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_0	0	1	1	0	1	0	0	0
v_1	1	0	0	1	0	0	1	0
v_2	1	0	0	1	0	1	0	0
v_3	0	1	1	0	0	0	0	1
v_4	1	0	0	0	0	1	1	0
v_5	0	0	1	0	1	0	0	1
v_6	0	1	0	0	1	0	0	1
v_7	0	0	0	1	0	1	1	0

A_G

Figura 2.1: Gráfica G junto con su matriz de adyacencia A_G.

En la Figura 2.1 indicamos a la diagonal cero como la línea verde, de igual manera marcamos de azul al triángulo inferior de la matriz y de rojo al triángulo superior.

La forma en la que recorremos el triángulo superior de la matriz de adyacencia es columna por columna; es decir, primero recorremos a todos los elementos de la primera columna, después a los de la segunda y así hasta terminar con la columna n.



	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_0	0	1	1	0	1	0	0	0
v_1	1	0	0	1	0	0	1	0
v_2	1	0	0	1	0	1	0	0
v_3	0	1	1	0	0	0	0	1
v_4	1	0	0	0	0	1	1	0
v_5	0	0	1	0	1	0	0	1
v_6	0	1	0	0	1	0	0	1
v_7	0	0	0	1	0	1	1	0

A_G

Figura 2.2: Recorrido del triángulo superior de la matriz A_G.

De esta forma, podemos construir un vector de tamaño $\frac{n(n-1)}{2}$. Por ejemplo, el vector obtenido al recorrer la matriz de la Figura 2.2 es el siguiente:

$$x = [1,1,0,0,1,1,1,0,0,0,0,0,1,0,1,0,1,0,0,1,0,0,0,0,1,0,1,1].$$

Ahora, nuestro objetivo es guardar este vector x como una cadena de bits. Para ello, primero observamos que podemos representar a x como un número binario. Después, necesitamos que la longitud de este binario sea múltiplo de 6, así que para ello le podemos agregar ceros hasta alcanzar la longitud deseada. Por convención, al tratarse de los bits correspondientes al conjunto de aristas de la gráfica, los ceros son agregados a la derecha. De esta forma, podemos dividir nuestro binario en grupos de 6. Siguiendo el ejemplo de la Figura 2.2, obtenemos lo siguiente:

$$x_{bin} = 110011100000101010010000 1011,$$

$$x'_{bin} = 110011\ 100000\ 101010\ 010000\ 101100.$$

Suponiendo que nuestros binarios siguen el orden *Big Endian*[1], una vez que tenemos nuestros grupos, los convertimos a decimal. Para asegurarnos que nuestros bytes puedan ser imprimibles (es decir que exista un carácter ASCII correspondiente al valor del byte) sumamos a cada grupo 63, y así el rango del valor de los bytes está

[1]El orden *Big Endian* indica que el primer bit del número binario es el más significativo. Y por ende, el último es el menos significativo, es decir, aquel al que le corresponde la primera potencia de 2, $2^0 = 1$.

entre 63 y 126; todos los bytes de este rango son imprimibles. El grupo de bytes obtenidos después de esta serie de transformaciones es denotado $R(x)$, donde x es el binario cuya longitud es múltiplo de 6. Concluyendo nuestro ejemplo, el valor $R(x)$ del vector obtenido de la Figura 2.2 es:

$$dec(x'_{bin}) = 51\ 32\ 42\ 16\ 44,$$
$$R(x'_{bin}) = 114\ 95\ 105\ 79\ 107.$$

Ya que tenemos las adyacencias ahora sólo nos hace falta ver qué hacer con el número de vértices n. Para ello definimos a la función $N(n)$ de la siguiente forma:

- Si $0 \leq n \leq 62$, definimos a $N(n)$ como el byte correspondiente a $n + 63$.
- Si $63 \leq n \leq 258047$, definimos a $N(n)$ como 4 bytes de la forma forma 126 $R(n')$ donde n' es el binario de 18-bits correspondiente a n.
- Si $258048 \leq n < 2^{36}$, definimos a $N(n)$ como 8 bytes de la forma 126 126 $R(n')$ donde n' es el binario de 36-bits correspondiente a n.

Mostramos los siguientes ejemplos de $N(n)$:

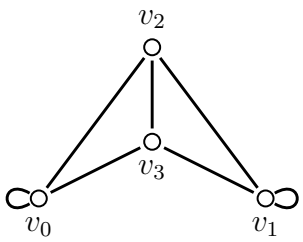
$$\begin{aligned} N(8) &= 8 + 63 = 71, \\ N(136) &= 126\ R(000000\ 000010\ 001000) = 126\ 63\ 65\ 71, \\ N(460175067) &= 126\ 126\ R(000000\ 011011\ 011011\ 011011\ 011011\ 011011) \\ &= 126\ 126\ 63\ 90\ 90\ 90\ 90\ 90. \end{aligned}$$

Y finalmente, una vez que ya tenemos $N(n)$ y $R(x)$, entonces la representación de la gráfica G en formato `graph6` es $N(n)\ R(x)$. Por ejemplo, la gráfica de la Figura 2.1 en formato `graph6` tiene los bytes 71 114 95 105 79 107, por lo tanto la representación en cadena de texto (convirtiendo cada byte a su correspondiente carácter ASCII) es `Gr_iOk`.

2.1.1. Loop6

Podemos extender el formato `graph6` para que podamos trabajar con gráficas con lazos pero sin aristas múltiples, a este nuevo formato lo llamamos `loop6`. La única diferencia es que ahora sí estamos considerando a la diagonal de la matriz de adyacencia pues esta ya no necesariamente es cero. De esta manera, el vector que ahora estaríamos obteniendo al recorrer la matriz es de tamaño $\frac{n(n+1)}{2}$. La representación de

la gráfica G en formato `loop6` es 59 $N(n)$ $R(x)$; donde el byte 59 es un identificador para gráficas que pueden tener lazos pero no multiaristas.



G'

	v_0	v_1	v_2	v_3
v_0	1	0	1	1
v_1	0	1	1	1
v_2	1	1	0	1
v_3	1	1	1	0

Figura 2.3: Gráfica G' junto con su matriz de adyacencia.

Por ejemplo, siguiendo el procedimiento mencionado anteriormente con la gráfica de la Figura 2.3, el vector que obtenemos al recorrer la matriz de adyacencia es:

$$x = [1, 0, 1, 1, 1, 0, 1, 1, 1, 0].$$

Continuando, aplicamos la serie de transformaciones que definimos para `graph6`.

$$\begin{aligned} x_{bin} &= 1011101110, \\ x'_{bin} &= 101110\ 111000, \\ dec(x'_{bin}) &= 46\ 56, \\ R(x'_{bin}) &= 109\ 119, \\ N(4) &= 67. \end{aligned}$$

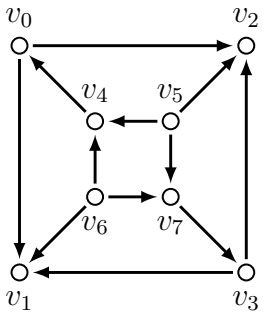
Concluyendo el ejemplo, el formato `loop6` de la gráfica de la Figura 2.3 en su grupo de bytes son 59 67 109 119, en cadena texto esto sería equivalente a `;Cmw`.

2.2. Digraph6

El formato `digraph6` nos permite guardar a una digráfica sin flechas múltiples dentro de un archivo de texto, y al igual que en el formato `graph6` guardamos el número de vértices, con la única diferencia de que ahora también guardamos a todos los elementos de la matriz de adyacencia correspondiente a la digráfica; a continuación explicamos a detalle este proceso.

Sea $D = (V_D, \mathcal{A}_D)$ la digráfica simple que deseamos guardar, tal que $|V_D| = n$ y $A_D = (a_{ij})$ es la matriz de adyacencia correspondiente a D. A comparación de las

gráficas simples, la matriz de adyacencia de una digráfica sólo cumple con que su diagonal es cero, si es que no tiene lazos; la matriz no necesariamente tiene que ser simétrica, ya que para $v_i, v_j \in V_D$, si v_i es adyacente a v_j, no necesariamente v_j es adyacente a v_i. Por esta razón es necesario que consideremos a toda la matriz.



H

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_0	0	1	1	0	0	0	0	0
v_1	0	0	0	0	0	0	0	0
v_2	0	0	0	0	0	0	0	0
v_3	0	1	1	0	0	0	0	0
v_4	1	0	0	0	0	0	0	0
v_5	0	0	1	0	1	0	0	1
v_6	0	1	0	0	1	0	0	1
v_7	0	0	0	1	0	0	0	0

A_H

Figura 2.4: Digráfica H junto con su matriz de adyacencia.

Recorremos la matriz de adyacencia renglón por renglón. Construyendo así un vector de tamaño n^2 que contendrá toda la información de las flechas de la digráfica.

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_0	0	1	1	0	0	0	0	0
v_1	0	0	0	0	0	0	0	0
v_2	0	0	0	0	0	0	0	0
v_3	0	1	1	0	0	0	0	0
v_4	1	0	0	0	0	0	0	0
v_5	0	0	1	0	1	0	0	1
v_6	0	1	0	0	1	0	0	1
v_7	0	0	0	1	0	0	0	0

A_H

Figura 2.5: Recorrido de la matriz de adyacencia renglón por renglón.

Por ejemplo, el vector x obtenido al recorrer la matriz de la Figura 2.5 es el siguiente:

$$\begin{aligned} x = \; & [0,1,1,0,0,0,0,0,\\ & 0,0,0,0,0,0,0,0,\\ & 0,0,0,0,0,0,0,0,\\ & 0,1,1,0,0,0,0,0,\\ & 1,0,0,0,0,0,0,0,\\ & 0,0,1,0,1,0,0,1,\\ & 0,1,0,0,1,0,0,1,\\ & 0,0,0,1,0,0,0,0]. \end{aligned}$$

Para convertir nuestro vector x en una cadena de bytes aplicamos las mismas transformaciones que definimos para $R(x)$ en el formato `graph6`. Es decir, representamos el vector x como un binario, si es necesario extendemos el binario para que su longitud sea múltiplo de 6, dividimos nuestro binario extendido en grupos de 6, después cada grupo lo convertimos a decimal siguiendo el orden *Big Endian*, y por último a cada grupo le sumamos 63. Por ejemplo, el valor $R(x)$ del vector obtenido en la Figura 2.5 es:

$$\begin{aligned} x_{bin} = \; & 011000000000000000000000011000001000000\\ & 001010010100100100010000, \end{aligned}$$

$$\begin{aligned} x'_{bin} = \; & 011000\ 000000\ 000000\ 000000\ 011000\ 001000\\ & 000000\ 101001\ 010010\ 010001\ 000000, \end{aligned}$$

$$dec(x'_{bin}) = \; 24\ 0\ 0\ 0\ 24\ 8\ 0\ 41\ 18\ 17\ 0,$$

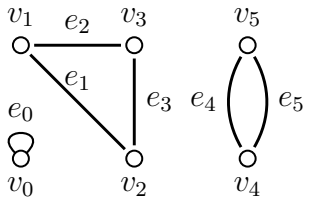
$$R(x) = \; 87\ 63\ 63\ 63\ 87\ 71\ 63\ 104\ 81\ 80\ 63.$$

Y para el número de vértices se utiliza la misma función $N(n)$ definida para `graph6`. De esta forma la digráfica simple D se representa como 38 $N(n)$ $R(x)$; el 38 es un identificador para indicar que la cadena representa a una digráfica, posiblemente con lazos, pero sin flechas múltiples. Terminando el ejemplo, el formato `digraph6` de la digráfica de la Figura 2.4 es 38 71 87 63 63 63 87 71 63 104 81 80 63, así, su representación en cadena de texto es `&GW???WG?hQP?`.

2.3. Sparse6

Los formatos anteriores utilizan alrededor de la mitad o toda la matriz de adyacencia, lo que nos lleva a tener que trabajar con vectores de tamaño $O(|V|^2)$. En algunos casos esto puede resultar contraproducente por la cantidad de espacio que se esta desperdiciando en indicar qué vértices son y no son adyacentes. Un caso extremo puede ser una gráfica con n vértices y tan solo una arista; a pesar de que sólo tenemos una arista, también tenemos que codificar que no existe ninguna otra adyacencia dentro de la gráfica. Por este motivo surge el formato `sparse6`, para guardar únicamente a los vértices que sí tienen alguna adyacencia. Además, también nos permite representar gráficas tanto con lazos como con multiaristas.

Sea $G = (V_G, E_G, \psi_G)$ la gráfica que deseamos guardar, tal que $|V_G| = n$. Primero vamos a explicar cómo guardamos a las adyacencias. Sabemos, por medio de la función de incidencia, que los elementos del conjunto de aristas E_G son parejas no ordenadas, por lo que podemos construir una sucesión de parejas no ordenadas que contenga a estos elementos. Una vez construida esta sucesión, ordenamos cada pareja con respecto a su máximo y después ordenamos a todas las parejas de menor a mayor. Cabe mencionar que esta ordenación la podemos realizar comparando los índices de los vértices.



G

$$E_G = \{e_0, e_1, e_2, e_3, e_4, e_5\},$$

$$\psi(e_0) = \{v_0, v_0\}, \quad \psi(e_1) = \{v_1, v_2\},$$

$$\psi(e_2) = \{v_1, v_3\}, \quad \psi(e_3) = \{v_2, v_3\},$$

$$\psi(e_4) = \{v_4, v_5\}, \quad \psi(e_5) = \{v_4, v_5\}.$$

Figura 2.6: Gráfica con multiaristas y lazos.

Por fines prácticos podemos ignorar las etiquetas de los vértices y enfocarnos únicamente en su índice. Por ejemplo, la sucesión $\mathcal{S}_G$ de la Figura 2.6 es:

$$\mathcal{S}_G = ((0,0),(2,1),(3,1),(3,2),(5,4),(5,4)).$$

Una vez que ya tenemos la sucesión, el siguiente paso es convertirla a un número binario. Para ello utilizamos el Algoritmo 10, el cual tiene como finalidad convertir

a cada arista en uno o dos bloques de $k+1$ bits[2] que representa a cada pareja.

El algoritmo funciona recorriendo únicamente a los vértices de la gráfica que tienen alguna adyacencia, los índices de estos se encuentran en la sucesión $\mathcal{S}_G$. Así, el primer bit del bloque es 0 si el vértice v_i en el que nos encontramos actualmente es adyacente con algún otro vértice v_j con $j < i$, entonces seguido del bit 0 sigue el binario de k bits correspondiente a j. De lo contrario, el primer bit del bloque es 1 para indicar que vamos a cambiar de vértice, sea v_j el vértice al que nos vamos a cambiar. Si $v_j = v_{i+1}$, entonces seguido del bit 1 ponemos el binario correspondiente al índice del vértice que es adyacente a v_j. De lo contrario, si $j > i+1$, entonces seguido del bit ponemos el binario correspondiente a j, después creamos otro bloque en donde el primer bit es 0 y seguido ponemos el binario correspondiente al índice del vértice que es adyacente a v_j. Empezamos colocándonos en el vértice v_0 y terminamos hasta recorrer a todo $\mathcal{S}_G$. La concatenación de todos estos bloques resulta en el binario correspondiente a las adyacencias de la gráfica.

Siguiendo el ejemplo de la Figura 2.6, nos colocamos en el vértice v_0 y empezamos a recorrer $\mathcal{S}_G$. El primer elemento que recorremos es $(0, 0)$, por lo que el bloque es 0 000. Después, recorremos $(2, 1)$, por lo que necesitamos dos bloques, uno para indicar que nos movemos a v_2 y otro para indicar la adyacencia entre v_1 y v_2, entonces los bloques construidos son 1 010 0 001. El siguiente elemento es $(3, 1)$, como $3 = 2+1$, entonces esta adyacencia la podemos representar como un solo bloque 1 001, y el vértice actual es v_3. Seguimos con $(3, 2)$, como no tenemos que cambiar de vértice, entonces el bloque es únicamente 0 010. Para $(5, 4)$ tenemos que cambiar de vértice de nuevo y como $5 \neq 3+1$ entonces necesitamos dos bloques, 1 101 0 100, y el vértice actual ahora es v_5. Finalmente, para $(5, 4)$ sólo necesitamos un bloque 0 100. Por lo que el binario correspondiente al ejemplo es:

$$\mathcal{S}_{bin} = 0\ 000\ 1\ 010\ 0\ 001\ 1\ 001\ 0\ 010\ 1\ 101\ 0\ 100\ 0\ 100.$$

Se agregan a la derecha del binario los suficientes bits 1 para que su longitud sea múltiplo de 6. Dividimos el binario en grupos de 6 y a cada grupo le sumamos 63. Al grupo de bytes obtenidos después de esta serie de transformaciones lo denotamos $\mathcal{R}(x)$. Así, el $\mathcal{R}(x)$ correspondiente a las adyacencias de la gráfica en la Figura 2.6 es:

$$\begin{aligned} \mathcal{S}'_{bin} &= 000010\ 100001\ 100100\ 101101\ 010001\ 001111, \\ dec(\mathcal{S}'_{bin}) &= 2\ 33\ 36\ 45\ 17\ 15, \\ \mathcal{R}(x) &= 65\ 96\ 99\ 108\ 80\ 78. \end{aligned}$$

[2]Donde k es el número de bits necesarios para representar a $n-1$ en binario.

Para el número de vértices se utiliza la función $N(n)$ definida tanto para `graph6` como `digraph6`. Así la gráfica G se representa como los bytes 58 $N(n)$ $\mathcal{R}(x)$; donde el byte 58 es el identificador para gráficas que pueden tener lazos o multiaristas. Terminando con el ejemplo, los bytes del formato `sparse6` de la gráfica en la Figura 2.6 son 58 69 65 96 99 108 80 78, y la cadena de texto correspondiente es `:EA‘clPN`.

Algoritmo 10: $\mathcal{S}_G$ a Binario

Input: La sucesión $\mathcal{S}_G$ y n el número de vértices de la gráfica G
Output: Un binario b que representará a las adyacencias de la gráfica G

```
1  b ← ε
2  v ← 0
3  k ← ⌈log₂(n − 1)⌉
4  for i ∈ {0, 1, ..., |S_G| − 1} do
5  |  p ← la pareja en la posición i de S_G
6  |  p₀ ← el primer elemento de p
7  |  p₁ ← el segundo elemento de p
8  |  if p₀ = v then
9  |  |  Agrega a la derecha de b el bit 0
10 |  else if p₀ = v + 1 then
11 |  |  Agrega a la derecha de b el bit 1
12 |  |  v ← v + 1
13 |  else
14 |  |  Agrega a la derecha de b el bit 1
15 |  |  Agrega a la derecha de b el binario de p₀ utilizando k bits
16 |  |  Agrega a la derecha de b el bit 0
17 |  |  v ← p₀
18 |  end
19 |  Agrega a la derecha de b el binario de p₁ utilizando k bits
20 end
21 while b ≢ 0 (mod 6) do
22 |  Agrega a la derecha de b el bit 1
23 end
```

Capítulo 3

Isomorfismo de árboles

¿Cuándo dos gráficas se pueden considerar como la misma? Informalmente, podemos decir que dos gráficas G y H son *la misma* cuando podemos dibujarlas de la misma manera. Formalmente, tenemos que definir un **isomorfismo** entre G y H. Decimos que G y H son **isomorfas**, si existen biyecciones $\theta\colon V_G \to V_H$ y $\phi\colon E_G \to E_H$ tales que $\psi_G(e) = \{u, v\}$ si y sólo si $\psi_H(\phi(e)) = \{\theta(u), \theta(v)\}$. En el caso de las gráficas simples, como es en el caso de los árboles, decimos que dos gráfica simples G y H son isomorfas si existe una biyección $\varphi\colon V_G \to V_H$ tal que $uv \in E_G$ si y sólo si $\varphi(u)\varphi(v) \in E_H$.

Ahora abordaremos el problema del isomorfismo de árboles; es decir, determinar de manera eficiente cuándo dos árboles se pueden considerar como el mismo. En 1974, Aho, Hopcroft, y Ullman presentaron la idea de un algoritmo lineal capaz de resolver una variante del problema del isomorfismo de árboles, el isomorfismo de árboles enraizados [5]. Después, en el 2002, Valiente de igual forma presenta un algoritmo para resolver otra variante del problema anterior, una más restrictiva, el isomorfismo de árboles ordenados [6]. En este capítulo vamos a explorar ambas propuestas, primero la de Valiente, y después la de Aho, Hopcroft, y Ullman. Dando una explicación formal de sus algoritmos y también demostrando que efectivamente son correctos. Por último, una vez exploradas estas dos propuestas, proponemos y demostramos un algoritmo lineal capaz de resolver el problema general del isomorfismo de árboles.

3.1. Árboles enraizados

Definimos a un **árbol enraizado** en r como una digráfica D tal que su gráfica subyacente es un árbol, y tiene un vértice distinguido r al que le llamaremos **raíz**, de tal forma que para cualquier vértice $v \in V_D$, existe una única rv-trayectoria dirigida

en D. Por consistencia con la mayoría de los textos de Ciencias de la Computación, a los vértices de los árboles los llamamos **nodos**.

Sea $v \in V_D$ un nodo en el árbol, decimos que v es **padre** de otro vértice $w \in V_D$ si existe la flecha $(v, w) \in \mathcal{A}_D$. Así, los **hijos** de un nodo $v \in V_D$, son el conjunto de vértices $W \subset V$ tales que para todo $w \in W$ existe la flecha $(v, w) \in \mathcal{A}_D$. A aquellos nodos que no tengan hijos los llamamos **hojas**. Por último, si dos nodos tienen el mismo padre entonces decimos que son **hermanos**.

Definimos la **profundidad** de un nodo $v \in V_D$ como la longitud de la trayectoria dirigida entre la raíz r y v. La **profundidad máxima** es entonces el máximo entre todas las profundidades del árbol. Definimos la **altura** de un nodo $v \in V_D$ como la longitud de la trayectoria dirigida más larga del subárbol enraizado en v; otra forma de verlo, es como la profundidad máxima del subárbol enraizado en v. La **altura máxima** es el máximo entre todas las alturas del árbol. Nótese que la profundidad de la raíz siempre será cero, al igual que la altura de una hoja es cero.

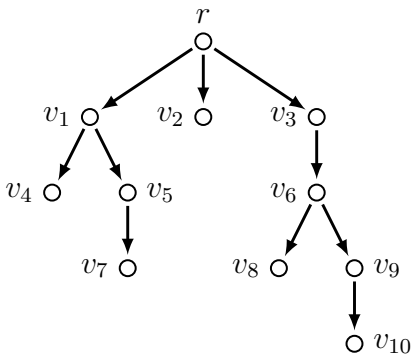


Figura 3.1: Ejemplo de un árbol T enraizado en r.

Por ejemplo, en la figura 3.1 tenemos que la raíz r es padre de los nodos v_1, v_2 y v_3. Los nodos v_4 y v_5 son hermanos pues su padre es v_1. Los nodos v_2, v_4, v_7, v_8 y v_{10} son hojas. La profundidad de r es 0, la de v_6 es 2, y la de v_7 es 3. La profundidad máxima de T es 4. La altura de v_1 es 2, la de v_3 es 3, y la de v_7 es 0. La altura máxima de T es 4.

3.2. Árboles ordenados

Definimos a T, un **árbol ordenado**, como un árbol enraizado en r para el cual el conjunto de hijos de cada vértice está totalmente ordenado. Así, sea $(v, w) \in \mathcal{A}$. Decimos que $w \in V$ es el **primer hijo** si no tiene hermano $w' \in V$ tal que $w' < w$. Y $w \in V$ es el **último hijo** si no tiene hermano $w' \in V$ tal que $w < w'$. De esta forma, decimos que $z \in V$ es el **hermano siguiente** del nodo $w \in V$ si se tiene que $w < z$ y no existe otro hermano x de w tal que $w < x < z$. De igual forma, decimos que $z \in V$ es el **hermano anterior** de $w \in V$, si w es el hermano siguiente de z. Dentro de los dibujos, el orden de los hijos será representado de izquierda a derecha; es decir, el primer hijo será el que más a la izquierda se encuentre, y el último hijo será el que se encuentre más a la derecha. Retomando el ejemplo de la figura 3.1, v_1 es el primer hijo de r y v_3 es su último hijo, v_2 es el hermano siguiente de v_1 y el hermano anterior de v_3.

Cabe mencionar que la función de parentesco obtenida por BFS y DFS (algoritmos 1 y 2) es capaz de construir un árbol no enraizado T en un árbol enraizado en r. Lo único que tenemos que hacer es ejecutar uno de estos algoritmos sobre T y con el vértice distinguido r. Obtenida la función de parentesco, definimos al árbol enraizado en r D como:

$$V_D = V_T,\ \mathcal{A}_D = \{(p(v), v) \colon v \in V_D - \{r\}\}.$$

Como el número de aristas de un árbol es $|V|-1$, entonces construir un árbol ordenado a partir de uno no ordenado por medio de BFS o DFS toma tiempo $O(|V|)$.

3.3. Isomorfismo de árboles ordenados

Dos árboles ordenados T_1 y T_2, con $T_1 = (V_1, \mathcal{A}_1)$ y $T_2 = (V_2, \mathcal{A}_2)$, son isomorfos si existe una biyección $\varphi \colon V_1 \to V_2$ tal que $\varphi(r_1) = r_2$ en donde r_1 y r_2 son las respectivas raíces de los árboles, y se cumplen las siguientes condiciones:

1. Si $v \in V_1$ y $w \in V_2$ son nodos no hojas con $\varphi(v) = w$, entonces la imagen bajo φ del primer hijo de v debe ser igual al primer hijo de w.

2. Si $v \in V_1$ y $w \in V_2$ son hijos no últimos con $\varphi(v) = w$, entonces la imagen bajo φ del hermano siguiente de v debe de ser igual al hermano siguiente de w.

Si todas las condiciones anteriores se cumplen entonces φ es un **isomorfismo de árboles ordenados** de T_1 a T_2, y lo denotamos por $T_1 \overset{\varphi}{\cong} T_2$, o simplemente $T_1 \cong T_2$,

si no es necesario indicar el nombre del isomorfismo. En otras palabras, dos árboles ordenados son isomorfos si existe una biyección entre sus conjuntos de vértices y ésta preserva la estructura de los árboles ordenados; con esto último nos referimos a que las raíces se correspondan entre sí y los nodos preserven sus relaciones de parentesco y hermandad.

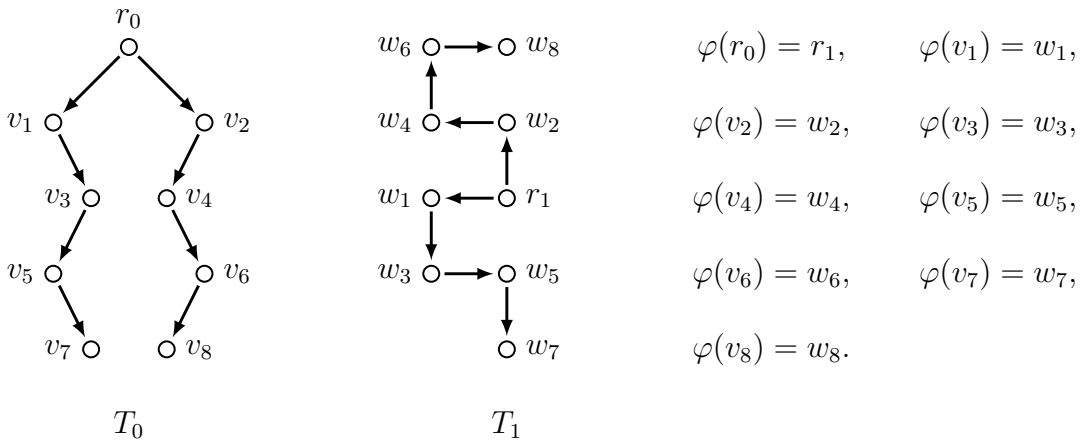


Figura 3.2: Ejemplo de dos árboles ordenados isomorfos.

