



UNIVERSIDAD AUTÓNOMA DEL
ESTADO DE MORELOS



UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MORELOS
INSTITUTO DE INVESTIGACIÓN EN CIENCIAS BÁSICAS Y APLICADAS
CENTRO DE INVESTIGACIÓN EN CIENCIAS

Clasificación algorítmica en gráficas de tipo \mathbb{D}_n

T E S I S
para obtener el Grado de
LICENCIADO EN CIENCIAS
área terminal en ciencias computacionales y computación científica

Presenta
Rey David Gutiérrez Torres

Director de Tesis:
Dr. Daniel Rivera López

CUERNAVACA, MORELOS

21 de julio de 2022

Jurado revisor de tesis

Presidente Dra. Larissa Sbitneva,
Universidad Autónoma del Estado de Morelos

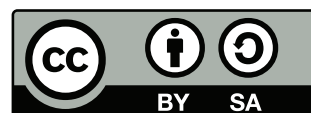
Secretario Dr. Rogelio Valdez Delgado,
Universidad Autónoma del Estado de Morelos

Vocal Dr. Antonio Daniel Rivera López,
Universidad Autónoma del Estado de Morelos

Suplente Dr. Jorge Hermosillo Valadez,
Universidad Autónoma del Estado de Morelos

Suplente Dr. Mario Alberto Abarca Sotelo,
Universidad Autónoma del Estado de Morelos

Esta obra está bajo una licencia Creative Commons
“Reconocimiento-CompartirIgual 4.0 Internacional”.



Contenido

Lista de símbolos	4
1 Introducción	1
1.1 FORMAS CUADRÁTICAS	1
1.2 FORMAS UNITARIAS	6
2 Antecedentes	15
2.1 EL MÉTODO DE LAS INFLACIONES	15
2.2 ENSAMBLAJE POR A-BLOQUES	24
3 COMPONENTES TRICONEXAS	33
3.1 DIVIDIR UNA GRÁFICA EN COMPONENTES TRICONEXAS	33
3.1.1 INTRODUCCIÓN	33
3.1.2 Gráficas, conectividad y búsqueda en profundidad	34
3.1.3 La idea de el algoritmo de triconectividad	41
3.2 El algoritmo para encontrar componentes triconexas	42
3.2.1 Encontrando pares de separación	43
3.2.2 Encontrando componentes de separación	50
4 Clasificación \mathbb{D}_n	56
4.1 Componentes triconexas en gráficas de tipo \mathbb{D}_n	56
4.1.1 Idea general de la clasificación algorítmica de \mathbb{D}_n	57
4.1.2 Clasificación algorítmica de \mathbb{D}_n	58
5 Componentes triconexas en un bigrafo \mathbb{D}_n.	63
5.1 Algoritmo para encontrar componentes triconexas de un bigrafo de tipo \mathbb{D}_n . .	63
Bibliografía	64

LISTA DE SIMOLOS

$x_i \text{---} x_j$	arista solida entre los vértices i y j
$x_i \cdots x_j$	arista punteada entre los vértices i y j
A^{-1}	matriz inversa de A
\cup	unión
\cap	intersección
$/$	negación
$=$	asignación
$=$	igualdad
\mathbb{Z}	conjunto de los números enteros
$\Theta()$	cota superior asintótica
\in	x está en
$A \subseteq B$	A es subconjunto de B
T_{ij}^-	inflación
T_{ij}^+	deflación
$E(G)$	conjunto de vértices de G
$V(G)$	conjunto de aristas de G
$A(v)$	lista de adyacencia de v
\mathbb{R}	conjunto de los números reales
\mathbb{N}	conjunto de los números naturales
\mathbb{A}_n	grafica de Dynkin \mathbb{A}_n
\mathbb{D}_n	grafica de Dynkin \mathbb{D}_n
$F_{m,m'}$	\mathbb{A} -bloque de $m + m'$ aristas
A^T	traspuesta de A
B_q	gráfica asociado a la forma q
q_G	forma unitaria asociada a el gráfica G

En este capítulo se presenta una introducción a las formas cuadráticas, sus representaciones y su clasificación.

1.1. FORMAS CUADRÁTICAS

Esta sección fue adaptada de [18]. Algunas demostraciones aquí omitidas pueden consultarse en dicha referencia.

Fijamos un conjunto de variables $\{x_1, x_2, \dots, x_n\}$. Un **monomio** es un producto de la forma $x_1^{e_1} \cdot x_2^{e_2} \cdots x_n^{e_n}$ donde cada e_i es un número natural. Un **término** se forma al multiplicar a un monomio por una constante, a la cual llamamos **coeficiente** del término. Un **polinomio** es una suma finita de términos.

Ejemplo: $7x^2y + 5xz^4 - y$ es un polinomio sobre las variables x , y y z con tres monomios: x^2y , xz^4 y y cuyos respectivos coeficientes son 2, 4, 1 respectivamente.

Una **forma cuadrática** es un polinomio en el que $q : \mathbb{R}^n \rightarrow \mathbb{R}$ (con $n > 0$) en el que cada monomio del mismo es una variable al cuadrado o la multiplicación de dos variables. Esto es equivalente a decir que q se puede expresar como

$$q(x_1, x_2, \dots, x_n) = \sum_{i=1}^n q_{ii}x_i^2 + \sum_{j=2}^n \sum_{i=1}^{j-1} q_{ij}x_i x_j$$

Los ejemplos más usuales aparecen al lado izquierdo del signo igual de las ecuaciones de las cónicas con centro en el origen

$$ax^2 + 2bxy + cy^2 = d$$

y de las superficies cuadráticas con centro en el origen

$$ax^2 + 2dxy + 2exz + by^2 + 2fyz + cz^2 = g$$

donde a, b, c, d, e, f y g son números reales. Este tipo de polinomios surgen de manera natural en diversas áreas de la ingeniería, procesamiento de señales, cinética, economía, geometría diferencial y estadística.

En algunos curso básicos de álgebra lineal se suele definir el concepto de forma cuadrática como una función que se puede escribir como

$$q(\vec{x}) = \frac{1}{2} \vec{x}^T A \vec{x}$$

para alguna matriz simétrica A . En realidad esta representación matricial es equivalente a nuestra definición con monomios; a continuación veremos como pasar de una representación a otra. Primero notemos que $\frac{1}{2} \vec{x}^T A \vec{x}$ se puede reescribir como sigue:

$$\begin{aligned} \frac{1}{2} \vec{x}^T A \vec{x} &= \frac{1}{2} [x_1, x_2, \dots, x_n] \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ &= \frac{1}{2} \left[\sum_{i=1}^n a_{i1} x_i, \sum_{i=1}^n a_{i2} x_i, \dots, \sum_{i=1}^n a_{in} x_i \right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ &= \frac{1}{2} \left[\sum_{i=1}^n a_{i1} x_i x_1 + \cdots + \sum_{i=1}^n a_{in} x_i x_n \right] \\ &= \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j \end{aligned}$$

Al desarrollar esto se puede ver que el coeficiente de $x_i x_j$ es $q_{ij} = \frac{1}{2} (a_{ij} + a_{ji})$ porque $x_i x_j = x_j x_i$, pero como dijimos que A es simétrica ($a_{ij} = a_{ji}$) entonces $q_{ij} = a_{ij}$. En particular cuando $i = j$ se tiene que el coeficiente de x_i^2 es $q_{ii} = \frac{1}{2} a_{ii}$. Por lo cuál tenemos la siguiente identidad:

$$\frac{1}{2} \vec{x}^T A \vec{x} = \sum_{i=1}^n \frac{1}{2} a_{ii} x_i^2 + \sum_{j=2}^n \sum_{i=1}^{j-1} a_{ij} x_i x_j \quad (1.1)$$

La matriz simétrica A de esta identidad se conoce como **matriz asociada a la forma cuadrática** q . Se denota por A_q , y según la ecuación 1.1 se puede calcular como sigue:

$$a_{ij} = \begin{cases} q_{ij} & \text{si } i < j \\ 2q_{ij} & \text{si } i = j \\ q_{ji} & \text{si } i > j \end{cases} \quad (1.2)$$

Por ejemplo, para $q(x, y, z) = 6x^2 + 2y^2 + 4z^2 - 2xy + 10xz - 6yz$

$$q(x, y, z) = \frac{1}{2} [x \ y \ z] \begin{bmatrix} 12 & -2 & 10 \\ -2 & 4 & -6 \\ 10 & -6 & 8 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

CAMBIO DE VARIABLE

Un **cambio de variable** es una ecuación de la forma

$$\vec{x} = P\vec{y} \quad \text{o bien} \quad \vec{y} = P^{-1}\vec{x}. \quad (1.3)$$

donde P es una matriz invertible y \vec{y} es un nuevo vector variable en \mathbb{R}^n . Si se aplica el cambio de variable 1.3 sobre una forma cuadrática $q(\vec{x}) = \frac{1}{2}\vec{x}^T A \vec{x}$ se obtiene una matriz cuadrática $q'(\vec{y})$ cuya matriz asociada es $P^T A P$:

$$\frac{1}{2}\vec{x}^T \vec{x} = \frac{1}{2}(P\vec{y})^T A (P\vec{y}) = \frac{1}{2}(\vec{y}^T P^T) A (P\vec{y}) = \frac{1}{2}\vec{y}^T (P^T A P) \vec{y}$$

Garantizamos que $P^T A P$ es una matriz simétrica (y por tanto que en verdad corresponde a otra forma cuadrática) porque

$$\begin{aligned} (P^T A P)^T &= P^T A^T (P^T)^T \\ &= P^T A^T P \\ &= P^T A P \end{aligned}$$

Mediante el cambio de variable $\vec{y} = P^{-1}\vec{x}$ tenemos que $q(\vec{x}) = q'(\vec{y})$ y diremos que q y q' son **equivalentes** mediante la matriz invertible P . Si denotamos con $L(\vec{x})$ a la transformación dada por, $L(\vec{x}) = P\vec{x}$ y tomamos $\vec{y} = L(\vec{x})$, entonces $q'(\vec{y}) = q'(L(\vec{x})) = (q' \circ L)(\vec{x})$, por lo tanto

$$q' = q \circ L$$

Ejemplo 1.1. Consideremos la forma cuadrática $x^2 - 5y^2 - 8xy$ con matriz asociada

$$A = \begin{bmatrix} 2 & -8 \\ -8 & 10 \end{bmatrix}.$$

y el cambio de variable definido por

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \\ -\frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} z \\ w \end{bmatrix} = \begin{bmatrix} \frac{2z+w}{\sqrt{5}} \\ \frac{-z+2w}{\sqrt{5}} \end{bmatrix}.$$

La matriz $P = \begin{bmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \\ -\frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{bmatrix}$ es invertible, y su inversa es

$$P^{-1} = \begin{bmatrix} \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{bmatrix} = P^T$$

Calculamos $P^T A P = \begin{bmatrix} 6 & 0 \\ 0 & -14 \end{bmatrix}$ para concluir que

$$x^2 - 5y^2 - 8xy = 3z^2 - 7w^2$$

Esto es lo que se hubiese obtenido de hacer las sustituciones

$$\begin{aligned} x &\leftarrow \frac{2z+w}{\sqrt{5}} \\ y &\leftarrow \frac{-z+2w}{\sqrt{5}} \end{aligned}$$

sobre la expresión $x^2 - 5y^2 - 8xy$.

Las cosas a destacar en este ejemplo son:

- P resultó ser una **matriz ortogonal**, en otras palabras, $P^{-1} = P^T$.
- A es una matriz **ortogonalmente diagonalizable**, en otras palabras, existe una matriz ortogonal P tal que $P^{-1} A P$ es una matriz diagonal.

Teorema 1.1. Toda matriz es ortogonalmente diagonalizable si y solo si es simétrica

El teorema 1.1 se dejara sin demostración. Lo importante es que, a partir de este teorema, y dado que las matrices asociadas a las formas cuadráticas son simétricas, se sigue el siguiente resultado:

Teorema 1.2. *(de los ejes principales). Toda forma cuadrática $q(\vec{x})$ es equivalente mediante una matriz ortogonal P a una forma cuadrática*

$$q'(\vec{y}) = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 \quad (1.4)$$

Demostración. La matriz A asociada a q es simétrica, luego por teorema anterior existe una matriz P tal que

$$P^{-1} A P = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$$

Haciendo $\vec{x} = P\vec{y}$ y se obtiene $q'(\vec{y}) = \frac{1}{2} \vec{y}^T (P^T A P) \vec{y} = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2$

□

Como se vera, esta representación es muy útil para clasificar formas cuadráticas. Diremos que una forma cuadrática q es **definida positiva** si $q(\vec{x}) > 0$, $\forall \vec{x} \neq \vec{0}$

Lema 1.3. *Si dos formas cuadráticas $q(\vec{x})$ y $q'(\vec{y})$ son equivalentes mediante el cambio de variable $\vec{y} = P\vec{x}$ y si una de ellas es definida positiva entonces la otra también lo es.*

Demostración. Supongamos que $q(\vec{x})$ es definida positiva y definimos la transformación lineal $L(\vec{x}) = P\vec{x}$, entonces $L(\vec{0}) = L(\vec{0} + \vec{0}) = L(\vec{0}) + L(\vec{0})$, de donde $L(\vec{0}) = \vec{0}$. Como P es una matriz invertible entonces $L^{-1}(\vec{x}) = P^{-1}\vec{x}$. En particular L debe ser inyectiva y por lo tanto $L(\vec{x}) = \vec{0}$ implica que $\vec{x} = \vec{0}$. Si q es definida positiva entonces $q(\vec{x}) = q'(\vec{y}) > 0$ para todo $\vec{y} \neq \vec{0}$ y se cumple que $\vec{x} \neq \vec{0} \Leftrightarrow \vec{y} = P^{-1}\vec{x} \neq \vec{0}$; por lo tanto q' es definida positiva, entonces aplicando el mismo razonamiento con $L(\vec{y}) = P^{-1}\vec{y}$ se llega a la conclusión de que q también es definida positiva. □

Teorema 1.4. *Sea q una forma cuadrática y supongamos que*

$$q(\vec{x}) = q'(y) = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2$$

entonces q es definida positiva si y solo si todos los $\lambda_i > 0$

Demostración. Por el lema anterior q es definida positiva si y solo si q' es definida positiva. Por contradicción, si $\lambda_i \leq 0$ entonces definimos $\vec{y}' = (y_1, y_2, \dots, y_n)$ donde todos los $y_k = 0$ excepto $y_i = 1$. Claramente $\vec{y}' \neq \vec{0}$ pero $q(\vec{y}') = \lambda_i \leq 0$; por lo tanto q no es definida positiva. □

1 Introducción

Como las formas cuadráticas se pueden representar mediante matrices entonces podemos decir que una matriz A es definida positiva si su forma cuadrática asociada $q(\vec{x}) = \frac{1}{2} \vec{x}^T A \vec{x}$ es definida positiva.

El siguiente teorema típico de los cursos de análisis numérico nos da un criterio computacionalmente eficiente para decidir cuando una matriz simétrica es definida positiva (sin demostración).

Teorema 1.5. *Una matriz A es definida positiva si y solo si tiene **factorización de Cholesky**, es decir, se puede escribir como $A = R^T R$ donde R es una matriz triangular superior con entradas positivas.*

El siguiente teorema es el criterio de Sylvester para que una forma cuadrática sea positiva definida.

Teorema 1.6. *Sean V un espacio vectorial real de dimensión n , $q \in \mathcal{Q}(V)$, B una base de V . Denotamos por q a la matriz asociada a la forma cuadrática q respecto a la base B . Entonces las siguientes condiciones son equivalentes:*

1. $q > 0$, esto es, $q(x) > 0$ para todo $x \in V \setminus \{0\}$.
2. todos los menores principales de qB son positivos:

$$\forall I \subseteq 1, \dots, n \quad \delta_I(qB) > 0.$$

3. todos los menores de la matriz qB son positivos: para todo $k \in 1, \dots, n$

$$\Delta_k(qB) > 0.$$

1.2. FORMAS UNITARIAS

Esta sección fue adaptada de [22] y [9]. Las **formas cuadráticas enteras** son un caso especial de las formas cuadráticas donde todos los coeficientes q_{ij} son todos números enteros. Si además exigimos que $q_{ii} = 1$ para $i = 1, 2, \dots, n$ entonces las llamamos **formas cuadráticas unitarias**. En el resto de este trabajo solo se trabaja con formas cuadráticas enteras que son unitarias por lo que las llamaremos **formas unitarias**.

CAMBIO DE VARIABLE ENTERO

Una **matriz entera** M es una matriz tal que todas sus entradas son números enteros, y es **\mathbb{Z} -invertible** si además su matriz inversa M^{-1} también es una matriz entera. Un **cambio de variable entero** es un cambio de variable $\vec{y} = P\vec{x}$ donde P es una matriz \mathbb{Z} -invertible.

Los cambios de variables enteros tienen la propiedad de que transforman formas cuadráticas enteras en formas cuadráticas enteras (demostración: sean A y P matrices enteras, entonces $P^T A P$ es una matriz entera). Cuando $q(\vec{x}) = q'(\vec{y})$ mediante el cambio de variable entero $\vec{y} = P\vec{x}$ diremos que q y q' son \mathbb{Z} -**equivalentes** mediante la matriz \mathbb{Z} -invertible P .

Bi-gráficas asociadas a formas unitarias

Si q es una forma unitaria entonces le asociaremos una bigráfica B_q construida de la siguiente manera:

- Existe un vértice x_i para cada variable x_i
- Si $q_{ij} > 0$ entonces trazamos aristas punteadas entre los vértices x_i y x_j con peso q_{ij} .

$$x_i \overset{q_{ij}}{\cdots} x_j$$

- Si $q_{ij} < 0$ entonces trazamos aristas sólidas entre los vértices x_i y x_j con peso $|q_{ij}|$.

$$x_i \overset{|q_{ij}|}{\text{---}} x_j$$

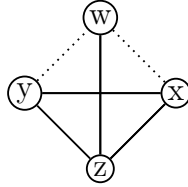


Figura 1.1: Bigráfica asociada a $2wx - 2yz - xz - xy - wz + wy$

Este proceso se puede revertir y asociar a toda bigráfica G (posiblemente con aristas punteadas) una forma unitaria que denotaremos por q_G . Ahora, toda la información de q_G está codificada en G : La existencia de un vértice x nos dice que la forma cuadrática está definida sobre alguna variable x y que contiene el término x^2 (porque q es una forma cuadrática unitaria). El coeficiente del monomio xy es $c = a_p - a_s$ donde a_p es la cantidad de aristas punteadas entre x y y , y a_s es la cantidad de aristas sólidas entre estos mismos vértices. Más aún cabe recalcar que nosotros estamos descartando gráficas con lazos, por lo que cualquier gráfica en verdad define a una forma cuadrática unitaria.

Las gráficas de Dynkin se presentan en la figura 1.2. Hay que resaltar que las gráficas \mathbb{D}_n están definidas para $n \geq 4$ mientras que las gráficas \mathbb{E}_n solo se definen para $n = 6, 7, 8$. Lo importante de estas gráficas radica en que permite dar una caracterización elegante de las formas unitarias que son definidas positivas tal como se explica a continuación.

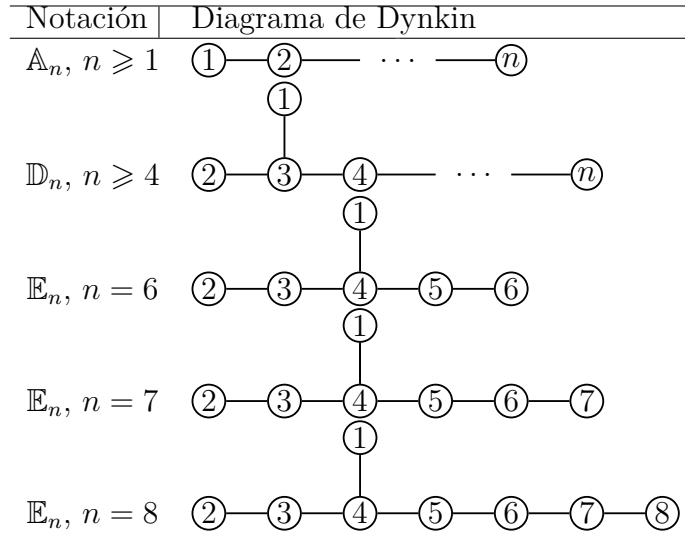


Figura 1.2: Gráficas de Dynkin. El subíndice n indica la cantidad de vértices que tiene la bigráfica.

Toda forma unitaria q manejada en este trabajo es definida positiva.

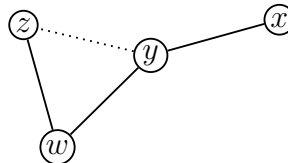
Teorema 1.7. *Toda forma unitaria B_q es conexa, así q es \mathbb{Z} -equivalente a una forma unitaria cuya bigráfica asociada es un diagrama de Dynkin.*

Para poder hacer la demostración hay que comprender el teorema. El teorema dice que la forma unitaria y conexa $q(\vec{x}) = \frac{1}{2} \vec{x}^T A \vec{x}$ es definida positiva si, y sólo si, se puede llevar, mediante un cambio de variable entero $\vec{y} = P \vec{x}$, a la forma $q'(\vec{y}) = \frac{1}{2} \vec{y}^T (P^T A P) \vec{y}$ donde $B_{q'}$, tiene la propiedad de que cada una de sus componentes conexas es una gráfica de Dynkin. A dicha gráfica $B_{q'}$ se le llama el **tipo Dynkin** de q .

Ejemplo 1.2. [1] *La forma unitaria*

$$q(w, x, y, z) = x^2 + y^2 + z^2 + w^2 - xy + yz - yw - zw \quad (1.5)$$

tiene asociada la bigráfica



y su matriz asociada

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & 1 & -1 \\ 0 & 1 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}$$

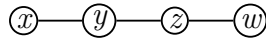
$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ con inversa } P^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} P^T A P &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & 1 & -1 \\ 0 & 1 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 1 & -1 & 0 \\ 0 & 1 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \end{aligned}$$

Entonces podemos concluir que q es \mathbb{Z} -equivalente a la forma unitaria

$$q'(x, y, z, w) = x^2 + y^2 + z^2 + w^2 - xy - yz - zw$$

con gráfica asociada



Esta gráfica es isomorfa a \mathbb{A}_4 , por lo tanto ese el tipo Dynkin es \mathbb{A}_4 .

Para demostrar el teorema 1.7 lo dividiremos en dos partes:

1. Demostramos que las gráficas de Dynkin son las únicas gráficas conexas de aristas sólidas que definen formas unitarias que son definidas positivas (las aristas multiples las consideraremos como aristas con peso, ver corolario 1.10) .
2. Demostramos que siempre es posible hacer cambios de variable enteros de tal manera que la gráfica resultante no contenga aristas solidas.

1 Introducción

La última afirmación será demostrada en 2.1

Lema 1.8. *Los diagramas de Dynkin son las únicas gráficas conexas de aristas sólidas que tienen asociadas formas unitarias definidas positivas.*

Para comenzar necesitamos convencernos en verdad de que los diagramas de Dynkin definen formas unitarias definidas positivas. Comencemos con las gráficas de tipo \mathbb{A}_n :

$$x_1 - x_2 - \cdots - x_n$$

Consideremos la identidad:

$$\frac{1}{2} (x_i - x_j)^2 = \frac{1}{2} x_i^2 - x_i x_j + \frac{1}{2} x_j^2$$

Podemos reescribir la forma unitaria asociada a \mathbb{A}_n como

$$\begin{aligned} q_{\mathbb{A}_n}(\vec{x}) &= \sum_{i=1}^n x_i^2 - \sum_{i=1}^{n-1} x_i x_{i+1} \\ &= \frac{1}{2} x_1^2 + \sum_{i=1}^{n-1} \left(\frac{1}{2} x_i^2 + \frac{1}{2} x_{i+1}^2 \right) + \frac{1}{2} x_n^2 + \sum_{i=1}^{n-1} (-x_i x_{i+1}) \\ &= \frac{1}{2} x_1^2 + \sum_{i=1}^{n-1} \left(\frac{1}{2} x_i^2 - x_i x_{i+1} + \frac{1}{2} x_{i+1}^2 \right) + \frac{1}{2} x_n^2 \\ &= \frac{1}{2} x_i^2 + \sum_{i=1}^{n-1} \frac{1}{2} (x_i - x_{i+1})^2 + \frac{1}{2} x_n^2 \end{aligned}$$

Por el teorema 1.4 se concluye que $q_{\mathbb{A}_n}$ es definida positiva. La demostración para la forma unitaria $q_{\mathbb{D}_n}$ es similar solo que en este caso se usa la identidad:

$$\frac{1}{2} \left[(x_3 - x_2 - x_1)^2 + (x_2 - x_1)^2 \right] = x_1^2 + x_2^2 + \frac{1}{2} x_3^2 - x_1 x_3 - x_2 x_3$$

de tal forma que

$$\begin{aligned} q_{\mathbb{D}_n}(\vec{x}) &= \sum_{i=1}^n x_i^2 - x_1 x_3 - \sum_{i=2}^{n-1} x_i x_{i+1} \\ &= \frac{1}{2} \left[(x_3 - x_2 - x_1)^2 + (x_2 - x_1)^2 + \sum_{i=3}^{n-1} (x_i - x_{i+1})^2 + x_n^2 \right] \end{aligned}$$

Para las gráficas \mathbb{E}_6 , \mathbb{E}_7 y \mathbb{E}_8 usaremos el siguiente razonamiento:
 Supongamos que A_Δ es la matriz asociada a la gráfica Δ (con $\Delta = \mathbb{E}_6, \mathbb{E}_7, \mathbb{E}_8$); si existe una matriz R_Δ triangular superior, con entradas positivas en la diagonal principal tal que $A_\Delta = R_\Delta^T R_\Delta$ entonces por teorema 1.5 se sigue que Δ define una forma unitaria definida positiva.

En efecto tenemos que:

$$\begin{aligned}
 A_{\mathbb{E}_6} &= \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ -1 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \\
 R_{\mathbb{E}_6} &= \begin{bmatrix} \sqrt{2} & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & \text{sqrt}(2) & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} & 0 & 0 \\ 0 & 0 & 0 & \frac{\sqrt{5}}{\sqrt{6}} & -\frac{\sqrt{6}}{\sqrt{5}} & 0 \\ 0 & 0 & 0 & 0 & \frac{2}{\sqrt{5}} & -\frac{\sqrt{5}}{2} \\ 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{3}}{2} \end{bmatrix} \\
 A_{\mathbb{E}_7} &= \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \\
 R_{\mathbb{E}_7} &= \begin{bmatrix} \sqrt{2} & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & \text{sqrt}(2) & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\sqrt{5}}{\sqrt{6}} & -\frac{\sqrt{6}}{\sqrt{5}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{2}{\sqrt{5}} & -\frac{\sqrt{5}}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{3}}{2} & -\frac{2}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{\sqrt{3}} \end{bmatrix} \\
 A_{\mathbb{E}_8} &= \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}
 \end{aligned}$$

$$R_{\mathbb{E}_8} = \begin{bmatrix} \sqrt{2} & 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{3}}{\sqrt{2}} & -\frac{\sqrt{2}}{\sqrt{3}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\sqrt{5}}{\sqrt{6}} & -\frac{\sqrt{6}}{\sqrt{5}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{2}{\sqrt{5}} & -\frac{\sqrt{5}}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{3}}{2} & -\frac{2}{\sqrt{3}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{\sqrt{3}} & -\frac{\sqrt{3}}{\sqrt{2}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}$$

Aun no hemos terminado la demostración del lema 1.8 . Falta demostrar que los diagramas de Dynkin son las únicas bigráficas de aristas sólidas que definen formas unitarias definidas positivas. Comenzaremos con el siguiente lema, el cual muestra que toda bigráfica que tenga aristas con mayor peso no corresponde a ninguna forma unitaria definida positiva.

