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

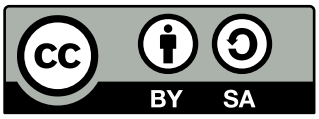
Presidente Dra. Larissa Sbitneva,
Universidad Autónoma del Estado de Morelos

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



Contenido

LISTA DE SIMOLOS

x_i——x_j	arista solida entre los vértices i y j
x_i·····x_j	arista punteada entre los vértices i y j
A^{-1}	matriz inversa de A
$\cup$	unón
$\cap$	intersección
$/$	negación
$=$	asignación
$=$	igualdad
$\mathbb{Z}$	conjunto de los números enteros
$\Theta()$	cota superor 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

1 Introducción

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 $\{x_1, x_2, \ldots, x_n\}$. 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: $7x^2y + 5x^z4 - y$ 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 $q : \mathbb{R}^n \to \mathbb{R}$(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

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

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

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

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 $\frac{1}{2}\overrightarrow{x}^T A \overrightarrow{x}$ se puede reescribir como sigue:

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

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:

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

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:

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

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

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

CAMBIO DE VARIABLE

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

$$\overrightarrow{x} = P\overrightarrow{y} \quad \text{o bien} \quad \overrightarrow{y} = P^{-1}\overrightarrow{x}. \tag{1.3}$$

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

$$\frac{1}{2}\overrightarrow{x}^T\overrightarrow{x} = \frac{1}{2}\,(P\overrightarrow{y})^T\ A\ (P\overrightarrow{y}) = \frac{1}{2}\left(\overrightarrow{y}^T P^T\right)\ A\ (P\overrightarrow{y}) = \frac{1}{2}\overrightarrow{y}^T\left(P^T\ A\ P\right)\overrightarrow{y}$$

.

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

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

Mediante el cambio de variable $\overrightarrow{y} = P\ \overrightarrow{x}$ tenemos que $q(\overrightarrow{x}) = q'(\overrightarrow{y})$ y diremos que q y q' son **equivalentes** mediante la matriz invertible P. Si denotamos con $L(\overrightarrow{x})$ a la transformación dada por, $L(\overrightarrow{x}) = P\overrightarrow{x}$ y tomamos $\overrightarrow{y} = L(\overrightarrow{x})$, entonces $q'(\overrightarrow{y}) = q'\,(L(\overrightarrow{x})) = (q' \circ L)\,(\overrightarrow{x})$, 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 expresion $x^2 - 5y^2 - 8xy$.

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 diagonizable**, 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 diagonizable 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 siguente resultado:

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

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

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

□

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

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

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

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

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

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 $\lambda_i \leqslant 0$ entonces definimos $y' = (y_1, y_2, \ldots, y_n)$ donde todos los $y_k = 0$ excepto $y_i = 1$. Claramente $\overrightarrow{y} \neq \overrightarrow{0}$ pero $q(\overrightarrow{y}) = \lambda_i \leqslant 0$; por lo tanto q no es definida positiva. □

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

$$\forall I \subseteq 1, ..., n \quad \delta_I(q_{\mathcal{B}}) > 0.$$

3. *todos los menores de la matriz $q_{\mathcal{B}}$ son positivos: para todo $k \in 1, \ldots, n$*

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

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 $i = 1, 2, \ldots, n$ 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 $\overrightarrow{y} = P\overrightarrow{x}$ 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 $q(\overrightarrow{x}) = q'(\overrightarrow{y})$ mediante el cambio de variable entero $\overrightarrow{y} = P\overrightarrow{x}$ 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}.

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

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

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

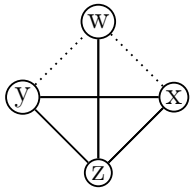


Figura 1.1: Bigráfica asociada a $2wx - 2yz - xz - xy - wz + wy$

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 $n \geqslant 4$ mientras que las gráficas $\mathbb{E}_n$ solo se definen para $n = 6, 7, 8$. Lo impotante 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.

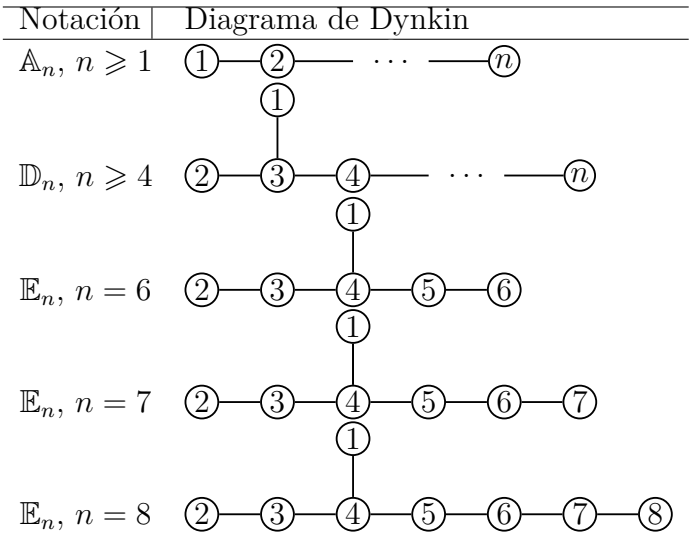


Figura 1.2: Gráficas de Dynkin. El subíndice n indica la cantidad de vértices que tiene la bigráfica.

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

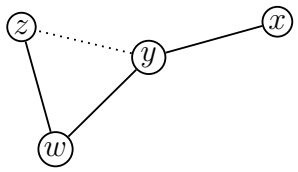
Teorema 1.7. *Toda forma unitaria B_q es conexa, así q es $\mathbb{Z}$-equivalente a una forma unitaria cuya bigráfica asociada es un diagrama de Dynkin.*

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

Ejemplo 1.2. *[1] La forma unitaria*

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

tiene asociada la bigráfica



y su matriz asociada

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

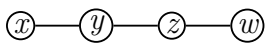
$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ con inversa } P^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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.

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 $\mathbb{A}_n$:

$$x_1\text{-}x_2\text{-}\cdots\text{-}x_n$$

Consideremos la identidad:

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

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

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

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:

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

de tal forma que

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

Para las gráficas $\mathbb{E}_6$, $\mathbb{E}_7$ y $\mathbb{E}_8$ usaremos el siguente 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:

$$A_{\mathbb{E}_6} = \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ -1 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

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

$$A_{\mathbb{E}_7} = \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 00 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

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

$$A_{\mathbb{E}_8} = \begin{bmatrix} 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

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

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 $q(\overrightarrow{x})$ es definida positiva entonces $q_{ij} \in \{-1, 0, 1\}$ para todo $1 \leqslant i \leqslant j \leqslant n$.*

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

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 $|q_{ij}| \leqslant 1$ 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 $\boldsymbol{B}_q$ 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 solidas).

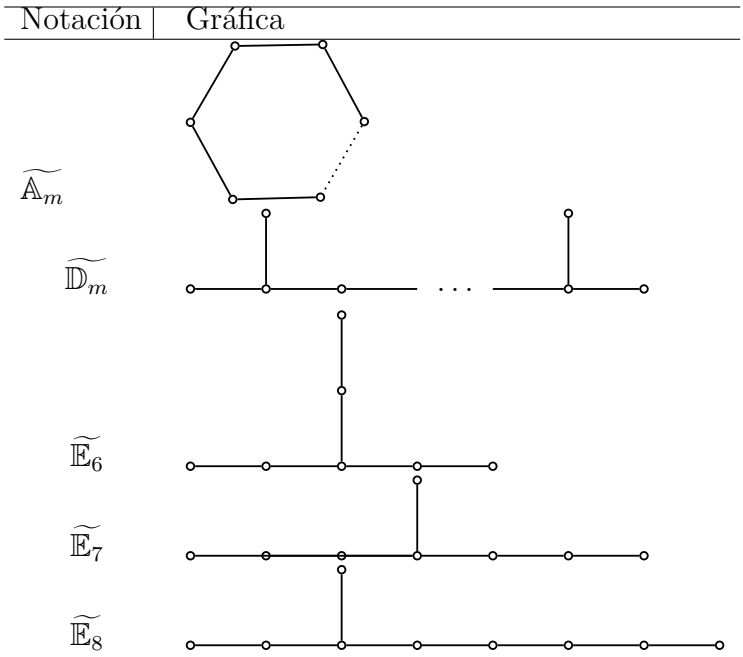
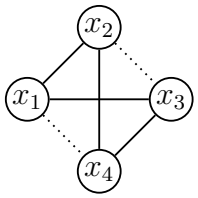


Figura 1.3: Diagramas Euclidianos. La cantidad de vértices que tiene cada gráfica es $n = m + 1$.

Hasta ahora hemos asociado vértices con variables(Ⓧ 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 ordemn 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 $\overrightarrow{x} \neq 0$ tal que $q(\overrightarrow{x}) = 0$, mostrando así que $q(\overrightarrow{x})$ no es definida positiva. Un simple cálculo nos muestra que esta evaluación produce un vector $\overrightarrow{x} \neq \overrightarrow{0}$ tal que $q(\overrightarrow{x}) = 0$.

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 $r \leqslant n$, pero habíamos supuesto que $G \neq \mathbb{D}_n$ por lo tanto hay tres casos:

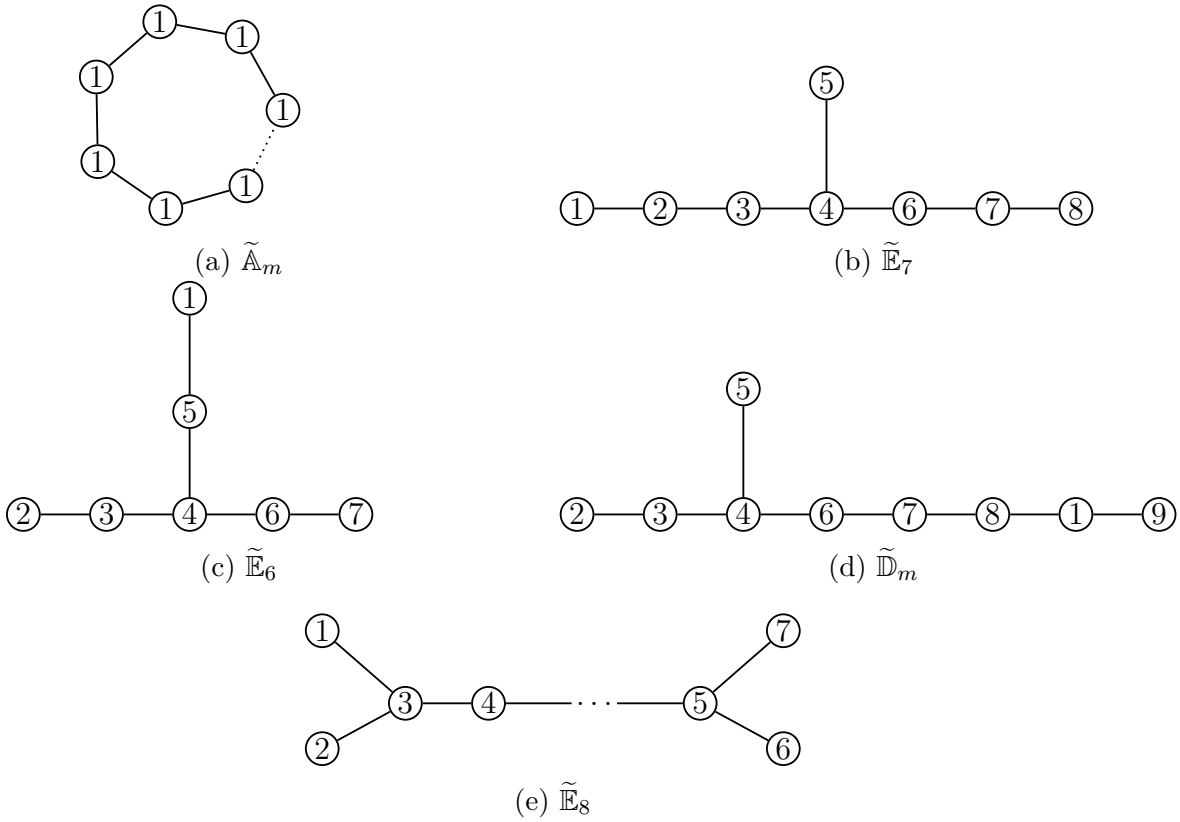


Figura 1.4: Formas unitarias asociadas a las gráficas Euclidianas.

1. Si v tiene grado estrictamente mayor a 3 entonces G contiene a $\widetilde{D_4}$
2. Si G contiene otro vértice w de grado 3 o más, entonces G contiene a $\widetilde{D_m}$ 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 habiamos supuesto que $G \neq \mathbb{E}_6$, por tanto $n > 6$. Eneste caso G debe contener a $\widetilde{\mathbb{E}}_6$ o $\mathbb{E}_7$. Si tenemos que $G \neq \mathbb{E}_7$ entonces $n > 8$., de donde obtenemos que G contiene a $\widetilde{\mathbb{E}}_7$ o $\mathbb{E}_8$, pero si $G \neq \mathbb{E}_8$ entonces $n > 8$ y por lo tanto G contiene a $\widetilde{\mathbb{E}}_8$.
Resumiendo, las gráficas de Dynkin definen formas unitarias positivas, y cualquier otra gráfica conexa y de aristas solidas 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 conluye la demostración del lema 1.8.
En la sección 2.1 se dará termino a la demostración del teorema 1.7

2 Antecedentes

Este capitulo 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 aqui.