3.3.1. Algoritmo para el isomorfismo de árboles ordenados

Sean T_1 y T_2 árboles ordenados, y sea s_{T_i} la sucesión de vértices obtenidos al recorrer T_i[1], para $i \in \{1, 2\}$. Si las longitudes de ambas sucesiones son iguales, entonces la función $\varphi\colon V_1 \to V_2$ inducida por s_{T_1} y s_{T_2}, dada por $\varphi(v_i) = w_i$ para cada $i \in [0, \ldots, |s_{T_1}|]$ es una biyección. En otras palabras, como recorremos a ambos árboles con el mismo método, entonces ambas sucesiones deben de contener a los mismos vértices en el mismo orden. Por lo que en caso de que sí se cumpla lo anterior, entonces podemos dar con la biyección φ que es el isomorfismo entre los dos árboles.

Cabe mencionar que es posible atribuirle etiquetas a los nodos y flechas. Nótese que las etiquetas para flechas no son necesarias dentro de los árboles ordenados, ya que un nodo dentro del árbol va a tener a lo más una flecha incidente en él, por lo que la etiqueta correspondiente a la flecha puede ser atribuida directamente al nodo al que incide. Entonces nos referimos a un **árbol ordenado etiquetado** como un árbol ordenado en donde sólo sus nodos han sido etiquetados. Denotamos

[1]Naturalmente, T_1 y T_2 deben ser recorridos con el mismo método.

a la etiqueta de un nodo con $e(v)$, donde $v \in V$. Entonces dos árboles ordenados etiquetados son isomorfos, si los dos árboles ordenados son isomorfos, y además los correspondientes nodos y aristas comparten la misma etiqueta. Así, usamos el algoritmo 11 para identificar isomorfismos entre árboles ordenados etiquetados. A continuación demostraremos que el algoritmo 11 es correcto.

Algoritmo 11: Isomorfismo de Árboles Ordenados Etiquetados

Input: Los árboles ordenados etiquetados T_1 y T_2
Output: Un booleano que indica si los árboles son o no son isomorfos.

```
1  if |V_{T_1}| ≠ |V_{T_2}| then
2  |  return False
3  s_{T_1} ← La sucesión obtenida al recorrer T_1
4  s_{T_2} ← La sucesión obtenida al recorrer T_2
5  for i ∈ {0, 1, ..., |V_{T_1}| − 1} do
6  |  v ← s_{T_1}[i]
7  |  w ← s_{T_2}[i]
8  |  if e(v) ≠ e(w) then
9  |  |  return False
10 |  if v es hoja y w no es hoja o viceversa then
11 |  |  return False
12 |  if v no es hoja then
13 |  |  v_f ← el primer hijo de v
14 |  |  w_f ← el primer hijo de w
15 |  |  if e(v_f) ≠ e(w_f) then
16 |  |  |  return False
17 |  if v tiene hermano siguiente y w no o viceversa then
18 |  |  return False
19 |  if v tiene hermano siguiente then
20 |  |  v_b ← el hermano siguiente de v
21 |  |  w_b ← el hermano siguiente de w
22 |  |  if e(v_b) ≠ e(w_b) then
23 |  |  |  return False
24 |  |  end
25 |  end
26 end
27 return True
```

Lema 3.3.1. *El algoritmo 11 es capaz de determinar si dos árboles ordenados etiquetados son isomorfos en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Sean T_1 y T_2 árboles ordenados etiquetados. Supongamos primero que T_1 y T_2 son isomorfos, y veamos que el algoritmo devuelve **verdadero**.

Como T_1 y T_2 son isomorfos, el número de vértices en ambos es el mismo, por lo que la condición en la línea 1 falla, y el algoritmo continúa su ejecución. Se recorren ambos árboles y se obtienen las sucesiones correspondientes s_{T_1} y s_{T_2}. Nótese que aunque las sucesiones sean iguales, esto no es suficiente para poder determinar si los dos árboles son isomorfos. Por lo que es necesario realizar verificaciones adicionales sobre la estructura de estos dos árboles; es decir, necesitamos comparar entre los dos árboles a los hijos y hermanos de los nodos.

Para ello recorremos simultáneamente a s_{T_1} y s_{T_2}. De tal forma que para cada pareja de nodos v y w de las correspondientes sucesiones, verificamos que sigan la misma estructura. Primero verificamos que tengan la misma etiqueta. Después que ambos nodos sean hojas o que ambos no sean hojas. En caso de que ambos no sean hojas, comparamos que los primeros hijos tengan las mismas etiquetas. Por último, que ambos tengan hermano siguiente o que ambos no tengan hermano siguiente. En caso de que ambos sí tengan hermano siguiente, comparamos que estos tengan la misma etiqueta. Como T_1 y T_2 son isomorfos, entonces todas las verificaciones anteriores van a pasar sin problema alguno, por lo que el algoritmo terminará su ejecución sin detenerse, llegando a la línea 32 y regresando **verdadero**.

Supongamos ahora que T_1 y T_2 son no isomorfos, y veamos que el algoritmo devuelve **falso**. Si el número de vértices en los árboles no coincide, entonces el algoritmo devolverá **falso** en la línea 2. Por otro lado, si existe algún índice i tal que el i-ésimo vértice de s_{T_1} no tiene la misma etiqueta que el i-ésimo vértice de s_{T_2}, entonces el algoritmo devolverá **falso** en la línea 9. Como ya mencionamos, aún cuando T_1 y T_2 no sean isomorfos, las etiquetas en ambas sucesiones podrían coincidir. Sin embargo, al ser T_1 y T_2 no isomorfos, su estructura debe diferir en al menos un vértice. Recordemos que la definición de isomorfismo nos dice que la raíz de los árboles debe coincidir, si v y w son nodos no hojas tales que w es la imagen de v, entonces la imagen bajo el isomorfismo del primer hijo de v debe de coincidir con la del primer hijo de w, y si v y w son hijos no últimos tales que w es la imagen de v, entonces la imagen bajo el isomorfismo del hermano siguiente de v debe ser igual al hermano siguiente de w. Al ser T_1 y T_2 no isomorfos, alguna de estas condiciones debe fallar, por lo que la verificación correspondiente también fallará, y el algoritmo devolverá **falso**.

Ahora pasemos a un análisis de tiempo de nuestro algoritmo. Nótese que todas las comparaciones que realiza el algoritmo se pueden hacer en tiempo constante. Por

otro lado, para obtener nuestras sucesiones s_{T_1} y s_{T_2} es necesario recorrer a todos los nodos de ambos árboles, por lo que si $|V_{T_1}| = n$ y $|V_{T_2}| = m$, entonces obtener estas sucesiones toma un total de $O(n + m)$; en caso de que ambos árboles sean isomorfos entonces esta operación toma $O(n)$. Por último, como recorremos a las dos sucesiones simultáneamente, y por cada pareja hacemos a lo más ocho comparaciones, entonces esta operación toma tiempo $O(n)$. Concluyendo de esta forma que nuestro algoritmo toma un total de tiempo $O(1 + n + n) = O(n)$ para determinar si dos árboles ordenados etiquetados son isomorfos.

Para el análisis de espacio de nuestro algoritmo, notemos que el elemento de nuestro algoritmo que más información almacena son las sucesiones s_{T_1} y s_{T_2} pues contienen a todos los nodos de los árboles ordenados; es decir el tamaño en espacio de estos elementos es de $O(n)$. Todos los demás elementos tienen un tamaño constante. De esta manera, concluimos que nuestro algoritmo toma espacio $O(n)$ para determinar si dos árboles ordenados etiquetados son isomorfos. □

Aunque el algoritmo 11 es capaz de detectar el isomorfismo entre dos árboles ordenados etiquetados, podemos mejorar el tiempo en el que detectamos cuando dos árboles ya no son isomorfos. Lo anterior lo hacemos recorriendo los dos árboles simultáneamente y verificando en cada paso que la función inducida por los nodos sea un isomorfismo de árboles ordenados etiquetados; en otras palabras, en cada paso verificamos que la estructura de los dos árboles sea la misma. El algoritmo 12 incluye esta mejora de tiempo, notemos que este algoritmo es recursivo, y utiliza a la rutina auxiliar VerificaciónEstructural (algoritmo 13) quien es el que realmente realiza todo el trabajo de verificación. Veamos que el algoritmo 12 efectivamente es correcto.

Algoritmo 12: Isomorfismo de Árboles Ordenados Etiquetados Recursivo

Input: Los árboles ordenados T_1 y T_2
Output: Un booleano que indica si los árboles son o no son isomorfos.

```
1 if |V_{T_1}| ≠ |V_{T_2}| then
2 |   return False
3 r_{T_1} ← la raíz de T_1
4 r_{T_2} ← la raíz de T_2
5 return VerificacionEstructural(T_1, r_{T_1}, T_2, r_{T_2})
```

Lema 3.3.2. *El algoritmo 12 es capaz de detectar si dos árboles ordenados etiquetados son isomorfos en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Algoritmo 13: Verificación Estructural

Input: Los árboles ordenados T_1 y T_2, y sus respectivos nodos v_{T_1} y w_{T_2}
Output: Un booleano que indica si los nodos respetan la estructura de los dos árboles.

```
1  if e(v) ≠ e(w) then
2  |  return False
3  end
4  n_v ← el número de hijos de v
5  n_w ← el número de hijos de w
6  if n_v ≠ n_w then
7  |  return False
8  if n_v ≠ 0 then
9  |  v_h ← el primer hijo de v
10 |  w_h ← el primer hijo de w
11 |  if ¬VerificacionEstructural(T_1, v_h, T_2, w_h) then
12 |  |  return False
13 |  for i ∈ {2, ..., n_v} do
14 |  |  v_h ← el hermano siguiente de v_h
15 |  |  w_h ← el hermano siguiente de w_h
16 |  |  if ¬VerificacionEstructural(T_1, v_h, T_2, w_h) then
17 |  |  |  return False
18 |  |  end
19 |  end
20 end
21 return True
```

Demostración. Sean T_1 y T_2 árboles ordenados etiquetados. Supongamos primero que T_1 y T_2 son isomorfos y veamos que el algoritmo devuelve **verdadero**.

El objetivo de este algoritmo es recorrer simultaneamente a ambos árboles T_1 y T_2 sin la necesidad de construir una sucesión, para que tan pronto se encuentre una diferencia entre los árboles nos detengamos. Para ello utilizamos la rutina auxiliar VerificaciónEstructural (el algoritmo 13), que recibe el árbol T_1 y el nodo en el que nos encontramos dentro de nuestro recorrido, y tenemos lo mismo para T_2.

Sea v el nodo correspondiente a T_1 y w el nodo correspondiente a T_2. Vamos a verificar que las etiquetas de ambos nodos sean la misma. En caso de que sí lo sean entonces se verifica que el número de hijos de ambos nodos sean el mismo, y si es

así entonces se aplica el algoritmo 13 a los respectivos hijos de w y v. Visto de otra manera, la rutina auxiliar determina si el subárbol de T_1 enraizado en v es isomorfo al subárbol de T_2 enraizado en w.

Procedemos por inducción sobre el número de vértices de los árboles T_1 y T_2.

Para el caso base, cuando $|V_{T_1}| = 1 = |V_{T_2}|$, observamos que la condición de la línea 1 falla, y entramos a la rutina auxiliar. Dentro de la rutina, como ambos árboles son isomorfos entonces v y w tienen la misma etiqueta, por lo que la condición de la línea 1 falla y continuamos con la ejecución. Debido a que ambos vértices son hojas entonces todas las demás condiciones fallan, la rutina regresa **verdadero**, y así nuestro algoritmo regresa **verdadero**.

Para el paso inductivo, cuando $|V_{T_1}| = n = |V_{T_2}|$, observamos que la condición de la línea 1 falla, y entramos a la rutina auxiliar. Dentro de la rutina, como ambos árboles son isomorfos entonces v y w tienen la misma etiqueta y la misma cantidad de hijos, por lo que las condiciones de la línea 1 y 6 fallan, continuamos con la ejecución. Después para cada pareja de hijos de v y w, sean v_h y w_h estos, llamamos la rutina auxiliar con estos valores. Lo que hacemos en este paso es verificar que el subárbol de T_1 enraizado en v_h y el subárbol de T_2 enraizado en w_h tengan la misma estructura. Sean T_1' y T_2' los respectivos árboles, nótese que $|V_{T_1'}| < |V_{T_1}|$ y $|V_{T_2'}| < |V_{T_2}|$, por lo que por hipótesis de inducción la rutina auxiliar regresa **verdadero** para estos valores. Como esto aplica para todos los hijos de v y w, entonces ninguna de las condiciones restantes falla, y llegamos a la línea 21. De esta manera la rutina auxiliar regresa **verdadero**, y así nuestro algoritmo regresa **verdadero**.

Supongamos ahora que T_1 y T_2 son no isomorfos y veamos que el algoritmo devuelve **falso**. Si el número de vértices no coincide entonces el algoritmo regresa **falso** en la línea 2. Por otro lado, si existe un índice i tal que en el i-ésimo paso de nuestro recorrido tenemos que la etiqueta del vértice v de T_1 no coincide con la del vértice w de T_2, entonces la rutina auxiliar regresa **falso** en la línea 1, lo que hace que el algoritmo regrese **falso**. Sin embargo, como se mencionó anteriormente, las etiquetas de todos los vértices pueden coincidir y aún así los árboles pueden ser no isomorfos. Por lo que si la estructura de T_1 y T_2 difieren en al menos un vértice, esto implica que existe un índice i tal que en el i-ésimo paso de nuestro recorrido tenemos v y w en donde el número de hijos de estos dos vértices no coinciden, así la rutina regresa **falso** en la línea 7, lo que hace que el algoritmo también regrese **falso**.

Para el análisis de tiempo observamos que en el peor de los casos hay un total de $|V_{T_1}| = n = |V_{T_2}|$ llamadas a la rutina auxiliar, ya que vamos a recorrer simultaneamente a todos los nodos de T_1 y T_2. Nótese que por cada llamada a la rutina auxiliar recorremos exactamente un vértice de ambos árboles, después el número de llamadas recursivas que haremos a la rutina es igual al número de hijos que tiene el vértice en

el que nos encontramos actualmente el recorrido. Por lo que la rutina auxiliar toma un total de tiempo $O(n)$, resultando en que el algoritmo igual toma tiempo $O(n)$.

Para el análisis de espacio vemos que en cada llamada a la rutina auxiliar ocupamos espacio constante pues sólo estamos guardando a los vértices en los que nos encontramos actualmente en el recorrido. Sin embargo, como en el peor caso la profundidad de la recursión puede ser $O(n)$, entonces la cantidad de espacio que puede ocupar el algoritmo en total es $O(n)$. □

3.4. Isomorfismo de árboles enraizados

Dos árboles enraizados T_1 y T_2, con $T_1 = (V_1, \mathcal{A}_1)$ y $T_2 = (V_2, \mathcal{A}_2)$, son isomorfos si existe un biyección $\varphi\colon V_1 \to V_2$ tal que $\varphi(r_1) = r_2$ en donde r_1 y r_2 son las respectivas raíces de los árboles, y se cumple la siguiente condición:

- Si $v \in V_1$ y $w \in V_2$ son nodos no raíces con $\varphi(v) = w$, entonces la imagen bajo φ del padre de v debe de ser igual al padre de w.

Si todas las condiciones anteriores se cumplen entonces φ es un **isomorfismo de árboles enraizados** de T_1 a T_2, y lo denotamos por $T_1 \overset{\varphi}{\cong} T_2$, o simplemente $T_1 \cong T_2$, si no es necesario indicar el nombre del isomorfismo.

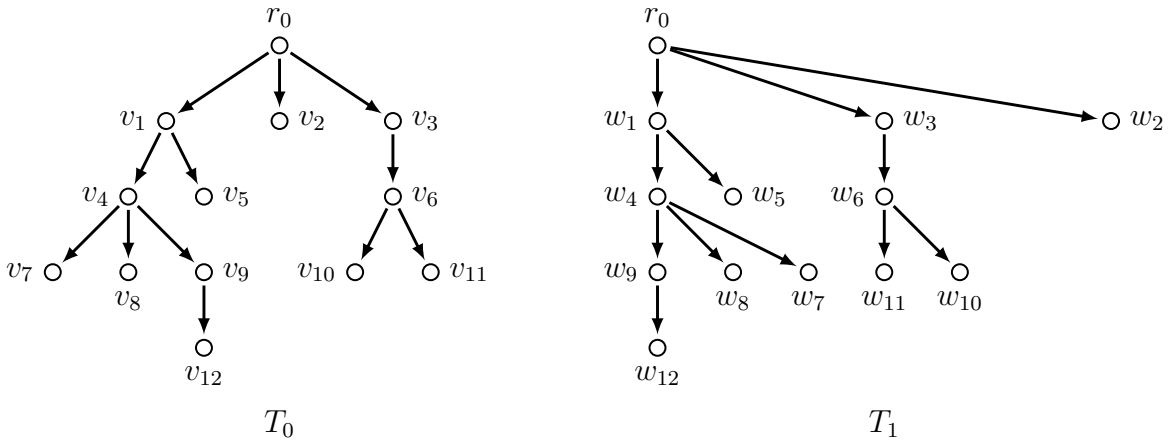


Figura 3.3: Ejemplo de dos árboles enraizados isomorfos.

3.4.1. Algoritmos para el isomorfismo de árboles enraizados

El objetivo principal de esta sección es presentar el algoritmo 23 para identificar el isomorfismo entre árboles enraizados. Iniciamos con una explicación informal y un

ejemplo de la forma en la que trabaja el algoritmo.

Sean T_1 y T_2 los árboles enraizados a identificar. La idea general del algoritmo es determinar, nivel por nivel, si los nodos en el nivel i de T_1 tienen la misma "estructura" que los nodos del nivel i de T_2. Si para todos los niveles se cumple que T_1 y T_2 tienen la misma estructura, entonces T_1 y T_2 son isomorfos. Recordemos que en estos árboles no existen los conceptos de primer hijo, último hijo, hermano siguiente o hermano anterior. No podemos tomar a un nodo arbitrario y compararlo con otro que esté en el mismo nivel del otro árbol. Sin embargo, lo que sí podemos hacer es ver qué tipo de estructura tienen todos los nodos del nivel i del árbol T_1 (el número de hojas en este nivel, el número de nodos con un solo hijo, el número de nodos con dos hijos, etc.) y verificar que esta estructura sea la misma para el conjunto de nodos en el nivel i de T_2.

Veamos un ejemplo de una ejecución del algoritmo 23 para detectar un isomorfismo entre dos árboles enraizados. A continuación presentamos los dos árboles con los que vamos a trabajar:

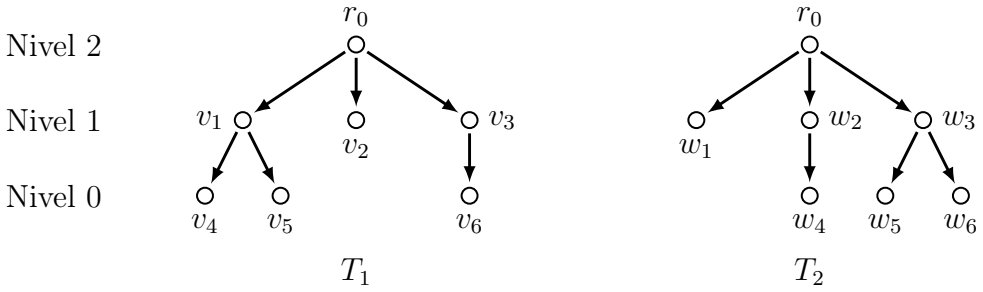


Figura 3.4: Los dos árboles enraizados isomorfos sobre los que trabajará el algoritmo.

Lo primero que hace el algoritmo 23 es determinar, para toda $i \in [0, h]$, cuál es el conjunto de nodos de T que se encuentran en el nivel i, en donde h es la altura del árbol. Nótese que el número de niveles que tiene un árbol T es igual a la altura de éste, en donde en el nivel h únicamente se encuentra la raíz y en el nivel 0 solo tenemos hojas. Por este motivo, definimos a la función $nvl : \mathbb{N} \to \mathcal{P}(V)$ dentro de nuestro algoritmo para la que, dado $i \in \mathbb{N}$, $nvl(i)$ es el conjunto de nodos que se encuentran en el nivel i.

La función nvl se calcula por medio de la rutina auxiliar Obtener Niveles (el algoritmo 14), el cual es un algoritmo recursivo que utiliza una subrutina auxiliar (el algoritmo 15), quien es el que realiza toda la construcción de la función nvl. La construcción de la función nvl se realiza mediante un recorrido DFS. Empezando por indicar que el único nodo en el nivel h es la raíz r, después para cada nodo que

recorramos, indicamos que éste se encuentra en el nivel $i-1$, en donde i es el nivel en el que se encuentra su padre. Definimos a la función nvl_1 para los nodos y niveles de T_1, y análogamente nvl_2 para T_2. Siguiendo con el ejemplo de la figura 3.4, los valores de las variables antes mencionadas son los siguientes:

$$\begin{aligned} nvl_1[2] &= \{r_0\}, & nvl_2[2] &= \{r_0\}, \\ nvl_1[1] &= \{v_1, v_2, v_3\}, & nvl_2[1] &= \{w_1, w_2, w_3\}, \\ nvl_1[0] &= \{v_4, v_5, v_6\}, & nvl_2[0] &= \{w_4, w_5, w_6\}. \end{aligned}$$

Como podemos observar, para toda $i \in [0, 2]$ se cumple que $|nvl_1[i]| = |nvl_2[i]|$, por lo que la ejecución del algoritmo continúa.

Ahora necesitamos determinar qué tipo de estructura tiene cada nivel de los árboles para poderlos comparar, y así verificar que sean el mismo. Para ello, a cada nodo le vamos a asociar un valor que corresponde al tipo de estructura que éste tiene. Por ejemplo, a las hojas por omisión les corresponde el valor 0, a la siguiente estructura que difiera de ser una hoja le asignamos el valor 1, y así consecutivamente para cada estructura diferente que encontremos. Así, definimos a la función $vals\colon V \to \mathbb{N}$ dentro de nuestro algoritmo para la que, dado $v \in V$, $vals(v)$ es el valor asociado al vértice v.

Por medio de la rutina auxiliar Preparar Valores (el algoritmo 16), vamos a indicar el valor inicial de los nodos, el cual le atribuye a las hojas valor 0 y a los nodos no hojas valor ∞. El algoritmo funciona recursivamente mediante una subrutina auxiliar (el algoritmo 17), la cual realiza un recorrido DFS sobre el árbol para asignar los valores iniciales. Respectivamente, definimos a la función $vals_1$ para los valores de T_1 y $vals_2$ para T_2. Continuando con el ejemplo de la figura 3.4, los valores iniciales de los nodos son los siguientes:

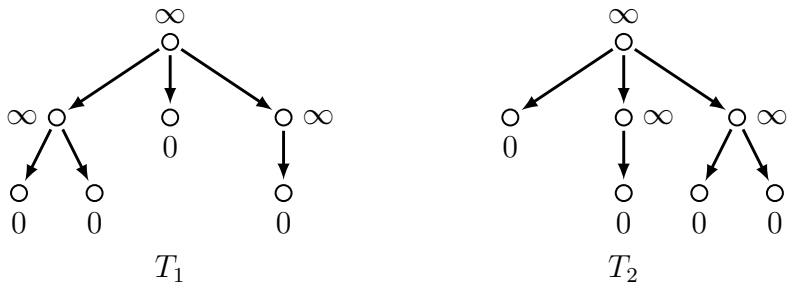


Figura 3.5: Los valores iniciales de los nodos de los árboles enraizados isomorfos.

Después nos colocamos en el nivel 1 de ambos árboles, con el objetivo de asignarles a todos los nodos de este nivel su respectivo valor; nótese que no nos colocamos en el

nivel 0 pues todos los nodos de este nivel ya tienen un valor ya que son hojas. Como mencionamos anteriormente, el valor de cada nodo depende de la estructura que éste tenga. Para ello, a cada nodo no hoja v le vamos a asignar un k-multiconjunto que contiene los valores de cada uno de sus hijos, donde k es el número de hijos que tiene v. Entonces, definimos a la función $struct\colon V \to \mathcal{P}^m(\mathbb{N})$ dentro de nuestro algoritmo para la que, dado $v \in V$, $struct(v)$ es el multiconjunto que corresponde a los valores que tienen los hijos de v.

La función *struct* se calcula por medio de la rutina auxiliar Asignar Estructura (el algoritmo 18). Entonces, obtenemos $struct_1$ para T_1 y $struct_2$ para T_2. Por ejemplo, en la figura 3.4 tenenemos lo siguiente para los nodos del primer nivel:

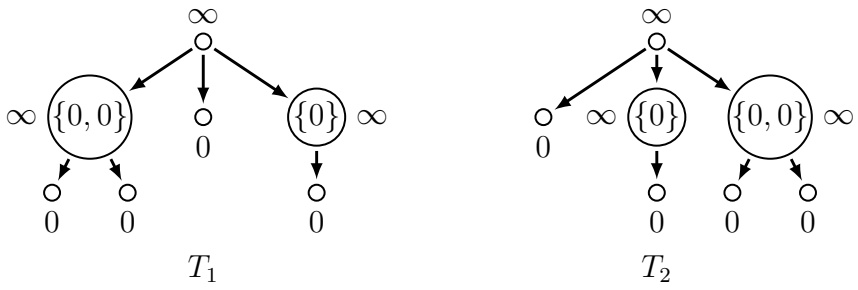


Figura 3.6: Los multiconjuntos correspondientes al primer nivel.

Una vez que todos los nodos del nivel ya tienen una estructura, construimos una sucesión que contenga a todas las estructuras que acabamos de definir, esto lo realizamos por medio de la rutina auxiliar Construir Sucesión (el algoritmo 19). Denotamos esta sucesión S_1 para T_1 y S_2 para T_2. El objetivo de construir estas sucesiones es compararlas y verificar que ambos árboles tengan la misma estructura en cada nivel. La comparación se realiza mediante la rutina Comparar Sucesiones (el algoritmo 20). Cabe mencionar que una vez construidas ambas sucesiones, estas son ordenadas utilizando el algoritmo 9, con la finalidad de poder determinar en tiempo lineal si ambas sucesiones son iguales; ya que si los árboles son isomorfos entonces las sucesiones tendrán los mismos elementos y en el mismo orden.