Lema 1.9. *Si la forma unitaria $q(\vec{x})$ es definida positiva entonces $q_{ij} \in \{-1, 0, 1\}$ para todo $1 \leq i \leq j \leq n$.*

Demostración. Denotemos con $\vec{e}_k = (d_1, d_2, \dots, d_n)$ al vector dado por $d_k = 1$ y $d_i = 0$ para toda $i \neq k$. Si $q_{ij} \geq 2$ entonces $q(\vec{e}_i + \vec{e}_j) = 2 - q_{ij} \leq 0$ pero $\vec{e}_i - \vec{e}_j \neq \vec{0}$, $i \neq j$. Si $q_{ij} \leq -2$ entonces $q(\vec{e}_i + \vec{e}_j) = 2 - q_{ij} \geq 4$ pero $\vec{e}_i - \vec{e}_j \neq \vec{0}$. De lo anterior, ya que q es definida positiva, entonces $|q_{ij}| \leq 2$ para todo $1 \leq i < j \leq n$. \square

Cada $|q_{ij}|$ nos dice el peso de las aristas que hay entre los vértices x_i y x_j de la gráfica \mathbf{B}_q ; por lo tanto si $|q_{ij}| \leq 1$ tenemos que \mathbf{B}_q es una bigráfica simple. Es decir que el lema 1.9 se puede reescribir como sigue:

Corolario 1.10. *Si q es una forma unitaria positiva entonces necesariamente \mathbf{B}_q es una bigráfica simple.*

Con base en este corolario diremos que una forma unitaria q es **simple** si su gráfica asociada \mathbf{B}_q es una bigráfica simple.

Ahora que hemos descartado a las bigráficas de aristas con peso mayor que uno el resto de la demostración es como sigue:

1. Demostraremos que toda gráfica que contenga a un diagrama Euclidiano (Figura 1.3) no define a una forma unitaria definida positiva
2. Demostraremos que toda gráfica conexa que no sea un diagrama de Dynkin necesariamente contiene una sub-gráfica Euclidiana (bigráfica con sólo aristas sólidas).

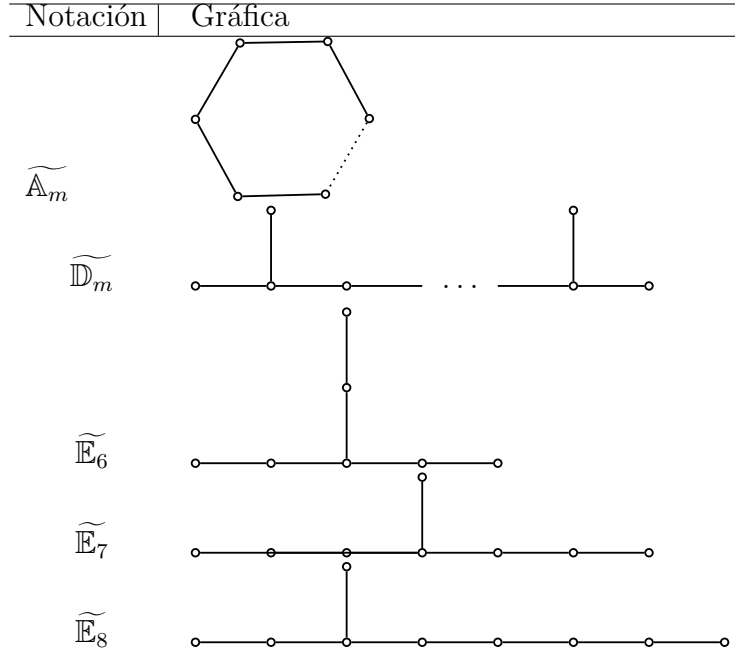
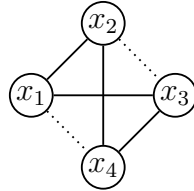


Figura 1.3: Diagramas Euclidianos. La cantidad de vértices que tiene cada gráfica es $n = m + 1$.

Hasta ahora hemos asociado vértices con variables (\otimes con x). Ahora a cada variable x_i le asignamos el vértice i , convenimos que le asignamos un orden a las variables y que el vértice i representa el lugar de la variable x_i en el orden m dado.



representa la evaluación $x = 2, y = 1, z = 2, w = -1$

Mostraremos que si \mathbf{B}_q contiene una subgráfica Euclidiana entonces existe un vector $\vec{x} \neq 0$ tal que $q(\vec{x}) = 0$, mostrando así que $q(\vec{x})$ no es definida positiva. Un simple cálculo nos muestra que esta evaluación produce un vector $\vec{x} \neq \vec{0}$ tal que $q(\vec{x}) = 0$.

Solamente falta demostrar que toda gráfica que no sea de Dynkin necesariamente contiene a una subgráfica Euclidiana. Sea G una gráfica conexa de n vértices distinta de $\mathbb{A}_n, \mathbb{B}_n, \mathbb{D}_n, \mathbb{E}_6, \mathbb{E}_7$ y \mathbb{E}_8 . Si G no es un árbol entonces G contiene un ciclo; es decir contiene una subgráfica de $\widetilde{\mathbb{A}}_m$ para algún $m < n$. Si G es un árbol, y dado que $G \neq \mathbb{A}_n$, entonces existe al menos un vértice v de grado 3 o más. Claramente v pertenece a una subgráfica de \mathbb{D}_r para algún $r \leq n$, pero habíamos supuesto que $G \neq \mathbb{D}_n$ por lo tanto hay tres casos:

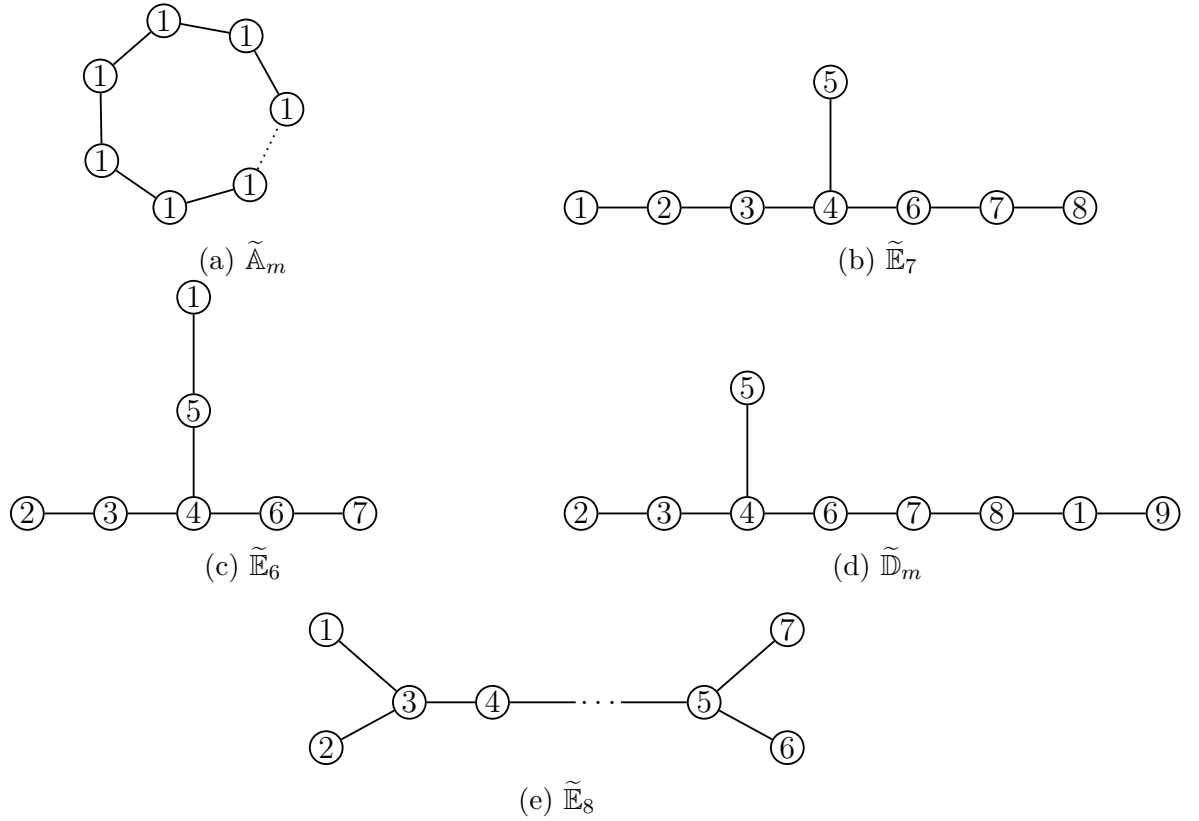


Figura 1.4: Formas unitarias asociadas a las gráficas Euclidianas.

1. Si v tiene grado estrictamente mayor a 3 entonces G contiene a \tilde{D}_4
2. Si G contiene otro vértice w de grado 3 o más, entonces G contiene a \tilde{D}_m para algún $m < n$
3. Si todos los demás vértices tienen grado menor a 3 entonces G debe contener a \mathbb{E}_6 como subgráfica.

Ahora ya que habíamos supuesto que $G \neq \mathbb{E}_6$, por tanto $n > 6$. En este caso G debe contener a $\tilde{\mathbb{E}}_6$ o \mathbb{E}_7 . Si tenemos que $G \neq \mathbb{E}_7$ entonces $n > 8$, de donde obtenemos que G contiene a $\tilde{\mathbb{E}}_7$ o \mathbb{E}_8 , pero si $G \neq \mathbb{E}_8$ entonces $n > 8$ y por lo tanto G contiene a $\tilde{\mathbb{E}}_8$.

Resumiendo, las gráficas de Dynkin definen formas unitarias positivas, y cualquier otra gráfica conexa y de aristas sólidas que no sea un diagrama de Dynkin necesariamente contiene una gráfica Euclidiana que la vuelve no positiva: por lo tanto las gráficas de Dynkin son las únicas gráficas conexas de aristas sólidas que definen formas unitarias definidas positivas. Esto concluye la demostración del lema 1.8.

En la sección 2.1 se dará término a la demostración del teorema 1.7

Antecedentes

Este capítulo está dedicado a presentar los resultados principales de [9] & [4] que permiten caracterizar a las formas \mathbb{A}_n

2.1. EL MÉTODO DE LAS INFLACIONES

La demostración del teorema 1.7 aparece en [9] y hace uso implícito de un algoritmo que describiremos aquí.

La idea intuitiva es pasar mediante cambios de variable enteros de la forma unitaria conexa q a la forma unitaria q' , donde $\mathbf{B}_{q'}$ no contiene aristas punteadas. Si q es definida positiva entonces por lemma 1.3 q' también es definida positiva; luego como q' solamente contiene aristas sólidas entonces por lema 1.8 se sigue que cada una de sus componentes conexas debe ser una gráfica de Dynkin (figura 1.2). El procedimiento para encontrar dicha forma cuadrática q' es el *método de las inflaciones* y es el tema de esta sección.

INFLACIONES Y DEFLACIONES

Tomemos matrices elementales E_{sr}^d , cuya transformación lineal

$$T_{sr}^d(A) = E_{sr}^d A$$

equivale a sumar d veces el renglón s de A al renglón r de A . Dado que la matriz inversa de E_{sr}^d es E_{sr}^{-d} , notamos que E_{sr}^d es \mathbb{Z} -invertible si y solamente si d es un número entero.

Consideremos el cambio de variable entero $\vec{y} = E_{sr}^d \vec{x}$, entonces

$$\begin{bmatrix} y_1 \\ \vdots \\ y_r \\ \vdots \\ y_n \end{bmatrix} = E_{sr}^d \begin{bmatrix} x_1 \\ \vdots \\ x_r \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_r + dx_s \\ \vdots \\ x_n \end{bmatrix}$$

y por tanto podemos concluir que T_{sr}^d como cambio de variable tiene el mismo efecto que sustituir $x_r \leftarrow x_r + dx_s$.

¿Qué efecto tiene T_{sr}^d si sobre una forma unitaria q ? Supongamos que q tiene matriz asociada $A = [a_{ij}]$ y sea

$$q'(\vec{x}) = \frac{1}{2} (E_{rs}^d)^T A E_{sr}^d$$

Dado que E_{sr}^d es la matriz identidad salvo por la d en la posición (s, r) -ésima, entonces $(E_{sr}^d)^T = E_{sr}^d$. Por lo tanto $(E_{sr}^d)^T$ tiene el efecto de sumar d veces el renglón r al renglón s . Por otra parte E_{sr}^d multiplicado por la derecha tiene el efecto de sumar d veces la columna r a la columna s , por lo tanto la matriz $C = (E_{sr}^d)^T A E_{sr}^d$ está dada por

$$c_{ij} = \begin{cases} a_{ij} & \text{si } i \neq s \text{ y } j \neq s \\ a_{sj} + da_{rj} & \text{si } i = s \text{ pero } j \neq s \\ a_{is} + da_{ir} & \text{si } j = s \text{ pero } i \neq s \\ d^2 a_{rr} + d(a_{sr} + a_{rs}) + a_{ss} & \text{si } i = s \text{ y } j = s \end{cases} \quad (2.1)$$

E_{rs}^d define un cambio de variable entero, por lo que C es la matriz asociada a alguna forma cuadrática entera, aunque no necesariamente unitaria. Para que $q'(\vec{x}) = \frac{1}{2} \vec{x}^T C \vec{x}$ sea una forma unitaria es necesario que $c_{ss} = 2$, o lo que es lo mismo (con $a_{rr} = a_{ss} = 2$ y $a_{rs} = a_{sr}$):

$$2d^2 + 2da_{rs} + 2 = 2.$$

Esta ecuación tiene dos soluciones: $d = 0$ y $d = -a_{rs}$. Estas soluciones se resumen en el siguiente lema:

Lema 2.1. *Si q es una forma unitaria entonces $q \circ T_{rs}^d$ es una forma unitaria si y solo si $q_{rs} = q_{sr} = -d$.*

Corolario 2.2. *Si q es una forma unitaria simple entonces $q \circ T_{rs}^d$ es una forma unitaria para un $d \neq 0$ en cualquiera de estos dos casos y solamente estos dos:*

1. $d = 1$ y \mathbf{B}_q contiene una arista punteada entre los vértices x_r y x_s .
2. $d = -1$ y \mathbf{B}_q contiene a una arista sólida entre los vertices x_r y x_s .

Este corolario no garantiza que $q \circ T_{rs}^d$ sea simple; en este caso por corolario 1.9 vemos que esto ocurre solamente cuando q (que es Z -equivalente a $q \circ T_{rs}^d$) no es definida positiva.

Con base al corolario anterior se justifica que definamos la transformación de **inflación** como

$$T_{rs}^- = T_{rs}^{-1}$$

y la transformación de la **deflación** como

$$T_{rs}^+ = T_{rs}^1$$

El corolario anterior dice que la inflación T_{rs}^- se puede aplicar solamente cuando \mathbf{B}_q contiene a la arista punteada $x_r \cdots x_s$ mientras que T_{rs}^+ solamente cuando \mathbf{B}_q contiene a la arista sólida $x_r - x_s$. A partir de la ecuación (2.1), y del hecho de que la matriz es simétrica. Podemos interpretar las inflaciones y deflaciones como un algoritmo de reconexión sobre la gráfica \mathbf{B}_q (bajo las condiciones del corolario anterior) que consiste de cuatro pasos:

1. *Duplicar*: Generar una copia de las aristas que inciden en x_i excepto la arista que conecta x_i con x_j .
2. *Invertir*: En caso de una inflación reemplazar $x_i \cdots x_j$ por $x_i - x_j$ y para cada arista duplicada del paso anterior intercambiar las aristas punteadas con sólidas y viceversa. En caso de una deflación simplemente reemplazar $x_i - x_j$ por $x_i \cdots x_j$.
3. *Arrastrar*: Cada una de las aristas duplicadas se desconecta de su x_i y se reconecta en x_j .
4. *Simplificar*: Si existen dos aristas $x_i \cdots x_j$ y $x_i - x_j$ incidentes en x_r y x_j , se borran.

Por ejemplo, consideremos la forma cuadrática q de la ecuación (1.5) y ordenemos a las variables como (w, x, y, z) . Entonces para aplicar T_{23}^- podemos calcular $(E_{23}^{-1})^T A E_{23}^{-1}$ donde A es la matriz asociada a q .

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & 1 & -1 \\ 0 & 1 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & -1 & 1 & 0 \\ -1 & 2 & -1 & -1 \\ 1 & -1 & 2 & 0 \\ 0 & -1 & 0 & 2 \end{bmatrix}$$

La matriz resultante nos describe la misma forma cuadrática que se hubiera obtenido reemplazando $y \rightarrow y - z$; en efecto:

$$x^2 + (y - z)^2 + z^2 + w^2 - x(y - z) + (y - z)z - (y - z)w - zw = x^2 + y^2 + z^2 + w^2 - xy + xz - yz - yw$$

También se puede hacer el cambio de variable de manera gráfica como se ilustra a continuación en la figura (2.1)

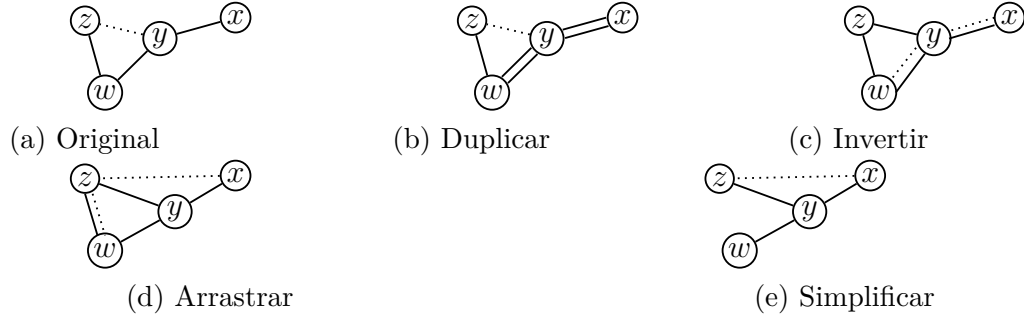


Figura 2.1: Ejemplo de una inflación.

DESCRIPCIÓN Y JUSTIFICACIÓN

El método que aparece en [9] consiste en aplicar sucesivamente inflaciones como se indica en el algoritmo 2.1 tanto como sea posible, es decir, hasta que no haya más aristas punteadas.

Algoritmo 2.1: Inflaciones(q)

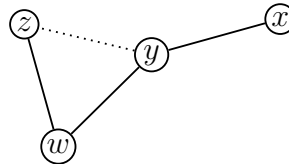
```

1 mientras exista una arista punteada  $x_r \dots x_s$  con  $r \leq s$  en  $B_q$ :
2    $q = q \circ T_{rs}^-$ 
3   si para algún coeficiente  $q_{ij}$  se tiene  $|q_{ij}| \geq 1$  :
4     devolver no es positiva
5 para cada componente conexa  $\mathcal{C}$  de  $B_q$ 
6    $n =$  número de vértices que contiene  $\mathcal{C}$ 
7   si  $\mathcal{C}$  no es isomorfa a  $A_n, D_n, E_6, E_7, E_8$ 
8     devolver
9     no es positiva
10 devolver es positiva y es de tipo Dynkin  $B_q$ 

```

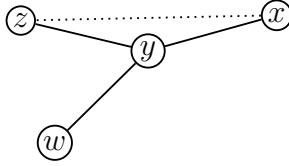
Veamos como funciona este algoritmo sobre la forma cuadrática de la ecuacion (1.5).

1. Comenzamos con la forma unitaria



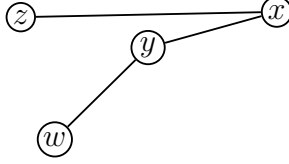
$$x^2 + y^2 + z^2 + w^2 - xy + yz - yw - zw$$

2. Tenemos a la arista $y \dots z$ y aplicamos T_{23}^- para obtener



$$x^2 + y^2 + z^2 + w^2 - xy + xz - yz - wy$$

3. Tenemos a la arista $x \cdots \cdots z$ y aplicamos T_{13}^- para obtener



$$x^2 + y^2 + z^2 + w^2 - xy - xz - wy$$

4. Esta gráfica es isomorfa a A_4 , por lo tanto es el tipo Dynkin de la forma cuadrática (1.5).

Para terminar la demostración del teorema 1.7 es necesario mostrar que si q es una forma unitaria definida positiva entonces este algoritmo termina. Esto lo podemos resumir de la siguiente manera.

- Decimos que $\vec{x} \in Z^n$ es una **raíz** de la forma unitaria q si $q(\vec{x}) = 1$.
- Primero mostramos que toda forma cuadrática se puede poner en la forma

$$\begin{aligned} q(\vec{x}) &= \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 \\ y_i &= x_i + \sum_{j=i+1}^n \rho_{ij} x_j \end{aligned}$$

- Vamos a suponer que q es una forma unitaria positiva, y usando la expresión anterior mostraremos que q tiene una cantidad finita de raíces. Más específicamente mostraremos que cada x_i está acotada en el intervalo $[-\sigma, \sigma]$ para alguna constante σ_i .
- Encontraremos cierto subconjunto de raíces que crece con cada iteración del algoritmo 2.1. Como el conjunto de raíces es finito, la cantidad de iteraciones también.

La **reducción de Lagrange** es un método que nos permite llevar toda forma cuadrática a

la forma (1.1). Primero necesitamos recordar la fórmula de **completar cuadrados**:

$$\begin{aligned}
 ax^2 + bx + c &= a \left[x^2 + \frac{b}{a}x \right] + c \\
 &= a \left[x^2 + \frac{b}{a}x + \left(\frac{b}{2a} \right)^2 - \left(\frac{b}{2a} \right)^2 \right] + c \\
 &= a \left(x + \frac{b}{2a} \right)^2 + \left(c - \frac{b^2}{4a} \right)
 \end{aligned} \tag{2.2}$$

Sea q una forma unitaria sobre las variables x_k, x_{k+1}, \dots, x_n . Para escribir $q(x_k, \dots, x_n)$ en la forma (1.1) se hace lo siguiente:

1. Si $q = q_{nn}x_n^2$ es una forma cuadrática en una variable entonces se definen $\lambda_n = q_{nn} = x_n$ y terminamos (en caso contrario seguimos).
2. Sea $\ell = k + 1$. Tomamos todos los términos que contienen la variable x_k :

$$q = q_{kk}x_k^2 + \sum_{j=\ell}^n q_{kj}x_kx_j + r$$

donde $r = \sum_{i=\ell}^n q_{ii}x_i^2 + \sum_{j=\ell+1}^n \sum_{i=\ell}^{j-1} q_{ij}x_ix_j$

3. Factorizamos $q_{k\ell}x_k$

$$q = q_{kk}x_k^2 + q_{k\ell}x_k \left(\sum_{j=\ell}^n \frac{q_{kj}}{q_{k\ell}}x_j \right) + r$$

4. Completando el cuadrado substituyendo en la ecuación (2.2) los valores de $a = q_{kk}$, $b = q_{k\ell} \left(\sum_{j=\ell}^n \frac{q_{kj}}{q_{k\ell}}x_j \right)$ y $c = r$

$$q = q_{kk} \left(x + \sum_{j=\ell}^n \frac{q_{kj}}{2q_{kk}}x_j \right)^2 - \frac{q_{k\ell}^2}{4q_{kk}} \left(\sum_{j=\ell}^n \frac{q_{kj}}{q_{k\ell}}x_j \right)^2 + r$$

5. Se definen

$$\begin{aligned}
 \lambda_k &= q_{kk} \\
 y_k &= x_k + \sum_{j=\ell}^n \frac{q_{kj}}{2q_{kk}}x_j \\
 q' &= -\frac{q_{k\ell}^2}{4q_{kk}} \left(\sum_{j=\ell}^n \frac{q_{kj}}{q_{k\ell}}x_j \right)^2 + r
 \end{aligned}$$

Desarrollando el cuadrado en q' vemos que q' es una forma cuadrática sobre las variables $(x_\ell, x_{\ell+1}, \dots, x_n)$ (una variable menos que q) y además

$$q = \lambda_1 y_1^2 + q'$$

6. Repetimos el mismo procedimiento para q'

Este método es una demostración constructiva (e inductiva) de el siguiente lema:

Lema 2.3. *Toda forma cuadrática $q(\vec{x})$ se puede reescribir como*

$$\begin{aligned} q(\vec{x}) &= \lambda_1 y_1^2 + \lambda_2 y_2^2 + \dots + \lambda_n y_n^2 \\ y_i &= x_i + \sum_{j=i+1}^n \rho_{ij} x_j \end{aligned} \quad (2.3)$$

Ejemplo 2.1. *Sea $q(x_1, x_2, x_3) = x_1^2 + 2x_2^2 - 7x_3^2 - 4x_1x_2 + 8x_1x_3$ la primera iteración hace lo siguiente:*

$$\begin{aligned} q(x_1, x_2, x_3) &= x_1^2 - 4x_1x_2 + 8x_1x_3 + \underbrace{2x_2^2 - 7x_3^2}_r \\ &= \underbrace{1}_a x_1^2 + x_1 \underbrace{(-4)(x_2 - 2x_3)}_b + \underbrace{2x_2^2 - 7x_3^2}_r \\ &= \underbrace{1}_{\lambda_1} \underbrace{(x_1 - 2x_2 + 4x_3)^2}_{y_1} - 4 \underbrace{(x_2 - 2x_3)^2 + 2x_2^2 - 7x_3^2}_{q'} \\ &= \lambda_1 y_1^2 - \underbrace{2x_2^2 - 23x_3^2 + 16x_2x_3}_{q'} \end{aligned}$$

La segunda iteración funciona sobre $q'(x_2, x_3) = -2x_2^2 - 23x_3^2 + 16x_2x_3$:

$$\begin{aligned} q'(x_2, x_3) &= \underbrace{-2}_a x_2^2 + x_2 \underbrace{(16x_3)}_b - \underbrace{23x_3^2}_{r'} \\ &= \underbrace{-2}_{\lambda_2} \underbrace{(x_2 + 4x_3)^2}_{y_2} + \underbrace{9x_3^2}_{q''} \\ &= \lambda_2 y_2^2 + \underbrace{9x_3^2}_{q''} \end{aligned}$$

Finalmente en la tercera iteración opera sobre $q''(x_3) = 9x_3^2$, que es una forma cuadrática en una sola variable, por tanto $\lambda_3 = 9$ y $y_3 = x_3$. Así hemos reescrito a q como

$$q(x_1, x_2, x_3) = y_1^2 - 2y_2^2 + 9y_3^2$$

2 Antecedentes

donde

$$\begin{array}{rcl} y_1 & = & x_1 - 2x_2 + 4x_3 \\ y_2 & = & x_2 - 4x_3 \\ y_3 & = & x_3 \end{array}$$

Supongamos que q es una forma unitaria definida positiva y que \vec{x} es una raíz de q . Usando la reducción de Lagrange escribimos

$$\begin{aligned} q(\vec{x}) &= \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 \\ y_i &= x_i + \sum_{j=i+1}^n \rho_{ij} x_j \end{aligned}$$

Como q es definida positiva entonces cada $\lambda_i > 0$. Más aún, como $\lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 = 1$ (porque habíamos supuesto que \vec{x} es una raíz) entonces $\lambda_i y_i^2 \leq 1$ para cada i .