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' tambien es definidad positiva; luego como q' solamente contiene aristas solidas 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^d_{sr}, cuya transformación lineal

$$T^d_{sr}(A) = E^d_{sr}A$$

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

Consideremos el cambio de variable entero $\overrightarrow{y} = E^d_{sr}\overrightarrow{x}$, entonces

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

y por tanto podemos concluir que T^d_{sr} como cambio de variable tiene el mismo efecto que sustituir $x_r \leftarrow x_r + dx_s$.
£Qué efecto tiene T^d_{sr} si sobre una forma unitaria q?. Supongamos que q tiene matriz asociada $A = [a_{ij}]$ y sea

$$q'(\overrightarrow{x}) = \frac{1}{2}\left(E^d_{rs}\right)^T A E^d_{sr}$$

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

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

E^d_{rs} 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 $q'(\overrightarrow{x}) = \frac{1}{2}\overrightarrow{x}^T C \overrightarrow{x}$ 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 $q \circ T^d_{rs}$ es una forma unitaria si y solo si $q_{rs} = q_{sr} = -d$.*

Corolario 2.2. *Si q es una forma unitaria simple entonces $q \circ T^d_{rs}$ es una forma unitaria para un $d \neq 0$ en cualquiera de estos dos casos y solamentes 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 $x_r \cdots\cdots x_s$ 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 reeplazar $x_i \cdots\cdots x_j$ 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 $x_i — xj$ por $x_i \cdots\cdots x_j$
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 $x_i \cdots\cdots x_j$ y $x_i — xj$ incidentes en x_r y x_j, se borran.

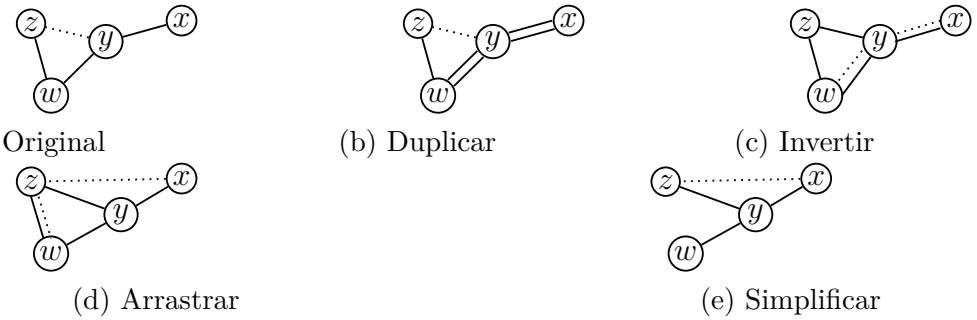
Por ejemplo, consideremos la forma cuadrática q de la ecuación (1.5) y ordenemos a las variables como (w, x, y, z). Entonces para aplicar T_{23}^- podemos calcular $\left(E_{23}^{-1}\right)^T A\, E_{23}^{-1}$ donde A es la matriz asociada a q.

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

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

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

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



(a) Original (b) Duplicar (c) Invertir

(d) Arrastrar (e) Simplificar

Figura 2.1: Ejemplo de una inflación.

DESCRIPCIÓN Y JUSTIFICACIÓN

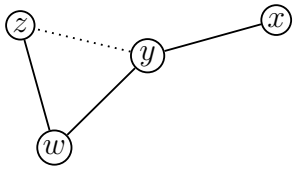
El método que aparece en [9] consiste en aplicar sucesivamente inflaciones como se indica en el algoritmo 2.1 tanto como sea posible, es decir, hasta que no haya más aristas punteadas.

Algoritmo 2.1: Inflaciones(q)

mientras *exista una arista punteada* $x_r \cdots\cdots x_s$ *con* $r \leqslant s$ *en* $\boldsymbol{B}_q$*:*
    $q = q \circ T_{rs}^{-}$
    si *para algun coeficiente* q_{ij} *se tiene* $|q_{ij}| \geqslant 1$:
        devolver no es positiva
para *cada componente conexa* $\mathcal{C}$ *de* $\boldsymbol{B}_q$
    $n =$ número de vértices que contiene $\mathcal{C}$
    si $\mathcal{C}$ *no es isomorfa a* $\mathbb{A}_n, \mathbb{D}_n, \mathbb{E}_6, \mathbb{E}_7, \mathbb{E}_8$
        devolver
    no es positiva
devolver es positiva y es de tipo Dynkin $\mathbf{B}_q$

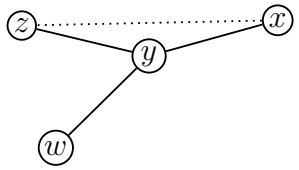
Veamos como funciona este algoritmo sobre la forma cuadrática de la ecuacion (1.5).

1. Comenzamos con la forma unitaria



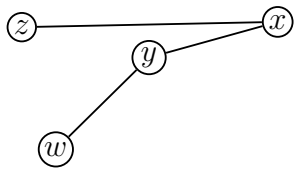
$x^2 + y^2 + z^2 + w^2 - xy + yz - yw - zw$

2. Tenemos a la arista $y \cdots\cdots z$ 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 $\mathbb{A}_4$, 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 $\overrightarrow{x} \in \mathrm{Z}^n$ es una **raíz** de la forma unitaria q si $q(\overrightarrow{x}) = 1$.
- Primero mostramos que toda forma cuadrática se puede poner en la forma

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

- 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 especificamente 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 tambien.

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**:

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

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

1. Si $q = q_{nn}x_n^2$ es una forma cuadratica 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 terminos que contienen la variable x_k:

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

   donde $r = \sum_{i=\ell}^{n} q_{ii}x_i^2 + \sum_{j=\ell+1}^{n}\sum_{i=\ell}^{j_1} q_{ij}x_i x_j$
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. Complentando el cuadrado sustituyendo 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 $c = 0$

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

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 $(x_\ell, x_{\ell+1}, \ldots, x_n)$ (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 $q(\overrightarrow{x})$ se puede reescribir como*

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

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:*

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

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

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

Finalmente en la tercera iteracíon 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

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

donde

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

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

$$q\left(\overrightarrow{x}\right) = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2$$
$$y_i = x_i + \sum_{j=i+1}^{n} \rho_{ij} x_j$$

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 $\overrightarrow{x}$ es una raíz) entonces $\lambda_i y_i^2 \leqslant 1$ para cada i.

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

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

de donde

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

Para continuar es necesaria la siguiente desigualdad:

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

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

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

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

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

Por lo tamto $\sigma_k = \frac{1}{\sqrt{\lambda_k}} + \sum_{j=k+1}^{n} |\rho_{kj}| \, \sigma_j$ es una constante tal que $-\sigma_k \leqslant x_k \leqslant \sigma_k$. Por inducción concluimos que si $\overrightarrow{x}$ 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:

$$S(q) = \{\overrightarrow{x} \in \mathbb{Z}^n | q\left(\overrightarrow{x}\right) = 1\}$$
$$R(q) = \{\overrightarrow{x} \in \mathbb{N}^n | q\left(\overrightarrow{x}\right) = 1\}$$

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 - - - - 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 $T_{rs}^{-} : \mathbb{Z}^n \to \mathbb{Z}^n$ está definida por una matriz $\mathbb{Z}$-invertible entonces T_{rs}^{-} es invertible y su inversa es T_{rs}^{+}.

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

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

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

Denotamos con $\overrightarrow{e} = (d_1, d_2, \ldots, d_n)$ al vector tal que d_k y d_i para todo $i \neq k$. Sea $\overrightarrow{z} = \overrightarrow{e_s} - \overrightarrow{e_r}$. Notamos que

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

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

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

$$|S(q_1)| = |S(q_2)| = |S(q_3)| = \cdots$$
$$|R(q_1)| < |R(q_2)| < |R(q_3)| < \cdots$$

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 pseudocodigo 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 habiamos 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 $m \geqslant m'$). La figura 2.3 muestra cómo dibujar un $\mathbb{A}$-bloque.[1][4]

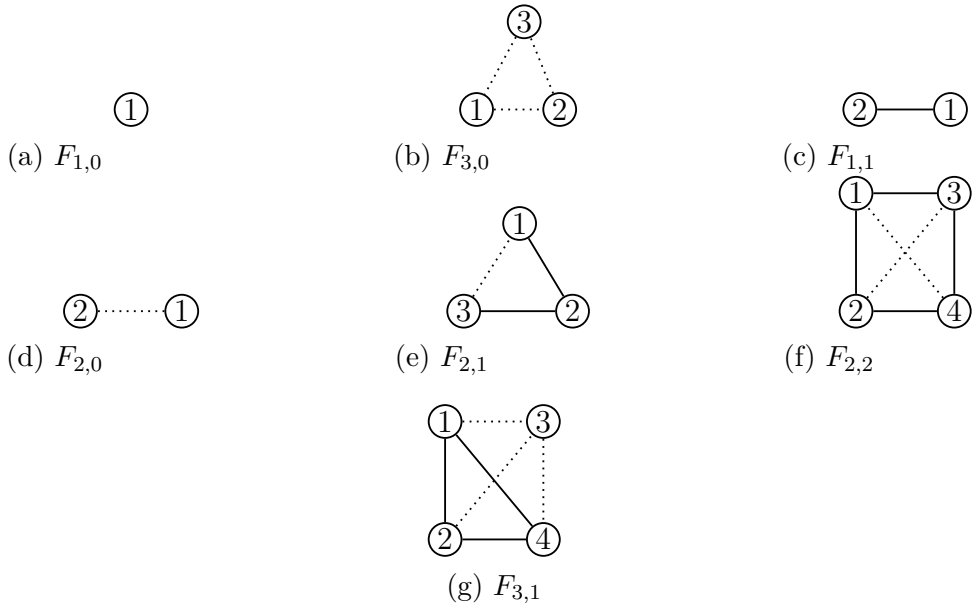


Figura 2.2: Algunos ejemplos de 𝔸-bloques.

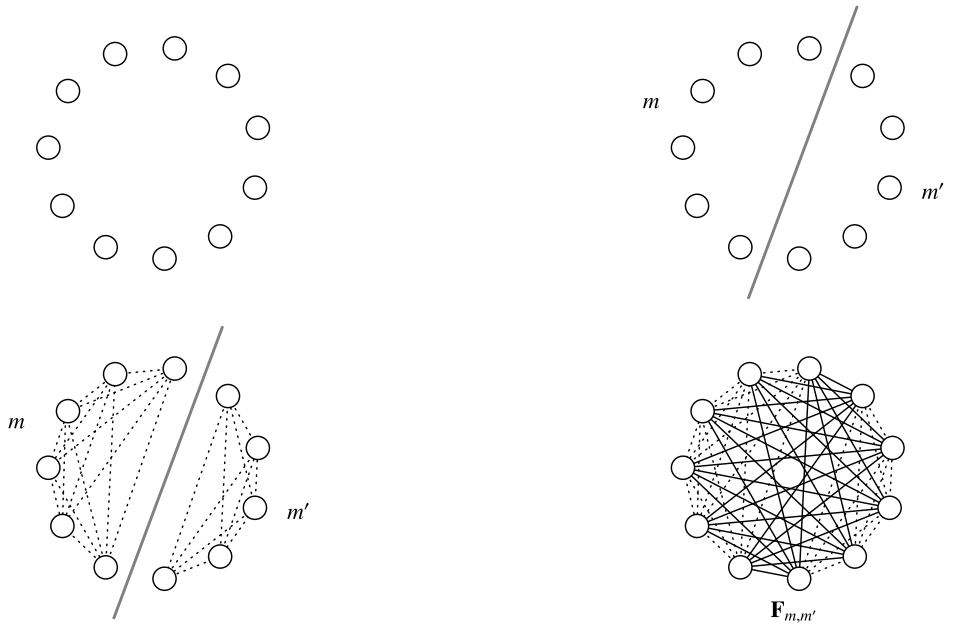
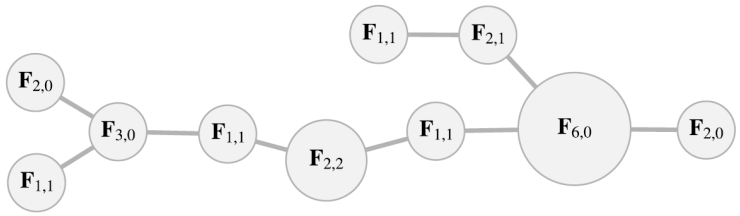


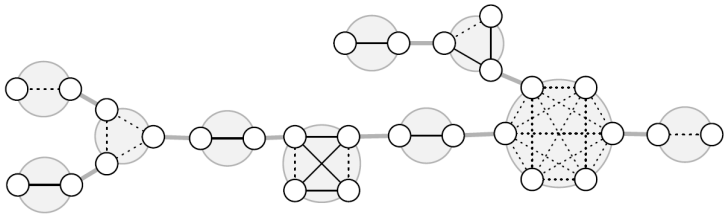
Figura 2.3: Como se construye un 𝔸-bloque.