Además de las sucesiones, definimos a la función $M_S\colon \mathcal{P}^m(\mathbb{N}) \to \mathcal{P}(V)$ dentro de nuestro algoritmo para la que, dado un multiconjunto s, $M_S(s)$ es el conjunto de nodos que tienen como estructura a s. El objetivo de esta función es determinar en tiempo constante al conjunto de nodos que les corresponde una estructura s. Esto será útil cuando vayamos a atribuirle a cada estructura un valor único. La construcción de esta función se realiza junto a la construcción de la sucesión S en la rutina Construir Sucesión. Continuando con el ejemplo de la figura 3.4, las sucesiones

ordenadas obtenidas son las siguientes:

$$S_1 = (\{0\}, \{0, 0\}), \quad S_2 = (\{0\}, \{0, 0\}).$$

De esta forma, ya podemos asignarle a cada nodo del nivel su respectivo valor utilizando la sucesión S, y la información que tenemos en la función M_S. En donde a cada estructura única s en la sucesión S le corresponde un valor k, tal que todos los vértices de $M_S(s)$ tendrán como valor a k. Lo anterior lo realizamos mediante la rutina auxiliar Asignar Valores (el algoritmo 21). Por ejemplo, con la sucesión que obtuvimos para el primer nivel de la figura 3.4, los valores de los nodos son los siguientes:

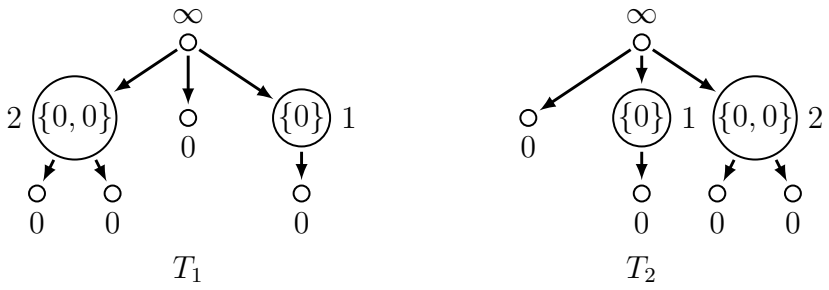


Figura 3.7: Los valores correspondientes al primer nivel.

Ahora que ya acabamos con el primer nivel, podemos continuar por asignarle un valor a todos los nodos del segundo nivel. En el caso del ejemplo, el segundo nivel es únicamente la raíz de ambos árboles, por lo que sólo tenemos que comparar que la sucesión obtenida en cada raíz sea la misma. Para ello se realiza el mismo procedimiento de asignarle a cada raíz su respectiva estructura y construir la sucesión correspondiente a partir de esta. Siguiendo nuestro ejemplo de la figura 3.4, nuestros árboles se ven de la siguiente forma:

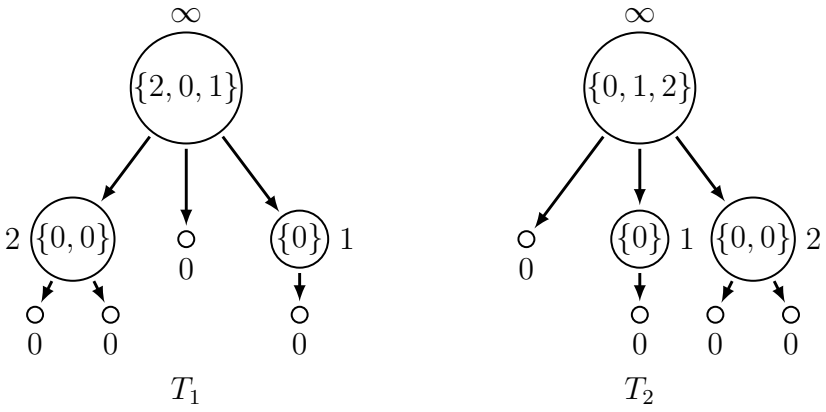


Figura 3.8: Las estructuras correspondientes a cada raíz.

Recordemos que ambas sucesiones son ordenadas antes de ser comparadas, sin embargo como solo hay un elemento en ambas sucesiones entonces no hay un cambio. Por lo que las sucesiones de cada una de las raíces son las siguientes:

$$S_{T_1} = (\{2, 0, 1\}) = (\{0, 1, 2\}) = S_{T_2}.$$

El algoritmo verifica que las sucesiones de ambas raíces sean la misma. Como T_1 y T_2 son isomorfos entonces esto es cierto, por lo que el algoritmo regresa **verdadero**. Concluyendo así el ejemplo de la figura 3.4.

El proceso de calcular el valor de todos los nodos del nivel i de los árboles T_1 y T_2 y comparar que sean el mismo, se realiza mediante la rutina Verificacion por Niveles (el algoritmo 22). En donde, utilizando los valores del nivel $i-1$, se calculan los valores del nivel i, y así recursivamente hasta llegar al último nivel; nótese que si en algún nivel previo al final se detecta que la estructura de ambos árboles no coincide, entonces en ese momento termina el algoritmo y regresa **falso**.

Para demostrar fácilmente la corrección del algoritmo 23, primero procedemos a demostrar que cada una de las rutinas auxiliares que utiliza el algoritmo es correcta.

Lema 3.4.1. *Dado T un árbol enraizado, el algoritmo 14 es capaz de regresar una función $M\colon \mathbb{N} \to \mathcal{P}(V)$ tal que, para $i \in \mathbb{N}$, $M(i)$ es el conjunto de vértices que se encuentran en el nivel i de T. El algoritmo 14 se ejecuta en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Veamos que la función M que construye el algoritmo 14 efectivamente es correcta. Para ello, sea $v \in V$, tal que v se encuentra en el nivel i, veamos que $v \in M(i)$. Procedemos por inducción sobre la distancia entre la raíz r y el vértice v.

Algoritmo 14: Obtener Niveles

Input: El árbol enraizado T, y la altura h de T.
Output: Una función $M\colon \mathbb{N} \to \mathcal{P}(V)$ que indica cuáles vértices de T están en cada nivel.

```
1 r ← la raíz de T
2 M ← ∅
3 return ObtenerNivelesAuxiliar(T, M, r, h)
```

Algoritmo 15: Obtener Niveles Auxiliar

Input: El árbol enraizado T, una función $M\colon \mathbb{N} \to \mathcal{P}(V)$, el vértice v en el que nos encontramos actualmente, y h la altura de v.
Output: Una función $M\colon \mathbb{N} \to \mathcal{P}(V)$ que indica en qué nivel se encuentran el vértice v junto con sus descendientes.

```
1 M[h] ← M[h] ∪ {v}
2 N ← los hijos de v
3 for n ∈ N do
4 |   M ← ObtenerNivelesAuxiliar(T, M, n, h − 1)
5 end
6 return M
```

El caso base es cuando $d(r, v) = 0$, es decir, $r = v$. Al ejecutar el algoritmo, la línea 3 ejecuta la subrutina auxiliar 14 con los parámetros T, M, r y h, con h la altura del árbol. Dentro de la subrutina, en la línea 1 indicamos que $M(h) = \{r\}$. Sabemos que el único vértice que se encuentra en el nivel h es la raíz, es decir v, y por el argumento anterior, podemos observar que efectivamente se cumple que $v \in M(h)$.

Para el paso inductivo, cuando $d(r, v) = k$. Sea $T = (v_1 = r, v_2, \ldots, v_{k-1}, v_k = v)$ la rv-trayectoria de longitud k en T, sabemos que $d(r, v_{k-1}) = k - 1$ por lo que por hipótesis inductiva se cumple que $v_{k-1} \in M(h)$, en donde h es el nivel en el que se encuentra v_{k-1} en T. Lo anterior implica que en algún momento de la ejecución de nuestro algoritmo, la subrutina auxiliar 14 fue llamada con los parámetros T, M, v' y h. Después de ejecutar la línea 1, obtenemos en la línea 2 a los hijos $\mathcal{N}$ de v', en donde sabemos que $v \in \mathcal{N}$. Por lo tanto, ejecutamos la subrutina auxiliar con los parámetros T, M, v y $h - 1$, asegurando de este modo que $v \in M(h - 1)$. Dado que v' tiene altura h, al ser v su hijo, éste tiene altura $h - 1$, como se esperaba de M.

Para el análisis de tiempo, la construcción de la función M depende del recorrido que hacemos sobre el árbol. Así, cuando acabemos de recorrer el árbol, ya habremos

construido toda la función M. El recorrido que estamos haciendo sobre el árbol es DFS, y por cada vértice que recorremos hacemos un número constante de operaciones, entonces el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, por cada llamada recursiva no estamos construyendo una nueva función M, sino que estamos trabajando sobre una misma durante toda la ejecución del algoritmo. Y como el tamaño final de esta función es $O(n)$ (ya que incluye a todos los vértices del árbol), entonces el espacio que utiliza este algoritmo es $O(n)$. □

Algoritmo 16: Preparar Valores

Input: El árbol enraizado T.
Output: Una función $M\colon V_T \to \mathbb{N}$ que indica qué valor tienen los vértices de T.

```
1 r ← la raíz de T
2 M ← ∅
3 return PrepararValoresAuxiliar(T, M, r)
```

Algoritmo 17: Preparar Valores Auxiliar

Input: El árbol enraizado T, una función $M\colon V_T \to \mathbb{N}$, y el vértice v en el que nos encontramos actualmente.
Output: Una función $M\colon V_T \to \mathbb{N}$ que indica qué valor tienen los vértices de T.

```
1  if v es una hoja then
2  |  M[v] ← 0
3  else
4  |  M[v] ← ∞
5  |  N ← los hijos de v
6  |  for n ∈ N do
7  |  |  M ← PrepararValoresAuxiliar(T, M, n)
8  |  end
9  end
10 return M
```

Lema 3.4.2. *Dado un árbol enraizado T, el algoritmo 16 es capaz de regresar una función $M\colon V_T \to \mathbb{N}$ en donde dado $v \in V_T$, $M(v)$ es el valor inicial asociado al nodo v. El algoritmo 16 se ejecuta en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Veamos que la función M que construye el algoritmo 16 efectivamente es correcta. Para ello, sea $v \in V$, veamos que $M(v) = 0$ si v es hoja y $M(v) = \infty$ si v es no hoja.

Sea $v \in V$, veamos qué ocurre dentro de la ejecución cuando v es hoja. El algoritmo 16 se basa de la subrutina auxiliar 17, el cual es un algoritmo recursivo para recorrer a todos los vértices del árbol; notemos que este es un recorrido DFS. Por lo que eventualmente, después de una serie de llamadas recursivas llegamos al vértice v y dado que v es una hoja entonces la condición de la línea 1 de la subrutina es verdadera, y por ende $M(v) = 0$, obteniendo así el resultado que buscábamos.

Ahora veamos qué ocurre dentro de la ejecución cuando v no es hoja. Como explicamos anteriormente, eventualmente llegamos al vértice v por medio del recorrido al árbol. En este punto, como v no es hoja, entonces la condición de la línea 1 de la subrutina es falsa, así, ejecutamos la línea 5 y obtenemos $M(v) = \infty$, consiguiendo el resultado que buscábamos.

Para el análisis de tiempo, solo estamos recorriendo el árbol y por cada vértice en el que nos paramos le asignamos un valor, esta última operación toma tiempo constante. Por lo que el algoritmo termina tan pronto como terminemos de recorrer todo el árbol, es decir, el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, por cada llamada recursiva no estamos construyendo una nueva función M, sino que estamos trabajando sobre una misma durante toda la ejecución del algoritmo. Y como el tamaño final de esta función es $O(n)$, entonces el espacio que utiliza este algoritmo es de $O(n)$. □

Lema 3.4.3. *Dado T un árbol enraizado, el algoritmo 18 es capaz de regresar una función struct: $V_T \to \mathcal{P}^m(\mathbb{N})$ tal que, para cualquier $v \in V_T$ en el nivel i, $struct(v)$ es el multiconjunto de los valores de los hijos de v. El algoritmo 18 se ejecuta en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Veamos que la función *struct* que construye el algoritmo 18 efectivamente es correcta. Para ello, sea $v \in V$ un vértice de T en el nivel i, y $struct(v) = (X_v, m_v)$, veamos que para cada hijo u de v tenemos que $vals(u) \in X$, y además $m(vals(u)) = k$, en donde k es el número de hijos de v que tienen como valor a $vals(u)$.

Sabemos que el vértice v está en el nivel i, y por ende sus hijos están en el nivel $i-1$. Y dado que en la línea 5 del algoritmo estamos recorriendo a todos los vértices del nivel $i-1$, entonces podemos asegurar que eventualmente vamos a recorrer a todos los hijos de v.

Sea u uno de los hijos de v. Siguiendo el algoritmo, en la línea 6 obtenemos a su padre (en este caso v) y también obtenemos al respectivo multiconjunto $struct(v) =$

Algoritmo 18: Asignar Estructura

Input: El árbol enraizado T, la función nvl a los niveles de T, la función $vals$ a los valores de los vértices de T, y el nivel i en el que nos encontramos en T.

Output: La función $struct$ en la que dado v, $struct[v]$ regresa el multiconjunto de naturales que corresponden a los valores de los hijos de v.

```
1  struct ← ∅
2  for u ∈ nvl[i] do
3  |  struct[v] ← (∅, ∅)
4  end
5  for u ∈ nvl[i − 1] do
6  |  v ← el padre de u
7  |  (X_v, m_v) ← struct[v]
8  |  if X_v = ∅ ó vals[u] ∉ X_v then
9  |  |  X_v ← X_v ∪ {vals[u]}
10 |  |  m_v[vals[u]] ← 1
11 |  else
12 |  |  m_v[vals[u]] ← m_v[vals[u]] + 1
13 |  end
14 |  struct[v] ← (X_v, m_v)
15 end
16 return struct
```

(X_v, m_v) en la línea 7. Después en la línea 8 verificamos si $vals(u)$ no se encuentra en X_v o si X_v es vacío. Si $vals(u)$ no se encuentra en X_v entonces esta condición se cumple y ejecutamos las líneas 9-10, asegurando de esta forma que $vals(u) \in X_v$. Si u ya se encuentra en X_v entonces ya tenemos lo que buscamos. Independientemente del caso, para cualquier hijo u de v, se cumple que $vals(u) \in X_v$.

Si $k = 1$, entonces la línea 10 del algoritmo nos asegura que $m_v(vals(u)) = k = 1$. Si $k > 1$, entonces por cada ocasión en la que encontremos de nuevo a $vals(u)$ en uno de los hijos de v, ejecutamos la línea 12 del algoritmo y aumentamos en uno el valor de $m_v(vals(u))$. Notemos que sólo recorremos una vez a todos los vértices del nivel $i - 1$, por lo que nunca repetimos vértices, asegurando de esta manera que $m_v(vals(u)) = k$.

Para el análisis de tiempo, en las líneas 2-4 recorremos a todos los vértices del nivel i para inicializar la función $struct$, y en las líneas restantes construimos el resto

de la función recorriendo a todos los vértices del nivel $i-1$. Notemos que por cada vértice que recorremos realizamos operaciones de tiempo constante. En el peor de los casos en el nivel i e $i-1$ se encontran todos los vértices del árbol, por lo que hacemos un total de n recorridos, es decir, el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, por cada iteración no estamos construyendo una nueva función *struct*, sino que trabajamos sobre una misma durante toda la ejecución del algoritmo. Además, sean $v_0, v_1, \ldots, v_m$ los vértices en el nivel i de T, en el peor caso todos los vértices del nivel $i-1$ tienen valores distintos, por lo que $|X_{v_0} \cup X_{v_1} \cup X_{v_m}| = |nvl(i-1)|$, y como $|nvl(i-1)| \leq n-1$, entonces la función *struct* tiene tamaño $O(n)$. □

Algoritmo 19: Construir Sucesion

Input: El árbol enraizado T, la función nvl a los niveles de T, la función *struct* a los multiconjuntos de los valores de los hijos de los vértices de T, y el nivel i en el que nos encontramos en T.

Output: La sucesión S que contiene a todos los multiconjuntos que contienen los valores de los hijos de los vértices en el nivel i, y la función M_S en donde dado un multiconjunto m, $M_S[m]$ es el conjunto de vértices para los que m es el multiconjunto de los valores de sus hijos.

```
1 S ← [];
2 M_S ← ∅
3 for u ∈ nvl[i] do
4 |   if u no es hoja then
5 |   |   Agregamos struct[u] al final de S
6 |   |   M_S[struct[u]] ← M_S[struct[u]] ∪ {u}
7 |   end
8 end
9 return S, M_S
```

Lema 3.4.4. *Dado un árbol enraizado T, el algoritmo 19 es capaz de regresar una sucesión de multiconjuntos S que corresponde a los multiconjuntos de los vértices del nivel i de T y una función $M_S \colon \mathcal{P}^m(\mathbb{N}) \to \mathcal{P}(V_T)$, en donde dado un multiconjunto m, $M_S(m)$ es el conjunto de vértices que tienen como estructura a m. El algoritmo 19 se ejecuta en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Sea $v \in V$ un vértice en el nivel i de T tal que v no es hoja, veamos que $struct(v) \in S$. Para ello observamos que en la línea 3 del algoritmo especificamos que vamos a recorrer a todos los vértices del nivel i, por lo que eventualmente vamos a recorrer a v. Siguiendo el algoritmo, en las líneas 4-7, si el vértice en el que nos encontramos no es hoja entonces agregamos su estructura a la sucesión S por medio de la línea 5. Dado que v no es hoja, entonces su estructura se agrega a S, asegurando de este modo que $struct(v) \in S$.

Después de ejecutar la línea 5, que agrega $struct(v)$ a S, ejecutamos la línea 6 del algoritmo en donde especificamos que $v \in M_S(struct(v))$. Por lo que para todo vértice v que recorremos, si $struct(v)$ es la estructura asociada a v, entonces $v \in M_S(struct(v))$, consiguiendo el resultado que buscábamos con la función M_S.

Para el análisis de tiempo, únicamente estamos recorriendo a todos los vértices del nivel i, y en cada vértice en el que nos paramos hacemos un par de operaciones de tiempo constante. Como la mayor cantidad de vértices que puede tener el nivel i es $n-1$, entonces el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, la longitud máxima que puede tener la sucesión S es $n-1$, pues esto implica que todos los vértices del nivel tienen una estructura asociada a ellos. Por lo que S tiene tamaño $O(n)$. Ahora, M_S también contiene a todos los vértices del nivel i, y por el argumento anterior, M_S tiene tamaño $O(n)$. Por lo tanto el algoritmo utiliza espacio $O(n)$. □

Algoritmo 20: Comparar Sucesiones

Input: Las sucesiones S_1 y S_2.
Output: Determina si las sucesiones son iguales.

```
1  n1 ← |S1|
2  n2 ← |S2|
3  if n1 ≠ n2 then
4  |  return False
5  for i ∈ {0, ..., n1 − 1} do
6  |  if S1[i] ≠ S2[i] then
7  |  |  return False
8  |  end
9  end
10 return True
```

Lema 3.4.5. *El algoritmo 20 es capaz de determinar si dos sucesiones S_1 y S_2 son iguales en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Veamos que si S_1 y S_2 son iguales entonces algoritmo regresa **verdadero**. Empezamos con la ejecución del algoritmo. Lo primero que comparamos en la línea 3 es la cardinalidad de ambas sucesiones, como S_1 y S_2 son iguales entonces la cardinalidad es la misma, por lo que la condición de la línea 3 no se cumple y ejecutamos la línea 5. Dentro de esta ejecución comparamos elemento por elemento para ver que ambas sucesiones tengan los mismos elementos en el mismo orden, como S_1 y S_2 son iguales entonces para todo par de elementos la condición de la línea 6 no se cumple y eventualmente llegamos a la línea 10. Una vez que ya comparamos que las sucesiones sean del mismo tamaño y que tengan los mismos elementos en el mismo orden, entonces concluimos que ambas sucesiones son iguales, regresando así **verdadero**.

Ahora veamos que si S_1 y S_2 son distintas entonces el algoritmo regresa **falso**. Si S_1 y S_2 tienen tamaños distintos entonces se cumple la condición de la línea 3 y el algoritmo regresa **falso**. Por otro lado, si ambas sucesiones tienen el mismo tamaño pero son distintas, entonces debe de existir una $i \in \mathbb{N}$ tal que $S_1[i] \neq S_2[i]$. En la línea 5 del algoritmo recorremos a todos los elementos de S_1 y S_2 al mismo tiempo, por lo tanto eventualmente vamos a dar con los dos elementos que son distintos y cuando sea así la condición de la línea 6 será cierta y el algoritmo regresará **falso**. Concluyendo de esta forma, que independientemente del caso, si S_1 y S_2 son distintas entonces el algoritmo regresa **falso**.

Para el análisis de tiempo, el peor caso es cuando S_1 y S_2 tienen el mismo tamaño y difieren únicamente en el último elemento, por lo que el algoritmo tiene que recorrer a todos los demás elementos hasta llegar a estos últimos. Como las sucesiones tienen tamaño n entonces el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, las únicas variables que utilizamos son aquellas en donde guardamos a las sucesiones y sus cardinalidades, la cardinalidad tiene tamaño constante y las sucesiones tienen tamaño n. Concluyendo así que el algoritmo utiliza espacio $O(n)$. □

Lema 3.4.6. *Dado un árbol enraizado T, el algoritmo 21 modifica la función vals para asignarle un valor a los vértices del nivel i de T. En donde dado $u, v \in V_T$ vértices del nivel i de T, $vals(u) = vals(v)$ si y sólo si $struct(u) = struct(v)$. El algoritmo 21 se ejecuta en tiempo $O(n)$ y utilizando espacio $O(n).$*

Demostración. Sean $u, v \in V_T$ vértices de T del nivel i. Veamos que la función que modifica el algoritmo 21 efectivamente es correcta, es decir, $vals(u) = vals(v)$ si y sólo si $struct(u) = struct(v)$.

Algoritmo 21: Asignar Valores

Input: La sucesión S de multiconjuntos, la función $M_S \colon \mathcal{P}^m(\mathbb{N}) \to \mathcal{P}(V)$, la función nvl a los niveles de T, la función $vals$ a los valores de los vértices de T, y el nivel i en el que nos encontramos en T.

Output: La función $vals$ con los nuevos valores de los vértices del nivel i de T.

```
1  k ← 1
2  n ← |S|
3  for j ∈ {0, ..., n − 1} do
4  |  if j ≠ 0 y S[j] = S[j − 1] then
5  |  |  continue
6  |  for v ∈ M_S[S[j]] do
7  |  |  vals[v] = k
8  |  end
9  |  k ← k + 1
10 end
11 return vals
```

Veamos primero que si $vals(u)$ es igual a $vals(v)$ entonces es porque u y v tienen la misma estructura. Para ello, sea S la sucesión ordenada de multiconjuntos, sabemos que S tiene la siguiente forma:

$$S = (m_1, m_2, \ldots, m_n).$$

Sea $struct(u) = m_j$ la estructura asociada a u, y $struct(v) = m_k$ la estructura asociada a v. El algoritmo recorre a todas las estructuras del nivel por medio de las línea 3-10, por lo que eventualmente vamos a recorrer a m_j y m_k.

Sin pérdida de generalidad supongamos que m_j ocurre antes que m_k en S. Si ya habíamos recorrido a una estructura m_l en S tal que $m_l = m_j$, entonces ya se ejecutaron las línea 6-8 del algoritmo en donde le asignamos a todos los vértices que tengan como estructura m_l un valor x, asegurando de esta forma que $vals(u) = x$. De otro modo, cuando recorremos a m_j le asignamos el valor x a u. Dado que $vals(u)$ es igual a $vals(v)$, la única forma en la que se diera esto es si para todas estructuras que siguen después de m_j se cumple la condición de la línea 4; es decir, todas las estructuras entre m_j y m_k son iguales. Concluyendo así lo que buscábamos.

Ahora veamos que si $vals(u)$ es distinto de $vals(v)$ entonces es porque u y v tienen estructuras diferentes. Sea S la sucesión ordenada de multiconjuntos, $struct(u) = m_j$

la estructura asociada a u y $struct(v) = m_k$ la estructura asociada a v. Sin pérdida de generalidad supongamos que m_j ocurre antes que m_k en S.

Al momento en el que recorremos a m_j, si es la primera vez que vemos a este tipo de estructura, entonces le asignamos un valor x a todos los vértices a los que les corresponda esta estructura, entre ellos a u, por medio de las líneas 6-8. Después ejecutamos la línea 9, en donde se especifica que el nuevo valor que se le va a asociar a las nuevas estructuras que veamos será de $x + 1$.

Seguimos recorriendo a las estructuras que siguen después de m_j, y para todas aquellas que sean igual a m_j, ejecutamos las líneas 4-5, en donde no modificamos el valor asociado a estas estructuras. Pero dado que $vals(u)$ es distinto a $vals(v)$, esto implica que existe una estructura después de m_j que difiere de las anteriores (puede ser m_k), a la cual se le atribuye el nuevo valor de $x + 1$. Asegurando de esta forma que u y v tienen valores distintos y por ende estructuras diferentes.

Cabe mencionar que la sucesión S está ordenada. Por lo que sean m_i, m_j, m_k multiconjuntos de S, en donde m_i ocurre antes que m_j y m_j ocurre antes que m_k, entonces no se puede dar el caso en donde $m_i \neq m_j$ pero $m_i = m_k$.

Para el análisis de tiempo, recorremos a cada estructura única contenida en S, y por cada estructura recorremos al conjunto de vértices a los que se les asoció dicha estructura. Sea n el tamaño de la sucesión S, en el peor caso tenemos n estructuras distintas pero esto implica que cada vértice del nivel i tiene una estructura diferente, por lo que por cada estructura que recorremos solo vamos a recorrer un vértice. Así incluso en el peor caso recorremos a los n elementos de S y a los n vértices del nivel una sola vez, es decir, hacemos un total de $n+n$ recorridos. El algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, usamos a la sucesión ordenada S de tamaño n, a la función M_S la cual sabemos que utiliza espacio $O(n)$, y a la función $vals$ que también tiene tamaño $O(n)$. En conjunto, el algoritmo utiliza espacio $O(n)$. □

Lema 3.4.7. *Dado dos árboles enraizados T_1, T_2 e $i \in \mathbb{N}$, el algoritmo 22 determina si ambos árboles comparten la misma estructura desde el nivel i en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Sean T_1 y T_2 dos árboles enraizados. Supongamos primero que T_1 y T_2 tienen la misma estructura en todos los niveles, veamos que el algoritmo 22 regresa **verdadero**. Procedemos por inducción sobre el número de niveles de T_1 que tenemos que verificar que sean iguales a los de T_2.

El paso inductivo es cuando tenemos que verificar los últimos k niveles de T_1. Empezamos con la ejecución del algoritmo, nos colocamos en el nivel $n - k$ de T_1 y

Algoritmo 22: Verificacion por Niveles

Input: Los árboles enraizados T_1 y T_2, las funciones $vals_1$ y $vals_2$ a los valores de los vértices de T_1 y T_2, las funciones nvl_1 y nvl_2 a los niveles de T_1 y T_2, el nivel i en el que nos encontramos actualmente en T y la altura h de T_1 y T_2.