Como $y_n = x_n$ entonces $\lambda_n x_n^2 \leq 1$, de donde $|x_n| \leq \frac{1}{\sqrt{\lambda_n}}$. Definbiendo $\sigma_n = \frac{1}{\sqrt{\lambda_n}}$ se ha mostrado que $-\sigma_n \leq x_n \leq \sigma_n$ está acotada por alguna constante σ_k . En efecto, tenemos que

$$\lambda_k y_k^2 = \lambda_k \left(x_k + \sum_{j=k+1}^n \rho_{kj} x_j \right)^2 \leq 1$$

de donde

$$\left| x_k + \sum_{j=k+1}^n \rho_{kj} x_j \right| \leq \frac{1}{\sqrt{\lambda_k}}$$

Para continuar es necesaria la siguiente desigualdad:

$$\left| \sum_{i=1}^m a_i \right| \geq |a_1| - \sum_{i=2}^m |a_i| \tag{2.4}$$

para cualesquiera números reales a_1, a_2, \dots, a_m . Esta desigualdad la demostramos por inducción (el caso base $m = 2$ establece que $|a_1 + a_2| \geq |a_1| - |a_2|$ y se comprueba por casos; para el paso inductivo escribimos $|\sum_{i=1}^m a_i| \geq |\sum_{i=1}^{m-1} a_i| - |a_m|$ y aplicamos la inducción). Usando la desigualdad (2.4) deducimos que

$$|x_k| - \sum_{j=k+1}^n |\rho_{kj} x_j| \leq \left| x_k + \sum_{j=k+1}^n \rho_{kj} x_j \right| \leq \frac{1}{\lambda_k}$$

de donde despejando $|x_k|$ y usando que $x_j \leq \sigma_j$ con $\sigma_j \geq 0$, obtenemos

$$|x_k| \leq \frac{1}{\sqrt{\lambda_k}} + \sum_{j=k+1}^n |\rho_{kj} x_j| \leq \frac{1}{\sqrt{\lambda_k}} + \sum_{j=k+1}^n |\rho_{kj}| \sigma_j$$

Por lo tanto $\sigma_k = \frac{1}{\sqrt{\lambda_k}} + \sum_{j=k+1}^n |\rho_{kj}| \sigma_j$ es una constante tal que $-\sigma_k \leq x_k \leq \sigma_k$. Por inducción concluimos que si \vec{x} es una raíz de q entonces todo x_i está acotado entre $-\sigma_i$ y σ_i para alguna constante σ_i . Como cada x_i debe ser un número entero entre $-\sigma_i$ y σ_i entonces concluimos el siguiente lema:

Lema 2.4. *Si q es una forma unitaria positiva entonces su conjunto de raíces es finito.*

En lo siguiente denotaremos por $S(q)$ al conjunto finito de raíces de q , y por $R(q)$ al conjunto de raíces no negativas de q , en otras palabras:

$$\begin{aligned} S(q) &= \{ \vec{x} \in \mathbb{Z}^n | q(\vec{x}) = 1 \} \\ R(q) &= \{ \vec{x} \in \mathbb{N}^n | q(\vec{x}) = 1 \} \end{aligned}$$

Claramente $R(q) \subseteq S(q)$.

Supongamos que q es una forma unitaria positiva. Entonces \mathbf{B}_q es una gráfica simple y el algoritmo 2.1 busca valores de r y s tales que existe una arista punteada $x_r - \dots - x_s$. Estos es lo mismo que decir que $q_{rs} = 1$ (ver Corolario 1.10). Sea $q' = q \circ T_{rs}^-$. Como la transformación $T_{rs}^- : \mathbb{Z}^n \rightarrow \mathbb{Z}^n$ está definida por una matriz \mathbb{Z} -invertible entonces T_{rs}^- es invertible y su inversa es T_{rs}^+ .

Supongamos que \vec{x} es una raíz de q , entonces

$$q'(T_{rs}^+(\vec{x})) = q(T_{rs}^-(T_{rs}^+(\vec{x}))) = q(\vec{x}) = 1$$

y por tanto $T_{rs}^+(\vec{x})$ es una raíz de q' . Esto nos dice que de hecho T_{rs}^+ es una función biyectiva entre los conjuntos $S(q)$ y $S(q')$, e inyectiva en los conjuntos $R(q)$ y $R(q')$. De aquí concluimos que $|S(q)| = |S(q')|$ y que $|R(q)| \leq |R(q')|$; mostraremos que esta desigualdad es estricta.

Denotamos con $\vec{e} = (d_1, d_2, \dots, d_n)$ al vector tal que d_k y d_i para todo $i \neq k$. Sea $\vec{z} = \vec{e}_s - \vec{e}_r$. Notamos que

$$q(\vec{z}) = (-1)^2 + 1^2 + q_{rs}(-1)(1) = 1$$

por lo que $\vec{z} \in S(q)$ pero $\vec{z} \notin R(q)$ (porque tiene un -1 en la entrada r -ésima). Sin embargo $T_{rs}^+(\vec{z}) = \vec{e}_s$, es decir que $\vec{z} \in R(q')$. Como $\vec{z} \in R(q')$ pero $\vec{z} \notin R(q)$, entonces $|R(q)| < |R(q')|$.

Finalmente, si denotamos por q_1, q_2, q_3, \dots a la sucesión de valores que toma la variable q de el algoritmo 2.1 entonces de lo anterior tenemos que

$$\begin{aligned} |S(q_1)| &= |S(q_2)| = |S(q_3)| = \dots \\ |R(q_1)| &< |R(q_2)| < |R(q_3)| < \dots \end{aligned}$$

Si estas sucesiones fuesen infinitas entonces eventualmente contradecimos que $R(q_i) \subseteq S(q_i)$, por lo tanto el algoritmo 2.1 solamente puede hacer un número finito de iteraciones de ciclo **mientras**. Una breve vista al pseudocódigo nos muestra que esto ocurre cuando \mathbf{B}_q ya no contiene aristas punteadas, por lo tanto al término de este ciclo \mathbf{B}_q contiene solamente aristas sólidas. Como habíamos supuesto que q es positiva y como cada q_i es \mathbb{Z} -equivalente a q , entonces el lema 1.8 nos garantiza que cada componente conexa de \mathbf{B}_q es una gráfica de Dynkin.

Con esto concluye la demostración de el teorema 1.7.

2.2. ENSAMBLAJE POR \mathbb{A} -BLOQUES

VISIÓN GENERAL DEL ENSAMBLE En [4] se aborda el problema de caracterizar las formas de tipo \mathbb{A}_n ; pero en lugar de basarse en transformaciones, se basa en la construcción de las gráficas Δ tales que q_Δ es de tipo \mathbb{A}_n . La idea es ir “pegando” ciertas gráficas como si fueran bloques de construcción. La estructura de estos bloques es muy sencilla: Se parte de dos conjuntos de vértices V_0 y V_1 ; entre los vértices (u, v) hay una arista punteada $u \cdots \cdots v$ si ambos $u, v \in V_0$ o ambos $u, v \in V_1$, y una arista sólida $u \text{---} v$ en otro caso. A las gráficas así obtenidas les llamaremos \mathbb{A} -bloques y las denotaremos por $F_{m, m'}$ donde $m = |V_0|$ y $m' = |V_1|$ (pero acostumbramos a escribir $m \geq m'$). La figura 2.3 muestra cómo dibujar un \mathbb{A} -bloque. [1][4]

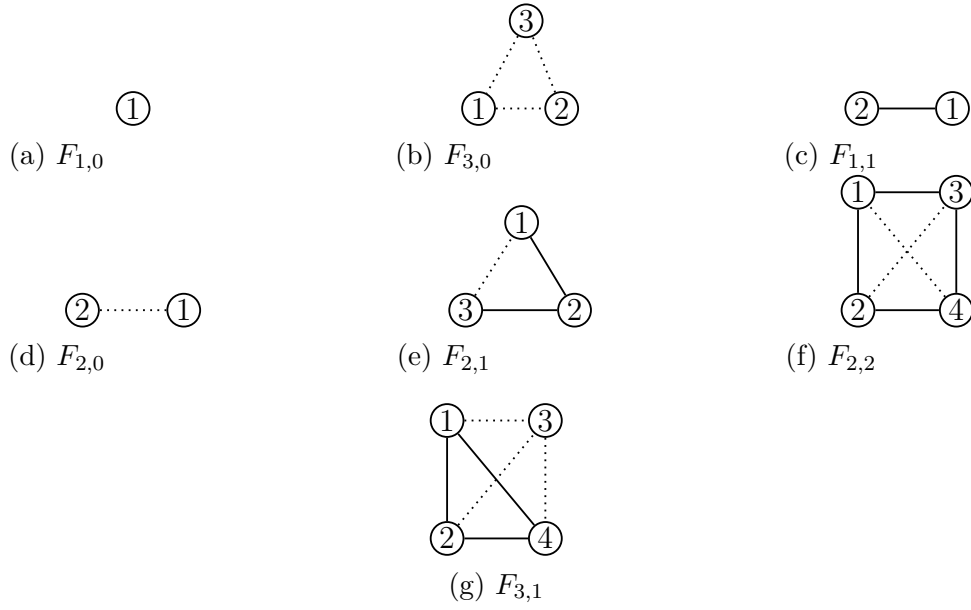


Figura 2.2: Algunos ejemplos de \mathbb{A} -bloques.

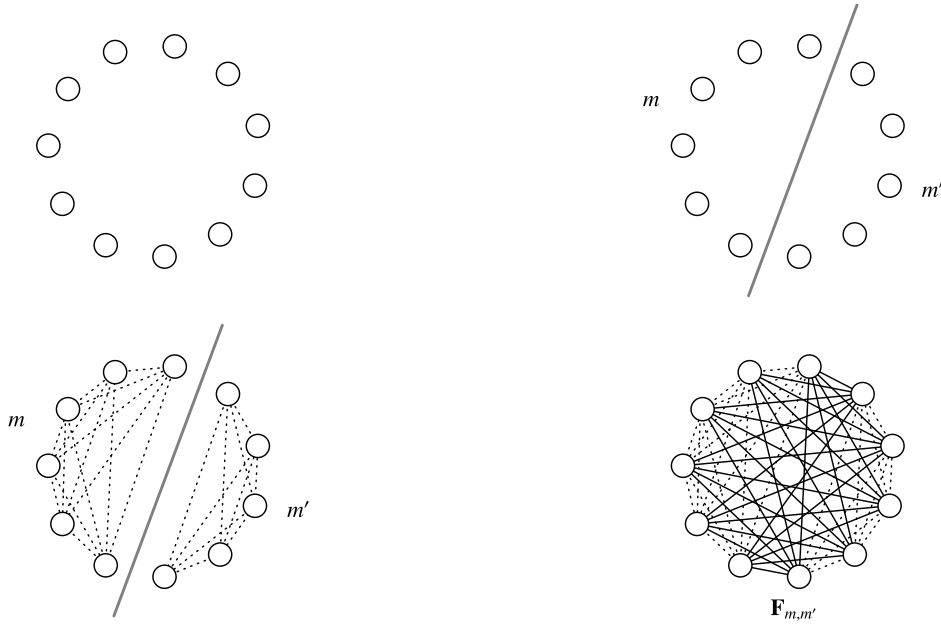
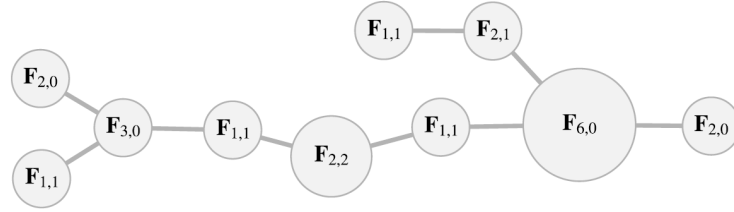


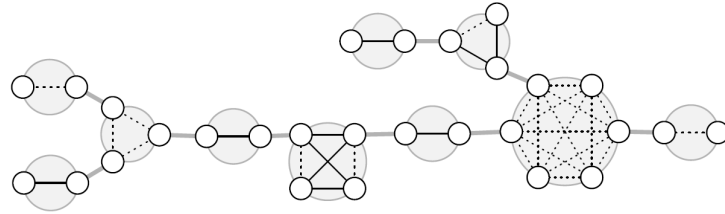
Figura 2.3: Como se construye un \mathbb{A} -bloque.

El resultado principal de [4] es que todas las formas unitarias de tipo \mathbb{A}_n se construyen pegando \mathbb{A} -bloques sobre un árbol como se explica a continuación:

1. Se comienza con un árbol T de vértices $\{1, 2, \dots, t\}$ y a cada vértice $i \in T$ se le asocia un \mathbb{A} -bloque \mathcal{B}_i .



2. A cada arista $i \text{---} j \in T$ se le asocian los vértices $\sigma_i(i \text{---} j) \in \mathcal{B}_i$ y $\sigma_j(i \text{---} j)$ de manera que cada función σ_k sea inyectiva.



3. Para cada arista $i \text{---} j \in T$ se identifican los vértices $\sigma_i(i \text{---} j)$ y $\sigma_j(i \text{---} j)$, es decir, se pegan para volverse uno solo. Continuando con este ejemplo obtenemos la figura 2.4

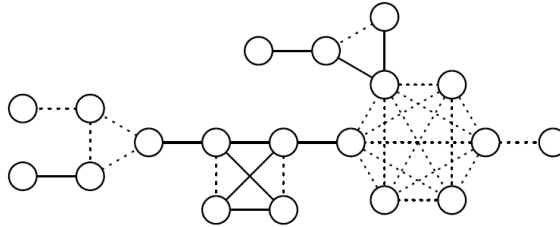


Figura 2.4: Esta gráfica define a una forma de tipo \mathbb{A}_{19}

LOS \mathbb{A} -BLOQUES

A continuación trataremos de justificar el ensamble de \mathbb{A} -bloques a partir del siguiente resultado tomado de [5]:

Teorema 2.5. *Si q_G es de tipo \mathbb{A}_n , y si H es una subgráfica conexa inducida por m vértices de G , entonces q_H es de tipo \mathbb{A}_m .*

Comencemos con un caso especial de gráficas: las gráficas circulares son aquellas en las que cada vértice está conectado con exactamente otros dos vértices. Nos interesa saber cuáles gráficas circulares definen formas cuadráticas de tipo \mathbb{A}_n . Ciertamente $F_{3,0}$ y $F_{2,1}$ son unas

de estas (lo podemos comprobar usando el algoritmo 2.1)

Trataremos de simplificar un poco el problema. Las gráficas circulares de la figura 2.5 definen formas \mathbb{Z} -equivalentes: podemos pasar de una a otra mediante las matrices elementales E_i^{-1} de tamaño n . No es difícil comprobar que estas matrices tienen el efecto de intercambiar aristas sólidas con punteadas siempre que estas incidan en el vértice x_i . De hecho, este mismo razonamiento muestra que toda gráfica circular es equivalente a otra gráfica circular estandarizada, que consiste de aristas sólidas y posiblemente una sola arista punteada.

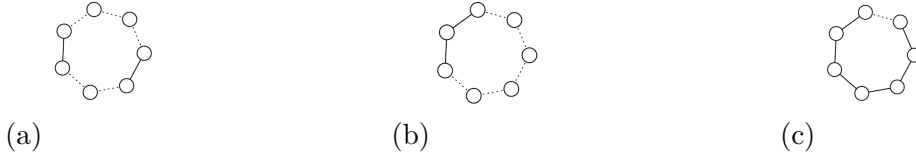
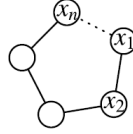


Figura 2.5: Ejemplos de gráficas circulares.

Podemos descartar a las gráficas que no tienen ninguna arista punteada por que son de tipo \tilde{A}_n , es decir, no son positivas. Denotemos con $C(n)$ a el grafo circular de n vértices que tiene exactamente una arista punteada $x_n \cdots x_1$ y las demás sólidas:



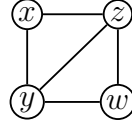
Con el algoritmo 2.1 podemos comprobar directamente que $C(4)$ tiene tipo Dynkin D_4 . Si $n \geq 4$, aplicando la inflación T_{n1}^- y borrando el vértice x_n de $C(n)$ se obtiene $C(n-1)$, de modo que por inducción y el teorema 2.5, todas las gráficas $C(n)$ con $n \geq 3$ no pueden ser de tipo \mathbb{A}_n . Hemos mostrado el siguiente lema:

Lema 2.6. *Las únicas gráficas circulares G tales que q_G es de tipo \mathbb{A}_n son $F_{3,0}$ y $F_{2,1}$.*

Una gráfica es k -conexa si no existe ningún subconjunto de $k-1$ vértices tales que su borrado desconecte a la gráfica. Dicho de otro modo, si κ denota la cantidad mínima de vértices que es necesario borrar para obtener una gráfica desconexa, entonces $\kappa \geq k$. Por ejemplo, toda gráfica conexa es automáticamente 1-conexa y toda gráfica completa K_n es k -conexa para todo $k \in \{1, 2, \dots, n\}$. De hecho, por definición tenemos que si G es una gráfica k -conexa entonces también es $(k-1)$ -conexa, $(k-2)$ -conexa, etc. Otro ejemplo: las gráficas circulares son gráficas biconexas (2-conexas). Ahora supongamos que \mathbf{B}_q es biconexa y que q es de tipo \mathbb{A}_n ; mostraremos que \mathbf{B}_q es un \mathbb{A} -bloque.

Primero mostraremos que \mathbf{B}_q debe ser una gráfica completa. Supongamos que no es así y sea S el conjunto de vértices más pequeño tal que al borrarlos de \mathbf{B}_q la gráfica se vuelve desconexa y defínase $\kappa = |S|$. Sabemos que \mathbf{B}_q es 2-conexa, y además que la ausencia de la arista $u \cdots v$ porque \mathbf{B}_q no es completa) nos permite obtener una gráfica desconexa borrando todos los

vértices excepto u y v ; por lo tanto tenemos que $2 \leq \kappa \leq n - 2$. Seleccionamos cualesquiera $\kappa - 2$ vértices del conjunto S y los borramos de la gráfica \mathbf{B}_q para obtener otra gráfica \mathbf{B}'_q . Por la definición de S sabemos que \mathbf{B}'_q es una gráfica biconexa pero no triconexa(3-conexa). Es decir, en \mathbf{B}'_q existen dos vértices x y y tales que al borrarlos de \mathbf{B}'_q se obtiene una gráfica de dos componentes conexos \mathcal{B}_0 y \mathcal{B}_1 . Consideremos el camino más corto $x \rightsquigarrow y$ que inicia en x pasa solamente por vértices de \mathcal{B}_0 y termina en y , juntemos este camino con el camino más corto $y \rightsquigarrow x$ que inicia en y , pasa solamente por vértices de \mathcal{B}_1 y termina en x . Esta unión forma un ciclo $x \rightsquigarrow y \rightsquigarrow x$, pero habíamos supuesto que q es de tipo \mathbb{A}_n , por tanto el lema 2.6 nos dice que \mathbf{B}'_q contiene la siguiente subgráfica inducida (sin tomar en cuenta si las aristas son sólidas o punteadas):



Aquí $z \in \mathcal{B}_0$, $w \in \mathcal{B}_1$ y no existe la arista $z \text{---} w$ (en otro caso \mathbf{B}_q seguiría conectada después de quitar los vértices x e y). Otra vez mediante el uso del algoritmo 2.1 se puede demostrar que no importa cuáles aristas sean sólidas o punteadas, esta gráfica no es de tipo \mathbb{A}_n , en contradicción con el teorema 2.5; por lo tanto tenemos:

Lema 2.7. Si $q : \mathbb{Z}^n \rightarrow \mathbb{Z}$ es una forma unitaria de tipo \mathbb{A}_n y si \mathbf{B}_1 es una gráfica biconexa, entonces \mathbf{B}_q es una gráfica completa.

El siguiente lema se puede leer *entre líneas* en el artículo mencionado; aquí solamente se está recalando su importancia porque haremos referencia a este lema en capítulos posteriores:

Lema 2.8. Si toda subgráfica de \mathbf{B}_q inducida por tres vértices es $F_{3,0}$ o $F_{2,1}$, entonces \mathbf{B}_q es un \mathbb{A} -bloque.

Demostración. Fijemos un vértice u y definamos los conjuntos

$$\begin{aligned} V_0 &= \{u\} \cup \{v \mid \text{existe una arista punteada } u \cdots v\} \\ V_1 &= \{v \mid \text{existe una arista solida } u \text{---} v\} \end{aligned}$$

Considérese a otros dos diferentes vértices v y w en \mathbf{B}_q . Por hipótesis tenemos que los vértices u , v y w inducen una subgráfica $F_{3,0}$ o $F_{2,1}$. Por la definición de V_0 y V_1 concluimos que si dos vértices v, w están en el mismo V_i entonces hay una arista punteada $v \cdots w$; y en otro caso (están en conjuntos diferentes) hay una arista sólida $v \text{---} w$; por lo tanto \mathbf{B}_q es un \mathbb{A} -bloque. \square

Vamos a recapitular lo que hemos visto hasta ahora:

1. El lema 2.7 nos dice que si q es de tipo \mathbb{A}_n y si \mathbf{B}_q es biconexa, entonces \mathbf{B}_q es, de hecho, completa.

2. Pero por el teorema 2.5 cada subgráfica inducida de B_q debe definir otra forma de tipo \mathbb{A}_n .
3. En particular, como \mathbf{B}_q es completa, el lema 2.6 nos dice que toda subgráfica inducida por cada tres vértices de B_q es $F_{3,0}$ o $F_{2,1}$.
4. Entonces, por lema 2.8 concluimos que \mathbf{B}_q es un \mathbb{A} -bloque.

Ahora vamos a mostrar el recíproco: que todo \mathbb{A} -bloque es biconexo y que define una forma de tipo \mathbb{A}_n . La biconexidad es obvia puesto que todo \mathbb{A} -bloque $F_{m,m}$ es una gráfica completa. Ahora bien, renombramos a los vértices en V_0 como x_1, x_2, \dots, x_m ; y a los de V_1 como $x_{m+1}, x_{m+2}, \dots, x_n$. Podemos deshilar a \mathbf{B}_q en dos etapas: en la primera quitamos todas las aristas punteadas que hay entre los vértices de V_1 usando las inflaciones T_{ii+1}^- en orden $i = n-1, n-2, \dots, m+1$ y en la segunda las punteadas que hay en V_0 usando las inflaciones T_{i+1i}^- en orden $i = 1, 2, \dots, m-1$.

Para mostrar que este método de deshilaro funciona, supongamos que $m \geq 0$ y $m' \geq 0$ y sean $V_0 = \{x_1, \dots, x_m\}$ y $V_1 = \{x_{m+1}, \dots, x_n\}$ los conjuntos que definen a $F_{m,m'}$. Sea x_r incidente en x_{n-1} .

- Si $x_r = x_n$ entonces la arista $x_{n-1} \cdots x_n$ se sustituye por $x_{n-1} \text{---} x_n$.
- Para todos los x_r tales que exista una arista punteada $x_r \cdots x_{n-1}$ se tiene que $x_r \in V_1$ y el arrastre forma una arista sólida $x_r \text{---} x_n$ que se cancela con la arista $x_r \cdots x_n$; por lo tanto luego de aplicar T_{n-1n}^- en el vértice x_n no incide ninguna arista punteada.
- Para todos los x_r tales que exista una arista sólida $x_r \text{---} x_{n-1}$ se tiene que $x_r \in V_0$ y el arrastre forma una arista punteada $x_r \cdots x_n$ que se cancela con la arista $x_r \text{---} x_n$; por lo tanto luego de aplicar T_{n-1n}^- en el vértice x_n no incide ninguna arista sólida.

De lo anterior concluimos que al aplicar T_{n-1n}^- a $F_{m,m'}$ se obtiene $F_{m,m'-1}$ unida con una arista sólida $x_{n-1} \text{---} x_n$; luego entonces por inducción se sigue la sucesión de inflaciones $(T_{ii+1}^-)_{i=n-1}^{m+1}$ sobre el grafo $F_{m,m'}$ produce la gráfica $F_{m,1}$ unido con $x_{m+1} \text{---} \cdots \text{---} x_n$.

Un razonamiento similar muestra que la sucesión de inflaciones $(T_{i+1i}^-)_{i=1}^{m-1}$ aplicadas a $F_{m,1}$ produce $F_{m,m'}$. Por lo tanto el método descrito transforma $F_{m,m'}$ en $\mathbb{A}_{m,m'}$. *

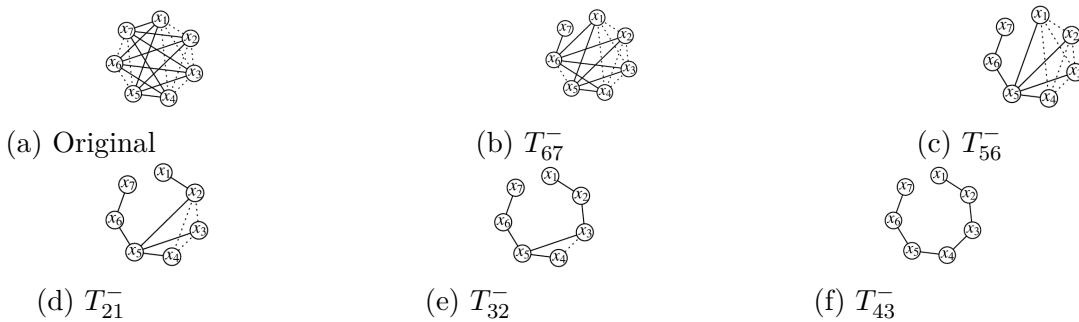


Figura 2.6: Deshilarando $F_{4,3}$

Se ha demostrado lo siguiente:

Lema 2.9. q es una forma unitaria de tipo \mathbb{A}_n con B_q biconexa si y solamente si B_q es un \mathbb{A} -bloque.