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

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



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



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

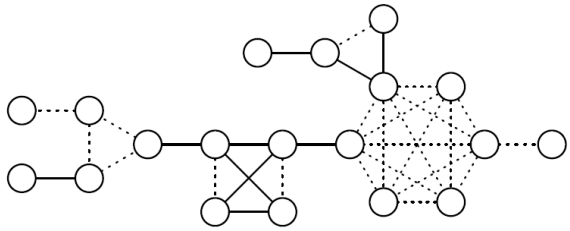


Figura 2.4: Esta gráfica define a una forma de tipo $\mathbb{A}_{19}$

LOS $\mathbb{A}$-BLOQUES

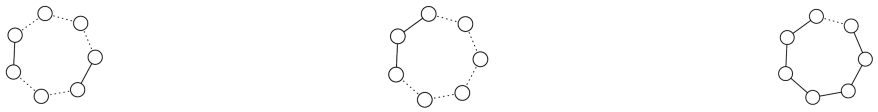
A continuación trataremos de justificar el ensamble de $\mathbb{A}$-bloques a partir del siguiente resultado tomado de [5]:

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 los 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)

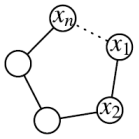
Trataremos de simplificar un poco el problema. Las gráficas circulares de la figura 2.5 definen formas $\mathbb{Z}$-equivalentes: podemos pasar de una a otra mediante las matrices elementales E_i^{-1} de tamaño n. No es difícil comprobar que estas matrices tienen el efecto de intercambiar aristas sólidas 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.



(a) (b) (c)

Figura 2.5: Ejemplos de gráficas circulares.

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



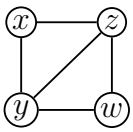
Con el algoritmo 2.1 podemos comprobar directamente que $C\,(4)$ tiene tipo Dynkin D_4. Si $n \geqslant 4$, aplicando la inflación T_{n1}^{-} y borrando el vértice x_n de $C\,(n)$ se obtiene $C\,(n-1)$, de modo que por inducción y el teorema 2.5, todas las gráficas $C\,(n)$ con $n \geqslant 3$ no pueden 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 disconexa, entonces $\kappa \geqslant k$. Por ejemplo, toda gráfica conexa es automáticamente 1-conexa y toda gráfica completa K_n es k-conexa para todo $k \in \{1, 2, \ldots, n\}$. 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 biconexa(2-conexos). 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 disconexa y defínase $\kappa = |S|$. Sabemos que $\mathbf{B}_q$ es 2-conexa, y además que la ausencia de la arista u——v porque $\mathbf{B}_q$ no es completa) nos permite obtener una gráfica disconexa borrando todos los

vértices excepto u y v; por lo tanto tenemos que $2 \leqslant \kappa \leqslant n-2$. 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 e y tales que al borrarlos de $B_q^{'}$ 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——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 $q : \mathbb{Z}^n \to \mathbb{Z}$ es una forma unitaria de tipo $\mathbb{A}_n$ y si $\boldsymbol{B}_1$ es una gráfica biconexa, entonces $\boldsymbol{B}_q$ es una gráfica completa.*

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

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

Demostración. Fijemos un vértice u y definamos los conjuntos

$$V_0 = \{u\} \cup \{v| \text{ existe una arista punteada } u \cdots\cdots v\}$$
$$V_1 = \{v| \text{ existe una arista solida } u \text{——} v\}$$

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 $v \cdots\cdots w$; y en otro caso (están en conjuntos diferentes) hay una arista sólida v——w; por lo tanto Bq es un $\mathbb{A}$-bloque. □

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 $F_{2,1}$.

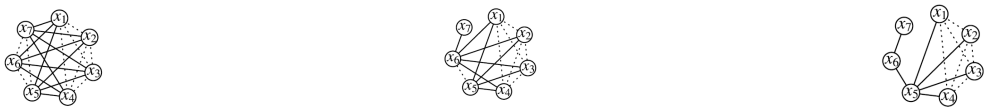
4. Entonces, por lema 2.8 concluimos que $\mathbf{B}_q$ es un $\mathbb{A}$-bloque.

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, renombremos a los vértices en V_0 como $x_1, x_2, \ldots, x_m$; y a los de V_1 como $x_{m+1}, x_{m+2}, \ldots, x_n$. 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 $i = n-1, n-2, \ldots, m+1$ y en la segunda las punteadas que hay en V_0 usando las inflaciones T^-_{i+1i} en orden $i = 1, 2, \ldots, m-1$.

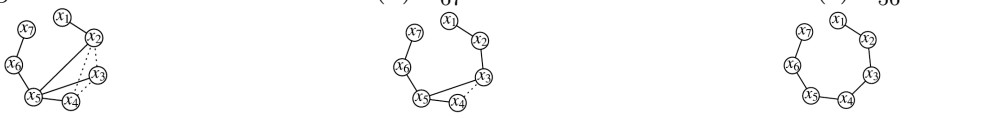
Para mostrar que este método de deshilado funciona, supongamos que $m \geqslant 0$ y $m' \geqslant 0$ y sean $V_0 = \{x_1, \ldots, x_m\}$ y $V_1 = \{x_{m+1}, \ldots, x_n\}$ los conjuntos que definen a $F_{m,m'}$. Sea x_r incidente en x_{n-1}.

- Si $x_r = x_n$ entonces la arista x_{n-1}·····x_n se sustituye por x_{n-1}——x_n.
- Para todos los x_r tales que exista una arista punteada x_r·····x_{n-1} se tiene que $x_r \in V_1$ y el arrastre forma una arista sólida x_r——x_n que se cancela con la arista x_r·····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——x_{n-1} se tiene que $x_r \in V_0$ y el arrastre forma una arista punteada x_r·····x_n que se cancela con la arista x_r——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}——x_n; luego entonces por inducción se sigue la sucesión de inflaciones $\left(T^-_{ii+1}\right)_{i=n-1}^{m+1}$ sobre el grafo $F_{m,m'}$ produce la gráfica $F_{m,1}$ unido con x_{m+1}——$\cdots$——x_n. Un razonamiento similar muestra que la sucesión de inflaciones $\left(T^-_{i+1i}\right)_{i=1}^{m-1}$ aplicadas a $F_{m,1}$ produce $F_{m,m'}$. Por lo tanto el método descrito transforma $F_{m,m'}$ en $\mathbb{A}_{m,m'}$. *



(a) Original (b) T^-_{67} (c) T^-_{56}



(d) T^-_{21} (e) T^-_{32} (f) T^-_{43}

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

Se ha demostrado lo siguiente:

Lema 2.9. *q es una forma unitaria de tipo $\mathbb{A}_n$ con $\boldsymbol{B}_q$ biconexa si y solamente si $\boldsymbol{B}_q$ 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 deshilado explicada justo antes del lema 2.9, pero renombrando al vértice v como $x_{m,m'}$, de manera que este deshilado 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:

(a) (b) (c) (d)

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 $i = m+m^{'}-1, m+m^{'}-2, \ldots, 3, 2, 1$ (porque precisamente este es el proceso inverso al de deshilado). 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. *

(a) Original (b) T_{23}^{-} (c) T_{32}^{+} (d) T_{21}^{+}

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 biconexos es mediante los **puntos de articulación**, que son los vértices de la gráfica que al quitarlos la deja desconectada.

(a) (b)

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 subgrafica biconexa, y por lo tanto pertenece a una subgrafica 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. □

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 $\mathcal{B}_1, mathcalB_2, \ldots, \mathcal{B}_t$ son las componentes biconexas de $\mathbf{B}_q$, y formemos una nueva gráfica T de vértices $\{1, 2, \ldots, t\}$, donde cada arista i——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 $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$.

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 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 siguente teorema:

Teorema 2.12. *Una forma unitaria q es de tipo $\mathbb{A}_n$ si y solamente si $\boldsymbol{B}_q$ 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 $\boldsymbol{B}_q$ es conexa, sus componentes biconexas son $\mathbb{A}$-bloques y todo punto de articulación pertenece a exactamente dos de estas componentes.*

3 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 electricos [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 dirigido. 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 busqueda primeron 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 el la gráfica tiene V vértices([2], [17]). El algoritmo descrito aquí requiere sustancialmente utilizamos lña combinacion de los articulos [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, $|f(x)| \leqslant k_1 |g(x)| + k_2$ para todo x

3.1.2. Gráficas, conectividad y busqueda 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 $G = (\mathscr{V}, \mathscr{E})$ consiste de un conjunto $\mathscr{V}$ que contiene V vertices y un conjunto $\mathscr{E}$ que contiene E aristas. Si $\mathscr{E}'$ es un conjunto de aristas en G, $\mathscr{V}\left(\mathscr{E}'\right)$ es el conjunto de vertices que inciden a uno o más aristas en $\mathscr{E}'$. Si S es un conjunto de vertices en G, $\mathscr{E}(\mathscr{S})$ es el conjunto de aristas incidentes a al menos un vertice en S.

Un **camino** $p : v \overset{*}{\Rightarrow} w$ en G es una secuencia de vertices y aristas que van de v a w. Un camino es simple si todos sus vertices son distintos. Un camino $p : v \overset{*}{\Rightarrow} v$ es un **ciclo** si todas las aristas son distintas y el único vertice que se repite en p es v y este está al principio y al final de la secuencia de vertices. Un gráfica no dirigida(sin dirección) es conexa(esta conectada) si para cada par de vertices (v, w) existe un camino entre (v, w). Sea $G = (\mathscr{V}, \mathscr{E})$ y sea $G' = \left(\mathscr{V}', \mathscr{E}'\right)$ dos gráficas tales que $\mathscr{V}' \subseteq \mathscr{V}$ y $\mathscr{E}' \subseteq \mathscr{E}$ entonces G' es una subgráfica de G. Una gráfica con exactamente dos vertices 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 $v \to w$. La relación "hay un camino de v a w en T" se denota por $v \overset{*}{\to} w$. Si $v \to w$, donde v es el padre de w, y w es un hijo de v.Si $v \overset{*}{\to} w$, entonces v en 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 arbol T es un **arbol generador** de G si T es un subgráfica de G y T contiene todos los vertices de G.

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

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

Entonces a P llamaremos árbol de recorrido. Las aristas $v - \to w$ 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.

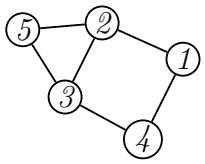
Los subgráficas G_i de G se denominan **componentes biconexas** de G. Las componentes biconexas de G son únicas.

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

1. *$\forall e \in E - F$, $e \tilde{=} e$*
2. *$\forall e, f \in E - F$ con $e : a_1\text{-}b_1$ y $f : a_2\text{-}b_2$, entonces $e \tilde{=} f$ si y solo si existe un camino de los siguientes tipos:*
   - *a_1—b_1—v_1—$v2$—$\cdots$—v_k—a_2—b_2 tal que $b_1, v_1, \ldots, v_k, a_2 \notin W$*
   - *a_1—b_1—v_1—$v2$—$\cdots$—v_k—b_2—a_2 tal que $b_1, v_1, \ldots, v_k, b_2 \notin W$*
   - *b_1—a_1—v_1—$v2$—$\cdots$—v_k—a_2—b_2 tal que $a_1, v_1, \ldots, v_k, a_2 \notin W$*
   - *b_1—a_1—v_1—$v2$—$\cdots$—v_k—b_2—a_2 tal que $a_1, v_1, \ldots, v_k, b_2 \notin W$*

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

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



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

La clase de equivalencia de 2—5 es el conjunto:

$$S_1 = \{2\text{—}5, 5 \cdots 3\}$$

La clase de equivalencia de $1 \cdots 4$ es el conjunto:

$$S_2 = \{2 \cdots 1, 1\text{—}4, 4\text{—}3\}$$

La clase de equivalencia de 2—3 es el conjunto:

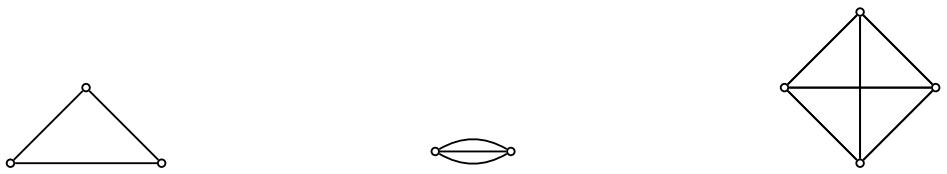
$$S_3 = \{2\text{—}3\}$$

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 $|E_1 = \bigcup_{i \in A} S_i| \geqslant 2$ y $|E_2 = \bigcup_{j \in B} S_j| \geqslant 2$. 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 $S_1, S_2, \ldots, S_k$ son las clases de separación del par $\{a, b\}$(las clases de equivalencia definidas por H).Sea A, B la partición del conjunto $\{1, 2, \ldots, k\}$ tal que $|E_1 = \bigcup_{i \in A} S_i| \geqslant 2$ y $|E_2 = \bigcup_{j \in B} S_j| \geqslant 2$. Si $H_1 = (V(E_1), E_1)$ y $H_2 = (V(E_2), E_2)$ entonces, $V(E_1) \cap V(E_1) = \{a, b\}$ 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**.