Output: Un booleano que nos indica si T_1 y T_2 comparten la misma estructura desde el nivel n

```
1  struct_1 ← AsignarEstructura(T_1, nvl_1, vals_1, i)
2  struct_2 ← AsignarEstructura(T_2, nvl_2, vals_2, i)
3  S_1, M_{S_1} ← ConstruirSucesion(T_1, S_1, M_{S_1}, nvl_1, struct_1, i)
4  S_2, M_{S_2} ← ConstruirSucesion(T_2, S_2, M_{S_2}, nvl_2, struct_2, i)
5  S'_1 ← la sucesión S_1 ordenada utilizando el algoritmo 9
6  S'_2 ← la sucesión S_2 ordenada utilizando el algoritmo 9
7  if ¬CompararSucesiones(S'_1, S'_2) then
8  |  return False
9  if i < h then
10 |  vals_1 ← AsignarValores(S'_1, M_{S_1}, nvl_1, vals_1, struct_1, i)
11 |  vals_2 ← AsignarValores(S'_2, M_{S_2}, nvl_2, vals_2, struct_2, i)
12 |  return VerificacionPorNiveles(T_1, T_2, vals_1, vals_2, nvl_1, nvl_2, i + 1, h)
13 else
14 |  return True
15 end
```

T_2. Ejecutamos las líneas 1-2, la cual le asignan una estructura a los vértices de este nivel por medio de las funciones $struct_1$ y $struct_2$. La construcción de estas funciones se realiza por medio del algoritmo 18. Así, ejecutamos las líneas 3-4, en donde construimos las sucesiones S_1 y S_2 para T_1 y T_2 respectivamente. Estas sucesiones contienen a las estructuras que acabamos de definir. De igual manera construimos las funciones M_{S_1} y M_{S_2}, en la cual dado un multiconjunto $s = (\mathbb{N}, m)$, $M_S(s)$ es el conjunto de vértices a los que les corresponde la estructura s. Lo definido anteriomente es construido mediante el algoritmo 19. Después, ejecutamos las líneas 5-6, en donde las sucesiones S_1 y S_2 son ordenadas por medio del algoritmo 9, para así comparar ambas sucesiones en la línea 7 utilizando el algoritmo 20. Como T_1 y T_2 tienen la misma estructura en todos los niveles, entonces estas sucesiones son iguales, y por ende la ejecución del algoritmo continúa. Dado que no nos encontramos en el último nivel de T_1, entonces ejecutamos las líneas 9-12, en las cuales utilizamos las sucesiones S_1 y S_2 para asignarle un nuevo valor a los vértices del nivel $n-k$ de T_1 y

T_2 por medio del algoritmo 21. Así, todos los vértices del nivel actual ahora cuentan con un valor y el número de niveles de T_1 por verificar ha disminuido en uno, y por hipótesis inductiva, el algoritmo regresa **verdadero**.

Ahora supongamos que existe $j \in \mathbb{N}$ tal que la estructura del nivel j de T_1 es diferente a la de T_2. Veamos que el algoritmo 22 regresa **falso** al ser ejecutado en el nivel j. Empezamos con la ejecución del algoritmo, notemos que las funciones y sucesiones construidas en las líneas 1-6 son correctas pues ya demostramos que cada una de las rutinas que usamos son correctas. De esta forma las sucesiones ordenadas S_1 y S_2 son diferentes, por lo que la condición de la línea 7 se cumple, y el algoritmo regresa **falso**.

Para el análisis de tiempo. Dada una sucesión de multiconjuntos S, el algoritmo 9 toma tiempo $O(\mathcal{M} + k)$ donde $\mathcal{M} = \sum_{M \in S} |M|$ y k es el valor más grande que encontramos en los multiconjuntos de S. En este caso, cada valor de los multiconjuntos corresponde a un vértice del nivel inferior, por lo que $\mathcal{M}$ es a lo más $n-1$. Por otro lado, si todos los vértices del nivel inferior tienen valores distintos, entonces el valor más grande que podemos alcanzar es $n-1$. El algoritmo anterior entonces toma tiempo $O((n-1)+(n-1)) = O(n)$. Para todas las demás rutinas utilizamos tiempo $O(m)$ donde m es el número de vértices en el nivel en el que nos encontramos. En el peor de los casos el nivel puede tener hasta $n-1$ vértices, por lo que cada una de las rutinas toma tiempo $O(n)$, por ende el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, para las variables *struct*, S, M_S ya demostramos que utilizan espacio $O(m)$ donde m es el número de vértices en el nivel, y como se argumentó anteriormente, en el peor de los casos estas funciones utilizan espacio $O(n)$. De igual forma, para ordenar a las sucesiones utilizamos espacio $O(\mathcal{M})$, y como se argumentó anteriormente, este valor es a lo más $n-1$; el algoritmo de ordenamiento (algoritmo 9) usa espacio $O(n)$. Así, el algoritmo toma espacio $O(n)$. □

Ahora que ya demostramos que cada una de las rutinas auxiliares son correctas, presentamos el pseudocódigo y procedemos a demostrar la corrección del algoritmo 23.

Teorema 3.4.8. *El algoritmo 23 es capaz de detectar si dos árboles enraizados son isomorfos en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Sean T_1 y T_2 dos árboles enraizados. Supongamos primero que T_1 y T_2 son isomorfos y veamos que el algoritmo devuelve **verdadero**. Procedemos por inducción sobre el número de niveles que tiene T_1.

El caso base es cuando T_1 tiene un solo nivel, el nivel cero. En este caso T_1 sólo está conformado por su raíz. Como T_2 es isomorfo a T_1, T_2 también sólo está

Algoritmo 23: Isomorfismo de Árboles Enraizados

Input: Los árboles enraizados T_1 y T_2.
Output: Un booleano que indica si los árboles son o no isomorfos.

```
1  if |V_{T_1}| ≠ |V_{T_2}| then
2  |  return False
3  h_1 ← la altura del árbol T_1
4  h_2 ← la altura del árbol T_2
5  if h_1 ≠ h_2 then
6  |  return False
7  if h_1 = 0 then
8  |  return True
9  else
10 |  nvl_1 ← ObtenerNiveles(T_1, h_1)
11 |  nvl_2 ← ObtenerNiveles(T_2, h_2)
12 |  for i ∈ {0, ..., h_1 − 1} do
13 |  |  if |nvl_1[i]| ≠ |nvl_2[i]| then
14 |  |  |  return False
15 |  end
16 |  vals_1 ← PrepararValores(T_1)
17 |  vals_2 ← PrepararValores(T_2)
18 |  return VerificacionPorNiveles(T_1, T_2, vals_1, vals_2, nvl_1, nvl_2, 1, h_1)
19 end
```

conformado por su raíz. Empezamos con la ejecución del algoritmo, observamos que la condición de la línea 1 falla pues ambos árboles tienen el mismo número de vértices, y la condición de la línea 5 también falla pues ambos árboles tienen la misma altura. Dado que T_1 tiene un solo nivel, es decir, altura cero, entonces la condición de la línea 7 es verdadera, ejecutamos la línea 8 y regresamos **verdadero**.

El paso inductivo es cuando T_1 tiene n niveles. Empezamos con la ejecución del algoritmo, y por los argumentos anteriores, podemos ver que las condiciones de las líneas 1 y 5 fallan. Dado que la altura del árbol ya no es cero, entonces la condición de la línea 7 falla, y por ende ejecutamos las líneas 9-19. En las líneas 10-11 se obtienen la funciónes nvl_1 y nvl_2 para T_1 y T_2 respectivamente, en la cual dado $i \in \mathbb{N}$, $nvl(i)$ es el conjunto de vértices que hay en el nivel i. Ya demostramos que el algoritmo 14 es correcto, por lo que la función nvl es construida correctamente. Después, en las líneas 12-15, utilizamos las funciones nvl_1 y nvl_2 para comparar que

en cada nivel T_1 y T_2 tengan la misma cantidad de vértices, como ambos árboles son isomorfos entonces esta condición se cumple y la ejecución continúa. En las líneas 16-17 definimos los valores iniciales de los vértices de T_1 y T_2 por medio de la construcción de las funciones $vals_1$ y $vals_2$. De igual manera, demostramos que el algoritmo 16 construye estas funciones correctamente. Así, ejecutamos la línea 18 y llamamos al algoritmo 22, el cual también sabemos que es correcto. Dentro de esta llamada, verificamos desde el nivel 1 que T_1 y T_2 tengan la misma estructura, dado que ambos árboles son isomorfos entonces todas las verificaciones se cumplen y de esta forma el algoritmo regresa **verdadero**.

Ahora supongamos que T_1 y T_2 no son isomorfos y veamos que el algoritmo 23 regresa **falso**. Si T_1 y T_2 no son isomorfos, puede ser porque difieren en el número de vértices, si este es el caso entonces se cumple la condición de la línea 1 y el algoritmo regresa **falso**. También es posible que T_1 y T_2 tengan la misma cantidad de vértices pero la altura de ambos es diferente, en dado caso se cumple la condición de la línea 5 y el algoritmo regresa **falso**. Por otro lado, es posible que T_1 y T_2 tengan la misma cantidad de vértices y la misma altura, pero exista $i \in \mathbb{N}$ tal que la cantidad de vértices que tiene el nivel i de T_1 es diferente a la de T_2, si este es el caso, las líneas 12-15 detectan esta diferencia y el algoritmo regresa **falso**. Si todo lo anterior se cumple, entonces existe un vértice v en un nivel $j \in \mathbb{N}$ de T_1, tal que la estructura de v no se encuentra presente en el nivel j de T_2. En la línea 18 indicamos que vamos a verificar la estructura de todos los niveles empezando por el nivel 1, entonces al llamar al algoritmo 22 en el nivel j, este regresa **falso**, por lo que algoritmo también regresa **falso**. Concluimos así que, independientemente del caso, si T_1 y T_2 no son isomorfos entonces el algoritmo 23 regresa **falso**.

Para el análisis de tiempo, observamos que las comparaciones, el acceso al elemento de un conjunto, y el obtener la imagen bajo una función M de un elemento se pueden hacer en tiempo constante. Obtener la cantidad de vértices y la altura de T_1 y T_2 lo podemos obtener haciendo un recorrido DFS, el cual sabemos que se puede realizar en tiempo $O(n)$. La construcción de las funciones nvl y $vals$ ya demostramos que toman tiempo $O(n)$. La comparación para verificar que para todo nivel $i \in \mathbb{N}$, T_1 y T_2 tengan la misma cantidad de vértices toma tiempo $O(k)$, donde k es el número de niveles que tiene T_1, sabemos que $k \leq n$ por lo que esta operación toma tiempo $O(n)$. También recordemos que el algoritmo 22 toma tiempo $O(m)$, donde m es el número de vértices que hay en el nivel, y como la suma de vértices en todos los niveles es n, entonces la suma de los tiempos de todas las llamadas de cada subrutina es $O(n)$. Concluyendo así, que el algoritmo 23 toma tiempo $O(n)$.

Para el análisis de espacio, observamos que las variables que guardan la cardinalidad y altura de los árboles utilizan espacio constante. Para las funciones nvl y $vals$

demostramos que utilizan espacio $O(n)$. Para las funciones *struct*, S y M_S demostramos que utilizan espacio $O(m)$ donde m es la cantidad de vértices que hay en el nivel, y como se argumentó anteriormente, la suma de vértices en todos los niveles es n, por lo que la suma de los espacios de todas las llamadas de cada subrutina es $O(n)$. Así, el algoritmo 23 toma espacio $O(n)$. □

Para terminar esta sección, notemos que en el teorema anterior utilizamos de forma implícita que árboles isomorfos tienen la misma estructura, sin embargo, esta observación no es del todo trivial. Veamos que efectivamente el que dos árboles enraizados tengan la misma estructura es equivalente a que sean isomorfos. Haremos esta demostración en las siguientes dos proposiciones.

Proposición 3.4.9. *Sean T_1 y T_2 árboles enraizados. Si T_1 y T_2 son isomorfos, entonces T_1 y T_2 tienen la misma estructura.*

Demostración. Sean $i \in \mathbb{N}$, $\varphi\colon V_{T_1} \to V_{T_2}$ un isomorfismo y $v \in V_{T_1}$. Veamos que v está en el nivel i de T_1 si y sólo si $\varphi(v)$ está en el nivel i de T_2, y que v y $\varphi(v)$ tienen la misma estructura. Procedemos por inducción sobre el nivel de T_1.

Para el caso base, el nivel 0, recordemos que todos los vértices en este nivel son hojas. De esta forma sólo queda demostrar que, v está en el nivel 0 si y sólo si $\varphi(v)$ está en el nivel 0. Sea v un vértice del nivel 0 de T_1, r la raíz de T_1, y $P = (v_0 = v, v_1, \ldots, v_k = r)$ una vr-trayectoria en T_1. Notemos que $\varphi(v_0)$ no puede tener un hijo $\varphi(u)$, pues en caso de que lo tuviera, por la definición de isomorfismo, u tendría que ser hijo de v_0, lo cual no es posible pues v_0 es una hoja. Concluyendo así que $\varphi(v_0)$ también es una hoja.

Ahora, si $\varphi(v_0)$ no está en el nivel 0, entonces existe otro vértice $\varphi(w_0)$ en T_2 que se encuentra en el nivel 0. Sea $Q' = (\varphi(w_0), \varphi(w_1), \ldots, \varphi(w_m) = \varphi(r))$ una $\varphi(w_0)\varphi(r)$-trayectoria en T_2. Como T_1 y T_2 son isomorfos, entonces $Q = (w_0, w_1, \ldots, w_m = r)$ es una w_0r-trayectoria en T_1. Sin embargo si comparamos a P con Q, podemos observar que Q tiene longitud mayor, lo que implicaría que existen vértices en niveles inferiores a v_0, contradiciendo que v_0 se encuentra en el nivel 0. Así, concluimos que $\varphi(v_0)$ también se encuentra en el nivel 0. Recíprocamente, si $\varphi(v)$ está en el nivel 0, es análogo verificar que v está en el nivel 0.

Para el paso inductivo, notemos que el mismo argumento del caso base funciona para demostrar que v está en el nivel k de T_1 si y sólo si $\varphi(v)$ está en el nivel k de T_2. Por lo que lo único que falta es demostrar que v y $\varphi(v)$ tienen la misma estructura.

Sea $\mathcal{N}_v$ el conjunto que contiene a los hijos de v, sabemos por la definición de isomorfismo que $w \in \mathcal{N}_v$ si y sólo si $\varphi(w) \in \mathcal{N}_{\varphi(v)}$. Además, sabemos que si v se encuentra en el nivel i, entonces cualquiera de sus hijos se encuentra en el nivel

$i-1$, y lo mismo para $\varphi(v)$ y sus hijos. Así, por hipótesis inductiva, todo vértice $w \in \mathcal{N}_v$ tiene la misma estructura que $\varphi(w) \in \mathcal{N}_{\varphi(v)}$, es decir, los hijos de v tienen la misma estructura que los hijos de $\varphi(v)$. Por ende, v y $\varphi(v)$ comparten la misma estructura. □

Proposición 3.4.10. *Sean T_1 y T_2 árboles enraizados. Si T_1 y T_2 tienen la misma estructura, entonces T_1 y T_2 son isomorfos.*

Demostración. Construiremos una biyección φ entre V_{T_1} y V_{T_2}, y verificaremos que es un isomorfismo de árboles enraizados de T_1 a T_2. La idea es realizar un recorrido BFS simultáneamente en los dos árboles, utilizando el hecho de que T_1 y T_2 tienen la misma estructura para decidir a qué vértice nos vamos a mover en cada recorrido.

Para empezar, sean r_1 y r_2 las raíces de T_1 y T_2, respectivamente. Definimos $\varphi(r_1) = r_2$. Si $v_0, \ldots, v_n$ es un recorrido de BFS de T_1, con $v_0 = r_1$, y suponiendo que $\varphi(v_j)$ está definido para cada $j \leq i$, veamos cómo definir a $\varphi(v_{i+1})$. Si v_i no es una hoja, entonces v_{i+1} es un hijo de v_i. Como T_1 y T_2 tienen la misma estructura, en T_2 existe un hijo w de $\varphi(v_i)$ que tiene la misma estructura que v_{i+1} y que no ha sido definido como la imagen de vértice alguno de T_1 bajo φ. Entonces definimos $\varphi(v_{i+1}) = w$. Si v_i es una hoja, entonces existe un ancestro v_j de v_i tal que v_{i+1} es un hijo de v_j. Análogamente al caso anterior, existe un hijo w de $\varphi(v_j)$ que tiene la misma estructura que v_{i+1} y que no ha sido definido como la imagen de vértice alguno de T_1 bajo φ. Definimos $\varphi(v_{i+1}) = w$. Vale la pena recalcar que la existencia de $\varphi(v_{i+1})$ está garantizada por la hipótesis, es decir, dado que T_1 y T_2 tienen la misma estructura. Por la forma en la que se construyó φ, resulta ser claramente una biyección.

Ahora, veamos que φ efectivamente es un isomorfismo de árboles enraizados de T_1 a T_2. Procedemos verificando la definición de isomorfismo de árboles enraizados por inducción sobre la distancia entre la raíz de T_1 y el vértice $v \in V_{T_1}$. Para el caso base, cuando $d(r_1, v) = 0$, tenemos que $r_1 = v$, y por la construcción de la función tenemos que $\varphi(r_1) = r_2$, la cual es la definición para el isomorfismo en la raíz. Para el paso inductivo, consideremos $d(v, r_1) = k$, y sea

$$T = (v_0 = v, v_1, \ldots, v_{k-1}, v_k = r_1),$$

la vr_1-trayectoria en T_1. Sabemos que $d(v_i, r_1) = k - i$ para $i \in \{1, \ldots, k\}$. Además, v_{i+1} es el padre de v_i para cada $i \in \{0, \ldots, k-1\}$. Por hipótesis inductiva, sabemos que φ esta bien definida para v_i con $i \in \{1, \ldots, k\}$ es decir, $\varphi(v_{i+1})$ es el padre de $\varphi(v_i)$. Lo anterior implica que en algún momento del recorrido BFS de T_1, estando en v_1 exploramos el vértice v, por lo que al construir a φ, en T_2 se eligió un hijo de $\varphi(v_1)$ con la misma estructura que v como $\varphi(v)$. Por lo tanto, el padre de $\varphi(v)$

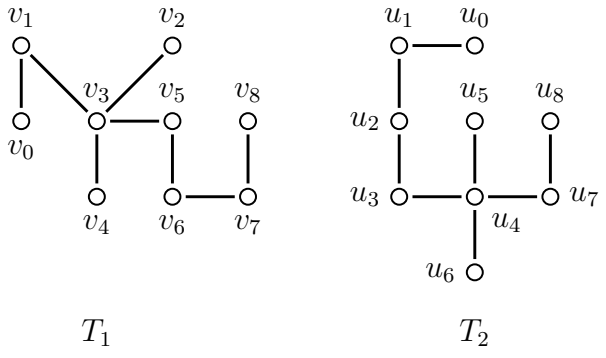
es $\varphi(v_1)$, como se quería demostrar. Concluimos que φ es un isomorfismo de árboles enraizados de T_1 a T_2. □

Lema 3.4.11. *Dos árboles enraizados son isomorfos si y sólo si tienen la misma estructura.*

Demostración. Utilizando los resultados obtenidos en 3.4.9 y 3.4.10, podemos observar que el enunciado es cierto. □

3.5. Isomorfismo de árboles arbitrarios

Dos árboles T_1 y T_2, con $T_1 = (V_1, E_1)$ y $T_2 = (V_2, E_2)$, son isomorfos si existe una biyección $\varphi\colon V_1 \to V_2$ tal que para cualesquiera vértices $u, v \in V_1$ se tiene que $uv \in E_1$ si y sólo si $\varphi(u)\varphi(v) \in E_2$. Si se cumple la condición anterior entonces φ es un **isomorfismo de árboles** de T_1 a T_2, y lo denotamos por $T_1 \overset{\varphi}{\cong} T_2$, o simplemente $T_1 \cong T_2$, si no es necesario indicar el nombre del isomorfismo.



$$\varphi(v_0) = u_8, \quad \varphi(v_5) = u_3,$$
$$\varphi(v_1) = u_7, \quad \varphi(v_6) = u_2,$$
$$\varphi(v_2) = u_6, \quad \varphi(v_7) = u_1,$$
$$\varphi(v_3) = u_4, \quad \varphi(v_8) = u_0,$$
$$\varphi(v_4) = u_5.$$

Figura 3.9: Ejemplo de dos árboles isomorfos.

3.5.1. Algoritmo para el isomorfismo de árboles arbitrarios

Sean T_1 y T_2 los árboles a comparar. La idea general del algoritmo es primero encontrar los centros de ambos árboles, después comparamos que tengan la misma cantidad de centros, si difieren entonces los árboles **no son isomorfos**, de lo contrario construimos dos árboles enraizados haciendo BFS desde el centro (o los centros) de T_1 y T_2 para así ejecutar el algoritmo 23 y determinar si son isomorfos o no.

Veamos un ejemplo de una ejecución del algoritmo 25 para detectar un isomorfismo entre dos árboles arbitrarios. Los dos árboles con los que vamos a trabajar son los de la figura 3.9.

Para poder encontrar el centro (o los centros) de un árbol T, vamos a tomar a un vértice arbitrario $v \in V_T$ y vamos a ejecutar BFS tomando a v como raíz. Sea x el último vértice recorrido, ejecutamos de nuevo BFS pero ahora tomando a x como raíz. Ahora sea y el último vértice recorrido, nótese que la longitud de la xy-trayectoria es el diámetro de T, por lo que los vértices intermedios de esta xy-trayectoria son los centros de T; todo lo anterior se explica a mayor profundidad en la demostración del algoritmo 24. Siguiendo el ejemplo, supongamos que los vértices tomados arbitrariamente son v_1 y u_5 para T_1 y T_2 respectivamente, ejecutar BFS desde estos vértices produce los siguientes árboles:

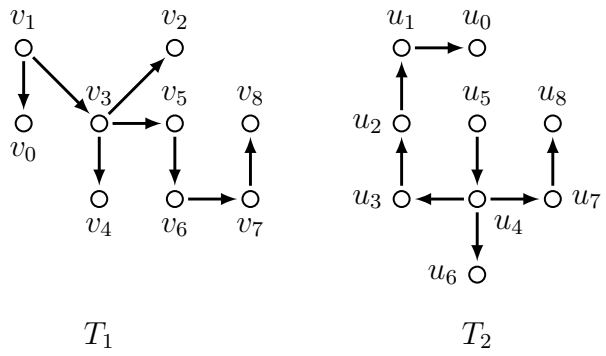


Figura 3.10: Árboles BFS obtenidos de tomar a v_1 y u_5 como raíz.

Como podemos observar, los últimos vértices en ser recorridos son v_8 y u_0 respectivamente. Ejecutamos BFS tomando a estos vértices como raíces para obtener los siguientes árboles:

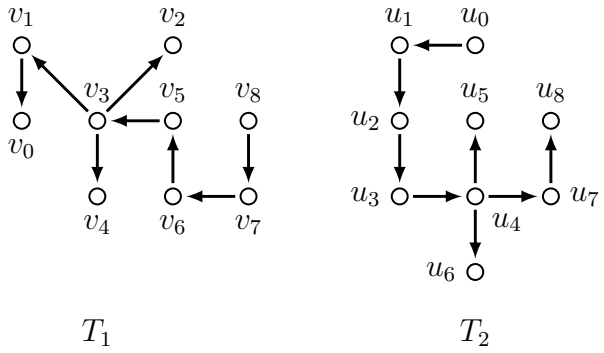


Figura 3.11: Árboles BFS obtenidos de tomar a v_8 y u_0 como raíz.

Ahora, los últimos vértices en ser recorridos son v_0 y u_8 respectivamente. La v_0v_8-trayectoria en T_1 es $P_1 = (v_0, v_1, v_3, v_5, v_6, v_7, v_8)$, por lo que el centro de T_1 es v_5. La u_0u_8-trayectoria en T_2 es $P_2 = (u_0, u_1, u_2, u_3, u_4, u_7, u_8)$, por lo que el centro de T_2 es u_3. Como ambos árboles tienen un solo centro entonces podemos construir un árbol enraizado en v_5 y u_3 mediante BFS, sean T_1' y T_2' respectivamente, los árboles se ven de la siguiente forma:

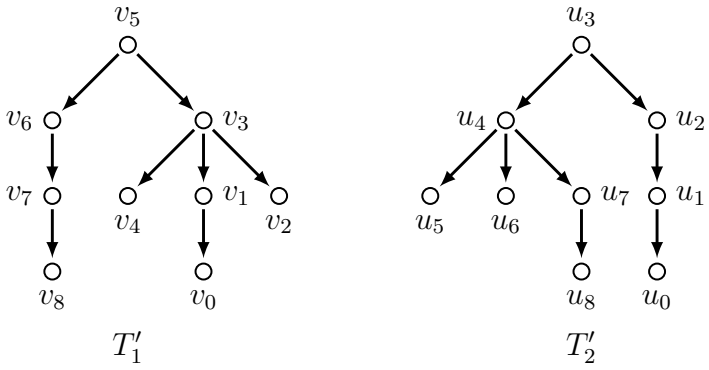


Figura 3.12: Árboles enraizados a partir de T_1 y T_2.

Así, al ejecutar el algoritmo 23 obtenemos **verdadero**, y concluimos de esta manera que T_1 y T_2 son dos árboles isomorfos.

La razón por la que funciona el procedimiento anterior es por el siguiente enunciado:

Proposición 3.5.1. *Si T_1 y T_2 son árboles, entonces T_1 y T_2 son isomorfos si y sólo si existen $u \in V_{T_1}$ y $v \in V_{T_2}$ tal que el árbol T_1 enraizado en u es isomorfo a T_2 enraizado en v.*

Demostración. Primero supongamos que T_1 y T_2 son isomorfos. Sea $\varphi\colon V_{T_1} \to V_{T_2}$ el isomorfismo de T_1 a T_2. Sea $v \in V_{T_1}$, enraizamos al árbol T_1 tomando a v como la raíz mediante BFS, y enraizamos al árbol T_2 tomando a $\varphi(v)$ como la raíz. Dado que T_1 y T_2 son isomorfos, entonces los árboles enraizados que obtenemos tienen la misma estructura y por ende también son isomorfos.