ENSAMBLAJE DE FORMAS \mathbb{A}_n

Ahora sí tenemos las herramientas necesarias para justificar el ensamble de \mathbb{A} -bloques. Primero veamos que todo ensamble de \mathbb{A} -bloques en verdad define una forma unitaria de tipo \mathbb{A}_n .

Supongamos que T es el árbol que subyace en un pegado de \mathbb{A} -bloques. Si este árbol tiene un solo vértice entonces ya acabamos por el lema 2.9 recién mostrado. En otro caso, como T es un árbol, debe haber un vértice t de grado 1 (podría decirse que t es una hoja del árbol). Entonces el \mathbb{A} -bloque asociado al vértice t , que es $\mathcal{B}_t = F_{m,m'}$, comparte exactamente un vértice v con algún otro \mathbb{A} -bloque \mathcal{B}_s (*). Ahora apliquemos la técnica deshilaro explicada justo antes del lema 2.9, pero renombrando al vértice v como $x_{m,m'}$, de manera que este deshilaro no afecta en absoluto a ningún otro vértice de \mathcal{B}_s . Lo que veremos (*) es que ahora del vértice v cuelga la siguiente gráfica:



Figura 2.7: Esquemática de la fusión de dos \mathbb{A} -bloques

$$x_{m+m'} \text{---} x_{m+m'-1} \text{---} \cdots \text{---} x_3 \text{---} x_2 \text{---} x_1$$

Esto se parece a lo que se obtiene durante los pasos intermedios para deshilar un \mathbb{A} -bloque; esto sugiere hacer el proceso inverso, es decir, utilizar las deflaciones.

Podemos usar las deflaciones para tejer los vértices de \mathcal{B}_t en \mathcal{B}_s aplicando sucesivamente $T_{i+1,i}$ en orden $i = m + m' - 1, m + m' - 2, \dots, 3, 2, 1$ (porque precisamente este es el proceso inverso al de deshilaro). Cada vez que hacemos esto estamos, en cierto sentido, agregando x_i a \mathcal{B}_s ; de manera que al término de estas deflaciones tendremos todos los vértices de \mathcal{B}_s y \mathcal{B}_t en un mismo \mathbb{A} -bloque \mathcal{B}'_s . Así hemos mostrado cómo fusionar dos \mathbb{A} -bloques del árbol en uno solo, repitiendo este proceso podemos fusionarlos todos, mostrando así que todo ensamble de \mathbb{A} -bloques es equivalente a un solo \mathbb{A} -bloque. *

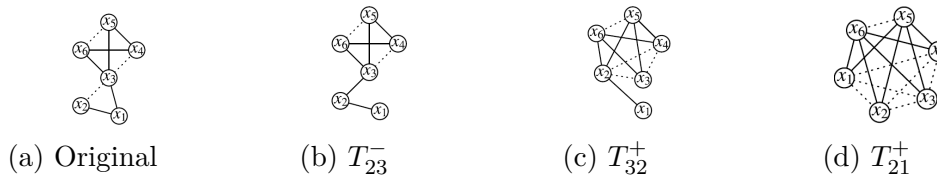


Figura 2.8: Ejemplo de fusión de dos \mathbb{A} -bloques

Lema 2.10. Si G es una gráfica construida por un ensamble de árbol de \mathbb{A} -bloques, entonces q_G es de tipo \mathbb{A}_n .

Falta mostrar el converso: que si G es una forma unitaria con tipo Dynkin \mathbb{A}_n entonces se puede construir mediante un ensamble de árbol de \mathbb{A} -bloques. Una **componente biconexa** de una gráfica, es una subgrafo biconexo maximal (no contenida propiamente en ninguna otra subgrafo biconexo). La manera más fácil de entender a las componentes biconexas es mediante los **puntos de articulación**, que son los vértices de la gráfica que al quitarlos la deja desconectada.



Figura 2.9: Una gráfica sus puntos de articulación y componentes biconexas

Lema 2.11. *Las componentes biconexas de una gráfica particionan al conjunto de aristas.*

Demostración. Cada arista es por sí misma una subgráfica biconexa, y por lo tanto pertenece a una subgráfica biconexa maximal; por otro lado ninguna arista puede pertenecer a dos componentes biconexas, por que si este fuera el caso entonces podríamos pegar ambas componentes por medio de la arista que tienen en común, mostrando así que no eran maximales. \square

Del teorema 2.5 y del lema 2.9 se concluye que si q es de tipo \mathbb{A}_n entonces las componentes biconexas de \mathbf{B}_q son \mathbb{A} -bloques. Supongamos que $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_t$ son las componentes biconexas de \mathbf{B}_q , y formemos una nueva gráfica T de vértices $\{1, 2, \dots, t\}$, donde cada arista $i \text{---} j$ representa que \mathcal{B}_i y \mathcal{B}_j , con $i \neq j$, tienen un vértice en común (este debe ser un punto de articulación de \mathbf{B}_q). Más aún, si \mathcal{B}_i y \mathcal{B}_j comparten el vértice v definiremos $\sigma_i(i \text{---} j) = \sigma_j(i \text{---} j) = v$.

Debido a la manera en que construimos las funciones σ_k y la gráfica T , tenemos que las siguientes tres afirmaciones son equivalentes:

1. Cada función σ_k es inyectiva
2. T es un árbol
3. Ningún punto de articulación pertenece a tres o más componentes biconexas

Por lo tanto basta mostrar cualquiera de estas afirmaciones. Mostraremos la tercera afirmación por inducción. Si \mathbf{B}_q no tiene puntos de articulación, por vacuidad ningún punto de articulación pertenece a tres o más componentes biconexas. Ahora supongamos que \mathbf{B}_q tiene al menos un punto de articulación v ; entonces $\mathbf{B}_q - v$ tiene componentes conexas $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\ell$.

Defínase D_i como la subgráfica de \mathbf{B}_q inducida por los vértices $V(\mathcal{C}_i) \cup \{v\}$. Por teorema 2.5 cada $q_{\mathcal{C}_i}$ es de tipo \mathbb{A}_{n_i} y por hipótesis de inducción, cada \mathcal{C}_i se construye haciendo un ensamble de árbol de \mathbb{A} -bloques (ningún \mathcal{C}_i tiene puntos de articulación que pertenezcan a tres o más componentes biconexas). Ahora, mediante el proceso previamente descrito, deshilemos a cada \mathcal{C}_i para convertirlo en un grafo \mathbb{A}_{n_i} que tenga a v como extremo. Luego entonces v es el centro de una gráfica de estrella de ℓ picos y sin aristas punteadas; pero habíamos supuesto que \mathbf{B}_q es de tipo \mathbb{A} , de manera que el algoritmo 2.1 nos dice que $\ell = 2$. Por lo tanto v , que es un punto de articulación cualquiera, solamente puede pertenecer a exactamente dos componentes biconexas.

Así tenemos el siguiente teorema:

Teorema 2.12. *Una forma unitaria q es de tipo \mathbb{A}_n si y solamente si \mathbf{B}_q se construye mediante un ensamble de árbol de \mathbb{A} -bloques.*

Sin embargo, para los fines de este trabajo es más conveniente reinterpretar este resultado utilizando otras palabras (pero la demostración es la misma):

Corolario 2.13. *Una forma unitaria q es de tipo \mathbb{A}_n si y solo si \mathbf{B}_q es conexa, sus componentes biconexas son \mathbb{A} -bloques y todo punto de articulación pertenece a exactamente dos de estas componentes.*

COMPONENTES TRICONEXAS

Para esta parte se utilizan los conceptos generados por [16] y mejorados por [11].

3.1. DIVIDIR UNA GRÁFICA EN COMPONENTES TRICONEXAS

3.1.1. INTRODUCCIÓN

Las propiedades de conectividad de las gráficas forman una parte importante de la teoría de gráficas. En [16] se considera el problema de separar una gráfica en sus componentes triconexas. Un algoritmo para esto es de utilidad para analizar circuitos eléctricos [2], para determinar si una gráfica es plana [6] y para determinar cuando dos gráficas planas son isomorfas [15]. Un algoritmo para planaridad puede ser usado en el diseño de tablas de circuitos; un algoritmo para isomorfismo de gráficas planas puede ser usado para probar el isomorfismo estructural de compuestos químicos [19] y en nuestro caso para ayudar a clasificar gráficas de Dynkin de tipo \mathbb{D}_n .

Una técnica que se ha utilizado para resolver problemas de conectividad es el recorrido primero en profundidad. En [23] y [13], se aplica la búsqueda primero en profundidad para obtener algoritmos eficientes para determinar las componentes biconexas de una gráfica no dirigida y para determinar las componentes fuertemente conexas de una gráfica dirigida. El método también se ha utilizado en un algoritmo eficiente para pruebas de planaridad ([25], [14]) y en un algoritmo para encontrar dominadores en un grafo de flujo [26]. Aquí se aplica la búsqueda primero en profundidad al problema de encontrar las componentes triconexas de una gráfica. Los métodos antiguos para determinar estos componentes requieren $\Theta(V^3)$ pasos o más, si la gráfica tiene V vértices ([2], [17]). El algoritmo descrito aquí requiere sustancialmente utilizar la combinación de los artículos [16] y mejorados por [11].

Se usa la siguiente notación para especificar límites: si f y g son funciones de x , digamos que

$f(x)$ es $\Theta(g(x))$ si, para algunas constantes k_1 y k_2 , $|f(x)| \leq k_1 |g(x)| + k_2$ para todo x

3.1.2. Gráficas, conectividad y búsqueda en profundidad

Las definiciones utilizadas aquí se ven [7] y [12]. Las componentes triconexas pueden definirse de varias maneras, todas más o menos equivalentes. Los resultados a continuación, se dan sin prueba, siguen de Saunders MacLaine [20]; estas definiciones se modifican un poco para hacerlos más adecuadas para aplicaciones computacionales.

Una gráfica $G = (\mathcal{V}, \mathcal{E})$ consiste de un conjunto \mathcal{V} que contiene V vértices y un conjunto \mathcal{E} que contiene E aristas. Si \mathcal{E}' es un conjunto de aristas en G , $\mathcal{V}(\mathcal{E}')$ es el conjunto de vértices que inciden a uno o más aristas en \mathcal{E}' . Si S es un conjunto de vértices en G , $\mathcal{E}(S)$ es el conjunto de aristas incidentes a al menos un vértice en S .

Un **camino** $p : v \xrightarrow{*} w$ en G es una secuencia de vértices y aristas que van de v a w . Un camino es simple si todos sus vértices son distintos. Un camino $p : v \xrightarrow{*} v$ es un **ciclo** si todas las aristas son distintas y el único vértice que se repite en p es v y este está al principio y al final de la secuencia de vértices. Un gráfica no dirigida(sin dirección) es conexa(esta conectada) si para cada par de vértices (v, w) existe un camino entre (v, w) . Sea $G = (\mathcal{V}, \mathcal{E})$ y sea $G' = (\mathcal{V}', \mathcal{E}')$ dos gráficas tales que $\mathcal{V}' \subseteq \mathcal{V}$ y $\mathcal{E}' \subseteq \mathcal{E}$ entonces G' es una subgráfica de G . Una gráfica con exactamente dos vértices v, w y uno o más aristas (v, w) se le dice **enlace**.

Un árbol T (dirigido, enraizado) es una gráfica dirigida cuya versión no dirigida es conexa, que tienen un vértice (llamado raíz) y en el que cualquier par de vértices están conectados por exactamente un camino. La relación “ (v, w) es una arista de T ” se denota por $v \rightarrow w$. La relación “hay un camino de v a w en T ” se denota por $v \xrightarrow{*} w$. Si $v \rightarrow w$, donde v es el padre de w , y w es un hijo de v . Si $v \xrightarrow{*} w$, entonces v es un ancestro de w y w es un descendiente de v . El conjunto de descendientes de un vértice v se denota por $D(v)$. Todo vértice es ancestro y descendiente de sí mismo. Si G es una gráfica no dirigida, un árbol T es un **árbol generador** de G si T es una subgráfica de G y T contiene todos los vértices de G .

Sea P una gráfica no dirigida que consta de dos conjuntos disjuntos de aristas, denotados por $v \rightarrow w$ y $v \dashrightarrow w$. Supongamos que P satisface las siguientes propiedades:

1. La subgráfica T que contiene las aristas $v \rightarrow w$ es un árbol generado de P .
2. Si $v \dashrightarrow w$, entonces $w \xrightarrow{*} v$. Es decir, cada arista que no está en el árbol generador T de P conecta el vértice w con uno de sus ancestros v en T .

Entonces a P llamaremos árbol de recorrido. Las aristas $v \dashrightarrow w$ las llamaremos aristas de retroceso de P .

Un gráfica conexa G es **biconexa** si por cada tripleta de vértices distintos v , w y a en V , hay un camino $p : v \xrightarrow{*} w$ tal que a no está en el camino p . Si hay una tripleta distinta v , w , a tal que a está en cada camino de $p : v \xrightarrow{*} w$, entonces se dice que a es un **punto de separación** (o **punto de articulación**) de G . Podemos particionar las aristas de G de manera que dos aristas están en el mismo bloque de la partición si y sólo si pertenecen a un **ciclo**. Sea $G_i = (V_i, E_i)$ donde E_i es el conjunto de aristas en el i -ésimo bloque de la partición, y $V_i = V(E_i)$. Entonces lo siguiente se cumple:

1. Cada G_i es biconexa.
2. Ningún G_i es una subgráfica propia de una subgráfica biconexa de G .
3. Cada vértice de G que no sea un punto de articulación de G está exactamente una vez entre los V_i y cada punto de articulación está al menos dos veces.
4. Para cada $i, j, i \neq j$, $V_i \cap V_j$ contiene como máximo un vértice; además, este el vértice (si existe) es un punto de articulación.

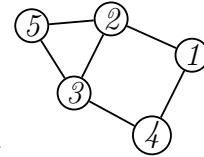
Los subgráficas G_i de G se denominan **componentes biconexas** de G . Las componentes biconexas de G son únicas.

Definición 3.1. Sea $G = (V, E)$ una gráfica y sea $H = (W, F)$ una subgráfica de G , definimos una relación de equivalencia sobre $E \setminus F$ como sigue:

1. $\forall e \in E \setminus F, e \approx e$
2. $\forall e, f \in E \setminus F$ con $e : a_1-b_1$ y $f : a_2-b_2$, entonces $e \approx f$ si y solo si existe un camino de los siguientes tipos:
 - $a_1-b_1-v_1-v_2-\dots-v_k-a_2-b_2$ tal que $b_1, v_1, \dots, v_k, a_2 \notin W$
 - $a_1-b_1-v_1-v_2-\dots-v_k-b_2-a_2$ tal que $b_1, v_1, \dots, v_k, b_2 \notin W$
 - $b_1-a_1-v_1-v_2-\dots-v_k-a_2-b_2$ tal que $a_1, v_1, \dots, v_k, a_2 \notin W$
 - $b_1-a_1-v_1-v_2-\dots-v_k-b_2-a_2$ tal que $a_1, v_1, \dots, v_k, b_2 \notin W$

Si $H = \{\{a, b\}, \emptyset\}$, las clases de equivalencia son llamadas clases de separación relativas al par $\{a, b\}$.

Definición 3.2. Sean S_1, S_2, \dots, S_k , las clases de separación relativas al par (a, b) . Si existe una partición (A, B) de $\{1, 2, \dots, k\}$ tal que $|E_1 = \bigcup_{i \in A} S_i| \geq 2$ y $|E_2 = \bigcup_{j \in B} S_j| \geq 2$ decimos que $\{a, b\}$ es un par de separación. Veamos esto en el siguiente ejemplo.



Ejemplo 3.1. Sea G el bigrafo y $H = \{\{2, 3\}, \emptyset\}$

3 COMPONENTES TRICONEXAS

La clase de equivalencia de $2-5$ es el conjunto:

$$S_1 = \{2-5, 5-3\}$$

La clase de equivalencia de $1-4$ es el conjunto:

$$S_2 = \{2-1, 1-4, 4-3\}$$

La clase de equivalencia de $2-3$ es el conjunto:

$$S_3 = \{2-3\}$$

Para saber si $\{2, 3\}$ es un par de separación hay que encontrar una partición de $\{1, 2, 3\} = A \cup B$, $A \cap B = \{\emptyset\}$ tal que se cumpla que $|E_1 = \bigcup_{i \in A} S_i| \geq 2$ y $|E_2 = \bigcup_{j \in B} S_j| \geq 2$. En este caso $A = \{1, 3\}$, $B = \{2\}$ es una partición posible que buscamos y entonces $\{2, 3\}$ es un par de separación.

Ahora supongamos $\{a, b\}$ es un par de separación. Si $H = \{\{a, b\}, \emptyset\}$ y S_1, S_2, \dots, S_k son las clases de separación del par $\{a, b\}$ (las clases de equivalencia definidas por H). Sea A, B la partición del conjunto $\{1, 2, \dots, k\}$ tal que $|E_1 = \bigcup_{i \in A} S_i| \geq 2$ y $|E_2 = \bigcup_{j \in B} S_j| \geq 2$. Si $H_1 = (V(E_1), E_1)$ y $H_2 = (V(E_2), E_2)$ entonces, $V(E_1) \cap V(E_2) = \{a, b\}$ donde la arista $a-b$ es llamada arista virtual. Sea $G_i = H_i + \{a, b\}$ para $i \in \{1, 2\}$. Los G_i son las gráficas de separación de G en $\{a, b\}$. A la operación de reemplazar la gráfica G por dos gráficas de separación llamaremos **separación** de G . Debe de haber muchas formas posibles de separar una gráfica, incluso con respecto a un par de separación fijo $\{a, b\}$. Una operación de separación se denota por $s(a, b, i)$; donde i es un etiqueta que distingue esta operación de separación de otras separaciones. Una arista virtual (a, b) asociada con la separación $s(a, b, i)$ se denotará por (a, b, i) . Si G es biconexa, cualquier gráfica de separación de G también es biconexa.

Si hay al menos dos clases de separación, entonces $\{a, b\}$ es una **par de separación** de G a menos que (i) haya exactamente dos clases de separación y una clase consta de una sola arista, o (ii) hay exactamente tres clases, cada una de las cuales consta de una sola arista.

Si G es una gráfica biconexa tal que ningún par $\{a, b\}$ es un par de separación de G , entonces G es **triconexa**.

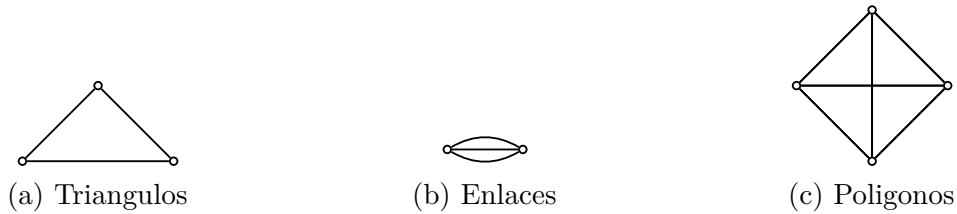


Figura 3.1: Ejemplos de componentes triconexas

Supongamos que separamos una gráfica G , las gráficas de separación se dividen, y así sucesivamente, hasta que no sean posibles más separaciones (cada gráfica restante es triconexa). Los gráficas construidas de esta manera se denominan **componentes de separación** de G . Los componentes de separación de una gráfica no son necesariamente únicos.

Lema 3.1. Sea $G = (V, E)$ un gráfica con $|E| \geq 3$. Sea G_1, G_2, \dots, G_m los componentes de separación de G . Entonces el número total de aristas en G_1, G_2, \dots, G_m está delimitado por $3|E| - 6$.

Demostración. El lema se demuestra por inducción sobre el número de aristas de G . Si G tiene 3 aristas, el lema es inmediato, porque G no se puede separar. Supongamos que el lema es cierto para gráficas con $n - 1$ aristas y supongamos que G tiene n aristas. Si G no puede ser dividido, el lema es verdadero para G . Supongamos, por otro lado, que G se puede separar en G' y G'' , donde G' tiene $k + 1$ aristas y G'' tiene $n - k + 1$ aristas tal que $2 \leq k \leq n - 2$. Por inducción, el número total de aristas en G_1, G_2, \dots, G_m debe estar acotado por $3(k + 1) - 6 + 3(n - k + 1) - 6 = 3n - 6$. Así, por inducción, el lema 1 es cierto. \square

Para obtener las componentes triconexas únicas, debemos volver a unir parcialmente los componentes de separación. Supongamos que $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son dos componentes de separación, ambos con una arista virtual (a, b, i) . Sea

$$G = (V_1 \cup V_2, (E_1 - \{(a, b, i)\}) \cup (E_2 - \{(a, b, i)\}))$$

Entonces a G se le llama una **gráfica de unión** de G_1 y G_2 ; la operación de unión se denotará por $m(a, b, i)$. La unión es la inversa de la separación; si realizamos un suficiente número de uniones en los componentes divididos de una gráfica, recreamos la gráfica original.

Los componentes de separación de una gráfica son de tres tipos:

1. enlaces triples de la forma $(\{a, b\}, \{(a, b), (a, b), (a, b)\})$ 3.1b
2. triángulos de la forma $(\{a, b, c\}, \{(a, b), (a, c), (b, c)\})$ 3.1a
3. gráficas triconexas 3.1c

Sea G una gráfica cuyos componentes de separación son un conjunto de enlaces triples \mathcal{B}_3 , un conjunto de triángulos \mathcal{T} y un conjunto de gráficas triconexas \mathcal{C} . Supongamos que los enlaces triples \mathcal{B}_3 se unen tanto como sea posible para dar un conjunto de enlaces \mathcal{B} y que los triángulos \mathcal{T} se unen tanto como sea posible para dar un conjunto de polígonos \mathcal{P} . Entonces el conjunto de gráficas $\mathcal{B} \cup \mathcal{P} \cup \mathcal{C}$ es el conjunto de componentes triconexas de G . Si G es una gráfica arbitraria, las componentes triconexas de las componentes biconexas de G se les llama **componentes triconexas** de G .

Lema 3.2. Las componentes triconexas de una gráfica G son únicas

Demostración. Prueba. Ver [20], [8] y [24]. \square

Tomemos como ejemplo el siguiente grafo para mostrar todo el procedimiento.

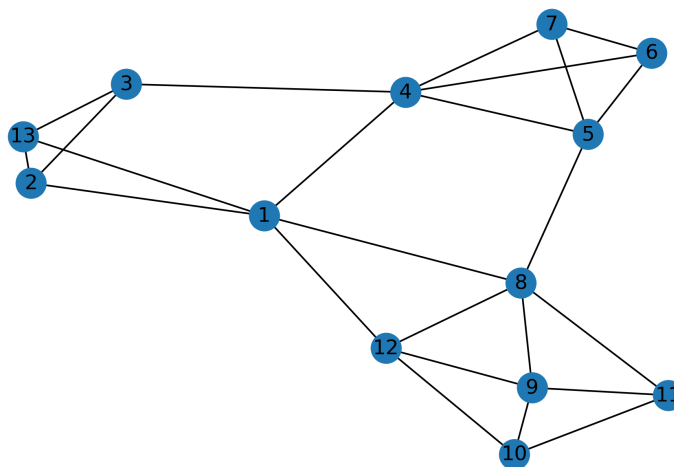


Figura 3.2: Una gráfica biconexa G con pares de separación $(1, 4), (4, 5), (1, 8), (12, 8)$

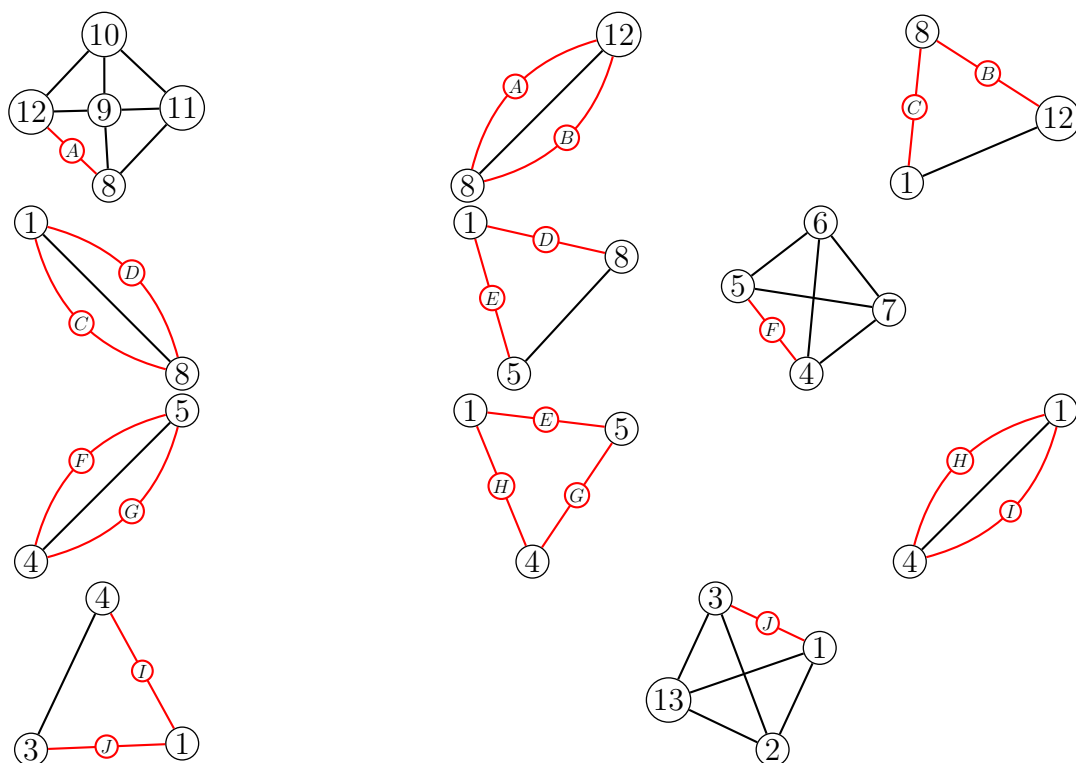


Figura 3.3: Componentes de separación de G las componentes triconexas se forma al unir los triángulos $(1, 4, 5)$ y $(1, 5, 8)$

Los algoritmos de gráficas requieren una forma sistemática de explorar una gráfica. En el artículo [16] se utiliza un método llamado **búsqueda en profundidad**. Para llevar a cabo una búsqueda en profundidad en G , se comienza desde algún vértice s y se elije una arista

que vaya desde s a otro vertice w en el gráfica, después se marca el vertice s como visitado y se elige ahora w como punto de partida ahora vamos a elegir alguna arista que conecta w cuyo vertice que la conecta a w aun no haya sido visitado si cumple esto elegimos esta arista y marcamos a w como visitado y continuamos así hasta que ya no haya vertices a los cuales visitar. Si G es conexa, cada arista se recorre exactamente una vez*.

Si G no es dirigida, una búsqueda sobre G impone una dirección en cada arista de G dada por la dirección en la que se recorre la arista durante la búsqueda. Así la búsqueda convierte G en una gráfica dirigida G' .