(a) Triangulos (b) Enlaces (c) Poligonos

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 $|E| \geqslant 3$. Sea $G_1, G_2, \ldots G_m$ los componentes de separación de G. Entonces el número total de aristas en $G_1, G_2, \ldots, G_m$ 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 $2 \leqslant k \leqslant n-2$. Por inducción, el número total de aristas en $G_1, G_2, \ldots G_m$ debe estar acotado por $3(k+1) - 6 + 3(n-k+1) - 6 = 3n - 6$. Así, por inducción, el lema 1 es cierto. □

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. triangulos 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 $\mathscr{B}_3$, un conjunto de triángulos $\mathscr{F}$ y un conjunto de gráficas triconexas $\mathscr{C}$. Supongamos que los enlaces triples $\mathscr{B}_3$ se unen tanto como sea posible para dar un conjunto de enlaces $\mathscr{B}$ y que los triángulos $\mathscr{F}$ se unen tanto como sea posible para dar un conjunto de polígonos $\mathscr{P}$. Entonces el conjunto de gráficas $\mathscr{B} \cup \mathscr{P} \cup \mathscr{C}$ 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]. □

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

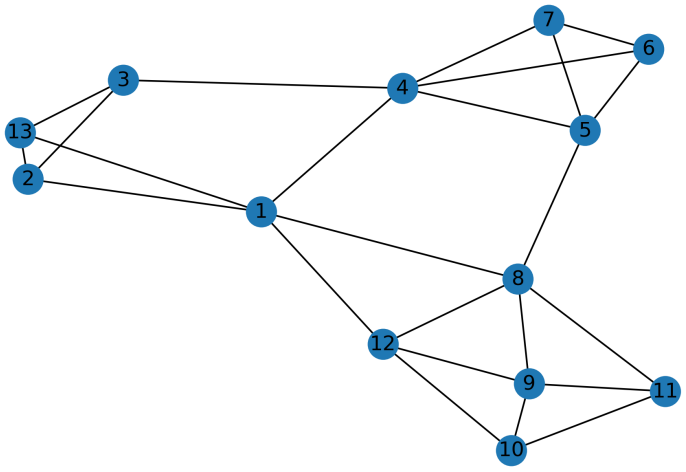


Figura 3.2: Una gráfica biconexa G con pares de separación $(1,4),(4,5),(1,8),(12,8)$

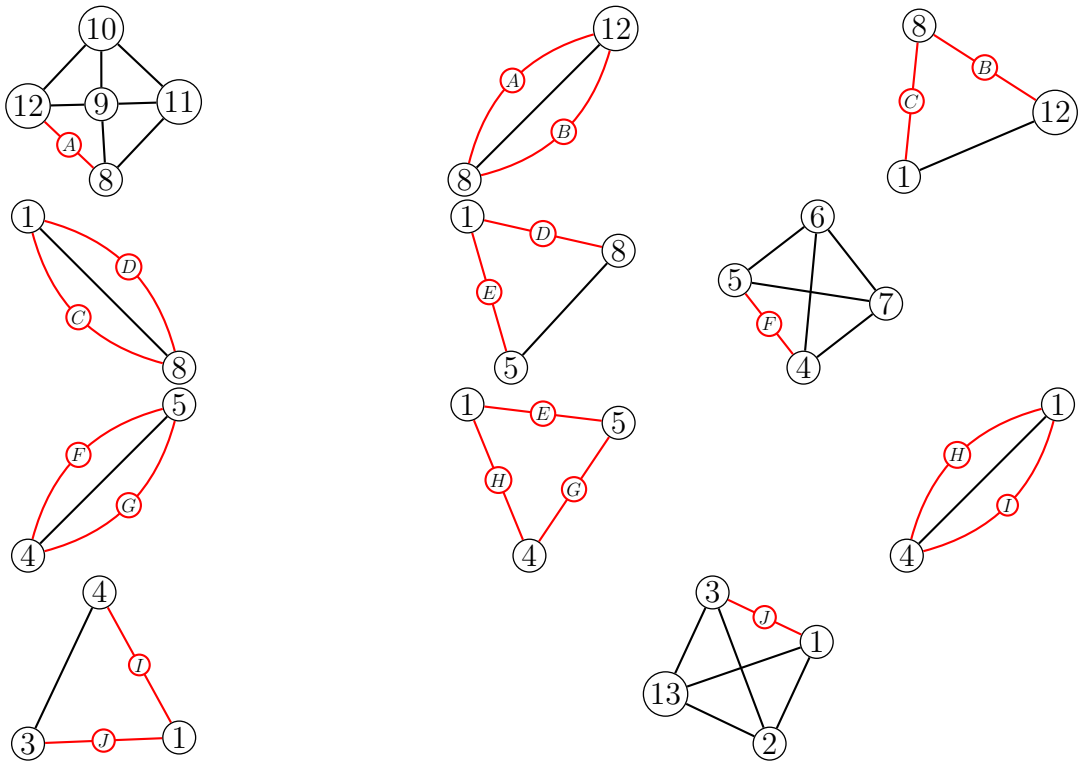


Figura 3.3: Componentes de separación de G las componentes triconexas se forma al unir los triangulos $(1,4,5)$ y $(1,5,8)$

Los algoritmos de gráficas requieren una forma sistemática de explorar una gráfica. En el articulo [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 vistado 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)

```
   // a: declaración vacía
1
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)
           // b: declaración vacía
6
7      si no (NUMBER(w) < NUMBER(v)) and ((w ≠ u) or ¬FLAG(v))
8          marcamos (v, w) como arista de retroceso
           // b: declaración vacía
9
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)
```

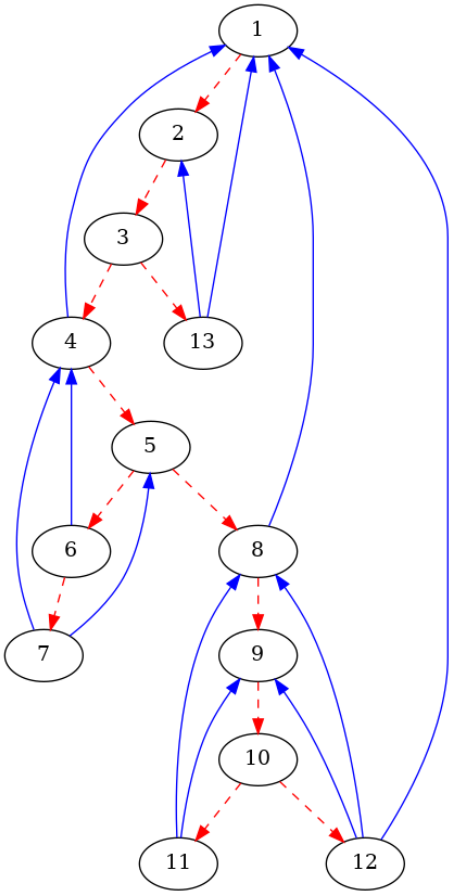


Figura 3.4: Árbol de recorrido generado por la busqueda primero en profundidad de G en 3.2

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 $S_1, \ldots, S_m$ 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 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 $\mathscr{B}_3$ y el conjunto de triángulos $\mathscr{T}$. 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 $S(\mathscr{B}_3)$ y $S(\mathscr{T})$ 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 el siguientes dos secciones. Con base en los resultados de estas secciones, toda el algoritmo de triconectividad esta acotado por $\Theta(V+E)$.

3.2.1. Encontrando pares de separación

Sea $G=(\mathscr{V},\mathscr{E})$ 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 separación y se describen a continuación. Las definiciones de las cantidades lowpt1, ND, etc., utilizadas en el el esquema se dará posteriormente.

Paso 1 Realice una búsqueda primero en profundidad en el gráfica G, convirtiendo G en un árbol P. Numeramos los vértices de G en el orden en que se alcanzan durante la búsqueda. Calculamos lowpt1(v), lowpt2(v), nd(v) y 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 lowpt1(v) y lowpt2(v) utilizando los nuevos números de los vértices. Calculamos A1(v), deg(v), y 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 $v \overset{*}{\rightarrow} w$ en P implica que $v \leqslant w$, si identificamos los vértices por su número. Para cualquier vértice v en P, sea padre(v) el padre de v en el árbol de P. Sea ND(v) el número de descendientes de v. Sea $\text{lowpt1}(v) = min\left(\{v\} \cup \{w | v \overset{*}{\rightarrow} \text{-} \rightarrow w\}\right)$. Es decir, 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 $\text{lowpt2}(v) = min\left[\{v\} \cup \left(\{w | v \overset{*}{\rightarrow} \text{-} \rightarrow w\} - \{lowpt1\,(v)\}\right)\right]$. Es decir, 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 lowpt1(v) = v. En este caso, lowpt2(v) = v.

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

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

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

Demostración. lowpt1(v) siempre identifica a un ancestro del vertice v. Además,
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, lowpt1(v) identifica a un único vértice independiente de la núméración, 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 lowpt2(v). □

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

$$lowpt1\,(v) = min\,(\{v\} \cup \{lowpt1\,(w)\,|v \rightarrow w\} U \{w|v\text{-} \rightarrow w\})$$

y

$$\begin{aligned} lowpt2\,(v) = min(\{v\} \cup ((\{&lowpt1\,(w)\,|v \rightarrow w\} \\ &\cup \{lowpt2\,(w)\,|v \rightarrow w\} \\ &\cup \{w|v\text{-} \rightarrow w\}) - \{lowpt1v\})). \end{aligned}$$

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 $\mathrm{nd}(v) = 1 + \sum_{v \to w} \mathrm{ND}(w)$. 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 $v \overset{*}{\to} w$ implica $v \leqslant w$.

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 $v \to w$, lowpt1$(w) \leqslant v$ 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 $\{1, 2, \ldots, 2V + 1\}$ definido por:

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

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 $1, 2, \ldots, 13$ 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 arbol de una gráfica biconexa G cuyos vértices son numerados de manera que $v \overset{*}{\to} w$ en P implica $v \leqslant w$. Entonces las estructuras de adyacencia aceptables de P son independientes del esquema de numeración exacto.*

Demostración. Si $v \to w$ en P entonces por el lema 3.5, lowpt2(w) es un ancestro de w. Por el lema 3.6 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 lowpt2(w) es menor que v es independiente de la numeración. Como G es biconexa, si $v \to w$ en P entonces lowpt1(w) $\leqslant v$ por el lema 3.7. Por el lema 3.5, lowpt1(w) es un ancestro de w. Dado que lowpt1(w) $\leqslant v$, lowpt1(w) debe ser un ancestro de v. Por e lema 3.6, el vertice correspondiente a lowpt1(w) es independiente de el esquema de numeración. De manera similar si $v\text{-} \to w$, 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 □

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 vertice. 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 $p : s \overset{*}{\Rightarrow} f$ es un camino generado, podemos formar un ciclo agregando el camino del árbol $f \overset{*}{\to} s$ 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 $v \overset{*}{\to} w$ implica $v \leqslant w$. Sea A1(v) el primer vértice de A(v). Si v- $\to w$ es la primera rama explorada en el paso 3 que termina en w, sea highpt(w) = v. Sea 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 $v \leqslant w$ si $v \overset{*}{\to} w$. El paso 3 también calcula lowpt1(v), highpt(v), A1(v) y 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: $1 \to 2 \to 3 \to 13 \hookrightarrow 1$	2: $13 \hookrightarrow 2$
3: $3 \to 4 \hookrightarrow 1$	4: $4 \to 5 \to 8 \hookrightarrow 1$
5: $8 \to 9 \to 10 \to 12 \hookrightarrow 1$	6: $12 \hookrightarrow 8$
7: $12 \hookrightarrow 9$	8: $10 \to 11 \hookrightarrow 8$
9: $11 \hookrightarrow 9$	10: $5 \to 6 \to 7 \hookrightarrow 4$
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 $u \to v$ 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 $u_0 \to u_1 \to \cdots \to u_n$ y u_i es un hijo primogénito de u_{i-1} para $1 \leqslant i \leqslant n$, entonces a u_n, se le llama el primer descendiente de u_0. La secuencia de ramas del árbol $u_0 \to u_1 \to u_2 \to \cdots \to u_n$ 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 $u \to v$ y $u \to w$ ramas del árbol con v antes de w en $A(u)$. Entonces $u \leqslant w \leqslant v$.*

Demostración. El paso 3 numera los vértices de V a 1 en el orden en que son los últimos examinados en la búsqueda. Si $u \to v$ se explora antes que $u \to w$, 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. □

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, $u \to v$ implica $u \leqslant v$ y por lo tanto por el lema 3.8, A es aceptable para la nueva numeración. □

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

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 $d(v) - d(w) = \{x | v \leqslant x \leqslant w\}$. □

Lema 3.12. *Sean $\{a, b\}$ un par de separación en G con $a \leqslant b$. Entonces $a \overset{*}{\to} b$ en el árbol generador T de P.1*

Demostración. Como $a \leqslant b$, a no puede ser descendiente de b. Supongamos que b no es un descendiente de a. Sean E_i, para $1 \leqslant i \leqslant k$, las clases de separación con respecto a $\{a, b\}$. Sea $S = \mathscr{V} - D(a) - D(b)$. 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 $a \neq 1$, 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. □

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 $b \to r$, lowpt1$(r) = a$, lowpt2$(r) \geqslant b$ 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 $a \to r \overset{*}{\to} b$; b es el primer descendiente de r (es decir, a, r y b se examinarn en una ruta generada); $a \neq 1$; cada rama $x\text{-} \to y$ con $r \leqslant x \leqslant b$ y $a \leqslant y$ y cada rama $x\text{-} \to y$ con $a \leqslant y \leqslant b$ y $b \to w \overset{*}{\to} x$ tiene lowpt1$(w) \geqslant a$. ($\{a, b\}$ 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 $1 \leqslant i \leqslant k$ 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 lowpt1$(r) = a$ y lowpt2$(r) \geqslant b$, 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, $S = \{x | r \leqslant x \leqslant b\}$ por el lema 3.1. Sean $b_1, b_2, \ldots, b_n$ los hijos de b en el orden ocurren en $A(b)$. Sea $i_0 = min\{i |$ lowpt1$(b) \geqslant a\}$. Por el ordenamiento impuesto en A, $i \leqslant i_0$ implica lowpt1$(b_i) \leqslant a$, y $i \geqslant i_0$ implica que lowpt1$(b_i) \geqslant a$. 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 $S \cup \{b\} \cup (\bigcup_{i \geqslant i_0} d(b_i))$. Cada arista con un vertice final en $d(b)$, $i \geqslant i_0$, 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 $S \cup (\bigcup_{i \geqslant i_0} d(b_i))$. 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, $a \overset{*}{\rightarrow} b$. Sea E_i, para $1 \leqslant i \leqslant k$, las clases de separación de G con respecto a $\{a, b\}$. Sea v el hijo de a tal que $a \rightarrow v \overset{*}{\rightarrow} b$, $S = d(v) - d(b)$, 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, lowpt1$(a) \leqslant a$. Esto significa que $E(d(a_i)) \subseteq E_2$. Sean $Y = X \cup (\bigcup_i d(a_i)$. Sean $b_1, b_2, \ldots b_n$ 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 $E(S), E(Y)$, $\{(a, b)\}, E(d(b_1)), E(d(b_2)), \cdots E(d(b_n))$.