Ahora supongamos que T_1 y T_2 no son isomorfos. Observamos que no es posible que existan $u \in V_{T_1}$ y $v \in V_{T_2}$ tal que T_1 enraizado en u sea isomorfo a T_2 enraizado en v, pues en caso de que sí lo fuera, el isomorfismo obtenido anteriormente sería un isomorfismo de T_1 a T_2. □

Como podemos observar, en general, si T_1 y T_2 son isomorfos, sea φ el isomorfismo de T_1 a T_2, entonces para todo vértice $v \in V_{T_1}$ se tiene que el árbol T_1 enraizado en v va a ser isomorfo al árbol T_2 enraizado en $\varphi(v)$. Sin embargo, dado que T_1 y T_2 son arbitrarios, entonces no podemos simplemente considerar dos vértices u y v arbitrarios de T_1 y T_2 y definir $\varphi(u) = v$. Los vértices u y v deben tener una propiedad que los distinga de todos los demás vértices, por ejemplo que sean hojas. De esta forma, si para cada posible pareja de hojas h_1 y h_2 en T_1 y T_2 comparamos que T_1 enraizado en h_1 sea isomorfo a T_2 enraizado en h_2, entonces eventualmente podríamos determinar si T_1 y T_2 son isomorfos. Pero lo anterior implica ejecutar el algoritmo 23 para cada posible pareja de hojas que haya en T_1 y T_2, y el número de hojas que puede tener un árbol está en el orden de $O(n)$, por lo que en el peor de los casos, un algoritmo para determinar si dos árboles son isomorfos basado en hojas puede tener complejidad $O(n^3)$; no es factible. Por este motivo buscamos a los centros de un árbol. Dado que los árboles tienen a lo más 2 centros, entonces reducimos el espacio de búsqueda de los vértices que menciona la proposición 3.5.1. Veamos entonces que lo anterior es cierto:

Proposición 3.5.2. *Sean T_1 y T_2 árboles, y $\mathcal{C}_1$ y $\mathcal{C}_2$ los conjuntos que contienen a sus respectivos centros. T_1 y T_2 son isomorfos si y sólo si existe una biyección $\phi\colon \mathcal{C}_1 \to \mathcal{C}_2$, tal que para todo $u \in \mathcal{C}_1$, se tiene que T_1 enraizado en u es isomorfo a T_2 enraizado en $\phi(u)$.*

Demostración. Primero supongamos que T_1 y T_2 son isomorfos, sea $\varphi\colon V_{T_1} \to V_{T_2}$ el isomorfismo entre ambos árboles. Sea $u \in \mathcal{C}_1$, sabemos por la proposición 3.5.1 que T_1 enraizado en u es isomorfo a T_2 enraizado en $\varphi(u)$. Así, entonces la excentricidad de $\varphi(u)$ debe de ser la misma que la excentricidad de u; de lo contrario la estructura de ambos árboles sería distinta y por ende no serían isomorfos. Además, no es posible que exista otro vértice $\varphi(w)$ con menor excentricidad a $\varphi(u)$ en T_2, ya que de ser así, w tendría menor excentricidad que u, y por ende u no sería uno de los centros

de T_1. De esta manera concluímos que si $u \in \mathcal{C}_1$ entonces $\varphi(u) \in \mathcal{C}_2$. El argumento anterior también funciona para determinar que si $v \in \mathcal{C}_2$ entonces $\varphi^{-1}(v) \in \mathcal{C}_1$. Concluyendo así que los centros de un árbol se preservan bajo un isomorfismo, es decir, el isomorfismo φ restringido a $\mathcal{C}_1$ es la función biyectiva que estábamos buscando.

Ahora supongamos que existe una biyección $\phi\colon \mathcal{C}_1 \to \mathcal{C}_2$, tal que para todo $u \in \mathcal{C}_1$, se tiene que T_1 enraizado en u es isomorfo a T_2 enraizado en $\phi(u)$. Sea φ el isomorfismo entre T_1 enraizado en u y T_2 enraizado en v, observemos que φ es un isomorfismo entre T_1 y T_2; concluyendo de esta forma que T_1 y T_2 son isomorfos. □

De esta forma, resulta práctico encontrar estos centros, y comparar para cada posible pareja de centros que los árboles enraizados en ellos sean isomorfos. Como en el peor de los casos estaríamos ejecutando el algoritmo 23 a lo más un total de 4 veces, entonces determinar si dos árboles son isomorfos o no lo podemos hacer en tiempo lineal. Así, presentamos el algoritmo 24 con el cual podemos encontrar los centros de un árbol.

Algoritmo 24: Encontrar los Centros de un Árbol

Input: Un árbol T.
Output: El conjunto $\mathcal{C}$ que contiene los centros del árbol T.

```
1  C ← ∅
2  v ← un vértice arbitrario de V_T
3  x ← el último vértice recorrido al hacer BFS en T con v como raíz
4  y ← el último vértice recorrido al hacer BFS en T con x como raíz
5  P ← la xy-trayectoria en T
6  n ← |P|
7  if n mód 2 ≠ 0 then
8  |  C ← {P[⌊n/2⌋]}
9  else
10 |  C ← {P[n/2 − 1], P[n/2]}
11 end
12 return C
```

Lema 3.5.3. *Dado un árbol T, el algoritmo 24 es capaz de encontrar los centros de T en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Sea $\mathcal{C}$ el conjunto de vértices que regresa el algoritmo. Veamos que $v \in \mathcal{C}$ efectivamente es un centro de T. Para ello observemos que definimos $\mathcal{C}$ a partir

de los vértices x y y que obtenemos en las líneas 3 y 4 del algoritmo. La razón es porque la longitud de la xy-trayectoria P es el diámetro de T, es decir, la trayectoria de máxima longitud, y por ende los vértices que se encuentran a la mitad de P son los centros de T. Veamos que efectivamente la longitud de P es el diámetro de T.

Sabemos que en todo árbol una trayectoria de longitud máxima debe tener hojas como vértices inicial y final. Por otro lado, al ser el último vértice en un recorrido BFS, sabemos que tanto x como y son hojas de T. Veamos entonces que si tomamos cualesquiera dos hojas z y w de T, la distancia de z a w es menor o igual que la distancia de x a y. Observemos primero que la distancia máxima entre un par de vértices en T se puede alcanzar tomando a x como uno de los vértices.

Sea B el árbol de BFS (B es un árbol enraizado) con raíz v, y sean v_{xz}, v_{xw} y v_{wz} los ancestros comunes más profundos de x y z, x y w, y w y z, respectivamente. Recordemos que todos los ancestros de un vértice s en B se encuentran en la única trayectoria vTs de v a s en T. Por lo tanto, v_{xz} y v_{xw} están emparentados en B, y análogamente v_{xz} y v_{wz}, y v_{xw} y v_{wz} están emparentados en B (es decir, uno es ancestro del otro, y posiblemente son iguales). Consideremos entonces los siguientes casos.

- Si $v_{xz} = v_{xw}$, entonces v_{xz} es ancestro común de w y de z, y por lo tanto, es ancestro de v_{wz}. Suponiendo, sin pérdida de generalidad, que la profundidad de z es mayor o igual a la profundidad de w, y recordando que la profundidad de x es mayor o igual a la de cualquier otro vértice, entonces tenemos que $\ell(wTv_{wz}) \leq \ell(xTv_{xz})$, de donde se sigue que $\ell(wTv_{wz}Tz) \leq \ell(xTv_{xz}Tv_{wz}Tz)$. Por lo tanto, $d(w, z) \leq d(x, z)$.

- Si $v_{xz} = v_{wz}$, entonces v_{wz} es ancestro común de x y w, por lo que también es ancestro de v_{xw}. Como la profundidad de x es mayor o igual que la profundidad de w en B, tenemos que $\ell(wTv_{xw}Tv_{xz}) \leq \ell(xTv_{xz})$. De lo anterior, se desprende que $\ell(wTv_{xw}Tv_{xz}Tz) \leq \ell(xTv_{xz}Tz)$, por lo que $d(w, z) \leq d(x, z)$. El caso en el que $v_{xw} = v_{wz}$ es análogo.

- Si v_{xz} es ancestro de v_{xw}, entonces, v_{xz} es ancestro común de w y z, por lo que también debe ser ancestro de v_{wz}. Si v_{xw} es ancestro de v_{wz}, entonces es ancestro común de x y z, por lo que resulta ser ancestro de v_{xz}, y tenemos $v_{xz} = v_{xw}$, caso que ya cubrimos. Por lo tanto, v_{wz} es ancestro de v_{xw}, y se sigue que v_{wz} es ancestro común de x y z, por lo que también es ancestro de v_{xz}. Esto último implica que $v_{xz} = v_{wz}$, caso que ya cubrimos. El caso en el que v_{xw} es ancestro de v_{xz} es análogo.

Como ya observamos, necesariamente v_{xz} es ancestro de v_{xw} o viceversa, por lo tanto, los casos considerados son exhaustivos. Se concluye que la distancia máxima entre un par de vértices en T se puede obtener tomando a x como uno de los dos vértices. Como y es un vértice a distancia máxima de x en T, se sigue que la trayectoria P es de longitud máxima en T.

Para el análisis de tiempo observemos que los dos recorridos BFS que usamos toman tiempo $O(n)$, en el peor de los casos la trayectoria P que construimos contiene a todos los vértices de T por lo que igual toma tiempo $O(n)$, y todas las demás operaciones toman tiempo $O(1)$. Concluyendo así que el algoritmo toma tiempo $O(n)$.

Para el análisis de espacio, el elemento que más información almacena en el algoritmo es la trayectoria P, y como se argumentó anteriormente, en el peor caso almacena a todos los vértices de T, por lo que P ocupa $O(n)$. Todos los demás elementos ocupan espacio $O(1)$. Concluyendo así que el algoritmo toma espacio $O(n)$. □

Una vez presentado el algoritmo 24 utilizado para encontrar los centros de un árbol, presentamos el algoritmo 25 para determinar si dos árboles arbitrarios son isomorfos.

Lema 3.5.4. *El algoritmo 25 es capaz de detectar si dos árboles arbitrarios son isomorfos en tiempo $O(n)$ y utilizando espacio $O(n)$.*

Demostración. Sean T_1 y T_2 dos árboles. Supongamos primero que T_1 y T_2 son isomorfos y veamos que el algoritmo regresa **verdadero**. Como ambos árboles son isomorfos entonces tienen la misma cantidad de vértices, por lo que la condición de la línea 1 no se cumple y continúa la ejecución. Obtenemos así a los respectivos centros de T_1 y T_2, $\mathcal{C}_1$ y $\mathcal{C}_2$, en las línea 4 y 5. Y dado que los árboles son isomorfos entonces ambos tienen la misma cantidad de centros, por lo que la condición de la línea 6 no se cumple y continúa la ejecución. Dado que ambos árboles son isomorfos, entonces existe $u \in \mathcal{C}_1$ y $v \in \mathcal{C}_2$ tal que el árbol T_1 enraizado en u es isomorfo al árbol T_2 enraizado en v, y como en las líneas 9-17 comparamos para cada posible pareja de centros de T_1 y T_2 que sus árboles enraizados sean isomorfos, en particular lo vamos a comparar con u y con v, por lo que la condición del línea 13 del algoritmo se cumple y así el algoritmo regresa **verdadero**.

Ahora supongamos que T_1 y T_2 no son isomorfos y veamos que el algoritmo regresa **falso**. Si T_1 y T_2 no son isomorfos, puede ser porque no tienen la misma cantidad de vértices, en dado caso se cumple la condición de la línea 1 del algoritmo y regresa **falso**. Es posible que T_1 y T_2 tengan la misma cantidad de vértices pero

Algoritmo 25: Isomorfismo de Árboles Arbitrarios

Input: Dos árboles T_1 y T_2.
Output: Un booleano que indica si los árboles son o no isomorfos.

```
1  if |V_{T_1}| ≠ |V_{T_2}| then
2  |  return False
3  end
4  C_1 ← EncontrarCentros(T_1)
5  C_2 ← EncontrarCentros(T_2)
6  if |C_1| ≠ |C_2| then
7  |  return False
8  end
9  for u ∈ C_1 do
10 |  for v ∈ C_2 do
11 |  |  T_u ← el árbol obtenido al hacer BFS sobre T_1 con raíz en u
12 |  |  T_v ← el árbol obtenido al hacer BFS sobre T_2 con raíz en v
13 |  |  if IsomorfismoArbolesEnraizados(T_u, T_v) then
14 |  |  |  return True
15 |  |  end
16 |  end
17 end
18 return False
```

difieran en la cantidad de centros que cada uno tiene, si ese es el caso, se cumple la condición de la línea 6 y el algoritmo regresa **falso**. Por otro lado, es posible que T_1 y T_2 tengan la misma cantidad de vértices y la misma cantidad de centros pero que no sean isomorfos, lo anterior implica que no existen $u \in \mathcal{C}_1$ y $v \in \mathcal{C}_2$ tal que el árbol T_1 enraizado en u sea isomorfo al árbol T_2 enraizado en v, por ende ninguna de las comparaciones realizadas en las líneas 9-17 son iguales y entonces continúa la ejecución del algoritmo. Ejecutamos la línea 18 y regresamos **falso**.

Para el análisis de tiempo, observamos que encontrar los centros de T_1 y T_2 toma tiempo $O(n)$, y como en el peor de los casos cada árbol tiene dos centros entonces ejecutamos el algoritmo 23 a lo más 4 veces. De esta forma, en el peor caso, el algoritmo 25 toma tiempo $O(6n)$, es decir, $O(n)$.

Para el análisis de espacio, los algoritmos 24 y 23 toman espacio $O(n)$. Los elementos $\mathcal{C}_1$ y $\mathcal{C}_2$ utilizan espacio $O(1)$ dado que solo pueden almacenar uno o dos vértices, y los elementos T_u y T_v ocupan espacio $O(n + (n-1))$ dado que guardan

a todos los vértices del árbol junto con sus respectivas adyacencias. Concluyendo de esta forma que el algoritmo ocupa espacio $O(n)$. □

Capítulo 4

Orden doblemente lexicográfico

Un **ordenamiento doblemente lexicográfico** de una matriz es un ordenamiento tal que sus renglones y columnas, como vectores, están ordenados lexicográficamente de mayor a menor. El **ordenamiento lexicográfico** de un vector v, denotado como $lex(v)$, es el orden que encontramos en un diccionario. Por ejemplo, dado el orden de un diccionario, los siguientes vectores están ordenados de mayor a menor:

$$(200, 111, 100, 020, 011, 010, 000).$$

Veamos ahora un ejemplo del ordenamiento doblemente lexicográfico de una matriz. En la siguiente figura 4.1, presentamos del lado izquierdo a la matriz original, y del lado derecho a la misma matriz pero ordenada lexicográficamente por renglones y columnas.

	c_1	c_2	c_3	c_4
r_1	0	1	0	1
r_2	0	1	0	0
r_3	0	0	1	0
r_4	0	0	0	1

Original

	c_4	c_2	c_3	c_1
r_1	1	1	0	0
r_4	1	0	0	0
r_2	0	1	0	0
r_3	0	0	1	0

Ordenada

Figura 4.1: Ordenamiento Doblemente Lexicográfico de una Matriz

Argumentando un poco más el ejemplo anterior. El orden original de los renglones y columnas de la matriz es (r_1, r_2, r_3, r_4) y (c_1, c_2, c_3, c_4), respectivamente. Ahora, dado el orden (r_1, r_4, r_2, r_3) y (c_4, c_2, c_3, c_1), observemos que los vectores conformados

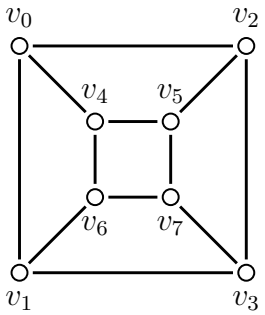
por los renglones y columnas están ordenados lexicográficamente de mayor a menor. Vale la pena recalcar que los intercambios de renglones son independientes a los intercambios de columnas, es decir, podemos intercambiar renglones sin intercambiar a las columnas correspondientes, y viceversa.

$$\begin{aligned} &\text{Renglones:} && (1100, 1000, 0100, 0010),\\ &\text{Columnas:} && (1100, 1010, 0001, 0000). \end{aligned}$$

Por lo tanto, el ordenamiento anterior efectivamente sí es un ordenamiento doblemente lexicográfico.

4.1. Ordenamientos lexicográficos de gráficas

Definimos a la **matriz de vecindades** de una gráfica G de n vértices como una matriz de tamaño $n \times n$, denotada $\mathcal{N}_G$, en donde $n_{ij} = 1$ si y sólo si $v_i = v_j$ o si v_i es adyacente con v_j. Observemos que la matriz es simétrica y además su diagonal es 1. Decimos que una gráfica está **ordenada lexicográficamente** si su matriz de vecindades tiene un orden doblemente lexicográfico.



	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_0	1	1	1	0	1	0	0	0
v_1	1	1	0	1	0	0	1	0
v_2	1	0	1	1	0	1	0	0
v_3	0	1	1	1	0	0	0	1
v_4	1	0	0	0	1	1	1	0
v_5	0	0	1	0	1	1	0	1
v_6	0	1	0	0	1	0	1	1
v_7	0	0	0	1	0	1	1	1

G $\mathcal{N}_G$

Figura 4.2: Gráfica G junto con su matriz de vecindades $\mathcal{N}_G$.

Dentro de la Teoría de Gráficas, el ordenamiento lexicográfico de gráficas ha resultado tener múltiples aplicaciones. Por ejemplo, Anna Lubiw en 1987 [10] demostró caracterizaciones que cumplen los ordenamientos de gráficas cordales y gráficas totalmente balanceadas. En este trabajo vamos a presentar y demostrar un algoritmo que regresa un ordenamiento lexicográfico de una gráfica en tiempo $O(|V|^2 \log |V|)$.

4.2. Algoritmo de ordenamiento lexicográfico

Antes de presentar el algoritmo, tenemos las siguientes definiciones. Una **partición ordenada** de un conjunto finito S es un lista ordenada $(S_1, S_2, \ldots, S_k)$ donde $\{S_1, S_2, \ldots, S_k\}$ es una partición de S; cada S_i, es una **parte** de la partición. La partición ordenada $(F_1, F_2, \ldots, F_\ell)$ es un **refinamiento** de la partición ordenada $(S_1, S_2, \ldots, S_k)$ si para cada $j \in \{1, \ldots, k\}$ existen índices únicos $i_1, i_2, \ldots, i_m$ tales que $(F_{i_1}, F_{i_2}, \ldots, F_{i_m})$ es una partición ordenada de S_j. Por ejemplo, si tenemos el conjunto $V = \{v_1, v_2, v_3, v_4, v_5\}$, una partición ordenada de V es $S_V = (\{v_1, v_2, v_3\}, \{v_4, v_5\})$ y un posible refinamiento de S_V es $F_V = (\{v_1, v_2\}, \{v_3\}, \{v_4\}, \{v_5\})$, ya que para $\{v_1, v_2, v_3\}$ tenemos que $(\{v_1, v_2\}, \{v_3\})$ es una partición ordenada y $(\{v_4\}, \{v_5\})$ de $\{v_4, v_5\}$.

Sea $M = (m_{ij})$ una matriz con una partición ordenada $\mathcal{R} = (R_1, R_2, \ldots, R_k)$ de su conjunto de renglones y una partición ordenada $\mathcal{C} = (C_1, \ldots, C_\ell)$ de su conjunto de columnas. Cada posible pareja (R_i, C_j) determina un **bloque** $B = M(R_i, C_j)$ de M. Decimos que un bloque B es **constante** si $m_{r,c}$ es el mismo para toda $r \in R_i$ y $c \in C_j$. La **entrada más grande** en B es definida como máx$(B) =$ máx$\{m_{r,c} \colon r \in R_i, c \in C_j\}$; como estamos trabajando con matrices de vecindades entonces máx(B) siempre es a lo más 1. Un **renglón separador** de B es un $r \in R_i$ tal que el conjunto $\{c \in C_j \colon m_{r,c} = \text{máx}(B)\}$ es un conjunto no vacío propio de C_j. Una **columna separadora** de B es una $c \in C_j$ tal que el conjunto $\{r \in R_i \colon m_{r,c} = \text{máx}(B)\}$ es un conjunto no vacío propio de R_i. Por último, sean $B = M(R_i, C_j)$ y $B' = M(R_{i'}, C_{j'})$ bloques, decimos que B' esta **arriba del bloque** B si $i' < i$ y $j' = j$; decimos que B' esta **a la izquierda del bloque** B si $i' = i$ y $j' < j$. A continuación presentamos un par de proposiciones que son de mucha utilidad para el algoritmo de ordenamiento.

Proposición 4.2.1. *Todo bloque B no constante contiene un renglón separador o una columna separadora.*

Demostración. Sea $B = M(R_i, C_j)$, al ser B no constante esto implica que existen $m, m' \in B$ tal que $m \neq m'$. Sean $m = m_{r,c}$ y $m' = m_{r',c'}$. Se puede dar el caso en el que $M(r, C_j)$, $M(R_i, c)$ o ambos sean bloques constantes. Si $M(r, C_j)$ es un bloque constante, tenemos que $m_{r,c} = m_{r,c'}$ pero $m_{r,c} \neq m_{r',c'}$ y por ende $m_{r,c'} \neq m_{r',c'}$, concluyendo de esta forma que $M(R_i, c')$ es un bloque no constante; el argumento anterior es análogo para los demás casos.

Si $M(R_i, c')$ es el bloque no constante, entonces podemos dar una partición $X = \{x \colon m_{x,c'} = 0, x \in R_i\}$ y $Y = \{y \colon m_{y,c'} = 1, y \in R_i\}$, concluyendo así que c' es una columna separadora. Si $M(r', C_j)$ es el bloque no constante, entonces podemos dar

una partición similar a la del caso anterior y concluir que r' es un renglón separador. □

Proposición 4.2.2. *Sea $B = M(R_i, C_j)$ un bloque constante, y sean S_R y S_C refinamientos de R_i y C_j, entonces para cualesquiera $R'_i \in S_R$ y $C'_j \in S_C$ tenemos que el bloque $B' = M(R'_i, C'_j)$ es constante. En otras palabras, el refinamiento de un bloque constante induce bloques constantes.*

Demostración. Dado que $B = M(R_i, C_j)$ es un bloque constante, entonces para todo $R'_i \subseteq R_i$ y $C'_j \subseteq C_j$ tenemos que $M(R'_i, C'_j)$ es un bloque constante. En particular, dados S_R y S_C, tenemos que cualquier $s_R \in S_R$ y $s_C \in S_C$ se cumple que $s_R \subseteq R_i$ y $s_C \subseteq C_j$, por lo que $M(s_R, s_C)$ también es un bloque constante. □

La idea del algoritmo es construir un refinamiento del conjunto de renglones $\mathcal{R}$ y el conjunto de columnas $\mathcal{C}$ de tal modo que la partición ordenada final de cada conjunto induzca un orden doblemente lexicográfico. Cabe mencionar que el algoritmo original funciona para matrices con entradas en $\{0, \ldots, k\}$ y de tamaño $n \times m$, donde $k, n, m \in \mathbb{N}$. Sin embargo, en este trabajo sólo nos vamos a enfocar en que funcione para matrices cuadradas y cuyas entradas son sólo 0 ó 1; ya que estos son los únicos valores que puede tener la matriz de vecindades.

Veamos un ejemplo de cómo podemos seleccionar renglones y columnas separadoras para ir construyendo un refinamiento sobre el conjunto de renglones y columnas, y de esta forma regresar un orden doblemente lexicográfico de una matriz. La matriz con la que vamos a trabajar es la de la figura 4.3. Junto con la matriz presentamos también a la partición ordenada de los renglones y columnas S_R y S_C, respectivamente. Por último, a cada posible bloque $B = M(R_i, C_j)$ con $R_i \in S_R$ y $C_j \in S_c$ lo vamos a indicar como un rectángulo dentro de la matriz.

	c_1	c_2	c_3	c_4
r_1	1	0	1	1
r_2	0	0	1	0
r_3	1	0	0	1
r_4	0	0	1	1

$$S_R = (\{r_1, r_2, r_3, r_4\}),$$

$$S_C = (\{c_1, c_2, c_3, c_4\}).$$

Figura 4.3: Matriz original junto con el orden S_R y S_C inicial.

El primer y único bloque con el que empezamos es $B = M(R_1, C_1)$, en donde sabemos que B no es constante. Observemos que r_1 es un renglón separador, por lo que si-

guiendo las líneas 7-8 del algoritmo obtenemos la siguiente partición: $C' = \{c_1, c_3, c_4\}$, $C'' = \{c_2\}$.

	c_1	c_3	c_4	c_2
r_1	1	1	1	0
r_2	0	1	0	0
r_3	1	0	1	0
r_4	0	1	1	0

$S_R = (\{r_1, r_2, r_3, r_4\})$,

$S_C = (\{c_1, c_3, c_4\}, \{c_2\})$.

Figura 4.4: Primera iteración del algoritmo.

Ahora $B = M(R_1, C_1)$ esta conformado por los mismos renglones pero con las columnas $\{c_1, c_3, c_4\}$. Este bloque no es constante, el renglón r_2 es separador, por lo que podemos obtener la siguiente partición: $C' = \{c_3\}$, $C'' = \{c_1, c_4\}$.

	c_3	c_1	c_4	c_2
r_1	1	1	1	0
r_2	1	0	0	0
r_3	0	1	1	0
r_4	1	0	1	0

$S_R = (\{r_1, r_2, r_3, r_4\})$,

$S_C = (\{c_3\}, \{c_1, c_4\}, \{c_2\})$.

Figura 4.5: Segunda iteración del algoritmo.

Continuando con la ejecución, ahora el bloque $B = M(R_1, C_1)$ esta conformado por los mismos renglones pero con la columna $\{c_3\}$. Este bloque no es constante, aunque cada renglón es constante; la columna c_3 es separadora. Obtenemos la siguiente partición: $R' = \{r_1, r_2, r_4\}$, $R'' = \{r_3\}$.

	c_3	c_1	c_4	c_2
r_1	1	1	1	0
r_2	1	0	0	0
r_4	1	0	1	0
r_3	0	1	1	0

$S_R = (\{r_1, r_2, r_4\}, \{r_3\})$,

$S_C = (\{c_3\}, \{c_1, c_4\}, \{c_2\})$.

Figura 4.6: Tercera iteración del algoritmo.

Así, los bloques $M(R_1, C_1)$ y $M(R_2, C_1)$ ya son constantes, por lo que nos podemos mover al bloque $M(R_1, C_2)$. Este bloque no es constante, el renglón r_4 es separador, por lo que obtenemos la siguiente partición: $C' = \{c_4\}$, $R'' = \{c_1\}$.

	c_3	c_4	c_1	c_2
r_1	1	1	1	0
r_2	1	0	0	0
r_4	1	1	0	0
r_3	0	1	1	0

$$S_R = (\{r_1, r_2, r_4\}, \{r_3\}),$$

$$S_C = (\{c_3\}, \{c_4\}, \{c_1\}, \{c_2\}).$$

Figura 4.7: Cuarta iteración del algoritmo.

El bloque $M(R_1, C_2)$ ahora está conformado por el mismo conjunto de renglones R_1 pero ahora $C_2 = \{c_4\}$. Este bloque no es constante, aunque todos los renglones son constantes, la columna c_4 es separadora. Obtenemos la siguiente partición: $R' = \{r_1, r_4\}$, $R'' = \{r_2\}$. Observemos que hacer este *intercambio* de renglones no afecta el trabajo que hemos realizado en los anteriores bloques. Esto se debe a que los bloques que están tanto arriba como a la izquierda del actual, son constantes, y por la proposición 4.2.2, los bloques que surjan a partir de esta partición seguirán siendo constantes; en otras palabras, no estamos afectando el orden lexicográfico de las partes de los vectores con los que ya hemos trabajado.

	c_3	c_4	c_1	c_2
r_1	1	1	1	0
r_4	1	1	0	0
r_2	1	0	0	0
r_3	0	1	1	0

$$S_R = (\{r_1, r_4\}, \{r_2\}, \{r_3\}),$$

$$S_C = (\{c_3\}, \{c_4\}, \{c_1\}, \{c_2\}).$$

Figura 4.8: Quinta iteración del algoritmo.

Todos los bloques a la izquierda y arriba de $M(R_1, C_3)$ son constantes, pero este bloque no es constante, la columna c_1 es separadora, obtenemos la siguiente partición: $R' = \{r_1\}$, $R'' = \{r_4\}$.

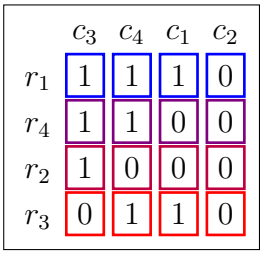


Figura 4.9: Sexta iteración del algoritmo.