Lema 3.3. *Sea P la gráfica dirigida generada por una búsqueda en profundidad de una gráfica no dirigida conexa G . Entonces P es un árbol de recorrido.*

Demostración. Véase [23]. □

La búsqueda primero en profundidad es importante porque la estructura de los caminos en un árbol es muy simple. Para implementar una búsqueda en profundidad de una gráfica, usamos un procedimiento recursivo simple que mantiene una pila de los viejos vértices con posiblemente aristas inexploradas. Para representar un gráfica, se utiliza un conjunto de **listas de adyacencia**, uno para cada vértice. Si v es un vértice la lista de adyacencia $A(v)$ contiene todos los w tales que (v, w) es una arista de G . Estas listas juntas comprenden una **estructura de adyacencia** para G . Si G no es dirigida, cada arista (v, w) se representa dos veces, una en $A(v)$ y otra en $A(w)$.

3.1 muestra un procedimiento recursivo para realizar una búsqueda en profundidad. La búsqueda exacta depende del orden de los aristas en las listas de adyacencia. Los números de procedimiento de los vértices del 1 al V en el orden en que se alcanzan durante el búsqueda, además de identificar arcos y ramas de el árboles de recorrido y aristas de retroceso que nos ayudan mas adelante. La referencia [23] da una prueba que el procedimiento es correcto y requiere tiempo $\Theta(V + E)$ para ejecutarse. Los vértices están numerados de modo que $NUMBER(v) \leqslant NUMBER(w)$ si $v \xrightarrow{*} w$ en el árbol generado.

Las declaraciones a, b, c, se reemplazarán cuando se use DFS para calcular otra información sobre el gráfica. En 3.4 representa el arbol formado aplicando DFS y tambien las aristas de backtracking $a \hookrightarrow b$ que son necesarias para nuestro algoritmo. En rojo estan coloreadas las aristas de DFS y en azul el Backtracking de estas aristas y identificar los valores de lowpt1 y lowpt2.

Algoritmo 3.1: DFS(v, u)

```

1 // a: declaración vacía
2 para cada  $w$  in  $A(v)$ 
3   si  $NUMBER(w) = 0$ 
4     marcamos a  $(v, w)$  como una rama de el árbol
5      $DFS(w, v)$ 
6     // b: declaración vacía
7   si no  $(NUMBER(w) < NUMBER(v))$  and  $((w \neq u) \text{ or } \neg FLAG(v))$ 
8     marcamos  $(v, w)$  como arista de retroceso
9     // b: declaración vacía
10  si  $w=u$ 
11     $FLAG(v) = false$ 
12   $n = 0$ 
13  para  $i = 1$  to  $V$ :
14     $NUMBER(i) = 0$ 
15     $FLAG(i) = True$ 
16   $DFS(1, 0)$ 

```

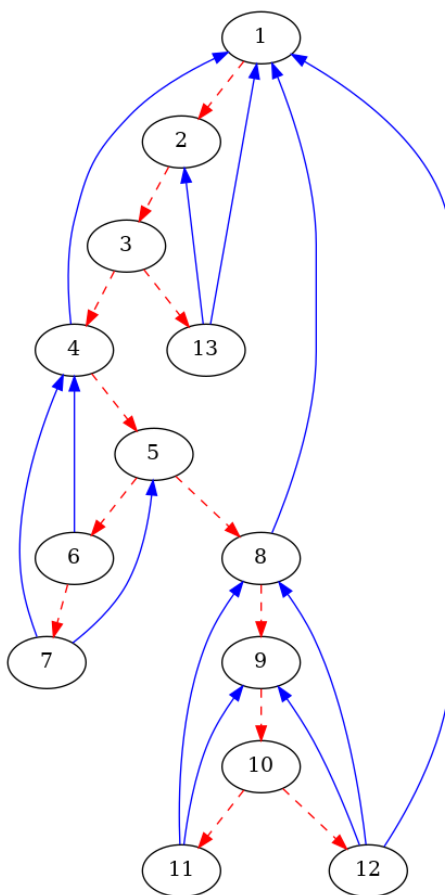


Figura 3.4: Árbol de recorrido generado por la búsqueda primero en profundidad de G en 3.2

3.1.3. La idea de el algoritmo de triconectividad

Esta sección esboza las ideas detrás del algoritmo de triconectividad. Las secciones posteriores desarrollan los componentes detallados. El algoritmo se basa en una idea de Auslander, Parter y Goldstein ([3], [10]) para probar la planaridad de los gráficas. La idea de Auslander, Parter y Goldstein da lugar a un algoritmo de tiempo $\Theta(V)$ para probar la planaridad, si la búsqueda primero en profundidad se utiliza para ordenar los cálculos de ([13], [14]). La misma idea da un tiempo $\Theta(V + E)$ para el algoritmo de encontrar componentes triconexas.

Sea G una gráfica biconexo arbitrario. Supongamos que un ciclo c se encuentra en G . Cuando se elimina el ciclo de G quedan ciertas piezas conectadas; a estos se les llama **segmentos**. Auslander y Parter [3] demuestran que G es plano si y solo si

1. Cualquier subgráfica de G que consta de c más un solo segmento es plano.
2. Los segmentos pueden combinarse consistentemente para dar una incrustación plana de todo la gráfica.

Se puede desarrollar un algoritmo de planaridad eficiente a partir de este resultado([13], [14]). A resultado similar se cumple para los pares de separación de G , es decir, el siguiente lema.

Lema 3.4. *Sea G una gráfica biconexa y sea c un ciclo en G . Seaa S_1, \dots, S_m las subgráficas de $G-c$ tales que e_1 y e_2 son aristas de S_i si y sólo si algún camino p en G contiene tanto e_1 como e_2 y ningún vértice de c se encuentra entre e_1 y e_2 en p . Los segmentos S_i y el ciclo c particionan las aristas de G . Sea $\{a, b\}$ un par de separación de G , entonces se cumplen las siguientes conclusiones.*

1. *Tanto a como b se encuentran en c , o bien a y b se encuentran ambos en algún segmento S_i .*
2. *Supongamos que tanto a como b están en c . Sean p_1 y p_2 los dos caminos que componen c los cuales unen a y b . Entonces o*
 - a) *Algún segmento S_i con al menos dos aristas tiene solo a y b en común con c , y algún vértice v no está en S_i ($\{a, b\}$ se le dice par de separación de "tipo 1"), o*
 - b) *Ningún segmento contiene un vértice $v \neq a, b$ en p_1 y un vértice $w \neq a, b$ en p_2 , p_1 y p_2 contienen cada uno un vértice además de a y b ($\{a, b\}$ se le dice par de separación de "tipo 2").*
3. *A la inversa, cualquier par $\{a, b\}$ que satisfaga (a) o (b) es un par de separación.*

El lema 3.4 da lugar a un algoritmo recursivo eficiente para encontrar componentes de separación. Encontramos un ciclo en G y determinamos los segmentos formados cuando es eliminado. Probamos cada segmento en busca de pares de separación aplicando el algoritmo recursivamente y probamos el ciclo para pares de separación verificando los criterios en lema 3.4. La aplicación recursiva del algoritmo requiere encontrar ciclos en subgráficas de G formados por la combinación de un segmento S_i y el ciclo inicial c .

Podemos hacer que este algoritmo sea muy eficiente ordenando los cálculos usando búsqueda en profundidad. Cada llamada recursiva en el algoritmo requiere que encontremos un ciclo en la parte de la gráfica que se va a probar para los pares de separación. Este ciclo constará de un camino simple de aristas las cuales no están en ciclos previamente encontrados más un camino simple de aristas en ciclos no encontrados. Usamos la búsqueda primero en profundidad para separar la gráfica en caminos simples que pueden ensamblarse en estos ciclos. El primer ciclo c consistirá en una secuencia de aristas del árbol seguidos por una arista en P , el árbol formado a partir de G por búsqueda primero en profundidad. La numeración de vértices es tal que los vértices están ordenados por número a lo largo del ciclo. Cada segmento consistirá en una sola arista (v, w) o en una arista (v, w) de el árbol más un subárbol con raíz w , además todas las ramas que salen del subárbol. La búsqueda explora los segmentos en orden decreciente de v y divide cada uno en caminos simples que consisten en una secuencia de ramas de el árbol seguido por una vértice.

Encontrar caminos en realidad requiere dos búsquedas porque la búsqueda de caminos debe llevarse a cabo en un orden especial para tener éxito, y ciertos cálculos preliminares son necesarios. La sección sobre cómo encontrar pares de separación describe la proceso de búsqueda de caminos en detalle e incluye una versión del lema 3.4 que caracteriza pares de separación en términos de los caminos generados. La sección de encontrar componentes de separación indica cómo se pueden usar estos resultados para determinar los componentes de separación de una gráfica biconexa en tiempo $\Theta(V + E)$.

Para determinar las componentes triconexas de una gráfica arbitrarias, encontramos las componentes biconexas de la gráfica usando el algoritmo $\Theta(V + E)$ descrito en [23] y [13]. A continuación, los componentes de separación de cada componente biconexa se encuentran utilizando el algoritmo descrito anteriormente y presentado en detalle en el siguiente dos secciones. Esto nos da los componentes de separación de toda la gráfica. El tamaño total de las componentes de separación es $\Theta(V + E)$, por el lema 3.1. A continuación identificamos el conjunto de enlaces triples \mathcal{B}_3 y el conjunto de triángulos \mathcal{T} . Para cada uno de estos dos conjuntos, construimos una gráfica auxiliar S cuyos vértices son los elementos del conjunto; dos componentes de separación están unidos por una arista en una gráfica auxiliar si tienen un arista virtual común. Los componentes conectados de $S(\mathcal{B}_3)$ y $S(\mathcal{T})$ corresponden a los enlaces y polígonos que son componentes triconexas de G . Encontrar estos enlaces y polígonos requiere tiempo $\Theta(V + E)$. A continuación se muestra un esquema de todo el algoritmo de acuerdo a [11].

3.2. El algoritmo para encontrar componentes triconexas

Algoritmo 3.2: TRICONNECTIVITY(G)

- 1 A: Encontramos las componentes biconexas de G ;
 - 2 **para** cada componente biconexa C de G
 - 3 B: encontrar los componentes de separación de C ;
 - 4 C: combinar los triangulos en poligonos al encontrar las componentes conexas correspondientes a las gráficas auxiliares;
-

Los pasos A y C requieren tiempo $\Theta(V + E)$ si se implementan correctamente. La implementación del paso A se describe en [23]. El paso difícil es el paso B, cuya implementación se describe en las siguientes dos secciones. Con base en los resultados de estas secciones, toda el algoritmo de triconectividad está acotado por $\Theta(V + E)$.

3.2.1. Encontrando pares de separación

Sea $G = (\mathcal{V}, \mathcal{E})$ una gráfica biconexa con vértices V y aristas E . El problema principal al separar G en sus componentes de separación radica en encontrar sus pares de separación. Esta sección da un criterio simple, basado en la búsqueda primero en profundidad, para identificar los pares de separación de una gráfica. Se deben realizar dos búsquedas en profundidad y algunos cálculos auxiliares. Estos cálculos forman la primera parte del algoritmo de componentes de separación y se describen a continuación. Las definiciones de las cantidades lowpt1 , ND , etc., utilizadas en el esquema se dará posteriormente.

Paso 1 Realice una búsqueda primero en profundidad en la gráfica G , convirtiendo G en un árbol P . Numeramos los vértices de G en el orden en que se alcanzan durante la búsqueda. Calculamos $\text{lowpt1}(v)$, $\text{lowpt2}(v)$, $\text{nd}(v)$ y $\text{padre}(v)$ para cada vértice v en P .

Paso 2 Construimos una estructura de adyacencia aceptable A para P ordenando las aristas en la estructura de adyacencia de acuerdo con los valores de lowpt1 y lowpt2 .

Paso 3 Realizar una búsqueda en profundidad sobre P utilizando la estructura de adyacencia A . Volvemos a numerar los vértices de A desde V a 1 en el orden en que se examinaron por última vez durante la búsqueda. Particionamos las aristas en caminos simples disjuntos. Recalculamos $\text{lowpt1}(v)$ y $\text{lowpt2}(v)$ utilizando los nuevos números de los vértices. Calculamos $\text{A1}(v)$, $\text{deg}(v)$, y $\text{highpt}(v)$ para cada vértice v .

Los detalles de estos cálculos aparecen a continuación. Desde los pasos 1, 2 y 3, obtenemos suficiente información para determinar rápidamente los pares de separación de G . lema 3.13 da una condición para este propósito.

Supongamos que G se explora primero en profundidad, dando un árbol P . Sean los vértices de P numerados desde 1 a V de modo que $v \xrightarrow{*} w$ en P implica que $v \leq w$, si identificamos los vértices por su número. Para cualquier vértice v en P , sea $\text{padre}(v)$ el padre de v en el árbol de P . Sea $\text{ND}(v)$ el número de descendientes de v . Sea $\text{lowpt1}(v) = \min(\{v\} \cup \{w | v \xrightarrow{*} - \rightarrow w\})$. Es decir, $\text{lowpt1}(v)$ es el vértice más bajo alcanzable desde v atravesando cero o más ramas de árbol en P seguido de en la mayoría una arista. Sea $\text{lowpt2}(v) = \min[\{v\} \cup (\{w | v \xrightarrow{*} - \rightarrow w\} - \{\text{lowpt1}(v)\})]$. Es decir, $\text{lowpt2}(v)$ es el segundo vértice más bajo alcanzable desde v atravesando cero o más ramas de el árbol seguidas por una arista de P como máximo, a menos que $\text{lowpt1}(v) = v$. En este caso, $\text{lowpt2}(v) = v$.

Lema 3.5. $\text{lowpt1}(v) \xrightarrow{*} v$ y $\text{lowpt2}(v) \xrightarrow{*} v$ en P .

Demostración. $\text{lowpt1}(v) \leq v$ por definición. Si $\text{lowpt1}(v) = v$, el resultado es inmediato. Si $\text{lowpt1}(v) < v$, hay una rama $u \rightarrow \text{lowpt1}(v)$ tal que $v \xrightarrow{*} u$. Como $u \rightarrow \text{lowpt1}(v)$ es una rama, $\text{lowpt1}(v) \xrightarrow{*} u$. Como P es un árbol, $v \xrightarrow{*} u$ y $\text{lowpt1}(v) \xrightarrow{*} u$, ya sea que $v \xrightarrow{*} \text{lowpt1}(v)$ o $\text{lowpt1}(v) \xrightarrow{*} v$. Pero $\text{lowpt1}(v) \leq v$. Así debe ser el caso que $\text{lowpt1}(v) \xrightarrow{*} v \xrightarrow{*} u$, y el lema se cumple para $\text{lowpt1}(v)$. La demostración es la misma para $\text{lowpt2}(v)$. \square

Lema 3.6. *Supongamos que $\text{lowpt1}(v)$ y $\text{lowpt2}(v)$ se definen en relación con alguna numeración para la cual $v \xrightarrow{*} w$ en P implica $\text{NUMERO}(v) < \text{NUMERO}(w)$. Entonces $\text{lowpt1}(v)$ y $\text{lowpt2}(v)$ identifican vértices únicos independientemente de la numeración usada.*

Demostración. $\text{lowpt1}(v)$ siempre identifica a un ancestro del vertice v . Además, $\text{lowpt1}(v)$ es el ancestro con el número más bajo de v con cierta propiedad relativa al árbol P . Dado que el orden de los ancestros de v corresponde al orden de sus números, $\text{lowpt1}(v)$ identifica a un único vértice independiente de la numeración, es decir, el primer ancestro de v a lo largo de el camino $1 \xrightarrow{*} v$ tiene la propiedad deseada (Cualquier numeración satisfactoria asigna 1 a la raíz de P .) La prueba es la misma para $\text{lowpt2}(v)$. \square

Los valores lowpt de un vértice v dependen solo de los valores lowpt de los hijos de v y en las ramas que salen de v ; es fácil ver que si los vértices se identifican por número, entonces

$$\text{lowpt1}(v) = \min(\{v\} \cup \{\text{lowpt1}(w) \mid v \rightarrow w\} \cup \{w \mid v \rightarrow w\})$$

y

$$\begin{aligned} \text{lowpt2}(v) = \min(\{v\} \cup ((\{\text{lowpt1}(w) \mid v \rightarrow w\} \\ \cup \{\text{lowpt2}(w) \mid v \rightarrow w\} \\ \cup \{w \mid v \rightarrow w\}) - \{\text{lowpt1}(v)\})). \end{aligned}$$

En el caso de la gráfica anterior los valores de lowpt1 y lowpt2 son los siguientes:

vertice	lowpt1	lowpt2
1	1	1
2	1	2
3	1	2
4	1	4
5	1	4
6	4	5
7	4	5
8	1	8
9	1	8
10	1	8
11	8	9
12	1	8
13	1	2

También tenemos $nd(v) = 1 + \sum_{v \rightarrow w} ND(w)$. Podemos calcular los valores de $lowpt$, nd y padre para todos los vértices en tiempo $\Theta(V + E)$ insertando las siguientes declaraciones para las declaraciones a, b, c en DFS. La numeración de los vértices en el orden en que se alcanzan durante la búsqueda garantiza claramente que $v \xrightarrow{*} w$ implica $v \leq w$.

Algoritmo 3.3: adiciones a DFS(G)

```

1 a: lowpt1( $v$ ) = lowpt2( $v$ ) = NUMERO( $v$ );
2   ND( $v$ ) = 1
3 b: si lowpt1( $w$ ) < lowpt1( $v$ )
4     lowpt2( $v$ ) = min{lowpt2( $v$ ), lowpt2( $w$ )}
5     lowpt1( $v$ ) = lowpt1( $w$ )
6   si no lowpt1( $w$ ) = lowpt1( $v$ )
7     lowpt2( $v$ ) = min{lowpt2( $v$ ), lowpt2( $w$ )}
8   en otro caso
9     lowpt2( $v$ ) = { lowpt2( $v$ ), lowpt1( $w$ )}
10  ND( $v$ ) = ND( $v$ ) + ND( $w$ )
11  FATHER( $w$ ) =  $v$ 
12 c: si NUMERO( $w$ ) < lowpt1( $v$ )
13     lowpt2( $v$ ) = lowpt1( $v$ )
14     lowpt1( $v$ ) = NUMERO( $w$ )
15   si no NUMERO( $w$ ) = lowpt1( $v$ )
16     lowpt2( $v$ ) = min{lowpt2( $v$ ), NUMERO( $w$ )}
```

Verificamos que DFS modificado anteriormente calculará $lowpt1$, $lowpt2$, ND y $FATHER$ correctamente en tiempo $\Theta(V + E)$. (Ver [14], [24].) $lowpt1$ puede usarse para probar la biconectividad de G , como se describe en [23]. El siguiente lema es importante.

Lema 3.7. Si G es biconexa y $v \rightarrow w$, $lowpt1(w) \leq v$ a menos que $v = 1$, en cuyo caso $lowpt1(w) = v = 1$. Además, $lowpt1(1) = 1$.

Demostración. Véase [5]. □

Sea ϕ el mapeo de las aristas de P en $\{1, 2, \dots, 2V + 1\}$ definido por:

$$\phi(x) = \begin{cases} 3lowpt1(w) & \text{si } e = v \rightarrow w \text{ y } lowpt2(w) < v \\ 3w + 1 & \text{si } e = v \leftrightarrow w \\ 3lowpt1(w) + 2 & \text{if } e = v \rightarrow w \text{ y } lowpt2(w) \geq v \end{cases}$$

Sea A una estructura de adyacencia para P . A se le dice aceptable si las aristas e en cada lista de adyacencia de A se ordena de forma creciente de acuerdo a $\phi(e)$.

Lo cual da el siguiente resultado de 3.2 en forma de mapeo a cada vertice $1, 2, \dots, 13$ los vertices adyacentes ordenados de acuerdo al ordenamiento de ϕ :

- 1: [2],
- 2: [3],
- 3: [13, 4],
- 4: [1, 5],
- 5: [8, 6],
- 6: [7, 4],
- 7: [4, 5],
- 8: [1, 9],
- 9: [10],
- 10: [12, 11],
- 11: [8, 9],
- 12: [1, 8, 9],
- 13: [1, 2]

Lema 3.8. *Sea P un árbol de una gráfica biconexa G cuyos vértices son numerados de manera que $v \xrightarrow{*} w$ en P implica $v \leq w$. Entonces las estructuras de adyacencia aceptables de P son independientes del esquema de numeración exacto.*

Demostración. Si $v \rightarrow w$ en P entonces por el lema 3.5, $\text{lowpt2}(w)$ es un ancestro de w . Por el lema 3.6 $\text{lowpt2}(w)$ es un vértice fijo independiente de la numeración. Ya que el orden de los ancestros es independiente de la numeración, la cuestión de si $\text{lowpt2}(w)$ es menor que v es independiente de la numeración. Como G es biconexa, si $v \rightarrow w$ en P entonces $\text{lowpt1}(w) \leq v$ por el lema 3.7. Por el lema 3.5, $\text{lowpt1}(w)$ es un ancestro de w . Dado que $\text{lowpt1}(w) \leq v$, $\text{lowpt1}(w)$ debe ser un ancestro de v . Por el lema 3.6, el vértice correspondiente a $\text{lowpt1}(w)$ es independiente de el esquema de numeración. De manera similar si $v \rightarrow w$, entonces por el lema 3.3 y la definición de árbol; w es un ancestro de v . Pero el orden de los ancestros de v es idéntico al orden de sus números y este orden es independiente de la numeración. Por lo tanto las estructuras de adyacencia aceptables A para P dependen solo de P y no de la numeración exacta \square

Ya despues de tener numeradas las aristas por esta función ordenamos las aristas. Hacemos el recorrido primero en profundidad sobre G

En el paso 3 de los cálculos realizamos una búsqueda en profundidad sobre P usando el estructura de adyacencia aceptable A dada por el paso 2. Esta búsqueda genera un conjunto de caminos de la siguiente manera: cada vez que atravesamos una arista, esta arista la agregamos al camino que se está construyendo. Cada vez que atravesamos una arista, la arista se convierte en la última arista del camino actual. Así, cada camino consta de una secuencia de ramas del árbol seguidos por un solo vértice. Debido al ordenamiento impuesto a A , cada camino termina en el vértice más bajo posible, el camino inicial es un ciclo, y cada camino excepto el primero es simple y sólo tiene en común su vértice inicial y terminal con las anteriores caminos generados ([14], [25]).

Si $p : s \xrightarrow{*} f$ es un camino generado, podemos formar un ciclo agregando el camino del árbol $f \xrightarrow{*} s$ a p . Los ciclos formados de esta manera son los ciclos generados por llamadas recursivas al algoritmo básico de triconectividad explicado en la última sección.

Solo necesitamos información mínima sobre los caminos. Sean los vértices de P numerados de manera que $v \xrightarrow{*} w$ implica $v \leq w$. Sea $A1(v)$ el primer vértice de $A(v)$. Si $v \rightarrow w$ es la primera rama explorada en el paso 3 que termina en w , sea $\text{highpt}(w) = v$. Sea $\text{deg}(v)$ el número de aristas que inciden en el vértice v . En el paso 3 numeramos los vértices de V a 1 en el orden en que se examinaron por última vez durante la búsqueda. Esta numeración garantiza que $v \leq w$ si $v \xrightarrow{*} w$. El paso 3 también calcula $\text{lowpt1}(v)$, $\text{highpt}(v)$, $A1(v)$ y $\text{deg}(v)$ con respecto a la nueva numeración.

El paso 3 enumera los vértices de V a 1 en el orden en que se alcanzaron por última vez durante la búsqueda. Sin embargo, a cada vértice en realidad se le debe asignar un número la primera vez que se alcanza, para que el cálculo de highpt se realice correctamente. Para lograr esto, la variable se establece igual a V cuando comienza la búsqueda (afirmación Z). El valor de i disminuye en uno cada vez que se descubre un nuevo vértice (afirmación Y). Así, cuando se alcanza por primera vez un vértice v , i es igual al número que desea asignar a v menos el número de vértices a encontrar antes de encontrar v por última vez. Pero los vértices a alcanzar entre el tiempo en que v se examinado por primera vez y el momento en que v se examinar por última vez son solo los descendientes propios de v . Por lo tanto, si asignamos el número $i - \text{nd}(v) + 1$ a v cuando v se examinar por primera vez (enunciado X), la numeración será correcta.

1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 13 \hookrightarrow 1$	2: $13 \hookrightarrow 2$
3: $3 \rightarrow 4 \hookrightarrow 1$	4: $4 \rightarrow 5 \rightarrow 8 \hookrightarrow 1$
5: $8 \rightarrow 9 \rightarrow 10 \rightarrow 12 \hookrightarrow 1$	6: $12 \hookrightarrow 8$
7: $12 \hookrightarrow 9$	8: $10 \rightarrow 11 \hookrightarrow 8$
9: $11 \hookrightarrow 9$	10: $5 \rightarrow 6 \rightarrow 7 \hookrightarrow 4$
11: $7 \hookrightarrow 5$	12: $6 \hookrightarrow 4$

Los caminos generados de las aristas ya ordenadas por phi en el paso 3 de la grafica 3.2.

Sea G una gráfica biconexa en el que se han realizado los pasos 1, 2 y 3, dando un árbol P y los conjuntos de valores definidos anteriormente. Sea A con una lista de adyacencia $A(v)$ la estructura de adyacencia aceptable construida en el paso 2. Sean los vértices de G identificados por los números asignados en el paso 3. Necesitamos una definición más. Si $u \rightarrow v$ y v es la primera entrada en $A(u)$, entonces a v se le llama el primer **hijo** de u . (Para cada vértice v , $A1(v)$, el primer hijo de v , si existe, se calcula en el paso 3). Si $u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_n$ y u_i es un hijo primogénito de u_{i-1} para $1 \leq i \leq n$, entonces a u_n , se le llama el primer descendiente de u_0 . La secuencia de ramas del árbol $u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_n$ es parte de un camino generado por el paso 3. Los lemas a continuación dan las propiedades que necesitamos para determinar los pares de separación de G .