Si $E(d(b)) = E_j$ para algún i y j, entonces lowpt1$(b_i) = a$ ya que G es biconexo y esto significa que lowpt1$(b_i) \leqslant b$ por el Lema 3.7. También, lowpt2$(b_i) \geqslant b$. 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 $i_0 = min\{i |$ lowpt1$(b_i) \geqslant a\}$. Si $i \geqslant i_0$, entonces como G es biconexo, debe ser el caso que lowpt1$(b) \leqslant b$, y estas clases de separación son $E_1 = E(S) \cup (\bigcup_{i \geqslant i_0} E(d(b_i)))$,
$E_2 = E(Y) \cup (\bigcup_{i \leqslant i_0} E(d(b_i)))$, $E_3 = \{(a, b)\}$. (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 $x\text{-} \rightarrow y$ es una rama con $v \leqslant x \leqslant b$, entonces $x \in S$, $(x, y) \in E_1$, y $a \geqslant y$. Si $x\text{-} \rightarrow y$ es una rama con $a \leqslant y \leqslant b$ y $b \rightarrow b_i \overset{*}{\rightarrow} x$, entonces $y \in S$, $(x, y) \in E_1$ y $i \geqslant i_0$, lo que significa que lowpt1$(b_i) \geqslant a$. 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, lowpt1$(v) \leqslant a$. 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 $x\text{-} \to y$ con $x \in d(v)$ y $y \leqslant a$ 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. □

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 $a \to r \overset{*}{\to} b$ y $i_0 = min\{i|lowpt2(b_i) \geqslant a\}$, donde $b_1, b_2, \ldots, b_n$, 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 $E(\{x|r \leqslant x \leqslant b_{i_0} + nd(b_{i_0})\} - \{b\})$. 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 $a \to v \to b$ 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 $1 \to v_1 \to v_2 \to \cdots \to v_n$ seguido de una rama $v_n\text{-} \to 1$. La numeración de los vértices es tal que $1 \leqslant v_1 \leqslant \cdots \leqslant v_n$. 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 $(h_1, a_1, b_1), (h_2, a_2, b_2), \cdots, (h_k, a_k, b_k)$ están en tstack, entonces $a_k \leqslant a_{k-1} \leqslant \cdots \leqslant a_2 \leqslant$

$a_1 \leqslant v_i \leqslant b_1 \leqslant b_2 \leqslant \cdots \leqslant b_k$. 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 $p : s \overset{*}{\Rightarrow} f$, borramos todos las tripletas (h_j, a_j, b_j) encima de la pila con $a \geqslant f$. Si p tiene un segundo vértice $v \neq f$, sea $x = v + ND(v) + 1$. De lo contrario, sea $x = s$. Sea $y = max\{h_j | triple(h_t, a_t, b_t)$ 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 $v_i \to v_{i+1}$ con $v \neq 1$, borramos todos entradas (h_j, a_j, b_j) encima de tstack que satisfacen 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 $v_i \to v_{i+1}$ 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 $a \neq$ 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 $\{v,$ 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 $p : s \overset{*}{\Rightarrow} f$ 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 el 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 $w_1, w_2, \ldots, w_k$ son los hijos de v tales que lowpt1$(w_i) = u$. Además, supongamos que $v\text{-} \to u$. Sea la w_i ordenada como en $A(v)$. Hay algún i_0 tal que $i \leqslant i_0 \Rightarrow$ lowpt2$(w_i) \leqslant v$ y $i \geqslant i_o \Rightarrow$ lowpt2$(w_i) \geqslant v$. En $A(v)$, u aparecerá después de el w_i con $1 \leqslant i \leqslant i_0$. Si $i \geqslant i_0$, entonces $\{u, w_i\}$ es un par de separación tipo 1; dividiendo los correspondientes componentes produce una nueva arista(virtual) $v\text{-} \to u$. Es importante que todos los w_i con $i \geqslant i_0$ 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 $e_1, e_2, \ldots, e_\ell$ aristas en ESTACK
3.1 $\mathcal{C}$ = nuevo componente $(e_1, \ldots, e_\ell)$

Algoritmo 3.5: PATHSEARCH(v)

para $e \in A(v)$*:*
  si $e = v \to w$
    si *e empieza un camino*
      eliminar todos los (h, a, b) con $a > lowpt1(w)$ de tstack
      si *no se eliminar tripletas*
        agregar a tstack $(w+ \text{ ND}(w) - 1, \text{ lowpt1}(w), v$
      en otro caso
        $y = max\{h|\,(h, a, b)$ eliminados de tstack$\}$
        sea (h, a, b) la ultima tripleta eliminada
        agregar a tstack $(max(y, w+ \text{ ND}(w) - 1), \text{ lowpt1}(w), b)$
      agregar a tstack (EOS)
    PATHSEARCH(w)
    agregamos a ESTACK $(v \to w)$
    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
        $y = max\{h|$ eliminado de tstack$\}$
        sea (h, a, b) la utima tripleta eliminada
        agregamos (y, w, b) a tstack
    si $w \in parent(v)$
      $\mathcal{C}$ = nuevo componente $(e, w \to v)$
      $e' =$ nueva arista virtual $(w, v, \mathcal{C})$
      crear arista vitual $\left(e', w \to v\right)$
    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* $a \leqslant x \leqslant h$ *y* $a \leqslant y \leqslant h$
        si $(x,y) = (a,b)$
          e_{ab} es el primer elemento que eliminamos de ESTACK
        en otro caso
          $\mathcal{C} = \mathcal{C} \cup \{$ primer elemento eliminado de ESTACK $\}$
    si $e_{ab} = nulo$
      $\mathcal{C}$ = nuevo componente $\left(e_{ab}, e^{'}\right)$ $e^{'}$ = nueva arista virtual $(v,b,\mathcal{C})$
    agregamos $e^{'}$ a ESTACK
    creamos la arista del árbol $\left(e^{'}, v \to b\right)$
    $w = b$

Algoritmo 3.7: checkTipo1(v)

6.1 **si** *lowpt2*$(w) \geqslant v$ *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* $w \leqslant x \leqslant w+ ND(w)$ *o* $w \leqslant y \leqslant w+ ND(w)$
    $\mathcal{C} = \mathcal{C} \cup \{$ESTACK.pop()$\}$
  $e^{'}$ = nueva arista virutal $(v,$ lowpt1$(w)\,,\mathcal{C})$
  si *ESTACK.pop()* $= (v, lowpt1\,(w))$
    $\mathcal{C}$ = nuevo componente $\left(\textbf{ESTACK.pop}(), e^{'}\right)$
    $e^{'}$ = 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$(w) \to v)$
  en otro caso
    $\mathcal{C}$ = nuevo componente $(e^{'},$ lowpt1$(w) \to v)$
    $e^{'}$ = nueva arista virtual (lowpt1$(w)\,, v, \mathcal{C})$
    crear arista de l árbol $(e^{'},$ lowpt1$(w) \to v)$

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 (lowpt1$(w) \neq 1$) o (padre$(v) \neq 1$) o $(w \geqslant 3)$ 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$(h_1, a_1, b_1), \cdots, (h_k, a_k, b_k)$ 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 $a_k \leqslant a_{k-1} \leqslant \cdots \leqslant a_1 \leqslant v \leqslant b_1 \leqslant \cdots \leqslant b_k$. 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, $a_k, a_{k-1} \cdots b_k$ 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 $v \to w$. La prueba (B, E en PATHSEARCH) establece que $a = v$, $v \neq 1$ y padre$(b) \neq a$. Se sigue que $r = A1\,(a) \neq b$ satisface $a \to r \overset{*}{\to} b$ 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 $x\text{-} \to y$ con $r \leqslant x \leqslant b$ tuviera $a \geqslant y$, 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 $x\text{-} \to y$ con $a \leqslant y \leqslant b$ y $b \to w \overset{*}{\to} x$ tenía lowpt1$(w) \leqslant a$, 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 $b_1, b_2, \ldots, b_n$, los hijos de b en el orden en que aparecen en $A\,(b)$. Sea $i_0 = min\{i|$ lowpt1$(b_i) \geqslant a\}$. Si i_0 existe, entonces (b_{i_0}+ND(b_{i_0}), lowpt1 $(b_i)\,, b)$ se colocará en tstack cuando la rama del árbol $b \to b_i$ se explora. Esta tripleta puede ser borrado de tstack, pero siempre será reemplazado por una tripleta de la forma (h, x, b), con lowpt1$(b) \geqslant x \geqslant a$. 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 $a \leqslant i$ y $j \leqslant b$. Si $i\text{-} \to j$, 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 $i \to j$, entonces (j+ND(j) ,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. □

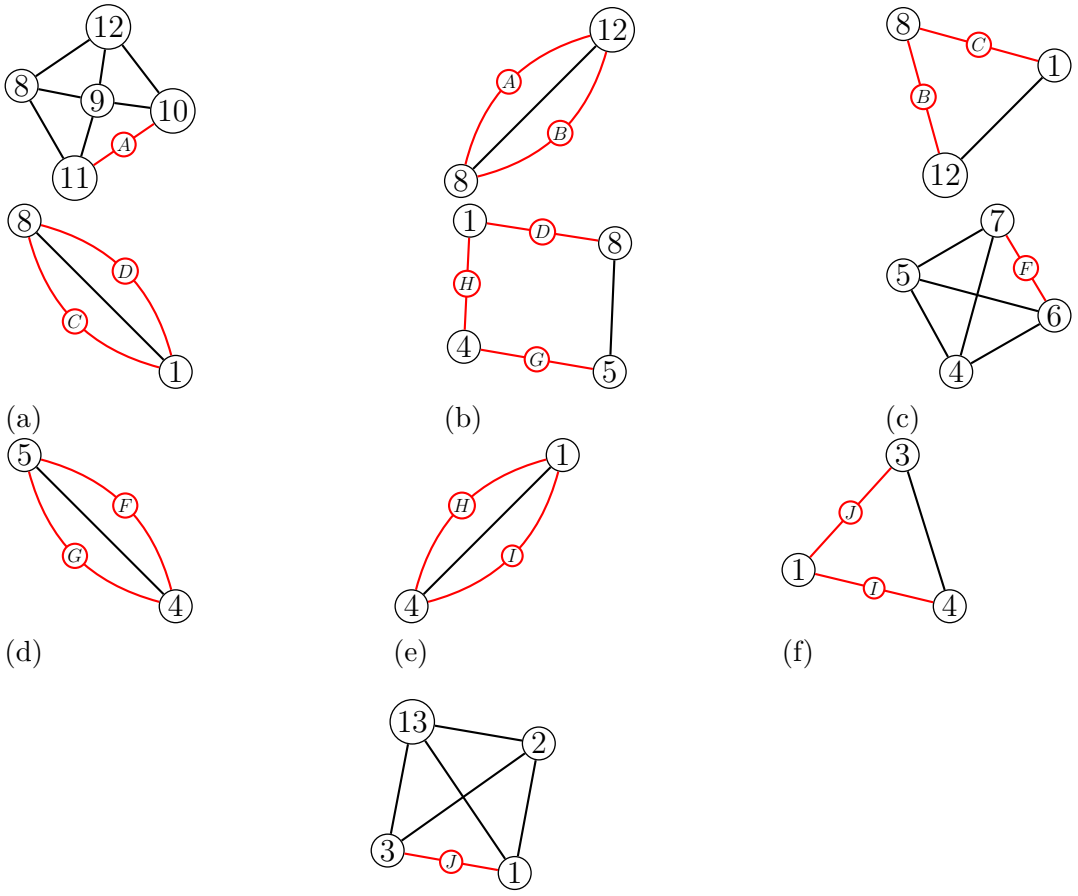


Figura 3.5: Componentes triconexas finales de la gráfica 3.2

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.. Considerese 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 $\Theta(V+E)$. □

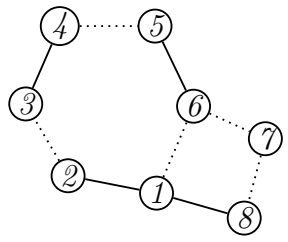
4 Clasificación $\mathbb{D}_n$

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.

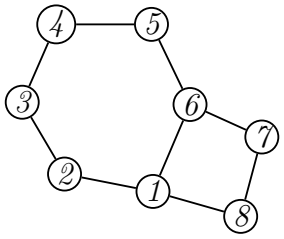


Una pareja de aristas paralelas se considera un ciclo de longitud 2.