Observemos ahora que todos los bloques que podemos formar son constantes por lo que finaliza la ejecución del algoritmo, y las particiones ordenadas S_R y S_C inducen un ordenamiento doblemente lexicográfico.

De esta forma, presentamos el algoritmo 26, el cual informalmente explica el proceso con el cual podemos obtener un ordenamiento doblemente lexicográfico de una matriz. Veamos primero un par de propiedades que cumple el algoritmo que nos facilitarán demostrar que es correcto.

Proposición 4.2.3. *Sea $B = M(R_i, C_j)$ un bloque de M, el refinamiento de R_i y C_j descrito en el algoritmo 26 produce bloques constantes.*

Demostración. Si B es constante, el resultado se sigue trivialmente. Supongamos que B no es constante, y procedamos por inducción sobre el número de renglones separadores de B.

Para el caso base, cuando B no tiene renglones separadores, entonces la línea 10 del algoritmo define una partición R' y R'', en donde observemos que los bloques $B' = M(R', C_j)$ y $B'' = M(R'', C_j)$ son constantes. En particular B' tiene únicamente unos y B'' tiene únicamente ceros.

Para el paso inductivo, cuando B tiene al menos un renglón separador r, entonces la línea 6 del algoritmo define una partición C' y C''. Donde tanto en $B' = M(R_i, C')$ como en $B'' = M(R_i, C'')$ tenemos que $M(r, C')$ y $M(r, C'')$ son constantes, por lo tanto B' y B'' tienen menos renglones separadores que B. Así, por hipótesis inductiva, el algoritmo aplicado sobre B' produce bloques constantes y como todos estos bloques se encuentran a la izquierda de B'', entonces eventualmente el algoritmo se coloca en el bloque B'', y de igual forma, produce bloques constantes.

Así, se concluye que el refinamiento de R_i y C_j descrito en el algoritmo produce bloques constantes. □

Lema 4.2.4. *Los bloques generados a partir de las particiones ordenadas S_R y S_C que regresa el algoritmo 26 son bloques constantes.*

Demostración. El bloque inicial es $B = M(\mathcal{R}, \mathcal{C})$, con $S_R = (\mathcal{R})$ y $S_C = (\mathcal{C})$. Por la proposición 4.2.3 sabemos que el algoritmo aplicado sobre B produce un refinamiento de $\mathcal{R}$ y $\mathcal{C}$ con bloques constantes. En particular, este refinamiento se encuentra definido por las particiones ordenadas S_R y S_C. □

Proposición 4.2.5. *Las particiones ordenadas S_R y S_C que regresa el algoritmo 26 inducen un orden lexicográfico sobre los renglones de M.*

Demostración. Sean $(r_1, r_2, \ldots, r_n)$ y $(c_1, \ldots, c_n)$ el orden de los renglones y columnas inducido por S_R y S_C respectivamente. Procedemos por contradicción, es decir, supongamos que el orden inducido por S_R no es un orden lexicográfico de los renglones de M.

Si el orden inducido por S_R no es un orden lexicográfico, entonces existen r_i y r_j, tal que $i < j$ y el vector conformado por el renglón r_i y r_j cumple que $lex(r_j) > lex(r_i)$. Lo anterior implica que existe una columna c_k donde $m_{r_i,c_k} = 0$ y $m_{r_j,c_k} = 1$, tal que a partir de esta columna $lex(r_j)$ es lexicográficamente mayor a $lex(r_i)$. Así, los vectores se ven de la siguiente forma:

$$lex(r_i) = (m_{r_i,c_1}, m_{r_i,c_2}, \ldots, m_{r_i,c_k} = 0, \ldots, m_{r_i,c_n}),$$
$$lex(r_j) = (m_{r_j,c_1}, m_{r_j,c_2}, \ldots, m_{r_j,c_k} = 1, \ldots, m_{r_j,c_n}).$$

En algún momento de la ejecución sabemos que m_{r_i,c_k} y m_{r_j,c_k} pertenecen al mismo bloque B, donde todos los bloques arriba y a su izquierda son constantes; ya sea el bloque inicial con el que empezamos la ejecución del algoritmo o en algún punto más adelante. Verifiquemos por inducción sobre el número de renglones separadores de B, que lo anterior no puede suceder.

Para el caso base, si B no tiene renglones separadores, entonces la línea 10 del algoritmo se ejecuta y define una partición donde $r_j \in R'$ y $r_i \in R''$, lo que implica que en el orden dado por S_R tenemos que r_j aparece antes que r_i, lo cual no puede suceder. Por lo tanto, hay al menos un renglón separador.

Para el paso inductivo, si B tiene al menos un renglón separador, entonces la línea 6 del algoritmo se ejecuta y define la partición C' y C''. Si $c_k \in C''$, entonces el algoritmo trabaja primero con todos los bloques conformados por las columnas que se encuentran antes de c_k en S_C. Si durante esto, el algoritmo en algún momento separa a r_i y r_j en R' y R'', esto implica que para alguna columna c_ℓ con $\ell < k$ tenemos que $m_{r_i,c_\ell} \neq m_{r_j,c_\ell}$. Si $r_i \in R'$ y $r_j \in R''$, entonces $m_{r_i,c_\ell} = 1$ y $m_{r_j,c_\ell} = 0$, en donde a partir de la columna c_ℓ el vector $lex(r_i)$ es lexicográficamente mayor a $lex(r_j)$ sin importar cómo ordenemos el resto de las columnas. Esto no puede suceder dado que $lex(r_i) < lex(r_j)$. Por otro lado, tampoco es posible que $r_j \in R'$ y $r_i \in R''$,

ya que r_i aparece antes que r_j en el orden inducido por S_R. Por lo tanto, el algoritmo no separa a r_i y r_j en partes distintas antes de revisar al bloque que contiene a la columna c_k. Sabemos por la proposicion 4.2.3 que todos los bloques producidos por B eventualmente son constantes. Por este motivo, el algoritmo eventualmente llega al bloque $B' = M(R'_i, C'_j)$, donde todos los bloques arriba y a la izquierda de él son constantes, y B' contiene m_{r_i,c_k} y m_{r_j,c_k}, además el número de renglones separadores es menor al de B. Así, por hipótesis inductiva, esta configuración no puede suceder. Sin embargo, para que el orden inducido por S_R y S_C no sea un orden lexicográfico, esta condición debe cumplirse en algún punto del refinamiento, lo que resulta en una contradicción.

Así las particiones ordenadas S_R y S_C inducen un orden lexicográfico sobre los renglones de M. □

Proposición 4.2.6. *Las particiones ordenadas S_R y S_C que regresa el Algoritmo 26 inducen un orden lexicográfico sobre las columnas de M.*

Demostración. Sean $(r_1, r_2, \ldots, r_n)$ y $(c_1, \ldots, c_n)$ el orden de los renglones y columnas inducido por S_R y S_C respectivamente. Procedemos por contradicción, es decir, el orden inducido por S_C no es un orden lexicográfico de las columnas de M.

Si el orden inducido por S_C no es un orden lexicográfico, entonces existen c_i y c_j, tal que $i < j$ y los vectores conformados por las columnas c_i y c_j cumplen que $lex(c_j) > lex(c_i)$. Lo anterior implica que existe un renglón r_k donde $m_{r_k,c_i} = 0$ y $m_{r_k,c_j} = 1$, tal que a partir de este renglón $lex(c_j)$ es lexicográficamente mayor a $lex(c_i)$. Así, los vectores se ven de la siguiente forma:

$$lex(c_i) = (m_{r_1,c_i}, m_{r_2,c_i}, \ldots, m_{r_k,c_i} = 0, \ldots, m_{r_n,c_i}),$$
$$lex(c_j) = (m_{r_1,c_j}, m_{r_2,c_j}, \ldots, m_{r_k,c_j} = 1, \ldots, m_{r_n,c_j}).$$

En algún momento de la ejecución sabemos que m_{r_k,c_i} y m_{r_k,c_j} pertenecen al mismo bloque B, donde todos los bloques arriba y a su izquierda son constantes, ya sea el bloque inicial con el que empezamos la ejecución del algoritmo o en algún punto más adelante. Observemos que r_k es un renglón separador. Verifiquemos por inducción sobre el número de renglones separadores de B que esto nunca puede ocurrir.

Para el caso base, cuando B sólo tiene un renglón separador, entonces ese renglón separador es r_k, por lo que la línea 6 del algoritmo define una partición C' y C'' donde $c_j \in C'$ y $c_i \in C''$, lo que implica que en S_C la columna c_j aparece antes que c_i, lo cual no puede ocurrir.

Para el paso inductivo, si B tiene al menos dos renglones separadores. Si tomamos el renglón r_k para hacer el refinamiento, caemos en el caso anterior. Por lo que

el algoritmo debe de tomar otro renglón distinto a r_k, y definir la partición R' y R''. Si $r_k \in R''$, entonces el algoritmo trabaja primero con todos los renglones que se encuentran antes de r_k en S_R. Si durante esto, el algoritmo en algún momento separa a c_i y c_j en C' y C'', esto implica que para algún renglón r_ℓ con $\ell < k$ tenemos que $m_{r_\ell,c_i} \neq m_{r_\ell,c_j}$. Si $c_i \in C'$ y $r_j \in C''$, entonces $m_{r_\ell,c_i} = 1$ y $m_{r_\ell,c_j} = 0$, por lo que a partir del renglón r_ℓ, el vector $lex(c_i)$ es lexicográficamente mayor a $lex(c_j)$ sin importar cómo ordenemos el resto de los renglones. Pero esto no puede ocurrir dado que $lex(r_i) < lex(r_j)$. Tampoco es posible que $c_j \in C'$ y $c_i \in C''$, ya que c_i aparece antes que c_j en el orden inducido por S_C. Por lo tanto, el algoritmo no separa a c_i y c_j en partes distintas antes de revisar al bloque que contiene al renglón r_k. Sabemos por la proposicion 4.2.3 que todos los bloques producidos por B eventualmente son constantes. Por este motivo, el algoritmo eventualmente llega al bloque $B' = M(R'_i, C'_j)$, donde todos los bloques arriba y a la izquierda de él son constantes, y B' contiene m_{r_k,c_i} y m_{r_k,c_j}, además el número de renglones separadores es menor al de B. Sin embargo, por hipótesis inductiva, esto no puede ocurrir, contradiciendo que dicha configuración debe existir.

Como la contradicción surge de suponer que S_C no es un orden lexicográfico para las columnas de M, se concluye que las particiones ordenadas S_R y S_C inducen un orden lexicográfico sobre las columnas de M. □

Corolario 4.2.7. *Las particiones ordenadas S_R y S_C que regresa el algoritmo 26 inducen un orden doblemente lexicográfico sobre M.*

Demostración. Utilizando los resultados obtenidos en las proposiciones 4.2.5 y 4.2.6, podemos observar que este enunciado es cierto. □

Algoritmo 26: Ordenamiento Doblemente Lexicográfico

Input: Una matriz M de tamaño $n \times n$ cuyas entradas son sólo 0 ó 1.
Output: Las particiones ordenadas S_R y S_C correspondientes al orden lexicográfico de M.

$\mathcal{R} \leftarrow$ el conjunto de renglones $\{r_1, \dots r_n\}$ de M
$\mathcal{C} \leftarrow$ el conjunto de columnas $\{c_1, \dots c_n\}$ de M
$S_R \leftarrow (\mathcal{R})$
$S_C \leftarrow (\mathcal{C})$
while *existan* $R \in S_R$ *y* $C \in S_C$ *tal que* $B = M(R, C)$ *es un bloque no constante en donde todos los bloques arriba y a la izquierda de* B *son constantes* **do**
    if *existe un renglón* r *tal que* $M(r, C)$ *es no constante* **then**
        $C' \leftarrow \{c \in C \colon m_{r,c} = 1\}$
        $C'' \leftarrow \{c \in C \colon m_{r,c} = 0\}$
        Reemplazar a C en S_C por C' y C'' en este orden
    else
        $R' \leftarrow \{r \in R \colon m_{r,c} = 1, c \in C\}$
        $R'' \leftarrow \{r \in R \colon m_{r,c} = 0, c \in C\}$
        Reemplazar a R en S_R por R' y R'' en este orden
    end
end
return S_R, S_C

4.3. Un algoritmo más concreto

Como lo mencionamos anteriormente, el algoritmo 26 únicamente explica de manera informal el proceso necesario para poder obtener un ordenamiento doblemente lexicográfico de una matriz. Por ende, éste se podría considerar como *genérico*, ya que existen múltiples formas de implementar las instrucciones que especifica el algoritmo. Y por lo tanto no podemos dar un análisis sobre tiempo o espacio de él. Por esta razón, necesitamos diseñar un algoritmo que funcione como el anterior, pero sobre el cual podamos hacer un análisis; con la ventaja adicional de que dicho algoritmo puede ser implementado en diversos lenguajes de programación.

4.3.1. La estructura de datos

Vamos a utilizar una estructura de datos llamada **pila**, *stack* en inglés. Esta estructura es similar a la cola (la estructura utilizada en el algoritmo 1) en que podemos agregar y eliminar elementos en tiempo constante. La estructura tiene un orden FILO (*First In, Last Out*), es decir, el primer elemento que agreguemos será el último en salir; un ejemplo visual puede ser una pila de tortillas, para sacar a la primera tortilla que pusimos, antes tenemos que sacar a todas las que se encuentren arriba de ella.

Sea P una pila. Definimos a la cabeza de P como el elemento que no tiene antecesor, es decir, es el elemento que se encuentra hasta arriba de P. El único elemento al que tenemos acceso es a la cabeza, y por ende es el único elemento que podemos eliminar. Sea x la cabeza de P, agregar un elemento w a P es simplemente ponerlo *arriba* de x en P; si P no tiene elemento alguno, entonces w se convierte en la cabeza de P. Por ejemplo, sea P_0 una pila, supongamos que agregamos a los elementos x, y y z en este orden. Entonces la cabeza de P_0 es z, y para poder acceder a y primero tenemos que eliminar a z, de igual forma si queremos acceder a x tenemos que eliminar a z y y en ese orden. Si agregamos w a P_0, entonces el orden que tiene P_0 es (w, z, y, x). Si eliminamos la cabeza de P_0, entonces el nuevo orden es (z, y, x), en donde es claro que z es la nueva cabeza de P_0.

4.3.2. Variables

Sea $M = (m_{ij})$ la matriz de tamaño $n \times n$, $\mathcal{R}$ el conjunto de índices para los renglones de M, $\mathcal{C}$ el conjunto de índices para las columnas de M, y B un bloque de M. Definimos el **tamaño de un bloque** respecto a S_R y S_C como la función $tam\colon (\mathcal{P}(\mathcal{R}) \times \mathcal{P}(\mathcal{C})) \to \mathbb{N}$, en donde, para un bloque $B = M(R_i, C_j)$, $tam(R_i, C_j)$ nos indica la cantidad de elementos no cero que hay en B. A cada bloque lo vamos a representar como una pareja ordenada (R_i, C_j) donde $R_i \in S_R$ y $C_j \in S_C$.

Hay un par de propiedades sobre la función tam que nos serán de utilidad al momento de diseñar un algoritmo más concreto. Por simplicidad, abusaremos de la notación para escribir $tam(r, C_j)$ en lugar de $tam(\{r\}, C_j)$ y $tam(R_i, c)$ en lugar de $tam(R_i, \{c\})$.

Proposición 4.3.1. *Sean M un matriz, y $B = M(R_i, C_j)$ un bloque de M.*

$$tam(R_i, C_j) = \sum_{r \in R_i} tam(r, C_j).$$

Demostración. Sabemos que el tamaño de un bloque B está definido como la cantidad de entradas no cero que hay en B. Dado que un bloque sólo puede tener entradas 0 ó 1, entonces podemos definir al tamaño de B como:

$$tam(R_i, C_j) = \sum_{r \in R_i} \sum_{c \in C_j} m_{r,c}.$$

En donde, renglón por renglón, sumamos las entradas de todas las columnas. Por otro lado, el tamaño del bloque $M(r, C_j)$ son todas las entradas no cero que hay en el renglón r tomando en cuenta sólo a las columnas de C_j, es decir,

$$tam(r, C_j) = \sum_{c \in C_j} m_{r,c}.$$

Si sumamos el tamaño de todos los bloques $M(r, C_j)$ con $r \in R_i$, entonces tenemos que:

$$\sum_{r \in R_i} tam(r, C_j) = \sum_{r \in R_i} \sum_{c \in C_j} m_{r,c}.$$

Concluyendo de este modo:

$$tam(R_i, C_j) = \sum_{r \in R_i} tam(r, C_j).$$

□

Proposición 4.3.2. *Un bloque $B = M(R_i, C_j)$ es constante si y sólo si $tam(R_i, C_j)$ es igual a 0 ó $|R_i||C_j|$.*

Demostración. Supongamos que $B = M(R_i, C_j)$ es constante. Podemos expresar el tamaño de B como

$$tam(R_i, C_j) = \sum_{r \in R_i} \sum_{c \in C_j} m_{ij}.$$

Si B es 0-constante, entonces:

$$tam(R_i, C_j) = \sum_{r \in R_i} \sum_{c \in C_j} m_{r,c} = \sum_{r \in R_i} \sum_{c \in C_j} 0 = \sum_{r \in R_i} 0 = 0.$$

Si es 1-constante, entonces

$$tam(R_i, C_j) = \sum_{r \in R_i} \sum_{c \in C_j} m_{r,c} = \sum_{r \in R_i} \sum_{c \in C_j} 1 = \sum_{r \in R_i} |C_j| = |R_i||C_j|.$$

Ahora supongamos que $B = M(R_i, C_j)$ no es constante, entonces existe un renglón r tal que la siguiente desigualdad se cumple:

$$0 < tam(r, C_j) < |C_j|.$$

Por lo que la siguiente desigualdad también se cumple:

$$0 < tam(R_i, C_j) = \sum_{r \in R_i} \sum_{c \in C_j} m_{r,c} < |R_i||C_j|.$$

□

4.3.3. Algoritmos auxiliares

Primero presentamos el algoritmo de refinamiento para columnas (algoritmo 27). Lo único que tenemos que hacer es recorrer el renglón r y guardar, en un conjunto C', a aquellas columnas c cuya entrada $m_{r,c}$ sea 1, y a todas las demás columnas guardarlas en C''.

Algoritmo 27: Refinamiento de Columnas

Input: Una matriz $M = (m_{ij})$ de tamaño $n \times n$, un renglón r y una parte C_j de la partición del conjunto de columnas de M.
Output: Un refinamiento del conjunto C_j

```
1  C' ← ∅
2  C'' ← ∅
3  for c ∈ C_j do
4  |  if m_{r,c} = 1 then
5  |  |  C' ← C' ∪ {c}
6  |  else
7  |  |  C'' ← C'' ∪ {c}
8  |  end
9  end
10 return C', C''
```

Proposición 4.3.3. *El algoritmo 27 regresa el refinamiento C', C'' del conjunto C_j descrito en el algoritmo 26 en tiempo $O(|C_j|)$.*

Demostración. Veamos que dado un renglón separador r, efectivamente los conjuntos C' y C'' que regresa el algoritmo coinciden con lo que describe el algoritmo 26, es decir:

$$C' = \{c \in C_j : m_{r,c} = 1\},$$
$$C'' = \{c \in C_j : m_{r,c} = 0\}.$$

Para ello observemos que dado el renglón r, en la línea 3 recorremos a todas las columnas c que se encuentran en C_j. Por cada columna vamos a comparar el valor de la entrada $m_{r,c}$, recordemos que la matriz sólo puede tener dos valores entonces $m_{r,c} \in \{0, 1\}$. Si $m_{r,c} = 1$, entonces la condición de la línea 4 se cumple y se agrega c a C'. De lo contrario, $m_{r,c} = 0$ y se ejecuta la línea 7, agregando c a C''. Por lo que al terminar de recorrer a todas las columnas, los conjuntos C' y C'' corresponden al refinamiento de C_j descrito en el algoritmo 26.

Tanto la comparación de la línea 4 como el agregar un elemento a un conjunto toman tiempo constante. Los operaciones anteriores se repiten un total de $|C_j|$ veces, concluyendo así que el algoritmo toma tiempo $O(|C_j|)$. □

Después presentamos el algoritmo de refinamiento para renglones (algoritmo 28). Sea $B = M(R_i, C_j)$, recordemos que el refinamiento de renglones únicamente se realiza si para todo $r \in R_i$ se tiene que $M(r, C_j)$ es un bloque constante. Por lo que, lo único que tenemos que hacer es tomar a una columna $c \in C_j$ y guardar en un conjunto R' a aquellos renglones r cuya entrada $m_{r,c}$ sea 1, y a todos los demás en R''.

Proposición 4.3.4. *El algoritmo 28 regresa el refinamiento R', R'' del conjunto R_i descrito en el algoritmo 26 en tiempo $O(|R_i|)$.*

Demostración. Veamos que efectivamente los conjuntos R' y R'' que regresa el algoritmo coinciden con lo que describe el algoritmo 26, es decir:

$$R' = \{r \in R_i : m_{r,c} = 1, c \in C\},$$

$$R'' = \{r \in R_i : m_{r,c} = 0, c \in C\}.$$

Dado que estamos buscando un refinamiento por renglones, entonces para todo $r \in R_i$ se tiene que $M(r, C_j)$ es un bloque constante, por lo que basta con tomar una columna cualquiera $c \in C_j$ y recorrer a todos los renglones $r \in R_i$. Dada la columna c, en la línea 4 recorremos a todas las columnas r que se encuentran en R_i. Por cada

Algoritmo 28: Refinamiento de Renglones

Input: Una matriz $M = (m_{ij})$ de tamaño $n \times n$, una parte R_i de la partición del conjunto de renglones de M y una parte C_j de la partición del conjunto de columnas de M.

Output: Un refinamiento del conjunto R_i

```
1  R' ← ∅
2  R'' ← ∅
3  c ← cualquier columna de C_j
4  for r ∈ R_i do
5  |  if m_{r,c} = 1 then
6  |  |  R' ← R' ∪ {r}
7  |  else
8  |  |  R'' ← R'' ∪ {r}
9  |  end
10 end
11 return R', R''
```

renglón comparamos el valor de la entrada $m_{r,c}$. Si $m_{r,c} = 1$, entonces la condición de la línea 5 se cumple y se agrega r a R'. De lo contrario, $m_{r,c} = 0$ y se ejecuta la línea 7, agregando r a R''. Por lo que al terminar de recorrer a todas los renglones, los conjuntos R' y R'' corresponden al refinamiento de R_j descrito en el algoritmo 26.

Tanto la comparación de la línea 5 como el agregar un elemento a un conjunto toman tiempo constante. Los operaciones anteriores se repiten un total de $|R_i|$ veces, concluyendo así que el algoritmo toma tiempo $O(|R_i|)$. □

El algoritmo 29 sirve para calcular el tamaño de un bloque $B = M(R_i, C_j)$.

Proposición 4.3.5. *Dado $B = M(R_i, C_j)$, el algoritmo 29 calcula el tamaño de los bloques B y $M(r, C_j)$ para todo $r \in R_i$ en tiempo $O(|R_i||C_j|)$.*

Demostración. Recordemos que el tamaño de un bloque B es la cantidad de entradas no-cero que hay en B. Además, dada la proposición 4.3.1, al calcular el tamaño de cada bloque $M(r, C_j)$ con $r \in R_i$ ya tenemos el tamaño del bloque B. Así, el algoritmo en la línea 2 recorre a cada renglón r y calcula el tamaño de $M(r, C_j)$. Lo anterior lo hace recorriendo a todas las columnas $c \in C_j$ en la línea 4, y dado que M sólo tiene entradas 0 y 1, entonces el tamaño de $M(r, C_j)$ se define como:

Algoritmo 29: Calcular Tamaño

Input: Una matriz $M = (m_{ij})$ de tamaño $n \times n$, un conjunto de renglones R_i, un conjunto de columnas C_j y la función tam.

Output: La función tam con el tamaño del bloque $B = M(R_i, C_j)$ y de los bloques $M(r, C_j)$ para toda $r \in R_j$.

```
1  t_B ← 0
2  for r ∈ R_i do
3  |   t_r ← 0
4  |   for c ∈ C_j do
5  |   |   t_r ← t_r + m_{r,c}
6  |   end
7  |   tam(r, C_j) ← t_r
8  |   t_B ← t_B + t_r
9  end
10 tam(R_i, C_j) ← t_B
11 return tam
```

$$tam(r, C_j) = \sum_{c \in C_j} m_{r,c}.$$

La suma anterior la guardamos en la variable t_r y al finalizar de recorrer a todas las columnas asignamos $tam(r, C_j) = t_r$. Después, en la línea 8 del algoritmo, sumamos t_r a t_B, esta última variable será el tamaño del bloque B. Así, al finalizar de recorrer a todas los renglones, tenemos que:

$$t_B = \sum_{r \in R_i} tam(r, C_j) = tam(R_i, C_j).$$

Es decir, t_B efectivamente es el tamaño de B. Asignamos $tam(R_i, C_j) = t_B$, y termina la ejecución del algoritmo.

El algoritmo recorre a todos los renglones de R_i y por cada renglón recorre a todas las columnas de C_j. Las operaciones de asignación y las sumas de enteros se hacen en tiempo constante. Concluyendo de esta forma que el algoritmo toma tiempo $O(|R_i||C_j|)$. □

El algoritmo 30 nos ayuda a encontrar un renglón separador dado un bloque $B = M(R_i, C_j)$.

Algoritmo 30: Obtener Renglón Separador

Input: Una matriz $M = (m_{ij})$ de tamaño $n \times n$, un bloque $B = M(R_i, C_j)$, y la función tam.

Output: Un renglón separador de $B = M(R_i, C_j)$, o $\varnothing$ si no existe alguno.

```
1 for r ∈ R_i do
2 |   if 0 < tam(r, C_j) < |C_j| then
3 |   |   return r
4 |   end
5 end
6 return ∅
```

Proposición 4.3.6. *Dado un bloque $B = M(R_i, C_j)$, el algoritmo 30 encuentra un renglón separador en tiempo $O(|R_i|)$.*

Demostración. Recordemos que la proposición 4.3.2 nos indica qué tamaño debe de tener un bloque para que tenga un renglón separador. Por lo que el algoritmo recorre todos los renglones $r \in R_i$ en la línea 1, y en la línea 2 verifica si el bloque $M(r, C_j)$ es constante, si no lo es, entonces r es un renglón separador.

Recorrer a todos los renglones toma tiempo $O(|R_i|)$, y por cada renglón obtenemos el valor de $tam(r, C_j)$ en tiempo constante. Concluyendo de esta forma que el algoritmo toma tiempo $O(|R_i|)$. □

Ahora, dado un bloque $B = M(R_i, C_j)$, y un refinamiento C', C'', es fácil obtener el tamaño de los bloques $M(R_i, C')$ y $M(R_i, C'')$ en tiempo $O(|R_i|(|C'| + |C''|)$. Sin embargo, podemos mejorar este tiempo aprovechando la información que ya teníamos sobre el tamaño de B; presentamos el algoritmo 31 el cual calcula de manera eficiente el tamaño de los bloques generados por el refinamiento.