Lema 3.9. *Sea $A(u)$ la lista de adyacencia del vértice u . Sean $u \rightarrow v$ y $u \rightarrow w$ ramas del árbol con v antes de w en $A(u)$. Entonces $u \leq w \leq v$.*

Demostración. El paso 3 numera los vértices de V a 1 en el orden en que son los últimos examinados en la búsqueda. Si $u \rightarrow v$ se explora antes que $u \rightarrow w$, v se examinarán en último lugar antes de que w se examine en último lugar, y v recibirá un número más alto. Claramente u será el último examinado después de que tanto v como w hayan sido examinados por última vez, por lo que u es el número más pequeño de los tres vértices. \square

Lema 3.10. *A es aceptable con respecto a la numeración dada por el paso 3.*

Demostración. El ordenamiento en el paso 2 crea una estructura de adyacencia aceptable para el numeración original. Por el lema 3.9, $u \rightarrow v$ implica $u \leq v$ y por lo tanto por el lema 3.8, A es aceptable para la nueva numeración. \square

Lema 3.11. *Si v es un vértice y $d(v)$ es el conjunto de descendientes de v , entonces $d(v) = \{x | v \leq x \leq v + nd(v)\}$. Si w es un primer descendiente de v , entonces $d(v) - D(w) = \{x | v \leq x \leq w\}$.*

Demostración. Supongamos que invertimos todas las listas de adyacencia $A(v)$ y las usamos para especificar una búsqueda en profundidad de P . Los vértices se examinarán por primera vez en sentido ascendente orden de 1 a V , si los vértices se identifican por su número del paso 3. Así a los descendientes de v se les asignan números consecutivos desde v a $v + nd(v) + 1$. Si w es el primer descendiente de v , los vértices en $d(w)$ se les asignarán números de acuerdo a los vértices en $d(v) - d(w)$. Así $d(v) - d(w) = \{x | v \leq x \leq w\}$. \square

Lema 3.12. *Sean $\{a, b\}$ un par de separación en G con $a \leq b$. Entonces $a \xrightarrow{*} b$ en el árbol generador T de P .*

Demostración. Como $a \leq b$, a no puede ser descendiente de b . Supongamos que b no es un descendiente de a . Sean E_i , para $1 \leq i \leq k$, las clases de separación con respecto a $\{a, b\}$. Sea $S = \mathcal{V} - D(a) - D(b)$. Los vértices S definen un subárbol en T que no contiene a a ni a b , por lo que $E(S)$ debe estar contenido en alguna clase de separación, digamos E_1 . Sea c cualquiera hijo de a . $E(D(c))$ debe estar contenido en alguna clase de separación. Pero como G es biconexa, y $a \neq 1$, $\text{lowpt1}(c) < a$, por el Lema 3.7. Por lo tanto, alguna arista incide en un vértice en S y a un vértice en $D(c)$. Así $E(D(c)) \subseteq E_1$. Un argumento similar demuestra que las aristas incidentes a cualquier descendiente de b están en E_1 . Pero esto significa que $E_1 = E$, y entonces a, b no puede ser un par de separación. \square

Lema 3.13. *Supongamos que $a < b$. Entonces $\{a, b\}$ es un par de separación de G si y solo si ya sea (i), (ii) o (iii) a continuación se cumple:*

1. Hay vértices distintos $r \neq a, b$ y $s \neq a, b$ tales que $b \rightarrow r$, $\text{lowpt1}(r) = a$, $\text{lowpt2}(r) \geq b$ y s no es un descendiente de r . (El par $\{a, b\}$ se le llama par de separación "tipo 1". Los pares de tipo 1 en 3.2 son $(1, 8), (1, 5), (4, 5), (1, 4), (1, 3)$).
2. Hay un vértice $r \neq b$ tal que $a \rightarrow r \xrightarrow{*} b$; b es el primer descendiente de r (es decir, a, r y b se examinan en una ruta generada); $a \neq 1$; cada rama $x \rightarrow y$ con $r \leq x \leq b$ y $a \leq y$ y cada rama $x \rightarrow y$ con $a \leq y \leq b$ y $b \rightarrow w \xrightarrow{*} x$ tiene $\text{lowpt1}(w) \geq a$. ($\{a, b\}$ se denomina par de separación de "tipo 2". Los pares de tipo 2 en 3.2 son $(8, 12)$).

Demostración. La parte inversa del lema es más fácil de probar. Supongamos que un par $\{a, b\}$ satisface (i), (ii) o (iii). Sea E_i para $1 \leq i \leq k$ las clases de separación de G con respecto a $\{a, b\}$. Supongamos que $\{a, b\}$ satisface (i). Entonces la arista (b, r) está contenido en alguna clase de separación, digamos E_1 . Cada rama del árbol con un vértice final en $d(r)$

tiene el otro extremo en $d(r) \cup \{a, b\}$. Además, dado que $\text{lowpt1}(r) = a$ y $\text{lowpt2}(r) \geq b$, toda rama con una vértice final en $D(r)$ tiene la otra vértice final en $D(r) \cup \{a, b\}$. Por lo tanto, E_1 consta de todas las aristas con un vértice final en $D(r)$. No hay otras aristas en E_1 , y las aristas incidentes al vértice s deben estar en alguna otra clase, digamos E_2 . Ya que E_1 y E_2 contienen cada uno dos o más aristas, $\{a, b\}$ es un par de separación.

Supongamos que $\{a, b\}$ satisface (ii). Sea $S = d(r) - d(b)$. Todas las aristas incidentes a un vértice en S están en la misma clase de separación, digamos E_1 . Dado que b es un primer descendiente de r , $S = \{x | r \leq x \leq b\}$ por el lema 3.1. Sean b_1, b_2, \dots, b_n los hijos de b en el orden ocurren en $A(b)$. Sea $i_0 = \min\{i | \text{lowpt1}(b_i) \geq a\}$. Por el ordenamiento impuesto en A , $i \leq i_0$ implica $\text{lowpt1}(b_i) \leq a$, y $i \geq i_0$ implica que $\text{lowpt1}(b_i) \geq a$. Por (ii), cada arista de retroceso con vertice inicial en S tiene su vertice final en $S \cup \{a\}$. También por (ii), cada arista de retroceso con vertice final en S tiene su vertice inicial en $S \cup \{b\} \cup (\bigcup_{i \geq i_0} d(b_i))$. Cada arista con un vertice final en $d(b)$, $i \geq i_0$, tiene su otro extremo en $S \cup \{a, b\} \cup d(b)$. Así la clase E_1 contiene al menos todas las aristas con un vertice final en S y como máximo todas las aristas con un vertice final en $S \cup (\bigcup_{i \geq i_0} d(b_i))$. Como $a \neq 1$, las aristas incidentes a la raíz de P no pueden estar en E_1 y por lo tanto $\{a, b\}$ es un par de separación.

Ahora debemos demostrar la parte directa del lema. Por 3.12, $a \xrightarrow{*} b$. Sea E_i , para $1 \leq i \leq k$, las clases de separación de G con respecto a $\{a, b\}$. Sea v el hijo de a tal que $a \rightarrow v \xrightarrow{*} b$, $S = d(v) - d(b)$, y $X = V - d(a)$. (O S o X o ambos pueden estar vacíos.). $E(S)$ y $E(X)$ están contenidos cada uno en una clase de separación, digamos $E(S) \subseteq E_1$ y $E(X) \subseteq E_2$.

Sea $a_i \neq v$ un hijo de a . Si a tiene un hijo así, $\text{lowpt1}(a) \leq a$. Esto significa que $E(d(a_i)) \subseteq E_2$. Sean $Y = X \cup (\bigcup_i d(a_i))$. Sean b_1, b_2, \dots, b_n los hijos de b en el orden en que aparecen en la lista de adyacencia de b . Sea $E(d(b))$ el conjunto de aristas con vertice final en $d(b)$. Las clases de separación deben ser uniones de los conjuntos $E(S), E(Y), \{(a, b)\}, E(d(b_1)), E(d(b_2)), \dots, E(d(b_n))$.

Si $E(d(b)) = E_j$ para algún i y j , entonces $\text{lowpt1}(b_i) = a$ ya que G es biconexo y esto significa que $\text{lowpt1}(b_i) \leq b$ por el Lema 3.7. También, $\text{lowpt2}(b_i) \geq b$. Ya que $\{a, b\}$ es un par de separación, debe haber una clase de separación distinta de E_j y $\{(a, b)\}$. Por tanto, existe un vértice s tal que $s \neq a$, $s \neq b$, y $s \notin d(b_i)$. Esto significa que $\{a, b\}$ satisface (i) donde r es b_i .

Supongamos ahora que ninguna $E(d(b))$ es una clase de separación. Sea $i_0 = \min\{i | \text{lowpt1}(b_i) \geq a\}$. Si $i \geq i_0$, entonces como G es biconexo, debe ser el caso que $\text{lowpt1}(b) \leq b$, y estas clases de separación son $E_1 = E(S) \cup (\bigcup_{i \geq i_0} E(d(b_i)))$, $E_2 = E(Y) \cup (\bigcup_{i \leq i_0} E(d(b_i)))$, $E_3 = \{(a, b)\}$. (E_3 puede estar vacío.) Tenemos $v \neq b$ ya que $\{a, b\}$ no es un par de tipo 1 y $a \neq 1$ ya que E_2 no es vacío. Si $x \rightarrow y$ es una rama con $v \leq x \leq b$, entonces $x \in S$, $(x, y) \in E_1$, y $a \geq y$. Si $x \rightarrow y$ es una rama con $a \leq y \leq b$ y $b \rightarrow b_i \xrightarrow{*} x$, entonces $y \in S$, $(x, y) \in E_1$ y $i \geq i_0$, lo que significa que $\text{lowpt1}(b_i) \geq a$. Debemos verificar una condición más para demostrar que (ii) se cumple, es decir, que b es un primer descendiente de v . Dado que G es biconexa, $\text{lowpt1}(v) \leq a$. Por lo tanto alguna arista con vertice inicial $d(v)$ tiene un vertice final $< a$. Por el ordenamiento impuesto a AS y la

definición de un primer descendiente, existe alguna arista de retroceso $x \rightarrow y$ con $x \in d(v)$ y $y \leq a$ tal que x es el primer descendiente de v . Si b no fuera el primer descendiente de v , entonces x estaría en S , y E_1 y E_2 no podrían ser clases de separación distintas. Por lo tanto, b es un primer descendiente de v , y (ii) se cumple con $r = v$. Esto completa la prueba de la parte directa del lema. \square

Vale la pena considerar cuidadosamente ?? y su demostración. El lema da tres condiciones fáciles de aplicar para pares de separación. Las condiciones (i) y (ii) identifican los pares de separación no triviales de la gráfica. La condición (iii) maneja aristas múltiples. La condición (i) requiere que se realice una prueba simple en cada rama del árbol P . Por lo tanto, la prueba de pares de tipo 1 requiere tiempo $\Theta(V)$. La prueba de pares de tipo 2 es algo más difícil, pero se puede hacer en tiempo $\Theta(V + E)$ usando otra búsqueda primero en profundidad. Sea $\{a, b\}$ un par de tipo 2 que satisfaga $a \rightarrow r \xrightarrow{*} b$ y $i_0 = \min\{i | \text{lowpt2}(b_i) \geq a\}$, donde b_1, b_2, \dots, b_n , son los hijos de b en el orden en que aparecen en $A(b)$. Entonces una clase de separación con respecto a $\{a, b\}$ es $E(\{x | r \leq x \leq b_{i_0} + nd(b_{i_0})\} - \{b\})$. Esto se sigue de la demostración de 3.13. La nueva numeración satisface la condición un tanto extraña en 3.9 por lo que hace fácil determinar la clases de separación y separar la gráfica cuando se encuentra un par de separación. Un algoritmo para encontrar los componentes de separación basado en 3.13 se da en el siguiente sección.

3.2.2. Encontrando componentes de separación

Encontramos los componentes de separación al examinar los caminos generados en orden y probamos los pares de separación con 3.13. Los pares de separación serán de varios tipos. Pares de tipo 1 son mas faciles de reconocer. También lo son los pares de tipo 2 $\{a, b\}$, donde $a \rightarrow v \rightarrow b$ y v tiene grado dos. Otros pares de tipo 2 son algo más difíciles de reconocer. Sea c el primer camino generado (un ciclo). El ciclo consta de un conjunto de aristas del árbol $1 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n$ seguido de una rama $v_n \rightarrow 1$. La numeración de los vértices es tal que $1 \leq v_1 \leq \dots \leq v_n$. Cuando se elimina c , la gráfica se divide en varias partes conectadas, llamadas **segmentos**. Cada segmento consta de una sola arista (v_i, v_j) o de una arista del árbol (v_i, w) más un subárbol con raíz w más todas las aristas que salen del subárbol. El orden del camino generado es tal que todas las rutas en un segmento se generan antes que las rutas en cualquier otro segmento, y los segmentos se exploran en orden decreciente de v_i . Supongamos que repetimos la búsqueda del camino, usándola ahora para encontrar los componentes de separación. Mantendremos una pila de aristas, agregando aristas a esta pila a medida que retrocedemos sobre ellos durante la búsqueda. Cada vez que encontramos un par de separación, eliminamos un conjunto de aristas de la pila correspondiente a un componente de separación. Agregamos una arista virtual correspondiente a la separación tanto al componente como a la pila de aristas. Nosotros también necesitamos actualizar varias piezas de información, ya que los padres de vértices y los grados de los vértices pueden cambiar cuando se divide una gráfica. El camino completo de la búsqueda creará un conjunto completo de componentes de separación. Unir los componentes de separación para dar los componentes triconexos es el paso final.

Para identificar los pares de tipo 2, mantenemos una pila (llamada tstack) de tripletas (h, a, b) . El par $\{a, b\}$ es un posible par de tipo 2 y h denota el vértice numerado más grande en el componente de separación correspondiente. Los pares están en orden anidado en la pila; es decir, si v_i es el vértice actual que está siendo examinado por la búsqueda de ruta y $(h_1, a_1, b_1), (h_2, a_2, b_2), \dots, (h_k, a_k, b_k)$ están en tstack, entonces $a_k \leq a_{k-1} \leq \dots \leq a_2 \leq$

$a_1 \leq v_i \leq b_1 \leq b_2 \leq \dots \leq b_k$. Además, todos los a_j y b_j son vértices. en el ciclo c .

Se actualiza tstack de las siguientes maneras:

1. Cada vez que recorremos un nuevo camino $p : s \xrightarrow{*} f$, borramos todos las tripletas (h_j, a_j, b_j) encima de la pila con $a \geq f$. Si p tiene un segundo vértice $v \neq f$, sea $x = v + ND(v) + 1$. De lo contrario, sea $x = s$. Sea $y = \max\{h_j | \text{triple}(h_j, a_j, b_j) \text{ que se eliminó de tstack}\}$. Si (h_k, a_k, b_k) fue la última tripleta eliminada, agregamos $(\max(x, y), f, s)$ a la pila. Si no se eliminó una tripleta, agregamos (x, f, s) a la pila.
2. Cuando retrocedemos sobre una rama del árbol $v_i \rightarrow v_{i+1}$ con $v \neq 1$, borramos todas las entradas (h_j, a_j, b_j) encima de tstack que satisfacen $\text{highpt}(v_i) > h_j$. esta prueba es necesaria para garantizar que las entradas que no correspondan a los pares de tipo 2 no se acumulan en tstack.

Usamos tstack para encontrar pares de separación de la siguiente manera: siempre que busquemos hacia atrás a lo largo de una rama del árbol $v_i \rightarrow v_{i+1}$ durante la búsqueda de ruta, examinamos la tripleta superior en (h_1, a_1, b_1) en tstack. Si $v \neq 1$, $a = v_i$, y $a \neq \text{padre}(b_i)$, $\{a_1, b_1\}$ es un par de separación de tipo 2. Si $\deg(v+1) = 2$ y v_{i+1} tiene un hijo, entonces v_i y el hijo de v_{i+1} forman un par de separación tipo 2. Dividimos los componentes correspondientes a los pares de tipo 2 hasta que estas dos condiciones no nos den más componentes. (Simultáneamente, probamos los componentes correspondientes a múltiples aristas y los dividimos.) Luego aplicamos el Lema 3.13 para probar si $\{v, \text{lowpt1}(v+1)\}$ es un par de tipo 1, dividiendo un componente si es necesario. (De nuevo, tenemos que comprobar si hay un componente de múltiples aristas).

Manejamos la parte recursiva del algoritmo de la siguiente manera: recorremos una ruta $p : s \xrightarrow{*} f$ que comienza en c significa que la búsqueda está entrando a un nuevo segmento. El vértice f debe ser el vértice más bajo en el segmento por el orden impuesto en la búsqueda de caminos. Después de actualizar tstack como se describe arriba, si p contiene más de una arista, colocamos un marcador de fin de pila en tstack y continuamos encontrando caminos. Esto corresponde a una llamada recursiva de la triconectividad básica del algoritmo. Cuando retrocedemos sobre el primer arista de p , eliminamos todas las entradas de tstack hasta el marcador de fin de la pila. esto corresponde a apareciendo de la recursividad.

Un punto más necesita explicación: la razón por la que usamos lowpt2 así como lowpt1 para construir A , la estructura de adyacencia aceptable que determina el orden de búsqueda de caminos. Este paso es necesario para que todas las aristas múltiples sean manejadas correctamente. Supongamos que v es un vértice, y w_1, w_2, \dots, w_k son los hijos de v tales que $\text{lowpt1}(w_i) = u$. Además, supongamos que $v \rightarrow u$. Sea la w_i ordenada como en $A(v)$. Hay algún i_0 tal que $i \leq i_0 \Rightarrow \text{lowpt2}(w_i) \leq v$ y $i \geq i_0 \Rightarrow \text{lowpt2}(w_i) \geq v$. En $A(v)$, u aparecerá después de el w_i con $1 \leq i \leq i_0$. Si $i \geq i_0$, entonces $\{u, w_i\}$ es un par de separación tipo 1; dividiendo los correspondientes componentes produce una nueva arista(virtual) $v \rightarrow u$. Es importante que todos los w_i con $i \geq i_0$ aparecen juntos en $A(v)$ para que estas aristas virtuales puedan ser localizadas y combinadas para dar componentes de separación que son enlaces.

Los siguientes algoritmos son dados en [11]

Algoritmo 3.4: SPLIT(G)

agregamos (EOS) a tstack
 PATHSEARCH(1)
 sea e_1, e_2, \dots, e_ℓ aristas en ESTACK
 3.1 \mathcal{C} = nuevo componente (e_1, \dots, e_ℓ)

Algoritmo 3.5: PATHSEARCH(v)

para $e \in A(v)$:
 si $e = v \rightarrow w$
 si e empieza un camino
 eliminar todos los (h, a, b) con $a > lowpt1(w)$ de tstack
 si no se eliminaron tripletas
 agregar a tstack $(w + ND(w) - 1, lowpt1(w), v)$
 en otro caso
 $y = \max\{h \mid (h, a, b) \text{ eliminados de tstack}\}$
 sea (h, a, b) la ultima tripleta eliminada
 agregar a tstack $(\max(y, w + ND(w) - 1), lowpt1(w), b)$
 agregar a tstack (EOS)
 PATHSEARCH(w)
 agregamos a ESTACK ($v \rightarrow w$)
 buscamos pares de tipo 2
 buscamos pares de tipo 1
 si e empieza un camino
 quitamos las tripletas en tstack hasta el EOS
 4.1 **mientras** (h, a, b) en tstack tiene $a \neq v$ y $b \neq v$ y $highpt(v) > h$
 eliminamos (h, a, b) de tstack
 en otro caso
 sea $e = v \hookrightarrow w$
 si e empieza un camino
 eliminar todos los (h, a, b) con $a > w$ de tstack
 si no se eliminaron tripletas
 agregamos (v, w, v) a tstack
 en otro caso
 $y = \max\{h \mid \text{eliminado de tstack}\}$
 sea (h, a, b) la ultima tripleta eliminada
 agregamos (y, w, b) a tstack
 si $w \in parent(v)$
 \mathcal{C} = nuevo componente ($e, w \rightarrow v$)
 e' = nueva arista virtual (w, v, \mathcal{C})
 crear arista virtual ($e', w \rightarrow v$)
 en otro caso
 agregamos e a tstack

Algoritmo 3.6: checkTipo2(v)

mientras $v \neq 1$ y $((h, a, b)$ en $tstack$ tiene $a = v$) o $(deg(v) = 2$ y $A1(w) > w$)
si $a = v$ y $FATHER(b) = a$
 eliminamos el primero de $tstack$
en otro caso
 $e_{ab} = nulo$ **si** $DEGREE(w) = y$ $A1 > w$
 \mathcal{C} = nuevo componente
 quitamos las primeras aristas (v, w) y (w, b) de $ESTACK$ y las agregamos a \mathcal{C}
 $e' =$ nueva arista virtual (v, x, \mathcal{C})
si eliminamos (v, b)
 e_{ab} es el primer elemento que eliminamos de $ESTACK$
en otro caso
 (h, a, b) = primer elemento eliminado de $tstack$
 \mathcal{C} = nuevo componente
mientras (x, y) en $ESTACK$ tiene $a \leq x \leq h$ y $a \leq y \leq h$
si $(x, y) = (a, b)$
 e_{ab} es el primer elemento que eliminamos de $ESTACK$
en otro caso
 $\mathcal{C} = \mathcal{C} \cup \{ \text{primer elemento eliminado de } ESTACK \}$
si $e_{ab} = nulo$
 \mathcal{C} = nuevo componente (e_{ab}, e') $e' =$ nueva arista virtual (v, b, \mathcal{C})
 agregamos e' a $ESTACK$
 creamos la arista del árbol $(e', v \rightarrow b)$
 $w = b$

Algoritmo 3.7: checkTipo1(v)

6.1 **si** $lowpt2(w) \geq v$ y $lowpt1(w) < v$ y $(FATHER(v) \neq 1)$ o v es adyacente a una rama de el árbol que aún no has sido visitada :
 \mathcal{C} = nuevo componente
mientras (x, y) en $ESTACK$ tiene $w \leq x \leq w + ND(w)$ o $w \leq y \leq w + ND(w)$
 $\mathcal{C} = \mathcal{C} \cup \{ESTACK.pop()\}$
 $e' =$ nueva arista virtual $(v, lowpt1(w), \mathcal{C})$
si $ESTACK.pop() = (v, lowpt1(w))$
 \mathcal{C} = nuevo componente $(ESTACK.pop(), e')$
 $e' =$ nueva arista virtual $(v, lowpt1(w), \mathcal{C})$
si $lowpt1(w) \neq FATHER(v)$
 agregar e' a $ESTACK$
 crear arista de l árbol $(e', lowpt1(w) \rightarrow v)$
en otro caso
 \mathcal{C} = nuevo componente $(e', lowpt1(w) \rightarrow v)$
 $e' =$ nueva arista virtual $(lowpt1(w), v, \mathcal{C})$
 crear arista de l árbol $(e', lowpt1(w) \rightarrow v)$

Lema 3.14. *SPLIT divide correctamente un gráfica biconexo G en sus componentes de separación.*

Demostración. Debemos probar dos cosas (i) si G es triconexo, SPLIT no lo divide y (ii) si G no es triconexo, el algoritmo lo separará. Una vez que tengamos estos dos hechos, podemos demostrar el lema por inducción sobre el número de aristas de la gráfica. Las pruebas para múltiples aristas, para pares de separación de tipo 1 y para vértices de grado 2 son sencillos. (La prueba de tipo 1 (G en PATHSEARCH) incluye la condición ($\text{lowpt1}(w) \neq 1$) o ($\text{padre}(v) \neq 1$) o ($w \geq 3$) para asegurarse que algún vértice está afuera del componente de separación correspondiente.) Estas pruebas descubren un par de separación del tipo correcto, si existe, y no informarán un par de separación si no existe uno. Por lo tanto, solo debemos mostrar que la prueba para el tipo 2 funciona correctamente en gráficas sin vértices de grado dos o pares de separación tipo 1 y habremos verificado (i) y (ii).

Supongamos que G es una gráfica biconexo sin vértices de grado dos, o pares de separación tipo 2. Consideremos la prueba de tipo 2 y el cambio contenido de tstack a medida de que avanza la búsqueda en G . Si $(h_1, a_1, b_1), \dots, (h_k, a_k, b_k)$ son los contenidos de tstack por encima del marcador de fin de pila más alto y si v es el vértice que se está examinando actualmente durante la búsqueda, entonces $a_k \leq a_{k-1} \leq \dots \leq a_1 \leq v \leq b_1 \leq \dots \leq b_k$. Esto se sigue por inducción de un examen de la posibles cambios que se pueden hacer en tstack (declaraciones A, B, C, D, E, F en PATHSEARCH). Además, $a_k, a_{k-1} \dots b_k$ se encuentran en el ciclo correspondiente a la llamada recursiva actual del algoritmo básico de triconectividad. Suponga que (h, a, b) en tstack satisface la prueba de tipo 2 cuando el la búsqueda regresa a lo largo de una rama del árbol $v \rightarrow w$. La prueba (B, E en PATHSEARCH) establece que $a = v$, $v \neq 1$ y $\text{padre}(b) \neq a$. Se sigue que $r = A1(a) \neq b$ satisface $a \rightarrow r \xrightarrow{*} b$ y que b es un primer descendiente de r (es decir, a, r y b se examinaron en un mismo camino generado). Si alguna rama $x \rightarrow y$ con $r \leq x \leq b$ tuviera $a \geq y$, la tripleta en tstack correspondiente a (h, a, b) se habría eliminado de tstack cuando se exploró la arista (A o F en PATHSEARCH). Similarmente, si alguna rama $x \rightarrow y$ con $a \leq y \leq b$ y $b \rightarrow w \xrightarrow{*} x$ tenía $\text{lowpt1}(w) \leq a$, la tripleta en tstack correspondiente a (h, a, b) habría sido eliminado por la prueba highpt cuando se examinó el vértice y (D en PATHSEARCH). Se sigue que $\{a, b\}$ es un par de separación tipo 2 por 3.13.

Por el contrario, supongamos que G tiene un par $\{a, b\}$ de tipo 2. Sean b_1, b_2, \dots, b_n , los hijos de b en el orden en que aparecen en $A(b)$. Sea $i_0 = \min\{i \mid \text{lowpt1}(b_i) \geq a\}$. Si i_0 existe, entonces $(b_{i_0} + \text{ND}(b_{i_0}), \text{lowpt1}(b_{i_0}), b)$ se colocará en tstack cuando la rama del árbol $b \rightarrow b_i$ se explora. Esta tripleta puede ser borrado de tstack, pero siempre será reemplazado por una tripleta de la forma (h, x, b) , con $\text{lowpt1}(b) \geq x \geq a$. Finalmente dicha tripleta satisfará la prueba de tipo 2, a menos que se encuentre primero algún otro par de tipo 2. Si i_0 no existe, sea (i, j) la primera arista recorrida después de llegar a b tal que $a \leq i$ y $j \leq b$. Si $i \rightarrow j$, entonces (i, j, i) se colocará en tstack, posiblemente modificado y eventualmente seleccionado como un par de tipo 2, a menos que algún otro par de tipo 2 sea examinado primero. Si $i \rightarrow j$, entonces $(j + \text{ND}(j), \text{lowpt1}(j), i)$ se colocará en tstack, posiblemente modificado y eventualmente seleccionado como un par de tipo 2 a menos que algún otro par de tipo 2 se examine primero. Por lo tanto, si existe algún par de tipo 2, al menos un par de tipo 2 será encontrado por el algoritmo. De ello se deduce que la prueba de tipo 2 funciona correctamente, y el algoritmo divide un gráfica si y solo si existe un par de separación.