- Una bigráfica cíclica $H = x_1 - x_2 - \cdots - x_h - x_1$ (todos los x_i distintos para $1 \leqslant i \leqslant h$) 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 algoritmica de $\mathbb{D}_n$

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

1. Descomponemos la bigrafica 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 bigrafica es de tipo $\mathbb{D}_n$ comn $n = |V|$

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

4.1.2. Clasificación algoritmica de $\mathbb{D}_n$

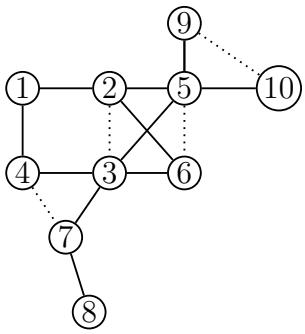
En [21] existe una implementación de el algoritmo de componentes triconexas en el lenguaje Python la cual verifica si es una gráfica triconexa y te devuelve las componentes triconexas finales de la gráfica ingresada solo funciona con gráficas por lo cual utilizaremos el marco de la bigráfica para poder 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:

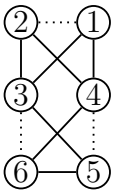
- con ciclo > 2
- con ciclo $= 2$

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 siquientes ejemplos:



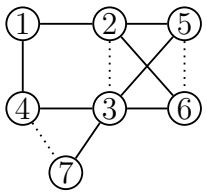
(a) Gráfica G_1 con ciclo > 2



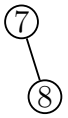
(b) Gráfica G_2 con ciclo $= 2$

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

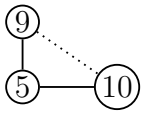
Descomponemos en su componentes biconexas:



(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:

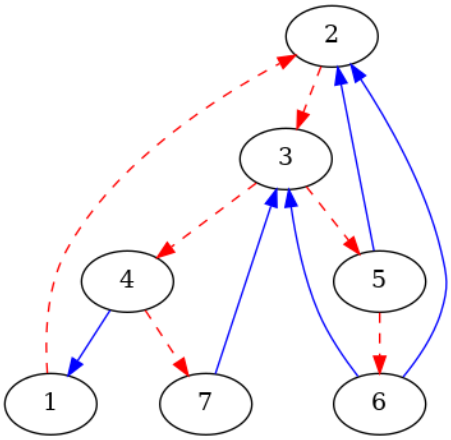


Figura 4.3: Árbol generado en el recorrido primero en profundida en $G_{1,1}$

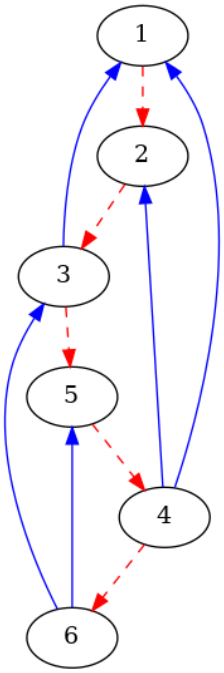


Figura 4.4: Árbol generado en el recorrido primero en profundida en G_2

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: $1 \to 2 \to 3 \to 4 \hookrightarrow 1$ 2: $4 \to 7 \hookrightarrow 3$
3: $3 \to 5 \to 6 \hookrightarrow 2$ 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: $1 \to 2 \to 3 \to 5 \to 4 \hookrightarrow 1$ 2: $4 \hookrightarrow 2$
3: $4 \to 6 \hookrightarrow 3$ 4: $6 \hookrightarrow 5$
5: $3 \hookrightarrow 1$

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

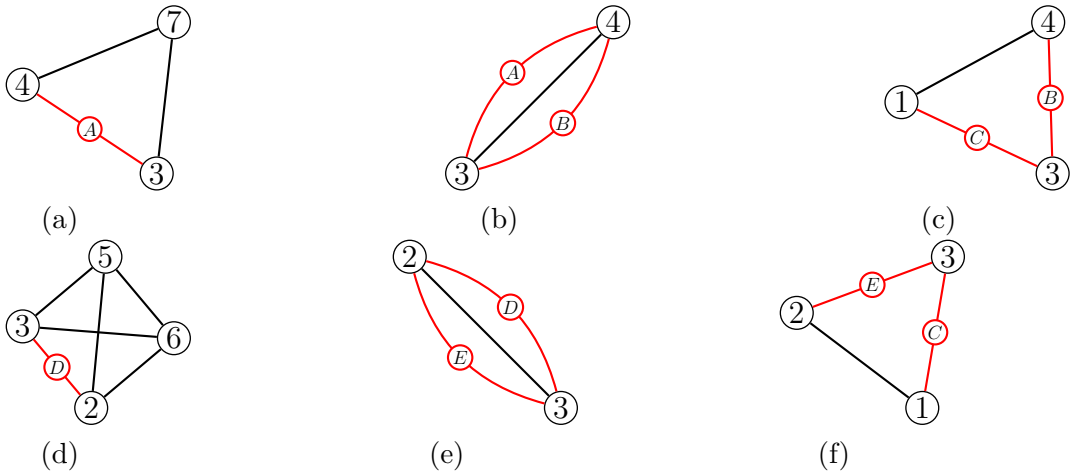


Figura 4.5: Componentes de separación de $G_{1,1}$

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

Las componentes triconexas finales de las bigráficas:

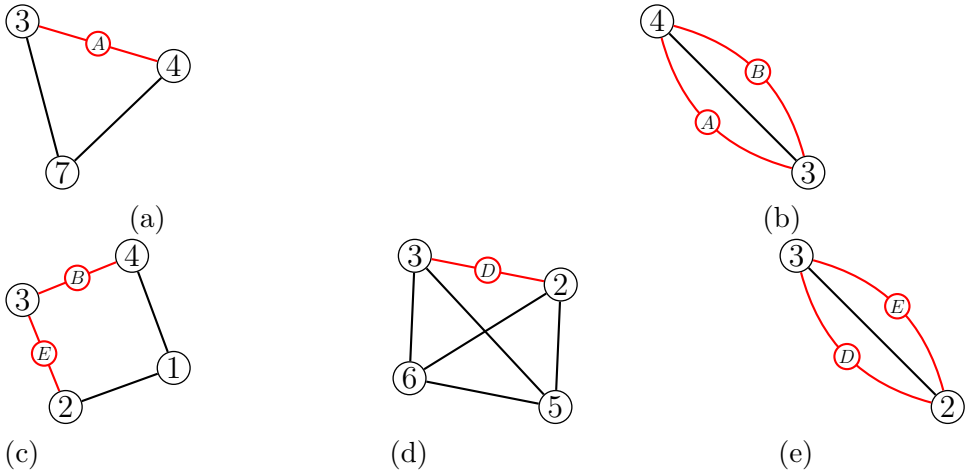
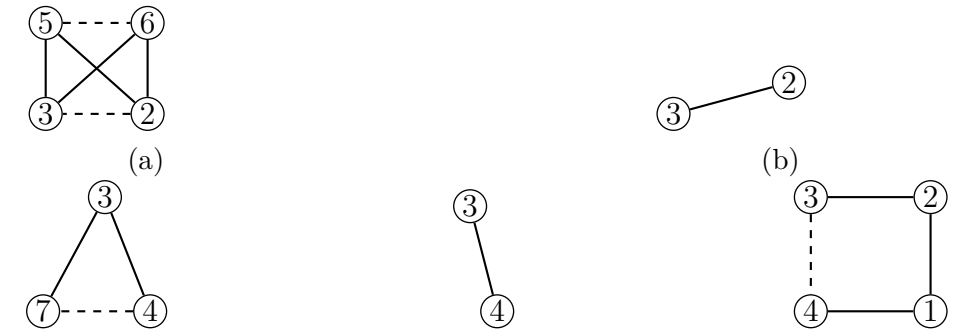


Figura 4.6: Componentes triconexas de $G_{1,1}$

(a) (b)

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.



(a) (b) (c) (d) (e)

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

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

(a) (b)

Figura 4.9: Componentes triconexas de G_2

5

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

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

```
1
2  class _LinkedListNode:
3
4          def __init__(self, data=None):
5
6                  self.data = data
7                  self.prev = None
8                  self.next = None
9                  self.set_data(self.data)
10
11         def set_data(self, data):
12                 self.data = data
13
14         def get_data(self):
15                 return self.data
16
```

```
1
2  class _LinkedList:
3
4          def __init__(self):
5
6                  self.head = None
7                  self.tail = None
8                  self.length = 0
9
10         def remove(self, node):
11
12                 if node.prev is None and node.next is None:
```