Proposición 4.3.7. *Dado un bloque $B = M(R_i, C_j)$, un refinamiento C', C'' de C_j, y la partición ordenada de renglones S_R. El algoritmo 31 calcula el tamaño de todos los bloques afectados por el refinamiento de columnas. Sea S'_R el conjunto de partes que van antes de R_i en S_R y S''_R el conjunto de partes que van después de R_{i-1} en S_R, y sea $\mathcal{R}''$ el conjunto de renglones en S''_R, entonces el algoritmo 31 corre en tiempo $O(|\mathcal{R}''||C^*| + |S'_R|)$, donde C^* es el conjunto de menor cardinalidad entre C' y C''.*

Demostración. En las líneas 1-6 obtenemos al conjunto C^* de menor cardinalidad entre C' y C''. Sin pérdida de generalidad supongamos que $|C'| \leq |C''|$. En la línea 7 empezamos el ciclo que nos va a ayudar a recorrer a todas las partes contenidas en la partición ordenada S_R. Sea R_k el elemento actual que estamos recorriendo.

Algoritmo 31: Actualizar Tamaño Columnas

Input: Una matriz $M = (m_{ij})$ de tamaño $n \times n$, un bloque $B = M(R_i, C_j)$, un refinamiento C' y C'' de C_j, la partición ordenada de renglones S_R y la función tam.

Output: La función tam con el tamaño de todos los bloques afectados por el refinamiento de columnas.

```
1  C* ← ∅, C⋆ ← ∅
2  if |C'| ≤ |C''| then
3  |  C* ← C', C⋆ ← C''
4  else
5  |  C* ← C'', C⋆ ← C'
6  end
7  while k < |S_R| do
8  |  R_k ← el elemento en la posición k de S_R
9  |  if k < i then
10 |  |  tam(R_k, C') ← 0
11 |  |  tam(R_k, C'') ← 0
12 |  else
13 |  |  tam ← CalcularTam(M, R, C*, tam)
14 |  |  for r ∈ R_k do
15 |  |  |  tam(r, C⋆) ← tam(r, C_j) − tam(r, C*)
16 |  |  end
17 |  |  tam(R_k, C⋆) ← tam(R_k, C_j) − tam(R, C*)
18 |  end
19 |  k ← k + 1
20 end
21 return tam
```

Si $k < i$, entonces el bloque $B' = (R_k, C_j)$ es un bloque constante, pues se encuentra arriba de B. Dado que B' es constante, entonces su refinamiento también produce bloques constantes, por lo tanto simplemente indicamos que $tam(R_k, C')$ y $tam(R_k, C'')$ son iguales a cero; como tal no buscamos recorrer a los elementos de estos bloques de nuevo, por lo que indicar que su tamaño es cero simplemente nos indica que estos bloques son constantes y ya no hay nada que hacer con ellos. Todo lo anterior lo hacemos en las líneas 9-11. Por otro lado, si $k \geq i$, entonces tanto para B como para todos los bloques debajo de él necesitamos calcular el tamaño de los

nuevos bloques producidos por el refinamiento. Así, calculamos el tamaño del bloque (R_k, C') y junto con $tam(R_k, C_j)$ podemos determinar el valor de $tam(R_k, C'')$ lo cual hacemos en las líneas 12-17.

Para el análisis de tiempo. Observemos que recorremos a todos los elementos de S_R. Para aquellos elementos que se encuentran antes de R_i hacemos únicamente operaciones constantes. Por otro lado, para tanto R_i como todos los elementos que le siguen debemos de calcular el tamaño de los bloques producidos. Así, para cada $R_k \in S''_R$ obtenemos el tamaño del bloque (R_k, C'). Para calcular el valor anterior necesitamos recorrer a todos los renglones de R_k y por cada renglón recorrer a todas las columnas $c \in C'$. Por lo tanto, al finalizar el algoritmo, habremos recorrido a todos los renglones contenidos en S'_R, sea $\mathcal{R}''$ este valor. Concluimos de esta forma que el algoritmo toma tiempo $O(|\mathcal{R}''||C'| + |S'_R|)$. □

De manera similar, al tener un refinamiento de renglones, podemos actualizar el tamaño de todos los bloques afectados enfocándonos únicamente en la parte del refinamiento más pequeña; el algoritmo 32 hace esto.

Proposición 4.3.8. *Dado un bloque $B = M(R_i, C_j)$, un refinamiento R', R'' de R_i, y el conjunto de partes pendientes P_C de S_C; es decir, las partes que siguen después de C_{j-1} en la partición ordenada S_C. El algoritmo 32 calcula el tamaño de los bloques producidos por el refinamiento de renglones en tiempo $O(|R^*||P_C|)$.*

Demostración. En las líneas 1-6 obtenemos el conjunto R^* de menor cardinalidad entre R' y R'', supongamos sin pérdida de generalidad que R' es el menor. A diferencia del algoritmo anterior (algoritmo 31), aquí no necesitamos considerar a los bloques a la izquierda de B pues ya no vamos a volver a trabajar con ellos, por ese motivo sólo necesitamos a las partes que se encuentran después de C_j en S_C. Así, para cada $C_j \in P_C$, obtenemos el tamaño de $M(R', Cj)$ sumando los valores $tam(r, C_j)$ para cada renglón $r \in R'$, y junto con $tam(R_i, C)$ podemos determinar directamente $tam(R'', C)$. Todo lo anterior lo hacemos en las líneas 8-13.

Para el análisis de tiempo. Para cada $C_j \in P_C$ necesitamos determinar el tamaño de los bloques producidos (R', C_j) y (R'', C_j), lo cual lo podemos hacer únicamente enfocándonos en la parte de menor cardinalidad, en este caso R'. Para calcular el tamaño de (R', C_j) necesitamos recorrer a todas los renglones en R', por lo que determinar el tamaño de los bloques anteriores toma tiempo $O(|R'|)$. Concluyendo de esta forma que el algoritmo en total toma tiempo $O(|R'||P_C|)$. □

Algoritmo 32: Actualizar Tamaño Renglones

Input: Una matriz $M = (m_{ij})$ de tamaño $n \times n$, el bloque $B = M(R_i, C_j)$, las partes por trabajar de la partición ordenada de columnas P_C, un refinamiento R' y R'' de R_i, y la función tam.

Output: La función tam con el tamaño de $M(R', C)$ y $M(R'', C)$, donde C son las partes de las columnas que están a la derecha de C_i.

$R^* \leftarrow \varnothing, R^\star \leftarrow \varnothing$
if $|R'| \leq |R''|$ **then**
  $R^* \leftarrow R', R^\star \leftarrow R''$
else
  $R^* \leftarrow R'', R^\star \leftarrow R'$
end
for $C_j \in P_C$ **do**
  $t_B \leftarrow 0$
  for $r \in R^*$ **do**
    $t_B \leftarrow tam(r, C_j)$
  end
  $tam(R^*, C_j) = t_B$
  $tam(R^\star, C_j) = tam(R_i, C_j) - tam(R^*, C_j)$
end
return tam

4.3.4. El diseño final

Finalmente, presentamos el algoritmo 33, que es una instancia más concreta del algoritmo 26 presentado anteriormente. Sin pérdida de generalidad, suponemos que tanto los renglones como las columnas están indexados del 1 al n. Vamos a utilizar una pila P_C guardar a las partes de las columnas con las que vayamos trabajando, y por cada parte vamos a recorrer a todas las partes R_i que se encuentran en S_R. De esta forma, podemos siempre obtener al respectivo bloque $B = (R_i, C_j)$. Ahora, tenemos que ver que efectivamente el algoritmo diseñado sigue los procedimientos especificados en el algoritmo original.

Proposición 4.3.9. *Dada una matriz M de tamaño $n \times n$ cuyas entradas son sólo 0 ó 1, el algoritmo 33 con entrada M produce la misma salida que la esperada por el algoritmo 26.*

Demostración. Al inicio de la ejecución simplemente indicamos que $S_R = \{\mathcal{R}\}, S_C = \{\mathcal{C}\}$ y agregamos $\mathcal{C}$ a la pila P_C. De esta forma, al iniciar el ciclo de la línea 7, el primer elemento que tomamos de P_C es $\mathcal{C}$, el primer elemento de S_R en la línea 9 es $\mathcal{R}$. Entonces el primer bloque con el que vamos a trabajar es el bloque inicial $(\mathcal{R}, \mathcal{C})$. Recordemos que el objetivo del algoritmo original (algoritmo 26) es obtener un refinamiento del bloque inicial de tal forma que al final terminemos con bloques constantes que induzcan un orden lexicográfico tanto en renglones como en columnas. Todo lo anterior ya lo demostramos en el lema 4.2.4 y en el corolario 4.2.7.

Supongamos entonces que estamos trabajando con el bloque $B = (R_i, C_j)$, es decir, la cabeza de la pila P_C es C_j y la parte de la partición ordenada S_R en la que nos encontramos es R_i. Si B es constante, entonces no hay nada que hacer y nos movemos al bloque de abajo, lo anterior se ve representado como ir a la parte que sigue de R_i, es decir, R_{i+1}. Si para todo $R_i \in S_R$ se tiene que (R_i, C_j) es un bloque constante, entonces ya no hay nada que hacer con la parte C_j, por lo que la eliminamos de la pila P_C, si la pila P_C es vacía entonces hemos terminado, de lo contrario ahora sacamos a C_{j+1}. Si B no es constante, entonces existe un renglón o columna separadora, para determinar si existe o no un renglón separador utilizamos el algoritmo 30.

Si existe un renglón separador r, entonces podemos dar un refinamiento por columnas. Utilizamos el algoritmo 27 para obtener el refinamiento C' y C'' de C_j descrito en el algoritmo original. Reemplazamos a C_j en S_C por C' y C'' en este orden. Ahora, tenemos que calcular el tamaño de los bloques producidos por este refinamiento, para ello utilizamos el algoritmo 29. Eliminamos a la cabeza C_j de P_C y la reemplazamos por C'' y C' en este orden. Dado que la partición ordenada S_C fue modificada, entonces necesitamos *salir* del ciclo actual para que podamos sacar a la nueva cabeza de P_C, la cual es C' y de esta forma obtener el bloque (R_i, C'); lo anterior lo hacemos con la instrucción `goto` con la cual nos movemos al inicio del ciclo de la línea 8, donde tomamos a la cabeza de P_C. Así, inductivamente nos enfocamos en producir un refinamiento de (R_i, C'). Todo lo anterior, lo hacemos en las líneas 15-20.

Si no existe un renglón separador, entonces podemos dar un refinamiento por renglones. Utilizamos el algoritmo 28 para obtener el refinamiento R' y R'' de R_i descrito en el algoritmo original. Reemplazamos a R_i en S_R por R' y R'' en este orden. Calculamos el tamaño de los bloques producidos por este refinamiento, para ello utilizamos el algoritmo 32. Dado que eliminamos a R_i de S_R y la reemplazamos por R' y R'', entonces las partes que se encuentra en S_R en las posiciones i e $i+1$ son R' y R'', para los cuales sabemos que los bloques (R', C_j) y (R'', C_j) son constantes. Por lo tanto la siguiente parte a recorrer es la parte R_{i+2}. Así, inductivamente nos

enfocamos en producir un refinamiento de (R_{i+2}, C_j). Todo lo anterior, lo hacemos en las líneas 22-24.

Los procedimientos anteriores, aunque tienen pasos adicionales, son equivalentes a los descritos en las líneas 6-14 del algoritmo original, el cual sabemos funciona por la proposición 4.2.3. Además, observemos que por la forma en la que trabajamos con la pila P_C y con la partición ordenada S_R, recorremos en un orden de *izquierda a derecha* a las partes de las columnas y en un orden de *arriba a abajo* a las partes de los renglones. En otras palabras, lo anterior nos permite trabajar siempre con el bloque no constante tal que todos los bloques arriba y la izquierda de él son constantes. Cumpliendo así con la condición de la línea 5 del algoritmo original. De esta forma, el algoritmo 33 sigue los procedimientos especificados en el algoritmo 26, en donde sabemos, por el corolario 4.2.7, que el procedimiento especificado por el algoritmo original regresa un orden doblemente lexicográfico sobre M. Por ende, el algoritmo 33 también regresa un orden doblemente lexicográfico sobre M. □

Algo importante a considerar del algoritmo 33 es el cómo utilizamos a la función *tam*. Al momento de sacar a la cabeza C_j de la pila P_C y de recorrer al elemento R_i de S_R, necesitamos determinar si el bloque $B = (R_i, C_j)$ es constante o no, para ello necesitamos del valor $tam(R_i, C_j)$. De igual forma, si B no es constante, para encontrar un renglón separador necesitamos los valores $tam(r, C_j)$ para todo $r \in R_i$. Por último al definir un refinamiento sobre C_j, necesitamos de nuevo los valores $tam(r, C_j)$ para toda $r \in R_i$ para calcular el tamaño de los nuevos bloques producidos (R_i, C') y (R_i, C''). Por otro lado, al definir un refinamiento sobre R_i, necesitamos los valores $tam(R_i, C)$ para toda $C \in P_C$. En otras palabras, la función *tam* necesita estar definida para todos los bloques con los que llegamos a trabajar durante la ejecución del algoritmo. Veamos que efectivamente lo anterior se cumple.

Proposición 4.3.10. *Durante la ejecución del algoritmo 33, para todo bloque $B = (R_i, C_j)$ sobre el cual necesitamos determinar si es constante o dar un refinamiento de R_i o C_j, los valores $tam(R_i, C_j)$ y $tam(r, C_j)$ para toda $r \in R_i$ ya se encuentran definidos.*

Demostración. Dado un bloque $B = (R_i, C_j)$ del cual tenemos el tamaño, veamos que el algoritmo 26 define el tamaño de todos los posibles sub-bloques de B definidos por el refinamiento de R_i o C_j.

El tamaño del bloque inicial $B = (\mathcal{R}, \mathcal{C})$ lo calculamos en la línea 5, es decir, los valores $tam(\mathcal{R}, \mathcal{C})$ y $tam(r, \mathcal{C})$ para toda $r \in \mathcal{R}$, ya se encuentran definidos. Por lo tanto sí existe un bloque del que tenemos su tamaño. Sea $B = (R_i, C_j)$ el bloque en el que nos encontramos.

Si B tiene un renglón separador r, entonces usamos el algoritmo 30 para obtener a r por medio de los valores $tam(r, C_j)$ que ya hemos calculado. Después, por medio del algoritmo 27 obtenemos un refinamiento de columnas C' y C'' de C_j. Ahora tenemos que calcular el tamaño de los bloques producidos por el refinamiento, para ello utilizamos el algoritmo 31. Lo anterior implica recorrer a todas las partes que se encuentran en S_R. Sea R_k la parte que estamos recorriendo de S_R. Si $k < i$, entonces el bloque $B' = (R_k, C_j)$ se encuentra arriba de B, y por definición del algoritmo, este es constante, por lo tanto simplemente indicamos que $tam(R_k, C') = tam(R_k, C'') = 0$; dado que ya no vamos a volver a recorrer a los elementos de este bloque, entonces solo asignamos sus tamaños a cero para indicar que son constantes. Si $k \geq i$, sin pérdida de generalidad supongamos que $|C'| \leq |C''|$, entonces calculamos el valor de $tam(R_k, C')$ y con este junto con $tam(R_k, C_j)$ podemos determinar el valor de $tam(R_k, C'')$. Concluyendo de esta forma, que en el refinamiento por columnas, calculamos correctamente el tamaño de los bloques (R_i, C'), (R_i, C''), y también para aquellos bloques tanto arriba como abajo de éstos. De esta forma, al eliminar a C' de P_C y pasar a C'', los valores $tam(R_i, C'')$ para todo $R_i \in S_R$ están bien definidos.

Si B no tiene un renglón separador, entonces tiene un refinamiento por renglones. Usamos el algoritmo 28 para obtener al refinamiento R' y R'' de R_i. Ahora tenemos que calcular el tamaño de los bloques producidos por este refinamiento, para ello utilizamos el algoritmo 32. Lo anterior implica recorrer tanto a C_j como a todas las partes que siguen después de él. Así, sin pérdida de generalidad supongamos que $|R'| \geq |R''$ y sea C_k para parte de columnas en la que nos encontramos. Calculamos el valor de $tam(R', C_k)$ y junto con $tam(R_i, C_k)$ determinamos el valor de $tam(R'', C_k)$. Observemos que ya tenemos los valores $tam(r, C_k)$ para todo $r \in R_i$, por lo tanto estos valores también están definidos para los valores $tam(r', C_k)$ para todo $r' \in R', R''$. A diferencia del caso anterior, aquí no es necesario considerar a los bloques que están a la izquierda de B, pues ya no los vamos a volver a recorrer. Por lo tanto, en el refinamiento por renglones, calculamos correctamente el tamaño de los bloques (R', C_j), (R'', C_j), y también para aquellos bloques a la derecha de estos. De esta forma, al eliminar C_j de P_C y pasar a C_{j+1} (si es que existe), los valores $tam(R', C_{j+1})$ y $tam(R'', C_{j+1})$ están bien definidos. □

La proposición anterior nos asegura el funcionamiento adecuado de nuestro algoritmo, el cual también sabemos que funciona como se esperaba. Ahora, lo último que falta por determinar es qué tan eficiente es. Lo anterior lo vamos a hacer por partes, demostrando cuánto tiempo toma cumplir cada objetivo del algoritmo.

Proposición 4.3.11. *Si M es una matriz de tamaño $n \times n$, y $\mathcal{B}$ la cantidad de bloques con los que tenemos que trabajar durante toda la ejecución del algoritmo 33,*

entonces $\mathcal{B} \in O(n^2)$.

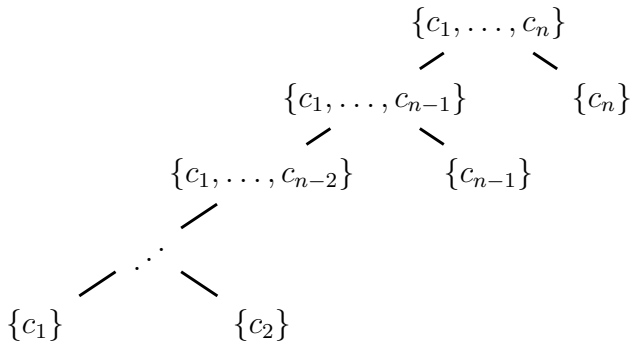
Demostración. Veamos cuál es el refinamiento más costoso con el que nos podemos encontrar. Sea $B = M(\mathcal{R}, \mathcal{C})$ el bloque inicial, donde $\mathcal{R} = \{1, \ldots, n\}$ y $\mathcal{C} = \{1, \ldots, n\}$. En el peor de los casos, los refinamientos finales son refinamientos en donde cada una de las partes sólo incluyen a un elemento, es decir, $|S_R| = n = |S_C|$, en donde S_R y S_C se ven de la siguiente forma:

$$S_R = (\{r_1\}, \ldots, \{r_n\}),\ S_C = (\{c_1\}, \ldots, \{c_n\}).$$

De esta forma, si (C', C'') es un refinamiento de C_j, en el peor de los casos tenemos que:

$$C' = C_j - \{c\},\ C'' = \{c\}.$$

En donde en cada refinamiento sólo separamos a un elemento de la parte original. De esta forma, definimos al árbol binario de particiones de columnas, $\mathcal{T}_{C_j}$, donde si un parte C_j tiene un refinamiento C' y C'', entonces C_j es adyacente a C' y C''. De esta forma, el árbol $\mathcal{T}_{\mathcal{C}}$ se ve de la siguiente forma:



En donde en cada nivel, excepto el último, tenemos dos nodos. Lo anterior se debe a que cada refinamiento produce exactamente dos nuevas partes. Observemos que para tener una partición ordenada de la forma $(\{c_1\}, \{c_2\}, \ldots, \{c_n\})$ necesitamos de $n-1$ refinamientos, ya que en cada refinamiento solo estamos separando a un elemento, por lo tanto el subárbol tiene n niveles. Concluyendo de esta forma que la cantidad de nodos en $\mathcal{T}_{\mathcal{C}}$ es $2(|\mathcal{C}| - 1) + 1 = 2|\mathcal{C}| - 1$; en particular dado que $|\mathcal{C}| = n$, entonces tenemos $2n - 1$ nodos. El árbol anterior representa, en el peor de los casos, a todas las partes de la partición ordenada de $\mathcal{C}$ con las que debemos de trabajar. Además, recordemos que por cada parte C_j que lleguemos a tener en P_C, en el peor de los casos debemos de recorrer a todas las partes de S_R para poder determinar si existe

un $R_i \in S_R$ tal que (R_i, C_j) no sea un bloque constante. En otras palabras, por cada parte $C_j \in P_C$ necesitamos recorrer a todos los renglones de M. Dado que podemos tener hasta $2n-1$ partes con las que trabajar y por cada una de ellas debemos de recorrer $|\mathcal{R}| = n$ renglones, entonces el máximo número de bloques podemos llegar a tener durante la ejecución del algoritmo es $n(2n-1) = 2n^2 - n$. Concluyendo de esta forma que $\mathcal{B} \in O(n^2)$. □

Proposición 4.3.12. *Dada una matriz M de tamaño $n \times n$. El algoritmo 33 calcula el tamaño de todos los bloques utilizados durante la ejecución en tiempo $O(n^2 \log n)$.*

Demostración. Veamos cuánto tiempo tarda el algoritmo en calcular el tamaño de todos los bloques generados a partir de un bloque $B = M(R_i, C_j)$.

Empecemos por el refinamiento de columnas. Recordemos que en cada refinamiento de columnas necesitamos recorrer a todos los elementos de S_R, el cual en el peor de los casos tiene hasta n elementos; uno por cada renglón en M.

Sea $R_k \in S_R$. Si $k < i$, entonces $B' = (R_k, C_j)$ se encuentra arriba de B y por lo tanto es constante. Por ende, determinar el tamaño de (R_k, C') y (R_k, C'') toma tiempo constante. Dado que la cantidad de bloques que están arriba de B está en el orden de $O(n)$, entonces determinar el tamaño de estos bloques toma tiempo $O(n)$. Por otro lado, si $i \geq k$, entonces los bloques abajo de B no necesariamente son constantes, por lo que hay que determinar el tamaño de los bloques producidos por el refinamiento. Observemos que podemos calcular los valores anteriores usando únicamente al conjunto de menor cardinalidad entre C' y C''. Sin pérdida de generalidad supongamos que $|C'| \leq |C''|$. Sea S'_R las partes que van después de R_{i-1} en S_R. Por cada $R \in S'_R$ tenemos que recorrer a $|R||C'|$ elementos para poder determinar los tamaños de los nuevos bloques producidos. Así, los refinamientos más costosos son aquellos en donde C' y C'' tienen la misma cardinalidad. Si además la cantidad de renglones en S'_R es $|\mathcal{R}|$, entonces en estos casos tenemos que recorrer $\frac{|\mathcal{R}||C_j|}{2}$ elementos.

Sea $\mathcal{T}_\mathcal{B}$ el árbol binario de bloques tal que si $B = (R_i, C_j)$ tiene un refinamiento de columnas, entonces los bloques (R_i, C') y (R_i, C'') son adyacentes a B; lo definimos igual para el refinamiento de renglones. Si el caso anterior sucede, en donde por cada refinamiento de columnas tenemos que las partes tienen la misma cardinalidad, entonces $\mathcal{T}_\mathcal{B}$ se ve como la figura 4.10, en donde indicamos a la cardinalidad de cada bloque en vez de su conjunto de elementos.

La altura del árbol $\mathcal{T}_B$ es $\log |C_j|$. Ahora, en el nivel i de $\mathcal{T}_B$ tenemos 2^{h-i} bloques, donde h es la altura del árbol. Si producimos un refinamiento sobre columnas para cada uno de estos bloques tal que C' y C'' tengan el mismo tamaño, entonces sólo necesitamos recorrer a uno de los producidos por cada bloque. De esta forma, en el nivel $i+1$ vamos a tener 2^{h-i+1} bloques, en donde para calcular el tamaño de todos

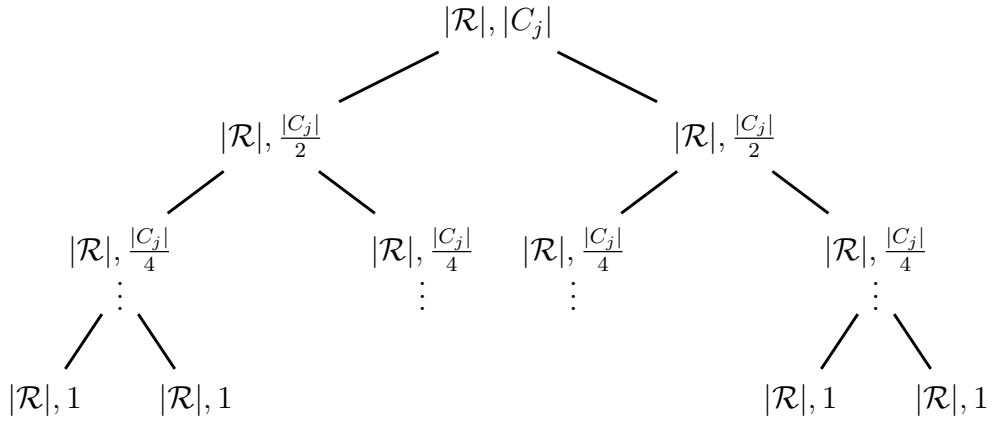


Figura 4.10: El árbol $\mathcal{T}_B$ donde en cada paso los refinamientos obtenidos tienen el mismo tamaño.

estos bloques sólo recorremos a 2^{h-i} bloques. Además, para calcular el tamaño de un bloque en el nivel $i+1$ tenemos que recorrer a $\frac{|\mathcal{R}||C_j|}{2^{h-i+1}}$ elementos. Por lo que para obtener el tamaño de todos los bloques en el nivel $i+1$ necesitamos recorrer a la siguiente cantidad de elementos:

$$2^{h-i} \cdot \frac{|\mathcal{R}||C_j|}{2^{h-i+1}} = \frac{|\mathcal{R}||C_j|}{2}.$$

Por ende, para cada nivel del árbol, excepto el último, calcular el tamaño de todos los bloques que se encuentran en el nivel siempre requiere de recorrer a $\frac{|\mathcal{R}||C_j|}{2}$ elementos de M. Concluyendo de esta forma que, en el peor de los casos, calcular el tamaño de todos los bloques producidos por todos los refinamientos por columnas requiere recorrer a la siguiente cantidad de elementos:

$$|\mathcal{R}||C_j| + \frac{|\mathcal{R}||C_j|}{2}((\log |C_j|) - 1).$$

En particular si $C_j = \mathcal{C}$ y $|\mathcal{R}| = |\mathcal{C}| = n$, entonces necesitamos recorrer a $n^2 + \frac{n^2}{2}((\log n) - 1)$ elementos; el primer n^2 se debe a que debemos de calcular el tamaño del bloque inicial.

De esta forma, por cada refinamiento por columnas, determinar el tamaño de los nuevos bloques constantes toma tiempo $O(n)$. Dado que podemos tener hasta $n-1$ refinamientos, entonces en total calcular el tamaño de todos los bloques constantes con los que lleguemos a trabajar toma tiempo $O(n^2)$. De igual forma, determinar el tamaño de todos bloques no constantes producidos toma en total tiempo $O(n^2 \log n)$.

Así, calcular el tamaño de todos los posibles bloques obtenidos por refinamientos de columnas toma tiempo $O(n^2)$ y $O(n^2 \log n)$; es decir, toma tiempo $O(n^2 \log n)$.