El lema se sigue por inducción sobre el número de aristas en G . Supongamos que el lema es cierto para gráficas con menos de k aristas. Sea G una gráfica con k aristas. Si G no puede ser dividido, el algoritmo funciona correctamente en G por el argumento anterior. Si G se puede separar, se separará. Consideremos la primera separación realizada por el algoritmo, produciendo gráficas de separación G_1 y G_2 . El comportamiento del algoritmo en G es un compuesto de su comportamiento en G_1 y G_2 . Dado que el algoritmo divide G_1 y G_2

correctamente por la hipótesis de inducción, debe separar G correctamente. El lema se sigue por inducción. \square

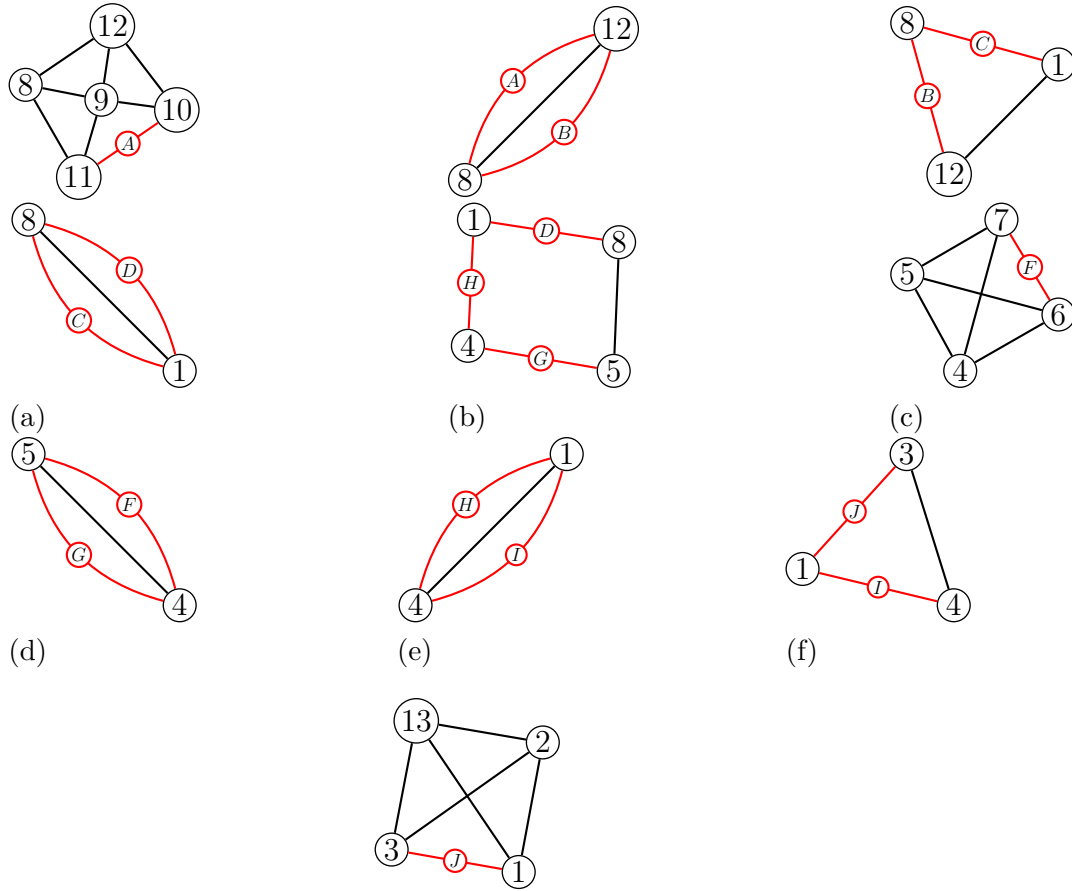


Figura 3.5: Componentes triconexas finales de la gráfica 3.2

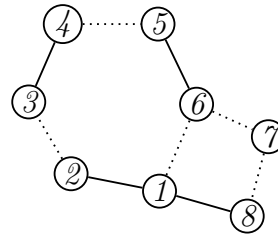
Lema 3.15. *El algoritmo de componentes triconexas procesa una gráfica G con V vértices y E aristas en tiempo $\Theta(V + E)$.*

Demostración. El número de aristas en un conjunto de componentes de separación de G está acotado por $3E - 6$, por el lema 3.1. Todos los pasos, excepto encontrar componentes de separación, requiere un tiempo $\Theta(V + E)$, por los resultados de las dos ultimas secciones.. Considere la ejecución de algoritmo SPLIT. Cada arista se coloca en ESTACK una vez y se elimina una vez. La búsqueda en profundidad en sí misma requiere un tiempo $\Theta(V + E)$, incluidas las diversas pruebas. El número de tripletas agregadas a tstack es $\Theta(V + E)$. Cada tripleta sólo puede modificarse si está encima de la pila. Por lo tanto, el tiempo necesario para mantener tstack es también $\Theta(V + E)$ y SPLIT requiere de tiempo $\Theta(V + E)$. \square

4.1. Componentes triconexas en gráficas de tipo \mathbb{D}_n

Antes de entrar en detalle con el tema de nuestro interes se definen dos conceptos que nos ayudaran a hacer la clasificación.

Definición 4.1. Una bigráfica cumple la **condición de ciclo** si todo ciclo tiene un número impar de aristas punteadas.



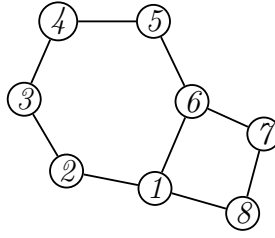
Ejemplo 4.1.

Una pareja de aristas paralelas se considera un ciclo de longitud 2.

- Una bigráfica cíclica $H = x_1 - x_2 - \cdots - x_h - x_1$ (todos los x_i distintos para $1 \leq i \leq h$) que satisface la condición de ciclo.
- A esta bigráfica H le llamaremos el \mathbb{D} -núcleo.

Definición 4.2. El **marco** $\Phi(G)$ de una bigráfica G es su gráfica subyacente.

Su diagrama se obtiene reemplazando aristas punteadas por sólidas.



Ejemplo 4.2.

4.1.1. Idea general de la clasificación algorítmica de \mathbb{D}_n

Aquí daremos la idea general de el algoritmo de clasificación cuando tenemos un ciclo > 2 :

1. Descomponemos la bigráfica en sus componentes biconexas.
2. Si todas excepto una son componentes biconexas entonces:
 - a) A la componente que no es biconexa aplicamos el algoritmo de componentes triconexas a el marco de la bigráfica.
 - b) Ya que obtengamos las componentes triconexas finales regresamos las aristas a su forma original.
 - c) a los enlaces les quitamos las aristas paralelas y hacemos que la arista sea solida.
 - d) y para las aristas virtuales de los poligonos o componentes triconexas con la misma etiqueta hacemos lo siguiente:
 - 1) Calculamos el camino mas corto hacia la arista virtual si el numero de aristas punteadas es par la arista virtual la hacemos punteada si el número es impar la arista virtual la hacemos solida.
 - e) Ya que tengamos una arista virtual definida si existe otra arista con la misma etiqueta solo cambiamos el tipo de arista en esa etiqueta (solida \rightarrow punteada, punteada \rightarrow solida).
 - f) Si tenemos que todas las componentes triconexas finales excepto una son \mathbb{A}_n entonces:
 - 1) Verificamos que la componente triconexa que no es un \mathbb{A} -bloque cumpla la condición de ciclo(si el ciclo es > 2).
 - 2) Si lo cumple entonces podemos decir que la bigráfica es de tipo \mathbb{D}_n con $n = |V|$

Si tenemos un ciclo $= 2$ entonces nuestras componentes triconexas van a ser necesariamente \mathbb{A} -bloques.

4.1.2. Clasificación algorítmica de \mathbb{D}_n

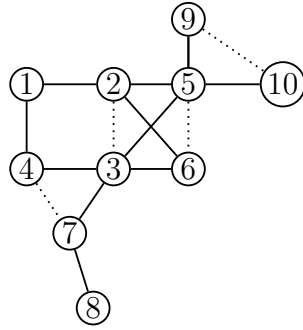
En [21] existe una implementación de el algoritmo de componentes triconexas en el lenguaje Python la cual verifica si es una gráfica triconexa y te devuelve las componentes triconexas finales de la gráfica ingresada solo funciona con gráficas por lo cual utilizaremos el marco de la bigráfica para poder utilizar este algoritmo y ya que tengamos las componentes finales volvemos las aristas a su forma original a las aristas virtuales si son punteadas o solidas lo definiremos conforme a la idea general.

Hay dos casos principales que ver en este problema de clasificación:

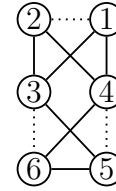
- con ciclo > 2
- con ciclo $= 2$

Las bigráficas con ciclo > 2 son mas faciles de ver que las bigráficas con ciclo $= 2$ A continuación veremos unos ejemplos de estos casos y como [21] lo descompone en sus componentes triconexas.

Vamos a describir el proceso de clasificación en los siguientes ejemplos:



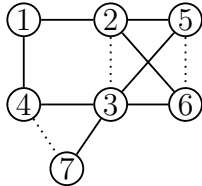
(a) Gráfica G_1 con ciclo > 2



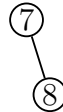
(b) Gráfica G_2 con ciclo $= 2$

Figura 4.1: Ejemplos de gráficas de tipo \mathbb{D}_n

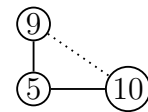
Descomponemos en su componentes biconexas:



(a) $G_{1,1}$



(b) $G_{1,2}$



(c) $G_{1,3}$

Figura 4.2: Descomposición de G_1 en sus componentes biconexas

Arboles generados de el recorrido en profundidad de la componente biconexa que no es de tipo \mathbb{A}_n de las gráficas:

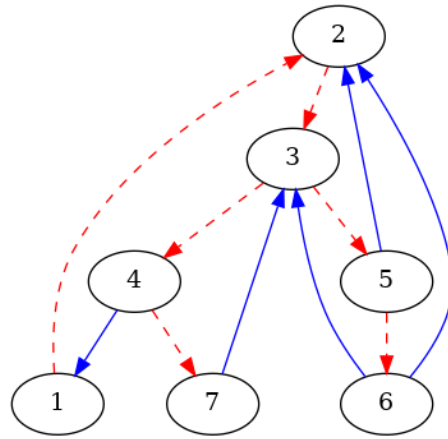


Figura 4.3: Árbol generado en el recorrido primero en profunda en $G_{1,1}$

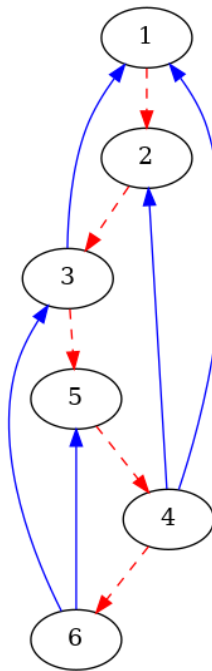


Figura 4.4: Árbol generado en el recorrido primero en profunda en G_2

Calculamos los lowpt1 y lowpt2 de cada gráfica:

vertice	lowpt1	lowpt2
1	1	1
2	1	2
3	1	2
4	1	3
5	2	3
6	2	3
7	3	5

Tabla 4.1: Calculos de lowpt1 y lowpt2 en $G_{1,1}$

vertice	lowpt1	lowpt2
1	1	1
2	1	2
3	1	2
4	1	2
5	1	2
6	3	4

Tabla 4.2: Calculos de lowpt1 y lowpt2 en G_2

El ordenamiento de ϕ en $G_{1,1}$ da el siguiente resultado:

- 1: [2],
- 2: [3],
- 3: [4, 5],
- 4: [1, 7],
- 5: [6, 2],
- 6: [2, 3],
- 7: [3]

Los caminos generados de $G_{1,1}$ son: 1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \hookrightarrow 1$ 2: $4 \rightarrow 7 \hookrightarrow 3$
3: $3 \rightarrow 5 \rightarrow 6 \hookrightarrow 2$ 4: $6 \hookrightarrow 3$
5: $5 \hookrightarrow 2$

Solo hay pares de separación de tipo 1 en $G_{1,1}$ los cuales son $(3, 4), (1, 3), (2, 3)$.

El ordenamiento de ϕ en G_2 da el siguiente resultado:

- 1: [2],
- 2: [3],
- 3: [5, 1],
- 4: [1, 2, 6],
- 5: [4],
- 6: [3, 5]

Los caminos generados de G_2 son:

1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \hookrightarrow 1$	2: $4 \hookrightarrow 2$
3: $4 \rightarrow 6 \hookrightarrow 3$	4: $6 \hookrightarrow 5$
5: $3 \hookrightarrow 1$	

Solo hay un par de separación de tipo 2 en G_2 el cual es $(3, 4)$.

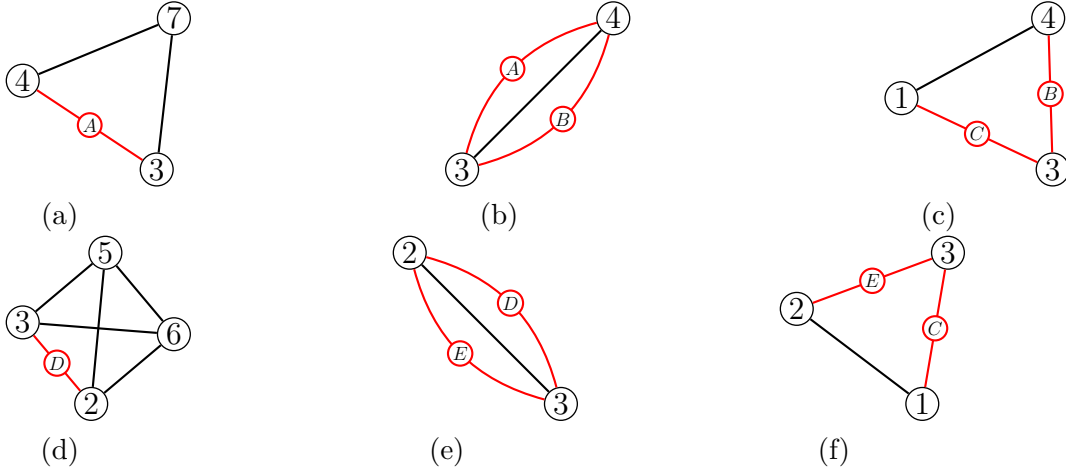


Figura 4.5: Componentes de separación de $G_{1,1}$

En el caso de G_2 los componentes de separación son iguales a las componentes triconexas finales.

Las componentes triconexas finales de las bigráficas:

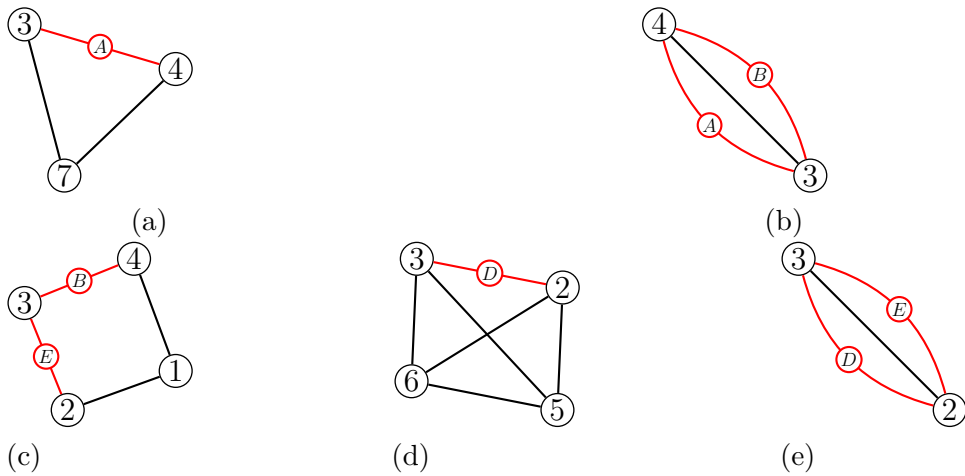


Figura 4.6: Componentes triconexas de $G_{1,1}$



Figura 4.7: Componentes triconexas de G_2

Obtenemos ahora la bigráfica original y aplicamos el criterio para decidir si la arista virtual es solida o punteada.

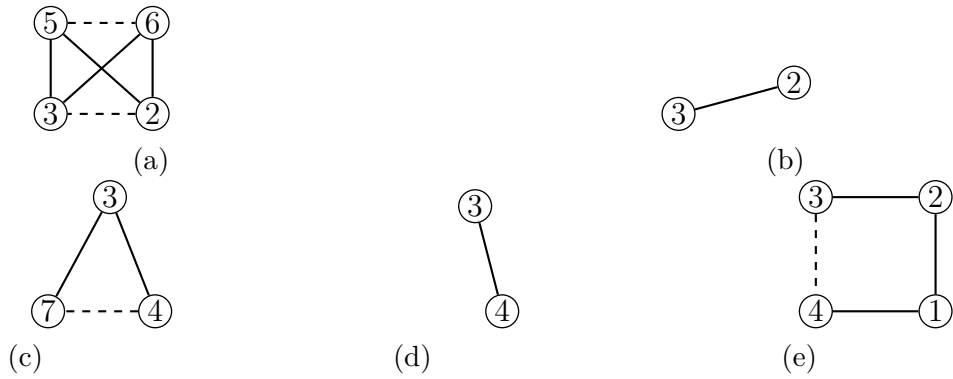


Figura 4.8: Componentes triconexas de $G_{1,1}$

De el capítulo 2 sabemos que estas gráficas excepto (e) son \mathbb{A} -bloques.



Figura 4.9: Componentes triconexas de G_2

Componentes triconexas en un bigrafo \mathbb{D}_n .

5.1. Algoritmo para encontrar componentes triconexas de un bigrafo de tipo \mathbb{D}_n

```

1  class _LinkedListNode:
2
3      def __init__(self, data=None):
4
5          self.data = data
6          self.prev = None
7          self.next = None
8          self.set_data(self.data)
9
10     def set_data(self, data):
11         self.data = data
12
13     def get_data(self):
14         return self.data
15
16
17 class _LinkedList:
18
19     def __init__(self):
20
21         self.head = None
22         self.tail = None
23         self.length = 0
24
25     def remove(self, node):
26
27         if node.prev is None and node.next is None:
28             self.head = None
29             self.tail = None
30         elif node.prev is None:

```

```

15         self.head = node.next
16         node.next.prev = None
17     elif node.next is None:
18         node.prev.next = None
19         self.tail = node.prev
20     else:
21         node.prev.next = node.next
22         node.next.prev = node.prev
23     self.length -= 1
24
25     def set_head(self, h):
26
27         self.head = h
28         self.tail = h
29         self.length = 1
30
31     def append(self, node):
32
33         if self.head is None:
34             self.set_head(node)
35         else:
36             self.tail.next = node
37             node.prev = self.tail
38             self.tail = node
39             self.length += 1
40
41     def get_head(self):
42         return self.head
43
44     def get_length(self):
45         return self.length
46
47     def push_front(self, node):
48
49         if self.head is None:
50             self.set_head(node)
51         else:
52             self.head.prev = node
53             node.next = self.head
54             self.head = node
55             self.length += 1
56
57     def to_string(self):
58
59         temp = self.head
60         s = ""
61         while temp:
62             s += " " + str(temp.get_data())
63             temp = temp.next
64         return s
65
66     def concatenate(self, lst2):
67

```

```

68         self.tail.next = lst2.head
69         lst2.head.prev = self.tail
70         self.tail = lst2.tail
71         self.length += lst2.length
72         lst2.head = None
73         lst2.length = 0

1  class _Component:
2
3     def __init__(self, edge_list, type_c):
4
5         self.edge_list = _LinkedList()
6         for e in edge_list:
7             self.add_edge(e)
8         self.component_type = type_c
9
10    def add_edge(self, e):
11        self.edge_list.append(_LinkedListNode(e))
12
13    def finish_tric_or_poly(self, e):
14
15        self.add_edge(e)
16        if self.edge_list.get_length() > 3:
17            self.component_type = 2
18        else:
19            self.component_type = 1
20
21    def __str__(self):
22
23        if self.component_type == 0:
24            type_str = "Bond: "
25        elif self.component_type == 1:
26            type_str = "Polygon: "
27        else:
28            type_str = "Triconnected: "
29        return type_str + self.edge_list.to_string()
30
31    def get_edge_list(self):
32
33        e_list = []
34        e_node = self.edge_list.get_head()
35        while e_node:
36            e_list.append(e_node.get_data())
37            e_node = e_node.next
38        return e_list

1  class _LinkedListNode:
2
3     def __init__(self, data=None):
4
5         self.data = data
6         self.prev = None
7         self.next = None

```

```

8         self.set_data(self.data)
9
10    def set_data(self, data):
11        self.data = data
12
13    def get_data(self):
14        return self.data
15
16
17    class TriconnectivitySPQR:
18
19        def __init__(self, G, check=True):
20
21            self.n = G.order()
22            self.m = G.size()
23            self.graph_name = G.name()
24
25            if self.n < 2:
26                raise ValueError("Graph is not biconnected")
27            elif self.n == 2 and self.m:
28
29                self.comp_list_new = [G.edges()]
30                self.comp_type = [0]
31                self.__build_spqr_tree()
32                return
33            elif self.m < self.n - 1:
34
35                raise ValueError("Graph is not connected")
36            elif self.m < self.n:
37
38                raise ValueError("Graph is not biconnected")
39
40            self.int_to_vertex = G.vertices()
41            self.vertex_to_int = {u:i for i,u in enumerate(self.int_to_vertex)}
42            self.int_to_original_edge_label = []
43            self.graph_copy = Graph(self.n, multiedges=True)
44            for i,(u, v, l) in enumerate(G.edge_iterator()):
45                self.graph_copy.add_edge(self.vertex_to_int[u], self.vertex_to_int[v], l)
46                self.int_to_original_edge_label.append(l)
47
48            self.edge_status = {e: 0 for e in self.graph_copy.edge_iterator()}
49
50            self.reverse_edges = set()
51
52            self.dfs_number = [0 for i in range(self.n+1)]
53
54            self.highpt = [_LinkedList() for i in range(self.n)]
55
56            self.in_high = {e:None for e in self.graph_copy.edge_iterator()}

```

```

46
47
48     self.old_to_new = [0 for i in range(self.n+1)]
49     self.newnum = [0 for i in range(self.n)]
50     self.node_at = [0 for i in range(self.n+1)]
51     self.lowpt1 = [None for i in range(self.n)]
52     self.lowpt2 = [None for i in range(self.n)]
53
54
55     self.adj = [_LinkedList() for i in range(self.n)]
56
57
58     self.in_adj = {}
59     self.nd = [None for i in range(self.n)]
60
61
62     self.parent = [None for i in range(self.n)]
63     self.degree = [None for i in range(self.n)]
64     self.tree_arc = [None for i in range(self.n)]
65     self.vertex_at = [1 for i in range(self.n)]
66     self.dfs_counter = 0
67     self.components_list = []
68     self.graph_copy_adjacency = [[] for i in range(self.n)]
69
70
71     self.starts_path = {e:False for e in self.graph_copy.edge_iterator()}
72
73     self.is_biconnected = True
74     self.cut_vertex = None
75
76
77     self.virtual_edge_num = 0
78
79     self.virtual_edges = set()
80
81     self.new_path = False
82
83
84     self.e_stack = []
85     self.t_stack_h = [None for i in range(2*self.m + 1)]
86     self.t_stack_a = [None for i in range(2*self.m + 1)]
87     self.t_stack_b = [None for i in range(2*self.m + 1)]
88     self.t_stack_top = 0
89     self.t_stack_a[self.t_stack_top] = -1
90
91
92     self.comp_list_new = []
93     self.comp_type = []
94
95     self.spqr_tree = None
96
97     self.type1 = []
98     self.type2 = []

```

```

99
100         for e in self.graph_copy.edge_iterator():
101             self.graph_copy_adjacency[e[0]].append(e)
102             self.graph_copy_adjacency[e[1]].append(e)
103
104         self.dfs_counter = 0
105         self.start_vertex = 0
106         self.cut_vertex = self.__dfs1(self.start_vertex, check=check)
107
108         if check:
109             # If graph is disconnected
110             if self.dfs_counter < self.n:
111                 raise ValueError("Graph is not connected")
112
113             # If graph has a cut vertex
114             if self.cut_vertex != None:
115                 raise ValueError("Graph has a cut vertex")
116
117
118         for e in self.graph_copy.edge_iterator():
119             up = (self.dfs_number[e[1]] - self.dfs_number[e[0]]) > 0
120             if (up and self.edge_status[e]==2) or (not up and self.edge_
121
122                 self.reverse_edges.add(e)
123
124         self.__build_acceptable_adj_struct()
125         self.__dfs2()
126
127         self.__path_search(self.start_vertex)
128
129
130         if self.e_stack:
131             e = self.__estack_pop()
132             c = _Component(self.e_stack, 0)
133             c.finish_tric_or_poly(e)
134             self.components_list.append(c)
135
136         self.__print_split()
137         self.__assemble_triconnected_components()
138
139         self.__build_spqr_tree()
140
141     def __tstack_push(self, h, a, b):
142
143         self.t_stack_top += 1
144         self.t_stack_h[self.t_stack_top] = h
145         self.t_stack_a[self.t_stack_top] = a
146         self.t_stack_b[self.t_stack_top] = b
147
148     def __tstack_push_eos(self):
149
150         self.t_stack_top += 1
151         self.t_stack_a[self.t_stack_top] = -1

```



```

152
153 def __tstack_not_eos(self):
154
155     return self.t_stack_a[self.t_stack_top] != -1
156
157 def __tstack_print(self):
158
159     return self.t_stack_a, self.t_stack_b, self.t_stack_h
160
161 def __estack_pop(self):
162
163     return self.e_stack.pop()
164
165 def __estack_print(self):
166
167     return self.e_stack
168
169 def __new_component(self, edges=[], type_c=0):
170
171     c = _Component(edges, type_c)
172     self.components_list.append(c)
173     return c
174
175 def __new_virtual_edge(self, u, v):
176
177     e = (u, v, "newVEdge"+str(self.virtual_edge_num))
178     self.virtual_edge_num += 1
179     self.virtual_edges.add(e)
180     return e
181
182 def __high(self, v):
183
184     head = self.highpt[v].get_head()
185     if head is None:
186         return 0
187     else:
188         return head.get_data()
189
190 def __del_high(self, e):
191
192     if e in self.in_high:
193         it = self.in_high[e]
194         if it:
195             if e in self.reverse_edges:
196                 v = e[0]
197             else:
198                 v = e[1]
199             self.highpt[v].remove(it)
200
201 def __bucket_sort(self, bucket, edge_list):
202
203     if len(bucket) == 1:
204         return