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

```
66
67          def concatenate(self, lst2):
68
69                  self.tail.next = lst2.head
70                  lst2.head.prev = self.tail
71                  self.tail = lst2.tail
72                  self.length += lst2.length
73                  lst2.head = None
74                  lst2.length = 0
75
```

```
1
2   class _Component:
3
4           def __init__(self, edge_list, type_c):
5
6                   self.edge_list = _LinkedList()
7                   for e in edge_list:
8                           self.add_edge(e)
9                   self.component_type = type_c
10
11          def add_edge(self, e):
12                  self.edge_list.append(_LinkedListNode(e))
13
14          def finish_tric_or_poly(self, e):
15
16                  self.add_edge(e)
17                  if self.edge_list.get_length() > 3:
18                          self.component_type = 2
19                  else:
20                          self.component_type = 1
21
22          def __str__(self):
23
24                  if self.component_type == 0:
25                          type_str = "Bond: "
26                  elif self.component_type == 1:
27                          type_str =  "Polygon: "
28                  else:
29                          type_str = "Triconnected: "
30                  return type_str + self.edge_list.to_string()
31
32          def get_edge_list(self):
33
34                  e_list = []
35                  e_node = self.edge_list.get_head()
36                  while e_node:
37                          e_list.append(e_node.get_data())
38                          e_node = e_node.next
39                  return e_list
40
```

```
1
```

```
class _LinkedListNode:

        def __init__(self, data=None):

                self.data = data
                self.prev = None
                self.next = None
                self.set_data(self.data)

        def set_data(self, data):
                self.data = data

        def get_data(self):
                return self.data



class TriconnectivitySPQR:

        def __init__(self, G, check=True):

                self.n = G.order()
                self.m = G.size()
                self.graph_name = G.name()


                if self.n < 2:
                        raise ValueError("Graph is not biconnected")
                elif self.n == 2 and self.m:

                        self.comp_list_new = [G.edges()]
                        self.comp_type = [0]
                        self.__build_spqr_tree()
                        return
                elif self.m < self.n -1:

                        raise ValueError("Graph is not connected")
                elif self.m < self.n:

                        raise ValueError("Graph is not biconnected")

                self.int_to_vertex = G.vertices()
                self.vertex_to_int = {u:i for i,u in enumerate(self.int_to_vertex)}
                self.int_to_original_edge_label = []
                self.graph_copy = Graph(self.n, multiedges=True)
                for i,(u, v, l) in enumerate(G.edge_iterator()):
                        self.graph_copy.add_edge(self.vertex_to_int[u], self.vertex_
                        self.int_to_original_edge_label.append(l)


                self.edge_status = {e: 0 for e in self.graph_copy.edge_iterator()}
```

```
38                 self.reverse_edges = set()
39
40                 self.dfs_number = [0 for i in range(self.n+1)]
41
42
43                 self.highpt = [_LinkedList() for i in range(self.n)]
44
45
46                 self.in_high = {e:None for e in self.graph_copy.edge_iterator()}
47
48
49                 self.old_to_new = [0 for i in range(self.n+1)]
50                 self.newnum = [0 for i in range(self.n)]
51                 self.node_at = [0 for i in range(self.n+1)]
52                 self.lowpt1 = [None for i in range(self.n)]
53                 self.lowpt2 = [None for i in range(self.n)]
54
55
56                 self.adj = [_LinkedList() for i in range(self.n)]
57
58
59                 self.in_adj = {}
60                 self.nd = [None for i in range(self.n)]
61
62
63                 self.parent = [None for i in range(self.n)]
64                 self.degree = [None for i in range(self.n)]
65                 self.tree_arc = [None for i in range(self.n)]
66                 self.vertex_at = [1 for i in range(self.n)]
67                 self.dfs_counter = 0
68                 self.components_list = []
69                 self.graph_copy_adjacency = [[] for i in range(self.n)]
70
71
72                 self.starts_path = {e:False for e in self.graph_copy.edge_iterator()}
73
74                 self.is_biconnected = True
75                 self.cut_vertex = None
76
77
78                 self.virtual_edge_num = 0
79
80                 self.virtual_edges = set()
81
82                 self.new_path = False
83
84
85                 self.e_stack = []
86                 self.t_stack_h = [None for i in range(2*self.m + 1)]
87                 self.t_stack_a = [None for i in range(2*self.m + 1)]
88                 self.t_stack_b = [None for i in range(2*self.m + 1)]
89                 self.t_stack_top = 0
90                 self.t_stack_a[self.t_stack_top] = -1
```

```
91
92
93                  self.comp_list_new = []
94                  self.comp_type = []
95
96                  self.spqr_tree = None
97
98                  self.type1 = []
99                  self.type2 = []
100
101                 for e in self.graph_copy.edge_iterator():
102                         self.graph_copy_adjacency[e[0]].append(e)
103                         self.graph_copy_adjacency[e[1]].append(e)
104
105                 self.dfs_counter = 0
106                 self.start_vertex = 0
107                 self.cut_vertex = self.__dfs1(self.start_vertex, check=check)
108
109                 if check:
110                         # If graph is disconnected
111                         if self.dfs_counter < self.n:
112                                 raise ValueError("Graph is not connected")
113
114                         # If graph has a cut vertex
115                         if self.cut_vertex != None:
116                                 raise ValueError("Graph has a cut vertex")
117
118
119                 for e in self.graph_copy.edge_iterator():
120                         up = (self.dfs_number[e[1]] - self.dfs_number[e[0]]) > 0
121                         if (up and self.edge_status[e]==2) or (not up and self.edge_
122
123                                 self.reverse_edges.add(e)
124
125                 self.__build_acceptable_adj_struct()
126                 self.__dfs2()
127
128                 self.__path_search(self.start_vertex)
129
130
131                 if self.e_stack:
132                         e = self.__estack_pop()
133                         c = _Component(self.e_stack, 0)
134                         c.finish_tric_or_poly(e)
135                         self.components_list.append(c)
136
137                 self.__print_split()
138                 self.__assemble_triconnected_components()
139
140                 self.__build_spqr_tree()
141
142         def __tstack_push(self, h, a, b):
143
```

```
144                 self.t_stack_top += 1
145                 self.t_stack_h[self.t_stack_top] = h
146                 self.t_stack_a[self.t_stack_top] = a
147                 self.t_stack_b[self.t_stack_top] = b
148 
149         def __tstack_push_eos(self):
150 
151                 self.t_stack_top += 1
152                 self.t_stack_a[self.t_stack_top] = -1
153 
154         def __tstack_not_eos(self):
155 
156                 return self.t_stack_a[self.t_stack_top] != -1
157 
158         def __tstack_print(self):
159 
160                 return self.t_stack_a, self.t_stack_b, self.t_stack_h
161 
162         def __estack_pop(self):
163 
164                 return self.e_stack.pop()
165 
166         def __estack_print(self):
167 
168                 return self.e_stack
169 
170         def __new_component(self, edges=[], type_c=0):
171 
172                 c = _Component(edges, type_c)
173                 self.components_list.append(c)
174                 return c
175 
176         def __new_virtual_edge(self, u, v):
177 
178                 e = (u, v, "newVEdge"+str(self.virtual_edge_num))
179                 self.virtual_edge_num += 1
180                 self.virtual_edges.add(e)
181                 return e
182 
183         def __high(self, v):
184 
185                 head = self.highpt[v].get_head()
186                 if head is None:
187                         return 0
188                 else:
189                         return head.get_data()
190 
191         def __del_high(self, e):
192 
193                 if e in self.in_high:
194                         it = self.in_high[e]
195                         if it:
196                                 if e in self.reverse_edges:
```

```
197                                     v = e[0]
198                             else:
199                                     v = e[1]
200                             self.highpt[v].remove(it)
201
202     def __bucket_sort(self, bucket, edge_list):
203
204             if len(bucket) == 1:
205                     return
206
207             # Create n bucket linked lists
208             bucket_list = []
209             for i in range(self.n):
210                     bucket_list.append(_LinkedList())
211
212             # Get the head pointer of the edge list
213             e_node = edge_list.head
214
215             # Link the n buckets w.r.t bucketId
216             while e_node:
217                     bucketId = bucket[e_node.get_data()]
218                     if bucket_list[bucketId].get_head():
219                             bucket_list[bucketId].tail.next = e_node
220                             bucket_list[bucketId].tail = bucket_list[bucketId].t
221                     else:
222                             bucket_list[bucketId].set_head(e_node)
223                     e_node = e_node.next
224
225             # Rearrange the `edge_list` Using bucket list
226             new_tail = None
227             for i in range(self.n):
228                     new_head = bucket_list[i].get_head()
229                     if new_head:
230                             if new_tail:
231                                     new_tail.next = new_head
232                             else:
233                                     edge_list.set_head(new_head)
234                             new_tail = bucket_list[i].tail
235
236             edge_list.tail = new_tail
237             new_tail.next = None
238
239     def __sort_edges(self):
240
241             # Create a linkedlist of edges
242             edge_list = _LinkedList()
243             for e in self.graph_copy.edges(sort=False):
244                     edge_list.append(_LinkedListNode(e))
245
246             bucketMin = {} # Contains the lower index of edge end point
247             bucketMax = {} # Contains the higher index of edge end point
248
249             # In `graph_copy`, every edge `(u, v)` is such that `u < v`.
```

```
250                 # Hence, `bucketMin` of an edge `(u, v)` will be `u`
251                 # and `bucketMax` will be `v`.
252                 for e in self.graph_copy.edge_iterator():
253                         bucketMin[e] = e[0]
254                         bucketMax[e] = e[1]
255
256                 # Sort according to the endpoint with lower index
257                 self.__bucket_sort(bucketMin, edge_list)
258                 # Sort according to the endpoint with higher index
259                 self.__bucket_sort(bucketMax, edge_list)
260
261                 # Return the head pointer to the sorted edge list
262                 return edge_list.get_head()
263
264         def __dfs1(self, v, u=None, check=True):
265
266                 first_son = None # For testing biconnectivity
267                 s1 = None # Storing the cut vertex, if there is one
268                 self.dfs_counter += 1
269                 self.dfs_number[v] = self.dfs_counter
270                 self.parent[v] = u
271                 self.degree[v] = self.graph_copy.degree(v)
272                 self.lowpt1[v] = self.lowpt2[v] = self.dfs_number[v]
273                 self.nd[v] = 1
274                 for e in self.graph_copy_adjacency[v]:
275                         if self.edge_status[e]:
276                                 continue
277
278                         w = e[0] if e[0] != v else e[1] # Opposite vertex of edge e
279                         if self.dfs_number[w] == 0:
280                                 self.edge_status[e] = 1 # tree edge
281                                 if first_son is None:
282                                         first_son = w
283                                 self.tree_arc[w] = e
284                                 s1 = self.__dfs1(w, v, check)
285
286                                 if check:
287                                         # Check for cut vertex.
288                                         # The situation in which there is no path from w to
289                                         # ancestor of v : we have identified a cut vertex
290                                         if (self.lowpt1[w] >= self.dfs_number[v]) and (w !=
291                                                 s1 = v
292
293                                 # Calculate the `lowpt1` and `lowpt2` values.
294                                 # `lowpt1` is the smallest vertex (the vertex x with smalle
295                                 # dfs_number[x]) that can be reached from v.
296                                 # `lowpt2` is the next smallest vertex that can be reached
297                                 if self.lowpt1[w] < self.lowpt1[v]:
298                                         self.lowpt2[v] = min(self.lowpt1[v], self.lowpt2[w])
299                                         self.lowpt1[v] = self.lowpt1[w]
300
301                                 elif self.lowpt1[w] == self.lowpt1[v]:
302                                         self.lowpt2[v] = min(self.lowpt2[v], self.lowpt2[w])
```

```
303
304                                 else:
305                                         self.lowpt2[v] = min(self.lowpt2[v], self.lo
306
307                                 self.nd[v] += self.nd[w]
308
309                         else:
310                                 self.edge_status[e] = 2 # frond
311                                 if self.dfs_number[w] < self.lowpt1[v]:
312                                         self.lowpt2[v] = self.lowpt1[v]
313                                         self.lowpt1[v] = self.dfs_number[w]
314                                 elif self.dfs_number[w] > self.lowpt1[v]:
315                                         self.lowpt2[v] = min(self.lowpt2[v], self.df
316
317                 return s1 # s1 is None if graph does not have a cut vertex
318
319
320         def __build_acceptable_adj_struct(self):
321
322                 max_size = 3*self.n + 2
323                 bucket = [[] for _ in range(max_size + 1)]
324
325                 for e in self.graph_copy.edge_iterator():
326                         edge_type = self.edge_status[e]
327
328                         # compute phi value
329                         # bucket sort adjacency list by phi values
330                         if e in self.reverse_edges:
331                                 if edge_type == 1: # tree arc
332                                         if self.lowpt2[e[0]] < self.dfs_number[e[1]]
333                                                 phi = 3*self.lowpt1[e[0]]
334                                         else:
335                                                 phi = 3*self.lowpt1[e[0]] + 2
336                                 else: # tree frond
337                                         phi = 3*self.dfs_number[e[0]]+1
338                         else:
339                                 if edge_type == 1: # tree arc
340                                         if self.lowpt2[e[1]] < self.dfs_number[e[0]]
341                                                 phi = 3*self.lowpt1[e[1]]
342                                         else:
343                                                 phi = 3*self.lowpt1[e[1]] + 2
344                                 else: # tree frond
345                                         phi = 3*self.dfs_number[e[1]]+1
346
347                         bucket[phi].append(e)
348
349                 # Populate `adj` and `in_adj` with the sorted edges
350                 for i in range(1, max_size + 1):
351                         for e in bucket[i]:
352                                 node = _LinkedListNode(e)
353                                 if e in self.reverse_edges:
354                                         self.adj[e[1]].append(node)
355                                         self.in_adj[e] = node
```

```
                                else:
                                        self.adj[e[0]].append(node)
                                        self.in_adj[e] = node

        def __path_finder(self, v):

                self.newnum[v] = self.dfs_counter - self.nd[v] + 1
                e_node = self.adj[v].get_head()
                while e_node:
                        e = e_node.get_data()
                        e_node = e_node.next
                        w = e[1] if e[0] == v else e[0] # opposite vertex of e
                        if self.new_path:
                                self.new_path = False
                                self.starts_path[e] = True
                        if self.edge_status[e] == 1: # tree arc
                                self.__path_finder(w)
                                self.dfs_counter -= 1
                        else:
                                # Identified a new frond that enters `w`. Add to `highpt[w]
                                highpt_node = _LinkedListNode(self.newnum[v])
                                self.highpt[w].append(highpt_node)
                                self.in_high[e] = highpt_node
                                self.new_path = True


        def __dfs2(self):

                self.in_high = {e:None for e in self.graph_copy.edge_iterator()}
                self.dfs_counter = self.n
                self.newnum = [0 for i in range(self.n)]
                self.starts_path = {e:False for e in self.graph_copy.edge_iterator()}

                self.new_path = True

                # We call the pathFinder function with the start vertex
                self.__path_finder(self.start_vertex)

                # Update `old_to_new` values with the calculated `newnum` values
                for v in self.graph_copy.vertex_iterator():
                        self.old_to_new[self.dfs_number[v]] = self.newnum[v]

                # Update lowpt values according to `newnum` values.
                for v in self.graph_copy.vertex_iterator():
                        self.node_at[self.newnum[v]] = v
                        self.lowpt1[v] = self.old_to_new[self.lowpt1[v]]
                        self.lowpt2[v] = self.old_to_new[self.lowpt2[v]]

        def __path_search(self, v):

                y = 0
                vnum = self.newnum[v]
                outv = self.adj[v].get_length()
```

```
409                 e_node = self.adj[v].get_head()
410                 while e_node:
411                         e = e_node.get_data()
412                         it = e_node
413
414                         if e in self.reverse_edges:
415                                 w = e[0] # target
416                         else:
417                                 w = e[1]
418                         wnum = self.newnum[w]
419                         if self.edge_status[e] == 1: # e is a tree arc
420                                 if self.starts_path[e]: # if a new path starts at ed
421                                         y = 0
422                                         # Pop all (h,a,b) from tstack where a > lowp
423                                         if self.t_stack_a[self.t_stack_top] > self.l
424                                                 while self.t_stack_a[self.t_stack_to
425                                                         y = max(y, self.t_stack_h[se
426                                                         b = self.t_stack_b[self.t_st
427                                                         self.t_stack_top -= 1
428                                                 self.__tstack_push(y, self.lowpt1[w]
429
430                                         else:
431                                                 self.__tstack_push(wnum + self.nd[w]
432                                         self.__tstack_push_eos()
433
434                                 self.__path_search(w)
435
436                                 self.e_stack.append(self.tree_arc[w])
437                                 temp_node = self.adj[w].get_head()
438                                 temp = temp_node.get_data()
439                                 if temp in self.reverse_edges:
440                                         temp_target = temp[0]
441                                 else:
442                                         temp_target = temp[1]
443
444                                 # Type-2 separation pair check
445                                 # while v is not the start_vertex
446                                 while vnum != 1 and ((self.t_stack_a[self.t_stack_to
447                                                                         (self.degre
448
449                                         a = self.t_stack_a[self.t_stack_top]
450                                         b = self.t_stack_b[self.t_stack_top]
451                                         e_virt = None
452                                         if a == vnum and self.parent[self.node_at[b]
453                                                 self.t_stack_top -= 1
454
455                                         else:
456                                                 e_ab = None
457                                                 if self.degree[w] == 2 and self.newn
458                                                         # found type-2 separation pa
459                                                         self.type2.append((v+1, temp
460                                                         e1 = self.__estack_pop()
461                                                         e2 = self.__estack_pop()
```

```
462                                     self.adj[w].remove(self.in_adj[e2])
463