El argumento anterior es análogo para calcular los tamaños de los bloques producidos por refinamientos de renglones. Concluyendo que el algoritmo toma tiempo $O(n^2 \log n)$ para calcular el tamaño de todos los posibles bloques con los que se trabajan durante la ejecución.

□

Lema 4.3.13. *Dada una matriz M de tamaño $n \times n$, el algoritmo 33 regresa un ordenamiento doblemente lexicográfico de M en tiempo $O(n^2 \log n)$.*

Demostración. Como se argumentó en la proposición 4.3.11, los refinamientos más costosos son aquellos en donde sólo separamos a un elemento de la parte original. Así, en el peor de los casos tenemos $n-1$ refinamientos de columnas y $n-1$ refinamientos de renglones.

Para poder construir a la sucesión S_R, por cada refinamiento necesitamos sustituir a R_i por R' y R'' en dicho orden. Obtener la posición en la que se encuentra R_i en S_R y sustituirla por R' y R'' toma tiempo $O(n)$, y como se argumentó anteriormente, en el peor caso tenemos $n-1$ refinamientos de $\mathcal{R}$. Por lo tanto, construir a la sucesión S_R toma tiempo $O(n^2)$. El argumento es análogo para S_C.

Determinar si un bloque $B = M(R_i, C_j)$ tiene un renglón separador toma tiempo $O(|R_i|)$. Tomemos en cuenta que sólo tratamos de encontrar a un renglón separador si el bloque en el que estamos no es constante. De esta forma, cada que hacemos un refinamiento (ya sea sobre renglones o columnas) implica que antes buscamos a un renglón separador. Dado que en el peor de los casos podemos tener $n-1$ refinamientos sobre renglones y $n-1$ refinamientos sobre columnas, y por cada refinamiento a lo más recorremos n renglones, lo anterior implica determinar el renglón separador para todos los refinamientos definidos toma tiempo $O(n^2)$.

Por otro lado, sabemos por la proposición 4.3.11 que el algoritmo toma tiempo $O(n^2)$ para recorrer a todos los bloques definidos durante el refinamiento de $\mathcal{R}$ y $\mathcal{C}$. Además, por la proposición 4.3.12, calcular el tamaño de todos los bloques utilizados por el algoritmo toma tiempo $O(n^2 \log n)$.

Así, nuestro diseño del algoritmo 26 tiene instrucciones que toman tiempo $O(n^2)$ y $O(n^2 \log n)$; el algoritmo 33 toma tiempo $O(n^2 \log n)$.

Finalmente, por la proposición 4.3.9 sabemos que nuestro diseño regresa la misma salida que el algoritmo original. En otras palabras, nuestro algoritmo regresa un ordenamiento doblemente lexicográfico de M en tiempo $O(n^2 \log n)$. □

Algoritmo 33: Ordenamiento Doblemente Lexicográfico

Input: Una matriz M de tamaño $n \times n$ cuyas entradas son sólo 0 ó 1.
Output: Las particiones ordenadas S_R y S_C correspondientes al orden lexicográfico de M.

```
1  R ← {1,...,n}, C ← {1,...,n}
2  S_R ← (R), S_C ← (C)
3  P_C ← []
4  tam ← (∅, ∅)
5  tam ← CalcularTam(R, C, tam)
6  agregar C a P_C
7  i ← 0
8  while P_C ≠ [] do
9      elegir la cabeza C_j de P_C
10     while i < |S_R| do
11         R_i ← el elemento en la posición i de S_R
12         if 0 < tam(R_i, C_j) < |Ri||C_j| then
13             r ← RenglonSeparador(R_i, C_j)
14             if r ≠ ∅ then
15                 C', C'' ← RefinamientoColumnas(C_j)
16                 reemplazar a C_j en S_C por C', C'' en este orden
17                 tam ← ActualizarTamCols(M, (R_i, C_j), S_R, C', C'', tam)
18                 eliminar la cabeza C_j de P_C
19                 agregar C'' y C' a P_C en este orden al final de P_C
20                 go to línea 8
21             else
22                 R', R'' ← RefinamientoRenglones(R_i)
23                 reemplazar a R_i en S_R por R', R'' en este orden
24                 tam ← ActualizarTamReng(M, (R_i, C_j), P_C, R', R'', tam)
25                 i ← i + 1
26             end
27         end
28         i ← i + 1
29     end
30     eliminar la cabeza C_j de P_C
31     i ← 0
32 end
33 return S_R, S_C
```

Capítulo 5

Conclusiones

A lo largo de este trabajo se presentaron una serie de algoritmos que pueden ser clasificados en las siguientes secciones:

1. Conversiones de formatos para guardar gráficas en memoria.

2. Detectar el isomorfismo entre dos árboles.

3. Producir un ordenamiento doblemente lexicográfico de una matriz.

En el caso de los algoritmos para la conversión de formatos, la descripción oficial proporcionada por Brendan McKay [4] explica la forma en la que podemos obtener una gráfica a partir de la cadena de texto correspondiente. Sin embargo, no hay una descripción oficial para la conversión de gráficas a texto ASCII. Por este motivo, en este trabajo nos enfocamos principalmente en el cómo convertir una gráfica a alguno de los formatos presentados. Guardar gráficas de forma persistente en memoria es de mucha utilidad al momento de trabajar con familias de gráficas que cumplan con ciertas propiedades. Por ejemplo, este tipo de conversiones nos permiten tener en un archivo de texto a todas las gráficas de cierta cantidad de vértices que no sean isomorfas entre sí. Actualmente hay múltiples programas y bibliotecas disponibles que nos permiten convertir gráficas a texto, y viceversa. Dentro de estas podemos encontrar a `networkx` una biblioteca de Python para trabajar con gráficas, a `rgraph6` una biblioteca de R diseñada específicamente para la conversiones, o incluso `MappleSoft` un software diseñado para la investigación matemática, entre otros. No obstante, los algoritmos de conversión presentados en este trabajo son valiosos, pues siempre es útil tener una descripción formal sobre estas conversiones por si es necesario o conveniente que uno lo implemente por su cuenta.

Sobre los algoritmos para detectar si dos árboles son isomorfos. El algoritmo original presentado por Aho, Hopcroft, y Ullman en 1974 [5], habla sobre un algoritmo lineal capaz de detectar el isomorfismo entre dos árboles enraizados pero la explicación de este algoritmo era informal y además no se presentaba una demostración sobre su corrección. En el 2002, Valiente [6] presenta una variante de este problema, en donde diseña un algoritmo para detectar el isomorfismo entre árboles ordenados enraizados en tiempo y espacio $O(n)$, después propone un algoritmo para resolver el problema original pero el rendimiento de este es de $O(n^2)$ en tiempo. De igual manera, en el 2010, Marthe Bonamy presenta un artículo [7] en donde estudia el isomorfismo de árboles, mencionando el algoritmo original e incluso presenta uno para detectar el isomorfismo de árboles no enraizados; no obstante, omite la implementación y demostración de sus algoritmos. En los casos anteriores el tema principal de los autores no era el isomorfismo de árboles, y por esta razón no fue necesario implementar el algoritmo o demostrar la corrección de estos. Sin embargo, todo lo anterior implica que no hay una demostración formal del algoritmo original. Por este motivo, el algoritmo presentado en este trabajo no solo sirve como un diseño lineal del algoritmo original para árboles enraizados sino también para demostrar formalmente que determinar el isomorfismo de árboles se puede llevar a cabo en tiempo y espacio lineal. Cabe mencionar que el isomorfismo de árboles ha resultado tener múltiples aplicaciones en las áreas de Bioinformática [9], en particular en el empalme de genes, el análisis de proteínas y el estudio de moléculas, ya que las estructuras químicas presentes aquí suelen ser árboles con millones de vértices. Y como lo mencionó Douglas M. Campbell en su momento, en este tipo de aplicaciones, la diferencia entre $O(n)$, $O(n \log n)$ y $O(n^2)$ no sólo es teórica sino también práctica.

Sobre los algoritmos para obtener un orden doblemente lexicográfico. El artículo original presentado por Anna Lubiw en 1987 [10] habla sobre un algoritmo capaz de regresar un orden doblemente lexicográfico de una matriz, en donde sus renglones y columnas, como vectores, están ordenados lexicográficamente de mayor a menor. Lubiw después describe y demuestra el algoritmo de una manera similar a como lo hacemos en el algoritmo 26, con la diferencia de que este toma tiempo $O(L \log^2 L)$ donde $L = e + m + n$ para matrices de tamaño $m \times n$ y con e entradas no cero. Ese mismo año Paige y Tarjan [11] describen una variante del algoritmo original, la cual toma tiempo $O(e \log(n+m)+m)$ y dan una breve descripción informal de cómo podría ser la implementación de este algoritmo. De igual forma, Hoffman, Sakarovich y Kolen [12] presentan una variante de este algoritmo, la cual toma tiempo $O(n^2 m)$; para este algoritmo existe una implementación hecha por François Brucker [13], la cual toma tiempo $O(n^3)$ para matrices cuyas entradas son sólo 0 o 1. En este trabajo demostramos de una manera más rigurosa que el algoritmo original efectivamente

siempre va a regresar un ordenamiento doblemente lexicográfico, adicional a esto, también presentamos un diseño más fiel a la variante propuesta por Paige y Tarjan, que toma tiempo $O(n^2 \log n)$ para matrices cuadradas cuyas entradas son sólo 0 o 1. El ordenamiento lexicográfico de (0,1)-matrices, como lo menciona Lubiw en su artículo [10], ha resultado tener múltiples aplicaciones en el estudio de gráficas totalmente balanceadas, matrices de subárboles, y gráficas cordales, por lo que siempre es útil tener algoritmos eficientes para el estudio de estas estructuras.

5.1. Sobre la implementación de las funciones

Los objetivos principales de los algoritmos presentados aquí son para poder justificar y demostrar sus respectivas complejidades de una manera formal. Aunque los algoritmos presentados aquí son lo suficientemente de *bajo nivel* como para poderlos implementar *casi* directamente en cualquier lenguaje de programación, hay un detalle importante a considerar, las funciones. El objetivo principal de nuestras funciones es el poder acceder en tiempo constante a recursos que de otra forma nos tomarían más tiempo obtener, y de esta forma conseguir las complejidades que estamos buscando. Por ejemplo, en el algoritmo 26, en el artículo presentado por Paige y Tarjan, mencionan que es posible determinar en tiempo $O(1)$ que un bloque de la forma $M(r, C_j)$ no es constante, sin embargo omiten los detalles sobre cómo podríamos realizar lo anterior. En situaciones como ésta se optó por usar una función sobre la cual siempre podamos acceder a sus elementos en tiempo constante, como es el caso de la función *tam* para el algoritmo anterior. Tenemos la misma situación en el isomorfismo de árboles enraizados al momento de tener que determinar a qué vértices les corresponde cierta estructura en el algoritmo 19, en donde necesitamos de una función M_S para poder guardar y tener acceso a estos elementos en tiempo constante. Mientras que en la teoría esto funciona bien y nos permite obtener las complejidades que buscamos, en la práctica esto es un poco diferente.

Las funciones dentro de la mayor parte de los lenguajes de programación se implementan por medio de lo que conocemos como **Tablas de Dispersión** (conocidas como *Hash Tables* en inglés). Estos objetos suelen utilizar una **función de dispersión** (*hash function*) con el objetivo de que objetos distintos sean mapeados a salidas distintas, es decir, buscan ser como una función inyectiva. Dependiendo de la función de dispersión que usemos será el qué tan eficiente sea el agregar y acceder a los elementos contenidos en la tabla de dispersión. En el caso en el que tengamos que mapear algún entero o cadena a algún otro elemento, ya existen funciones de dispersión *lo suficientemente buenas* como Sip Hash [15]. Desafortunadamente, algunas de nuestras funciones trabajan con valores distintos a lo que es un entero o

una cadena, como la función *tam* que trabaja con parejas de conjuntos, y la función M_S que trabaja con multiconjuntos. Una manera de resolver lo anterior es optar por trabajar con una representación en cadena de los objetos anteriores, como la que tenemos en el algoritmo 4, y de esta forma utilizar la función de dispersión antes descrita. Otro detalle importante es que, por cómo funcionan las tablas de dispersión internamente, en muy pocas ocasiones el agregar un elemento no toma tiempo $O(1)$ si no $O(n)$. Dado que el comportamiento anterior ocurre muy pocas veces, solemos decir que el agregar y acceder a elementos de una tabla de dispersión toma tiempo $O(1)$ **amortizado**. Por lo tanto, en caso de implementar los algoritmos utilizando las funciones dadas, la complejidad de estos algoritmos estarían amortizadas a $O(n)$ y $O(n^2 \log n)$ respectivamente.

Por otro lado, si estamos trabajando con lenguajes de programación estructurados u orientados a objetos, tenemos la posibilidad de definir objetos que puedan guardar la información que necesitamos y así no depender de una única función. Por ejemplo, en el caso del ordenamiento doblemente lexicográfico, podemos definir una estructura u objeto `Bloque` que tenga que tenga una variable para guardar el tamaño del bloque, un apuntador al bloque que sigue en el orden en el que trabajamos, un apuntador al bloque que se encuentra debajo de él, y otro apuntador al bloque que se encuentra a la derecha de él. Así, en el diseño anterior omitimos el uso de la función *tam*, mientras que preservamos el comportamiento del algoritmo original; claro que en los algoritmos 31 y 32, además de calcular el tamaño de los bloques producidos, también tendríamos que actualizar a los apuntadores de los bloques para reflejar estos cambios. Desde el punto de vista de estructuras de datos, podemos pensar en el conjunto de bloques como una lista doblemente ligada (*Doubly Linked List* en inglés), en donde los elementos de la lista tienen dos apuntadores adicionales que les indican quién está a la derecha y abajo. Presentamos la figura 5.1 para ejemplificar el argumento anterior, en donde las líneas rayadas rojas son el apuntador al bloque de abajo, las líneas punteadas azules son el apuntador al bloque de la derecha, y la línea negra es el apuntador al siguiente bloque en el orden específicado por el algoritmo original.

5.2. Resultados

Los algoritmos presentados aquí han sido implementados en el lenguaje de programación Java [17], en donde se probó la eficiencia de estos algoritmos con diferentes entradas con distintos tamaños.

En el caso del algoritmo para determinar el isomorfismo entre dos árboles, la prueba se realizó sobre una familia de árboles a las que les vamos a llamar **rehiletes**.

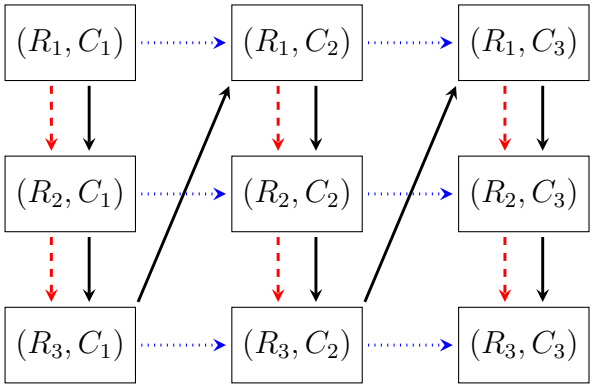


Figura 5.1: Un ejemplo visual de los apuntadores de las estructuras `Bloques`.

La definición de la gráfica rehilite es la siguiente:

Algoritmo 34: Árbol Rehilete Rh

Input: El número de ramas i que tendrá el rehilete.
Output: El árbol rehilete con i ramas.

$Rh_i \leftarrow$ la gráfica trivial con el vértice v_0
for $i \in \{1, \ldots, i\}$ **do**
  $P \leftarrow$ una trayectoria con $|V_{Rh_i}|$ vértices ajenos a Rh_i
  $Rh_i \leftarrow$ la gráfica obtenida de hacer adyacente a v_0 con uno de los extremos de P
end
return Rh_i

Como podemos observar, si el rehilite tiene i ramas entonces la cantidad de vértices que tiene la gráfica es $1 + 2 + 2^2 + \cdots + 2^i = 2^{i+1} - 1$. Por lo tanto, el tamaño de la gráfica rehilete pertenece a $O(2^n)$. Para la prueba, se tomó a la gráfica rehilete y a una copia de este árbol, de esta manera siempre vamos a tener el peor caso del algoritmo 25, en donde sí existe un isomorfismo entre los dos árboles. En la figura 5.2 podemos observar los tiempos obtenidos, en donde el tiempo que le toma al algoritmo corresponde linealmente con el número de vértices que hay en el árbol.

En el caso del algoritmo para obtener el ordenamiento doblemente lexicográfico, la prueba se realizó sobre múltiples (0,1)-matrices cuadradas con entradas *psuedoaleatorias*. Como podemos observar, si una matriz cuadrada tiene tamaño 50, entonces la matriz contiene un total de 250,000 elementos. En la figura 5.3 podemos observar los tiempos obtenidos, en donde el tiempo que le toma al algoritmo regresar un orden

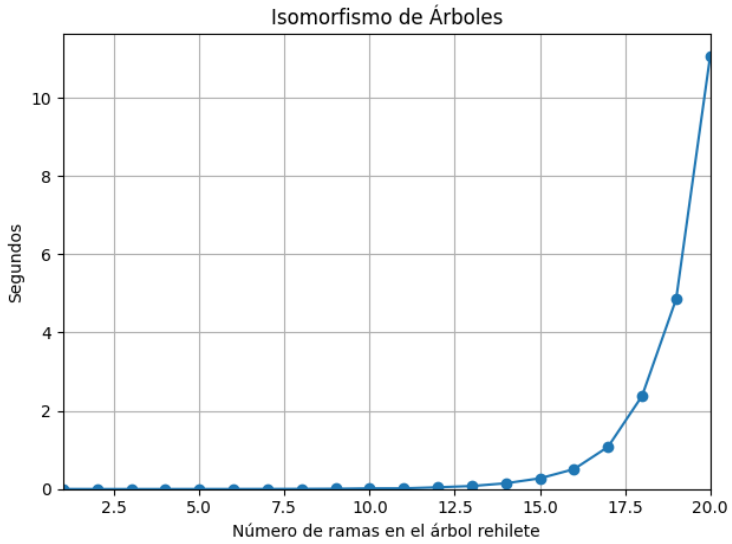


Figura 5.2: Los tiempos obtenidos para determinar el isomorfismo entre dos árboles rehiletes isomorfos.

doblemente lexicográfico corresponde con la complejidad obtenida. Hay ocasiones en donde entradas menores toman más tiempo que unas mayores, lo anterior se puede deber al renglón separador que estamos obteniendo. Recordemos que en el peor de los casos, un renglón separador puede determinar un refinamiento donde ambas partes tengan la misma cantidad de elementos, por lo que toma una mayor cantidad de tiempo determinar el tamaño de los bloques producidos. De igual manera ocurre lo contrario, en donde una entrada mayor toma menos tiempo que una menor. Lo anterior también se puede deber al renglón separador que obtenemos, en el mejor de los casos, en el refinamiento obtenido tenemos una parte que solo contiene a un elemento; reduciendo significativamente el tiempo que toma determinar el tamaño de los nuevos bloques.

Por último, se decidió implementar el algoritmo 25 en Python [18] y compararlo con la implementación de la biblioteca `networkx` [19], la cual toma tiempo $O(n \log n)$. Al igual que en la prueba anterior, se toma al árbol y a una copia de este. En este caso la gráfica de la figura 5.4 representa el tiempo que tomó determinar un isomorfismo para todos los árboles no-isomorfos con cierta cantidad de vértices. Como podemos observar, conforme más crece la entrada, más aparente es la diferencia logarítmica entre ambos algoritmos.

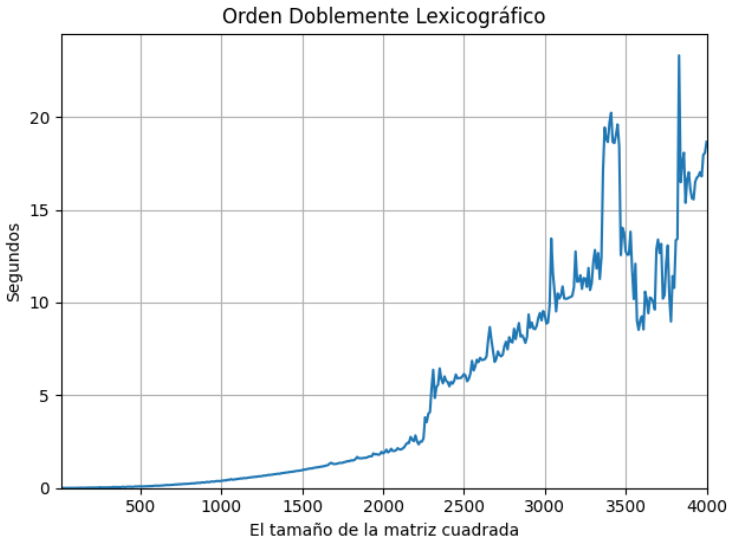


Figura 5.3: Los tiempos obtenidos para regresar un orden doblemente lexicográfico de una (0,1)-matriz cuadrada.

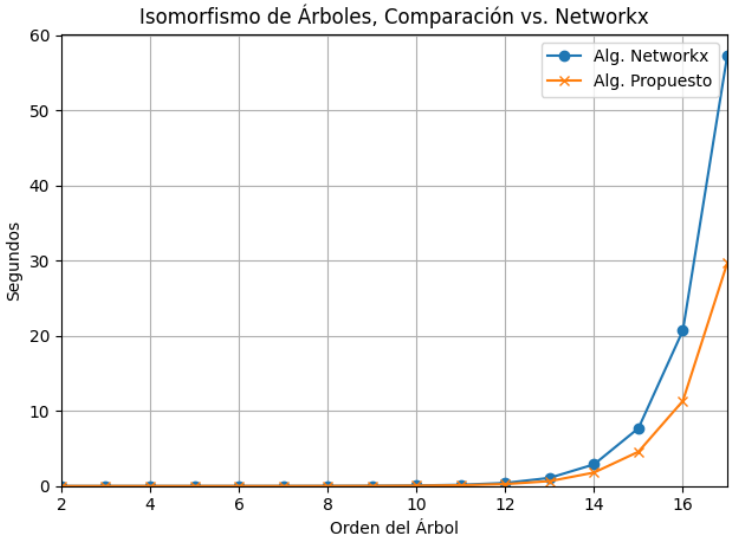


Figura 5.4: Comparación con la bibilioteca `networkx`.

5.3. Líneas futuras de investigación y trabajo

Como se mencionó anteriormente, el propósito de los algoritmos presentados en este trabajo es para poder argumentar y demostrar formalmente sus complejidades, tanto en tiempo como en espacio. Por esta razón, sus posibles implementaciones no son las más óptimas con respecto a las herramientas y utilidades que podemos encontrar en diversos lenguajes de programación; en especial con las tablas de dispersión.

Una potencial línea de trabajo es buscar una alternativa a las funciones para poder acceder a un elemento en tiempo constante; como el ejemplo presentado en la figura 5.1, en donde por medio de apuntadores y estructuras, podemos obtener el orden doblemente lexicográfico de una matriz.

Para el algoritmo 25 podemos tener una pequeña optimización al momento de combinar el propósito de los algoritmos 14 y 16 en un solo algoritmo para reducir el número de recorridos que hacemos sobre un árbol. Así, una futura línea de investigación, es determinar cuál es la menor cantidad de veces que tenemos que recorrer a todos los vértices de dos árboles para poder determinar si son isomorfos. De igual forma, investigar si es realmente necesario utilizar multiconjuntos u ordenar sucesiones de cadenas (o multiconjuntos) poder determinar si dos árboles comparten las mismas estructuras en un nivel, o si existe otra alternativa más eficiente y sencilla.

Por último, el diseño final del algoritmo 26 toma tiempo $O(n^2 \log n)$. Sin embargo, Paige y Tarjan mencionan que la complejidad en tiempo del algoritmo anterior es de $O(e \log n)$ donde e es la cantidad de entradas no cero en una matriz cuadrada. Una futura línea de investigación es determinar de qué manera podemos calcular el tamaño de un bloque recorriendo únicamente a aquellas entradas no cero. Por otro lado, la manera en la que elegimos a un renglón separador puede mejorar, de tal forma que siempre elijamos al óptimo; es decir, aquél que nos ayude a recorrer a la menor cantidad de elementos para calcular el tamaño de los bloques producidos por el refinamiento obtenido.

Bibliografía

[1] J. A. Bondy y U. S. R. Murty, **Graph Theory**, Springer, 2008.

[2] R. Diestel, **Graph Theory, Fifth Edition**, Springer, 2017.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, **Introduction to Algorithms, Third Edition**, The MIT Press, 2009.

[4] B. McKay, Description of graph6, sparse6 and digraph6 encodings (2022), https://users.cecs.anu.edu.au/~bdm/data/formats.txt. Acceso: Enero 24, 2023.

[5] A. V. Aho, J. E. Hopcroft y J. D. Ullman, **The Desing And Analysis of Computer Algorithms**, Addison Wesley Publishing Company, 1974.

[6] G. Valiente, *Algorithms on Trees and Graphs*, Springer, 2002.

[7] M. Bonamy, *A Small Report on Graph and Tree Isomorphism*, 2010.

[8] A. Smal. "Explanation for Tree Isomorphism talk", Joint Advanced Student School, Saint-Petersburg, Russia, 2008.

[9] D. M. Campbell y D. Radford, *Tree Isomorphism Algorithms: Speed vs. Clarity*, Mathematics Magazine, Mag. 64, No. 4, 1991, pp. 252-261.

[10] A. Lubiw, *Doubly Lexical Orderings of Matrices*, SIAM Journal on Computing, 1987.

[11] R. Paige, R. E. Tarjan, *Three Partition Refinement Algorithms*, SIAM Journal on Computing, 1987.

[12] A. J. Hoffman, M. Sakarovich, y A. Kolen, *Totally Balanced and Greedy Matrices*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 721-730.

[13] *Totally Balanced Structures: Doubly Lexical Order*, https://github.com/6ka/totally_balanced_structures/blob/master/tbs/contextmatrix/_order.py. Acceso: Enero 24, 2023.

[14] C. Peláez, **Estructuras de Datos con Java Moderno**, Las Prensas de Ciencias, 2018.

[15] J. P. Aumasson, D. J. Bernstein, *SipHash: a fast short-input PRF*, Cryptology ePrint Archive, 2012.

[16] T. Oetiker, H. Partl, I. Hyna, and E. Schlegl, The Not So Short Introduction to LaTeX Version 6.4 (2021), https://tobi.oetiker.ch/lshort/lshort.pdf.

[17] *Thesis Algorithms*, https://github.com/maucarrui/thesis-algorithms Acceso: Enero 24, 2023.

[18] *Networkx: Tree Isomorphism Alternative*, https://github.com/maucarrui/networkx/blob/improve-tree-iso-alg/networkx/algorithms/isomorphism/tree_isomorphism.py. Acceso: Enero 24, 2023.

[19] *Networkx: Tree Isomorphism*, https://github.com/networkx/networkx/blob/main/networkx/algorithms/isomorphism/tree_isomorphism.py. Acceso: Enero 24, 2023.

[20] *Rgraph6*, https://github.com/mbojan/rgraph6. Acceso: Enero 24, 2023.

[21] *MappleSoft: Graph6 Graph Format*, https://www.maplesoft.com/support/help/Maple/view.aspx?path=Formats%2FGraph6 Acceso: Enero 24, 2023.