```

```

205
206     # Create n bucket linked lists
207     bucket_list = []
208     for i in range(self.n):
209         bucket_list.append(_LinkedList())
210
211     # Get the head pointer of the edge list
212     e_node = edge_list.head
213
214     # Link the n buckets w.r.t bucketId
215     while e_node:
216         bucketId = bucket[e_node.get_data()]
217         if bucket_list[bucketId].get_head():
218             bucket_list[bucketId].tail.next = e_node
219             bucket_list[bucketId].tail = bucket_list[bucketId].tail.next
220         else:
221             bucket_list[bucketId].set_head(e_node)
222         e_node = e_node.next
223
224     # Rearrange the `edge_list` Using bucket list
225     new_tail = None
226     for i in range(self.n):
227         new_head = bucket_list[i].get_head()
228         if new_head:
229             if new_tail:
230                 new_tail.next = new_head
231             else:
232                 edge_list.set_head(new_head)
233             new_tail = bucket_list[i].tail
234
235     edge_list.tail = new_tail
236     new_tail.next = None
237
238     def __sort_edges(self):
239
240         # Create a linkedlist of edges
241         edge_list = _LinkedList()
242         for e in self.graph_copy.edges(sort=False):
243             edge_list.append(_LinkedListNode(e))
244
245         bucketMin = {} # Contains the lower index of edge end point
246         bucketMax = {} # Contains the higher index of edge end point
247
248         # In `graph_copy`, every edge `(u, v)` is such that `u < v`.
249         # Hence, `bucketMin` of an edge `(u, v)` will be `u`
250         # and `bucketMax` will be `v`.
251         for e in self.graph_copy.edge_iterator():
252             bucketMin[e] = e[0]
253             bucketMax[e] = e[1]
254
255         # Sort according to the endpoint with lower index
256         self.__bucket_sort(bucketMin, edge_list)
257         # Sort according to the endpoint with higher index

```

5.1 Algoritmo para encontrar componentes triconexas de un bigrafo de tipo \mathbb{D}_n

```

258         self.__bucket_sort(bucketMax, edge_list)
259
260         # Return the head pointer to the sorted edge list
261         return edge_list.get_head()
262
263     def __dfs1(self, v, u=None, check=True):
264
265         first_son = None # For testing biconnectivity
266         s1 = None # Storing the cut vertex, if there is one
267         self.dfs_counter += 1
268         self.dfs_number[v] = self.dfs_counter
269         self.parent[v] = u
270         self.degree[v] = self.graph_copy.degree(v)
271         self.lowpt1[v] = self.lowpt2[v] = self.dfs_number[v]
272         self.nd[v] = 1
273         for e in self.graph_copy_adjacency[v]:
274             if self.edge_status[e]:
275                 continue
276
277             w = e[0] if e[0] != v else e[1] # Opposite vertex of edge e
278             if self.dfs_number[w] == 0:
279                 self.edge_status[e] = 1 # tree edge
280                 if first_son is None:
281                     first_son = w
282                 self.tree_arc[w] = e
283                 s1 = self.__dfs1(w, v, check)
284
285             if check:
286                 # Check for cut vertex.
287                 # The situation in which there is no path from w to
288                 # ancestor of v : we have identified a cut vertex
289                 if (self.lowpt1[w] >= self.dfs_number[v]) and (w !=
290                     s1 = v
291
292             # Calculate the `lowpt1` and `lowpt2` values.
293             # `lowpt1` is the smallest vertex (the vertex x with smallest
294             # dfs_number[x]) that can be reached from v.
295             # `lowpt2` is the next smallest vertex that can be reached
296             if self.lowpt1[w] < self.lowpt1[v]:
297                 self.lowpt2[v] = min(self.lowpt1[v], self.lowpt2[w])
298                 self.lowpt1[v] = self.lowpt1[w]
299
300             elif self.lowpt1[w] == self.lowpt1[v]:
301                 self.lowpt2[v] = min(self.lowpt2[v], self.lowpt2[w])
302
303             else:
304                 self.lowpt2[v] = min(self.lowpt2[v], self.lowpt1[w])
305
306             self.nd[v] += self.nd[w]
307
308         else:
309             self.edge_status[e] = 2 # frond
310             if self.dfs_number[w] < self.lowpt1[v]:

```

```

311         self.lowpt2[v] = self.lowpt1[v]
312         self.lowpt1[v] = self.dfs_number[w]
313     elif self.dfs_number[w] > self.lowpt1[v]:
314         self.lowpt2[v] = min(self.lowpt2[v], self.dfs_number[w])
315
316     return s1 # s1 is None if graph does not have a cut vertex
317
318
319 def __build_acceptable_adj_struct(self):
320
321     max_size = 3*self.n + 2
322     bucket = [[] for _ in range(max_size + 1)]
323
324     for e in self.graph_copy.edge_iterator():
325         edge_type = self.edge_status[e]
326
327         # compute phi value
328         # bucket sort adjacency list by phi values
329         if e in self.reverse_edges:
330             if edge_type == 1: # tree arc
331                 if self.lowpt2[e[0]] < self.dfs_number[e[1]]:
332                     phi = 3*self.lowpt1[e[0]]
333                 else:
334                     phi = 3*self.lowpt1[e[0]] + 2
335             else: # tree frond
336                 phi = 3*self.dfs_number[e[0]]+1
337         else:
338             if edge_type == 1: # tree arc
339                 if self.lowpt2[e[1]] < self.dfs_number[e[0]]:
340                     phi = 3*self.lowpt1[e[1]]
341                 else:
342                     phi = 3*self.lowpt1[e[1]] + 2
343             else: # tree frond
344                 phi = 3*self.dfs_number[e[1]]+1
345
346         bucket[phi].append(e)
347
348         # Populate `adj` and `in_adj` with the sorted edges
349         for i in range(1, max_size + 1):
350             for e in bucket[i]:
351                 node = _LinkedListNode(e)
352                 if e in self.reverse_edges:
353                     self.adj[e[1]].append(node)
354                     self.in_adj[e] = node
355                 else:
356                     self.adj[e[0]].append(node)
357                     self.in_adj[e] = node
358
359 def __path_finder(self, v):
360
361     self.newnum[v] = self.dfs_counter - self.nd[v] + 1
362     e_node = self.adj[v].get_head()
363     while e_node:

```

```

364         e = e_node.get_data()
365         e_node = e_node.next
366         w = e[1] if e[0] == v else e[0] # opposite vertex of e
367         if self.new_path:
368             self.new_path = False
369             self.starts_path[e] = True
370         if self.edge_status[e] == 1: # tree arc
371             self.__path_finder(w)
372             self.dfs_counter -= 1
373         else:
374             # Identified a new frond that enters `w`. Add to `highpt[w]`
375             highpt_node = _LinkedListNode(self.newnum[v])
376             self.highpt[w].append(highpt_node)
377             self.in_high[e] = highpt_node
378             self.new_path = True
379
380
381     def __dfs2(self):
382
383         self.in_high = {e:None for e in self.graph_copy.edge_iterator()}
384         self.dfs_counter = self.n
385         self.newnum = [0 for i in range(self.n)]
386         self.starts_path = {e:False for e in self.graph_copy.edge_iterator()}
387
388         self.new_path = True
389
390         # We call the pathFinder function with the start vertex
391         self.__path_finder(self.start_vertex)
392
393         # Update `old_to_new` values with the calculated `newnum` values
394         for v in self.graph_copy.vertex_iterator():
395             self.old_to_new[self.dfs_number[v]] = self.newnum[v]
396
397         # Update lowpt values according to `newnum` values.
398         for v in self.graph_copy.vertex_iterator():
399             self.node_at[self.newnum[v]] = v
400             self.lowpt1[v] = self.old_to_new[self.lowpt1[v]]
401             self.lowpt2[v] = self.old_to_new[self.lowpt2[v]]
402
403     def __path_search(self, v):
404
405         y = 0
406         vnum = self.newnum[v]
407         outv = self.adj[v].get_length()
408         e_node = self.adj[v].get_head()
409         while e_node:
410             e = e_node.get_data()
411             it = e_node
412
413             if e in self.reverse_edges:
414                 w = e[0] # target
415             else:
416                 w = e[1]

```

```

417         wnum = self.newnum[w]
418         if self.edge_status[e] == 1: # e is a tree arc
419             if self.starts_path[e]: # if a new path starts at e
420                 y = 0
421                 # Pop all (h,a,b) from tstack where a > lowp
422                 if self.t_stack_a[self.t_stack_top] > self.lowp1[w]:
423                     while self.t_stack_a[self.t_stack_top] > self.lowp1[w]:
424                         y = max(y, self.t_stack_h[self.t_stack_top])
425                         b = self.t_stack_b[self.t_stack_top]
426                         self.t_stack_top -= 1
427                     self.__tstack_push(y, self.lowp1[w])
428
429             else:
430                 self.__tstack_push(wnum + self.nd[w])
431                 self.__tstack_push_eos()
432
433         self.__path_search(w)
434
435         self.e_stack.append(self.tree_arc[w])
436         temp_node = self.adj[w].get_head()
437         temp = temp_node.get_data()
438         if temp in self.reverse_edges:
439             temp_target = temp[0]
440         else:
441             temp_target = temp[1]
442
443         # Type-2 separation pair check
444         # while v is not the start_vertex
445         while vnum != 1 and ((self.t_stack_a[self.t_stack_top] > self.lowp1[w]
446                                or self.t_stack_b[self.t_stack_top] > self.degree[w])
447                                and (self.degree[w] > 2)):
448             a = self.t_stack_a[self.t_stack_top]
449             b = self.t_stack_b[self.t_stack_top]
450             e_virt = None
451             if a == vnum and self.parent[self.node_at[b]] == vnum:
452                 self.t_stack_top -= 1
453
454             else:
455                 e_ab = None
456                 if self.degree[w] == 2 and self.newnum[w] == vnum:
457                     # found type-2 separation pair
458                     self.type2.append((v+1, temp))
459                     e1 = self.__estack_pop()
460                     e2 = self.__estack_pop()
461                     self.adj[w].remove(self.in_a[e1])
462
463                     if e2 in self.reverse_edges:
464                         x = e2[0] # target
465                     else:
466                         x = e2[1] # target
467
468                     e_virt = self.__new_virtual_edge(x, w)
469                     self.graph_copy.add_edge(e_virt)

```

5.1 Algoritmo para encontrar componentes triconexas de un bigrafo de tipo \mathbb{D}_n

```

470         self.degree[v] -= 1
471         self.degree[x] -= 1
472
473         if e2 in self.reverse_edges:
474             e2_source = e2[1] # target
475         else:
476             e2_source = e2[0]
477         if e2_source != w:
478             raise ValueError("Graph is not a  $\mathbb{D}_n$ ")
479
480         comp = _Component([e1, e2, e_virt],
481                             self.components_list.append(comp))
482         comp = None
483
484         if self.e_stack:
485             e1 = self.e_stack[-1]
486             if e1 in self.reverse_edges:
487                 if e1[1] == x and e1[0] == w:
488                     e_ab = self._estack.pop()
489                     self.adj[x].add(e_ab)
490                     self._del_high(e_ab)
491             else:
492                 if e1[0] == x and e1[1] == w:
493                     e_ab = self._estack.pop()
494                     self.adj[x].add(e_ab)
495                     self._del_high(e_ab)
496
497         else: # found type-2 separation pair - (self.type2.append((self.node_at[a]+1,
498             self.type2.append((self.node_at[a]+1, self.node_at[a]),
499             h = self.t_stack_h[self.t_stack_top]
500             self.t_stack_top -= 1
501
502         comp = _Component([], 0)
503         while True:
504             xy = self.e_stack[-1]
505             if xy in self.reverse_edges:
506                 x = xy[1]
507                 xy_target = xy[0]
508             else:
509                 x = xy[0]
510                 xy_target = xy[1]
511             if not (a <= self.newnum[x]):
512                 break
513             if (self.newnum[x] == a and self.newnum[xy_target] == a):
514                 e_ab = self._estack.pop()
515                 if e_ab in self.reverse_edges:
516                     e_ab_source = e_ab[1]
517                 else:
518                     e_ab_source = e_ab[0]
519                 self.adj[e_ab_source].add(x)
520                 self._del_high(e_ab)
521
522         else:

```

```

523                                     eh = self._c
524                                     if eh in self._c:
525                                         eh_s = self._c[eh]
526                                     else:
527                                         eh_s = self._c[eh]
528                                     if it != self._c[eh_s]:
529                                         self._c[eh_s] = it
530                                     self._c[eh_s] = it
531
532                                     comp.add_edge(e_virt)
533                                     self.degree[x] += 1
534                                     self.degree[v] += 1
535
536                                     e_virt = self._new_virtual_edge(x, v)
537                                     self.graph_copy.add_edge(e_virt)
538                                     comp.finish_tric_or_poly(e_virt)
539                                     self.components_list.append(comp)
540                                     comp = None
541                                     x = self.node_at[b]
542
543                                     if e_ab is not None:
544                                         comp = _Component([e_ab, e_virt])
545                                         e_virt = self._new_virtual_edge(x, v)
546                                         self.graph_copy.add_edge(e_virt)
547                                         comp.add_edge(e_virt)
548                                         self.degree[x] -= 1
549                                         self.degree[v] -= 1
550                                         self.components_list.append(comp)
551                                         comp = None
552
553                                     self.e_stack.append(e_virt)
554                                     # Replace the edge 'it' with 'e_virt'
555                                     it.set_data(e_virt)
556
557                                     self.in_adj[e_virt] = it
558                                     self.degree[x] += 1
559                                     self.degree[v] += 1
560                                     self.parent[x] = v
561                                     self.tree_arc[x] = e_virt
562                                     self.edge_status[e_virt] = 1
563                                     w = x
564                                     wnum = self.newnum[w]
565
566                                     # update the values used in the while loop
567                                     temp_node = self.adj[w].get_head()
568                                     temp = temp_node.get_data()
569                                     if temp in self.reverse_edges:
570                                         temp_target = temp[0]
571                                     else:
572                                         temp_target = temp[1]
573
574                                     # start type-1 check
575                                     if self.lowpt2[w] >= vnum and self.lowpt1[w] < vnum

```


5.1 Algoritmo para encontrar componentes triconexas de un bigrafo de tipo \mathbb{D}_n

```

576 # type-1 separation pair - (self.node_at[self.lowpt1[w]], v)
577 self.type1.append((self.node_at[self.lowpt1[w]]+1, v))
578 # Create a new component and add edges to it
579 comp = _Component([], 0)
580 if not self.e_stack:
581     raise ValueError("stack is empty")
582 while self.e_stack:
583     xy = self.e_stack[-1]
584     if xy in self.reverse_edges:
585         xx = self.newnum[xy[1]] #source
586         y = self.newnum[xy[0]] #target
587     else:
588         xx = self.newnum[xy[0]] #source
589         y = self.newnum[xy[1]] #target
590
591     if not ((wnum <= xx and xx < wnum + self.newnum[v]) or
592            (wnum <= y and y < wnum + self.newnum[v])):
593         break
594
595     comp.add_edge(self.__estack_pop())
596     self.__del_high(xy)
597     self.degree[self.node_at[xx]] -= 1
598     self.degree[self.node_at[y]] -= 1
599
600 e_virt = self.__new_virtual_edge(v, self.node_at[self.lowpt1[w]])
601 self.graph_copy.add_edge(e_virt) # Add virtual edge
602 comp.finish_tric_or_poly(e_virt) # Add virtual edge
603 self.components_list.append(comp)
604 comp = None
605
606 if (xx == vnum and y == self.lowpt1[w]) or (y == vnum and x == self.lowpt1[w]):
607     comp_bond = _Component([], type_c=0) # new component
608     eh = self.__estack_pop()
609     if self.in_adj[eh] != it:
610         if eh in self.reverse_edges:
611             self.adj[eh[1]].remove(self.node_at[eh[0]])
612         else:
613             self.adj[eh[0]].remove(self.node_at[eh[1]])
614
615     comp_bond.add_edge(eh)
616     comp_bond.add_edge(e_virt)
617     e_virt = self.__new_virtual_edge(v, self.node_at[self.lowpt1[w]])
618     self.graph_copy.add_edge(e_virt)
619     comp_bond.add_edge(e_virt)
620     if eh in self.in_high:
621         self.in_high[e_virt] = self.in_high[eh]
622     self.degree[v] -= 1
623     self.degree[self.node_at[self.lowpt1[w]]] -= 1
624     self.components_list.append(comp_bond)
625     comp_bond = None
626
627 if self.node_at[self.lowpt1[w]] != self.parent[v]:
628     self.e_stack.append(e_virt)

```

```

629
630                                     # replace edge `it` with `e_virt`
631                                     it.set_data(e_virt)
632
633                                     self.in_adj[e_virt] = it
634                                     if not e_virt in self.in_high and se
635                                         vnum_node = _LinkedListNode(
636                                             self.highpt[self.node_at[self
637                                             self.in_high[e_virt] = vnum_
638
639                                     self.degree[v] += 1
640                                     self.degree[self.node_at[self.lowpt1
641
642                                     else:
643                                         self.adj[v].remove(it)
644                                         comp_bond = _Component([e_virt], typ
645                                         e_virt = self._new_virtual_edge(sel
646                                         self.graph_copy.add_edge(e_virt)
647                                         comp_bond.add_edge(e_virt)
648
649                                         eh = self.tree_arc[v];
650                                         comp_bond.add_edge(eh)
651                                         self.components_list.append(comp_bon
652                                         comp_bond = None
653
654                                         self.tree_arc[v] = e_virt
655                                         self.edge_status[e_virt] = 1
656                                         if eh in self.in_adj:
657                                             self.in_adj[e_virt] = self.i
658                                         e_virt_node = _LinkedListNode(e_virt
659                                         self.in_adj[eh] = e_virt_node
660                                     # end type-1 search
661
662                                     # if an path starts at edge e, empty the tstack.
663                                     if self.starts_path[e]:
664                                         while self._tstack_not_eos():
665                                             self.t_stack_top -= 1
666                                         self.t_stack_top -= 1
667
668                                     while self._tstack_not_eos() and self.t_stack_b[self
669                                         self.t_stack_top -= 1
670
671                                     outv -= 1
672                                     else: # e is a frond
673                                         if self.starts_path[e]:
674                                             y = 0
675                                             # pop all (h,a,b) from tstack where a > w
676                                             if self.t_stack_a[self.t_stack_top] > wnum:
677                                                 while self.t_stack_a[self.t_stack_top] > wnum:
678                                                     y = max(y, self.t_stack_h[self
679                                                     b = self.t_stack_b[self.t_st
680                                                     self.t_stack_top -= 1
681                                                     self._tstack_push(y, wnum, b)

```

```

682
683         else:
684             self.__tstack_push(vnum, wnum, vnum)
685             self.e_stack.append(e) # add (v,w) to ESTACK
686             # Go to next edge in adjacency list
687             e_node = e_node.next
688
689     def __print_split(self):
690         self.comp_list_new1 = []
691         self.comp_type1 = []
692         for comp in self.components_list:
693             if comp.edge_list.get_length() > 0:
694                 e_list = comp.get_edge_list()
695                 e_list_new = []
696                 # For each edge, get the original source, target and label
697                 for u,v,l in e_list:
698                     source = self.int_to_vertex[u]
699                     target = self.int_to_vertex[v]
700                     if isinstance(l, str):
701                         label = l
702                     else:
703                         label = self.int_to_original_edge_label[l]
704                     e_list_new.append((source, target, label))
705                 self.comp_list_new1.append(e_list_new)
706         return self.comp_list_new1
707
708     def __assemble_triconnected_components(self):
709
710         comp1 = {} # The index of first component that an edge belongs to
711         comp2 = {} # The index of second component that an edge belongs to
712         item1 = {} # Pointer to the edge node in component1
713         item2 = {} # Pointer to the edge node in component2
714         num_components = len(self.components_list)
715         visited = [False for i in range(num_components)]
716
717         # For each edge, we populate the comp1, comp2, item1 and item2 values
718         for i in range(num_components): # for each component
719             e_node = self.components_list[i].edge_list.get_head()
720             while e_node: # for each edge
721                 e = e_node.get_data()
722                 if e not in item1:
723                     comp1[e] = i
724                     item1[e] = e_node
725                 else:
726                     comp2[e] = i
727                     item2[e] = e_node
728
729                 e_node = e_node.next
730
731         # For each edge in a component, if the edge is a virtual edge, merge
732         # the two components the edge belongs to
733         for i in range(num_components):
734             c1 = self.components_list[i]

```

```

735         c1_type = c1.component_type
736         l1 = c1.edge_list
737         visited[i] = True
738
739         if l1.get_length() == 0:
740             continue
741
742         if c1_type == 0 or c1_type == 1:
743             e_node = self.components_list[i].edge_list.get_head()
744             # Iterate through each edge in the component
745             while e_node:
746                 e = e_node.get_data()
747                 e_node_next = e_node.next
748                 # The label of a virtual edge is a string
749                 if not isinstance(e[2], str):
750                     e_node = e_node_next
751                     continue
752
753                 j = comp1[e]
754                 if visited[j]:
755                     j = comp2[e]
756                     if visited[j]:
757                         e_node = e_node_next
758                         continue
759                     e_node2 = item2[e]
760                 else:
761                     e_node2 = item1[e]
762
763                 c2 = self.components_list[j]
764
765                 # If the two components are not the same type
766                 if (c1_type != c2.component_type):
767                     e_node = e_node_next # Go to next edge
768                     continue
769
770                 visited[j] = True
771                 l2 = c2.edge_list
772
773                 # Remove the corresponding virtual edges in l2
774                 # and merge the components
775                 l2.remove(e_node2)
776                 l1.concatenate(l2)
777
778                 # if `e_node_next` was empty, after merging
779                 # more edges are added to the component.
780                 if not e_node_next:
781                     e_node_next = e_node.next # Go to next edge
782
783                 l1.remove(e_node)
784
785                 e_node = e_node_next
786
787         # Convert connected components into original graph vertices and edges

```

```

788     self.comp_list_new = []
789     self.comp_type = []
790     for comp in self.components_list:
791         if comp.edge_list.get_length() > 0:
792             e_list = comp.get_edge_list()
793             e_list_new = []
794             # For each edge, get the original source, target and label
795             for u,v,l in e_list:
796                 source = self.int_to_vertex[u]
797                 target = self.int_to_vertex[v]
798                 if isinstance(l, str):
799                     label = l
800                 else:
801                     label = self.int_to_original_edge_label[l]
802                 e_list_new.append((source, target, label))
803             # Add the component data to `comp_list_new` and `comp_type`
804             self.comp_type.append(comp.component_type)
805             self.comp_list_new.append(e_list_new)
806     #print(self.comp_list_new)
807
808     def __build_spqr_tree(self):
809
810         # Types of components 0: "P", 1: "S", 2: "R"
811         component_type = ["P", "S", "R"]
812
813         self.spqr_tree = Graph(multiedges=False, name='SPQR-tree of {}'.format(self))
814
815         if len(self.comp_list_new) == 1 and self.comp_type[0] == 0:
816             self.spqr_tree.add_vertex(('Q' if len(self.comp_list_new[0]) == 1 else
817                                     Graph(self.comp_list_new[0].get_vertices()))
818             return
819
820         int_to_vertex = []
821         partner_nodes = {}
822
823         for i in range(len(self.comp_list_new)):
824             # Create a new tree vertex
825             u = (component_type[self.comp_type[i]],
826                 Graph(self.comp_list_new[i], immutable=True, multiedges=True))
827             self.spqr_tree.add_vertex(u)
828             int_to_vertex.append(u)
829
830             # Add an edge to each node containing the same virtual edge
831             for e in self.comp_list_new[i]:
832                 if e in self.virtual_edges:
833                     if e in partner_nodes:
834                         for j in partner_nodes[e]:
835                             self.spqr_tree.add_edge(int_to_vertex[j], int_to_vertex[i])
836                         partner_nodes[e].append(i)
837                     else:
838                         partner_nodes[e] = [i]

```

Bibliografía

- [1] Abarca Sotelo, M. A. (2011). Algoritmo para decidir si una forma unitaria es de tipo.
- [2] Ariyoshi, H., Shirakawa, I., and Ozaki, H. (1971). Decomposition of a graph into compactly connected two-terminal subgraphs. *IEEE Transactions on Circuit Theory*, 18(4):430–435.
- [3] AUSLANDER, L. and PARTER, S. V. (1961). On imbedding graphs in the plane. *Journal of Applied Mathematics and Mechanics*, pages 517–523.
- [4] Barot, M. (1999). A characterization of positive unit forms.
- [5] Barot, M. and de la Pena, J. (1999). The dynkin type of a non-negative unit form. *Expositiones Mathematicae*, 17.
- [6] Bruno, J., Steiglitz, K., and Weinberg, L. (1970). A new planarity test based on 3-connectivity. *IEEE Transactions on Circuit Theory*, 17(2):197–206.
- [7] Busacker, R., Busacker, R., Saaty, T., and Grob, B. (1965). *Finite Graphs and Networks: An Introduction with Applications*. Number v. 10 in Finite Graphs and Networks: An Introduction with Applications. McGraw-Hill.
- [8] EDMONDS, J. and CUNNINGHAM, W. (1973). private communication.
- [9] Gabriel, P. (1997). *Representations of finite-dimensional algebras / P. Gabriel, A.V. Roiter*. Springer, Berlin ;.
- [10] Goldstein, A. (1963). An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conference, Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dept. of Mathematics, Princeton University, May 16-18*.
- [11] Gutwenger, C. and Mutzel, P. (2000). A linear time implementation of spqr-trees. In *Graph Drawing*.

- [12] Harary, F. (1971). *Graph Theory*. Addison Wesley series in mathematics. Addison-Wesley.
- [13] Hopcroft, J. and Tarjan, R. (1973a). Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372378.
- [14] Hopcroft, J. and Tarjan, R. (1974). Efficient planarity testing. *J. ACM*, 21(4):549568.
- [15] Hopcroft, J. E. and Tarjan, R. E. (1972). *Isomorphism of Planar Graphs (Working Paper)*, pages 131–152. Springer US, Boston, MA.
- [16] Hopcroft, J. E. and Tarjan, R. E. (1973b). Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–158.
- [17] Kleitman, D. (1969). Methods for investigating connectivity of large graphs. *IEEE Transactions on Circuit Theory*, 16(2):232–233.
- [18] Lay, D. C. (2001). *Algebra lineal y sus aplicaciones*. Pearson Educación, México, 2a ed. actualizada. edition.
- [19] Lederberg, J. (1964). Dendral-64: A system for computer construction, enumeration and notation of organic molecules as tree structures and cyclic graphs. part i. notational algorithm for tree structures.
- [20] Mac Lane, S. (1937). A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3).
- [21] Meghana M Reddy, Sai Harsh Tondomker, D. C. (2018). Triconnectivity linear time algorithm. <https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/connectivity.html#sage.graphs.connectivity.TriconnectivitySPQR>.
- [22] Ringel, C. M. (1985). Tame algebras and integral quadratic forms.
- [23] Tarjan, R. (1971a). Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121.
- [24] TARJAN, R. and HOPCROFT, J. (1972). Finding the triconnected components of a graph. *Dept. of Computer Science*, pages 72–140.
- [25] Tarjan, R. E. (1971b). An efficient planarity algorithm. Technical report, STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE.
- [26] Tarjan, R. E. (1974). Finding dominators in directed graphs. *SIAM J. Comput.*, 3:62–89.