464                                     if e2 in self.reverse_edges:
465                                             x = e2[0] # target
466                                     else:
467                                             x = e2[1] # target
468
469                                     e_virt = self.__new_virtual_edge(v,
470                                     self.graph_copy.add_edge(e_virt)
471                                     self.degree[v] -= 1
472                                     self.degree[x] -= 1
473
474                                     if e2 in self.reverse_edges:
475                                             e2_source = e2[1] # target
476                                     else:
477                                             e2_source = e2[0]
478                                     if e2_source != w:
479                                             raise ValueError("Graph is r
480
481                                     comp = _Component([e1, e2, e_virt],
482                                     self.components_list.append(comp)
483                                     comp = None
484
485                                     if self.e_stack:
486                                             e1 = self.e_stack[-1]
487                                             if e1 in self.reverse_edges:
488                                                     if e1[1] == x and e1
489                                                             e_ab = self.
490                                                             self.adj[x].
491                                                             self.__del_h
492                                             else:
493                                                     if e1[0] == x and e1
494                                                             e_ab = self.
495                                                             self.adj[x].
496                                                             self.__del_h
497
498                             else: # found type-2 separation pair - (sel
499                                     self.type2.append((self.node_at[a]+1
500                                     h = self.t_stack_h[self.t_stack_top]
501                                     self.t_stack_top -= 1
502
503                                     comp = _Component([],0)
504                                     while True:
505                                             xy = self.e_stack[-1]
506                                             if xy in self.reverse_edges:
507                                                     x = xy[1]
508                                                     xy_target = xy[0]
509                                             else:
510                                                     x = xy[0]
511                                                     xy_target = xy[1]
512                                             if not (a <= self.newnum[x]
513                                                     break
514                                             if (self.newnum[x] == a and
```

```
515                                                     e_ab = self.
516                                                     if e_ab in s
517                                                             e_ab
518                                                     else:
519                                                             e_ab
520                                                     self.adj[e_a
521                                                     self.__del_h
522
523                                             else:
524                                                     eh = self.__
525                                                     if eh in sel
526                                                             eh_s
527                                                     else:
528                                                             eh_s
529                                                     if it != sel
530                                                             self
531                                                             self
532
533                                                     comp.add_edg
534                                                     self.degree[
535                                                     self.degree[
536
537                                     e_virt = self.__new_virtual_
538                                     self.graph_copy.add_edge(e_v
539                                     comp.finish_tric_or_poly(e_v
540                                     self.components_list.append(
541                                     comp = None
542                                     x = self.node_at[b]
543
544                             if e_ab is not None:
545                                     comp = _Component([e_ab, e_v
546                                     e_virt = self.__new_virtual_
547                                     self.graph_copy.add_edge(e_v
548                                     comp.add_edge(e_virt)
549                                     self.degree[x] -= 1
550                                     self.degree[v] -= 1
551                                     self.components_list.append(
552                                     comp = None
553
554                             self.e_stack.append(e_virt)
555                             # Replace the edge `it` with `e_virt
556                             it.set_data(e_virt)
557
558                             self.in_adj[e_virt] = it
559                             self.degree[x] += 1
560                             self.degree[v] += 1
561                             self.parent[x] = v
562                             self.tree_arc[x] = e_virt
563                             self.edge_status[e_virt] = 1
564                             w = x
565                             wnum = self.newnum[w]
566
567                     # update the values used in the while loop
```

```
568                                     temp_node = self.adj[w].get_head()
569                                     temp = temp_node.get_data()
570                                     if temp in self.reverse_edges:
571                                             temp_target = temp[0]
572                                     else:
573                                             temp_target = temp[1]
574
575                             # start type-1 check
576                             if self.lowpt2[w] >= vnum and self.lowpt1[w] < vnum and (sel
577                                     # type-1 separation pair - (self.node_at[self.lowpt
578                                     self.type1.append((self.node_at[self.lowpt1[w]]+1, v
579                                     # Create a new component and add edges to it
580                                     comp = _Component([], 0)
581                                     if not self.e_stack:
582                                             raise ValueError("stack is empty")
583                                     while self.e_stack:
584                                             xy = self.e_stack[-1]
585                                             if xy in self.reverse_edges:
586                                                     xx = self.newnum[xy[1]] #source
587                                                     y = self.newnum[xy[0]] #target
588                                             else:
589                                                     xx = self.newnum[xy[0]] #source
590                                                     y = self.newnum[xy[1]] #target
591
592                                             if not ((wnum <= xx and  xx < wnum + self.nd
593                                                             (wnum <= y and y < wnum + se
594                                                     break
595
596                                             comp.add_edge(self.__estack_pop())
597                                             self.__del_high(xy)
598                                             self.degree[self.node_at[xx]] -= 1
599                                             self.degree[self.node_at[y]] -= 1
600
601                                     e_virt = self.__new_virtual_edge(v, self.node_at[sel
602                                     self.graph_copy.add_edge(e_virt) # Add virtual edge
603                                     comp.finish_tric_or_poly(e_virt) # Add virtual edge
604                                     self.components_list.append(comp)
605                                     comp = None
606
607                                     if (xx == vnum and y == self.lowpt1[w]) or (y == vnu
608                                             comp_bond = _Component([], type_c=0) # new t
609                                             eh = self.__estack_pop()
610                                             if self.in_adj[eh] != it:
611                                                     if eh in self.reverse_edges:
612                                                             self.adj[eh[1]].remove(self.
613                                                     else:
614                                                             self.adj[eh[0]].remove(self.
615
616                                             comp_bond.add_edge(eh)
617                                             comp_bond.add_edge(e_virt)
618                                             e_virt = self.__new_virtual_edge(v, self.nod
619                                             self.graph_copy.add_edge(e_virt)
620                                             comp_bond.add_edge(e_virt)
```

```
621                                             if eh in self.in_high:
622                                                     self.in_high[e_virt] = self.
623                                             self.degree[v] -= 1
624                                             self.degree[self.node_at[self.lowpt1
625                                             self.components_list.append(comp_bon
626                                             comp_bond = None
627
628                                     if self.node_at[self.lowpt1[w]] != self.pare
629                                             self.e_stack.append(e_virt)
630
631                                             # replace edge `it` with `e_virt`
632                                             it.set_data(e_virt)
633
634                                             self.in_adj[e_virt] = it
635                                             if not e_virt in self.in_high and se
636                                                     vnum_node = _LinkedListNode(
637                                                     self.highpt[self.node_at[sel
638                                                     self.in_high[e_virt] = vnum_
639
640                                             self.degree[v] += 1
641                                             self.degree[self.node_at[self.lowpt1
642
643                                     else:
644                                             self.adj[v].remove(it)
645                                             comp_bond = _Component([e_virt], typ
646                                             e_virt = self.__new_virtual_edge(sel
647                                             self.graph_copy.add_edge(e_virt)
648                                             comp_bond.add_edge(e_virt)
649
650                                             eh = self.tree_arc[v];
651                                             comp_bond.add_edge(eh)
652                                             self.components_list.append(comp_bon
653                                             comp_bond = None
654
655                                             self.tree_arc[v] = e_virt
656                                             self.edge_status[e_virt] = 1
657                                             if eh in self.in_adj:
658                                                     self.in_adj[e_virt] = self.i
659                                             e_virt_node = _LinkedListNode(e_virt
660                                             self.in_adj[eh] = e_virt_node
661                                     # end type-1 search
662
663                                 # if an path starts at edge e, empty the tstack.
664                                 if self.starts_path[e]:
665                                         while self.__tstack_not_eos():
666                                                 self.t_stack_top -= 1
667                                         self.t_stack_top -= 1
668
669                                 while self.__tstack_not_eos() and self.t_stack_b[sel
670                                         self.t_stack_top -= 1
671
672                                 outv -= 1
673                         else: # e is a frond
```

```
674                             if self.starts_path[e]:
675                                     y = 0
676                                     # pop all (h,a,b) from tstack where a > w
677                                     if self.t_stack_a[self.t_stack_top] > wnum:
678                                             while self.t_stack_a[self.t_stack_top] > wnu
679                                                     y = max(y, self.t_stack_h[self.t_sta
680                                                     b = self.t_stack_b[self.t_stack_top]
681                                                     self.t_stack_top -= 1
682                                             self.__tstack_push(y, wnum, b)
683
684                                     else:
685                                             self.__tstack_push(vnum, wnum, vnum)
686                             self.e_stack.append(e) # add (v,w) to ESTACK
687                     # Go to next edge in adjacency list
688                     e_node = e_node.next
689
690         def __print_split(self):
691                 self.comp_list_new1 = []
692                 self.comp_type1 = []
693                 for comp in self.components_list:
694                         if comp.edge_list.get_length() > 0:
695                                 e_list = comp.get_edge_list()
696                                 e_list_new = []
697                                 # For each edge, get the original source, target and label
698                                 for u,v,l in e_list:
699                                         source = self.int_to_vertex[u]
700                                         target = self.int_to_vertex[v]
701                                         if isinstance(l, str):
702                                                 label = l
703                                         else:
704                                                 label = self.int_to_original_edge_label[l]
705                                         e_list_new.append((source, target, label))
706                                 self.comp_list_new1.append(e_list_new)
707                 return self.comp_list_new1
708
709         def __assemble_triconnected_components(self):
710
711                 comp1 = {} # The index of first component that an edge belongs to
712                 comp2 = {} # The index of second component that an edge belongs to
713                 item1 = {} # Pointer to the edge node in component1
714                 item2 = {} # Pointer to the edge node in component2
715                 num_components = len(self.components_list)
716                 visited = [False for i in range(num_components)]
717
718                 # For each edge, we populate the comp1, comp2, item1 and item2 values
719                 for i in range(num_components): # for each component
720                         e_node = self.components_list[i].edge_list.get_head()
721                         while e_node: # for each edge
722                                 e = e_node.get_data()
723                                 if e not in item1:
724                                         comp1[e] = i
725                                         item1[e] = e_node
726                                 else:
```

```
727                                     comp2[e] = i
728                                     item2[e] = e_node
729
730                             e_node = e_node.next
731
732                     # For each edge in a component, if the edge is a virtual edge, merg
733                     # the two components the edge belongs to
734                     for i in range(num_components):
735                             c1 = self.components_list[i]
736                             c1_type = c1.component_type
737                             l1 = c1.edge_list
738                             visited[i] = True
739
740                             if l1.get_length() == 0:
741                                     continue
742
743                             if c1_type == 0 or c1_type == 1:
744                                     e_node = self.components_list[i].edge_list.get_head(
745                                     # Iterate through each edge in the component
746                                     while e_node:
747                                             e = e_node.get_data()
748                                             e_node_next = e_node.next
749                                             # The label of a virtual edge is a string
750                                             if not isinstance(e[2], str):
751                                                     e_node = e_node_next
752                                                     continue
753
754                                             j = comp1[e]
755                                             if visited[j]:
756                                                     j = comp2[e]
757                                                     if visited[j]:
758                                                             e_node = e_node_next
759                                                             continue
760                                                     e_node2 = item2[e]
761                                             else:
762                                                     e_node2 = item1[e]
763
764                                             c2 = self.components_list[j]
765
766                                             # If the two components are not the same ty
767                                             if (c1_type != c2.component_type):
768                                                     e_node = e_node_next # Go to next ed
769                                                     continue
770
771                                             visited[j] = True
772                                             l2 = c2.edge_list
773
774                                             # Remove the corresponding virtual edges in
775                                             # and merge the components
776                                             l2.remove(e_node2)
777                                             l1.concatenate(l2)
778
779                                             # if `e_node_next` was empty, after merging
```

```
780                                     # more edges are added to the component.
781                                     if not e_node_next:
782                                             e_node_next = e_node.next # Go to next edge
783
784                                     l1.remove(e_node)
785
786                                     e_node = e_node_next
787
788             # Convert connected components into original graph vertices and edges
789             self.comp_list_new = []
790             self.comp_type = []
791             for comp in self.components_list:
792                     if comp.edge_list.get_length() > 0:
793                             e_list = comp.get_edge_list()
794                             e_list_new = []
795                             # For each edge, get the original source, target and label
796                             for u,v,l in e_list:
797                                     source = self.int_to_vertex[u]
798                                     target = self.int_to_vertex[v]
799                                     if isinstance(l, str):
800                                             label = l
801                                     else:
802                                             label = self.int_to_original_edge_label[l]
803                                     e_list_new.append((source, target, label))
804                             # Add the component data to `comp_list_new` and `comp_type`
805                             self.comp_type.append(comp.component_type)
806                             self.comp_list_new.append(e_list_new)
807     #print(self.comp_list_new)
808
809     def __build_spqr_tree(self):
810
811             # Types of components 0: "P", 1: "S", 2: "R"
812             component_type = ["P", "S", "R"]
813
814             self.spqr_tree = Graph(multiedges=False, name='SPQR-tree of {}'.format(self.
815
816             if len(self.comp_list_new) == 1 and self.comp_type[0] == 0:
817                     self.spqr_tree.add_vertex(('Q' if len(self.comp_list_new[0]) == 1 el
818                                                                     Graph(self.comp_l
819                     return
820
821             int_to_vertex = []
822             partner_nodes = {}
823
824             for i in range(len(self.comp_list_new)):
825                     # Create a new tree vertex
826                     u = (component_type[self.comp_type[i]],
827                             Graph(self.comp_list_new[i], immutable=True, multiedges=Tru
828                     self.spqr_tree.add_vertex(u)
829                     int_to_vertex.append(u)
830
831                     # Add an edge to each node containing the same virtual edge
832                     for e in self.comp_list_new[i]:
```

```
833                                 if e in self.virtual_edges:
834                                         if e in partner_nodes:
835                                                 for j in partner_nodes[e]:
836                                                         self.spqr_tree.add_edge(int_
837                                                 partner_nodes[e].append(i)
838                                         else:
839                                                 partner_nodes[e] = [i]
840
```

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.