Universidad Nacional Autónoma de México
Posgrado en Ciencia e Ingeniería de la Computación
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Posgrado en Ciencia e Ingeniería de la computación

Sobre Algoritmos que generan Certificados de k-conexidad

TESIS

que para optar por el grado de:
maestro en ciencia e ingeniería de la computación

presenta:
víctor sánchez flores

Director de tesis:
Dr. Sergio Rajsbaum Gorodezky
Instituto de Matemáticas

Ciudad Universitaria, Cd. Mx., Febrero 2022

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

Esta tesis fue realizada bajo la supervisión del Dr. Sergio Rajbsaum, a quien me gustaría agradecer por su infinita paciencia, su tiempo y su constante apoyo desde un principio. Que en conjunto hicieron posibles la culminación de este trabajo.

Agradezco a D"s por todas las bendiciones que he recibido y por guiarme en el camino que Él ha trazado para mí.

Agradezco a mi hermano, a mis padres y mis abuelos por su apoyo y cariño, pero, especialmente a mi mamá por ser mi ejemplo a seguir.

Agradezco a mis amigos, Abraham, Cristian y Henry, por estar conmigo apoyándome en los buenos y malos momentos.

Agradezco a los doctores Adriana Ramírez, Ilán Goldfeder, David Flores y Ricardo Strausz, los cuales me hicieron el inmerecido favor de revisar, corregir y con ello mejorar este trabajo.

Y por último, agradecer al CONACYT por el apoyo recibido durante los dos años de maestría; que siga siendo la educación el motor del cambio y la emancipación.

Da che tu vuo' saver cotanto a dentro, dirotti brievemente
Dante Alighieri. Inferno, Canto II

Índice general

Agradecimientos **I**

1 Introducción **1**
§1.1 Contexto y problemática 1
§1.1.1 Conexidad 1
§1.1.2 Certificados de k-conexidad 1
§1.1.3 Gráficas con pesos 3
§1.2 Resultados 4
§1.3 Organización de la tesis 4

2 Sobre el Teorema de Menger **6**
§2.1 Conceptos de Conexidad 6
§2.1.1 Conexidad por vértices (local y global) 6
§2.1.2 Conexidad por aristas (local y global) 7
§2.1.3 Resultados de 2-conexidad 9
§2.1.4 Conexidad mixta 11
§2.2 Teorema de Menger: una prueba corta 12
§2.3 Algoritmos que calculan la conexidad 14

3 Una generalización del teorema de Menger **17**
§3.1 Gráficas con pesos 17
§3.2 Teorema de Menger generalizado 22

4 Certificados de 2-conexidad **27**
§4.1 `NI-Search` 27
§4.1.1 Propiedades de conexidad mixta 29
§4.2 `Scan-first Search` 31
§4.3 Algoritmo distribuido: `Distributed-NI` 33
§4.4 `Sparsify-2` 34

5 Certificados de k-conexidad **37**
§5.1 `NI-Search` 37
§5.2 `Scan-First Search` 40
§5.3 `Distributed-NI` 42
§5.4 Certificados para conexidad mixta 46
§5.5 Comparación de los algoritmos 50

6 Implementación y visualización **53**
§6.1 Implementación de gráficas en `Python` 54
§6.2 Implementación de `NI-Search` 56
§6.3 Implementación de `Scan-first Search` 58
§6.4 Visualización de los algoritmos 60
§6.4.1 Vinculación con `networkx` 62
§6.5 Evaluación y discusión 63

7 Conclusiones y trabajo futuro **65**

A Ejemplo de ejecución de `NI-Search` **67**

B Ejemplo de ejecución de `Scan-first Search` **69**

C Ejemplo de ejecución de `Distributed_NI` **72**

D Ejemplo de ejecución de `Sparsify-2` **76**

E Código de la clase `Gráfica` **78**

F Código de `Certificates_tk.py` **83**

Capítulo 1

Introducción

1.1. Contexto y problemática

La conexidad de gráficas, ya sea por vértices o aristas, es un tema fundamental en la *Teoría de gráficas*. Sin embargo, con el desarrollo tecnológico y científico de los últimos años este tema ha resultado ser de gran relevancia. Ya que ha tenido múltiples aplicaciones por ejemplo: estudio de comunidades en redes sociales [11], diseño asistido por computadora [13], comunicación en redes de computadoras [35], estudio del genoma en bioinformática [43], solo por mencionar algunas. Comencemos presentando los conceptos centrales involucrados en esta tésis.

1.1.1. Conexidad

Una gráfica G es conexa si entre cualesqiera par de vértices de G existe una trayectoria entre ellos. Especificamente, una gráfica G es **k-conexa por vértices** si $G - S$[1] es conexa para cualquier subconjunto $S \subset V(G)$ con $|S| < k$. Análogamente, una gráfica G es **k-conexa por aristas** si $G - L$ es conexa para cualquier subconjunto $L \subset E(G)$ con $|L| < k$. La **conexidad** de una gráfica G, denotada por $\kappa(G)$ es el mayor valor de k para el cual G es k-conexa. De manera similar, la **conexidad** por aristas de una gráfica G, denotada por $\lambda(G)$ es el mayor k para el cual G es k-conexa por aristas.

La generalización de ambos tipos de conexidades se conoce como **conexidad mixta**. Decimos que $x, y \in V(G)$, $x \neq y$, están **(k, l)-mixto-conectados**, con $k, l \in \mathbb{N}$, si para cualquier subconjunto $S \subseteq V(G) - \{x, y\}$ y cualquier subconjunto $L \subseteq E(G)$, tales que $|S| + |L| < k + l$, los vértices x, y pertenecen a la misma componente en $G - S - L$.

1.1.2. Certificados de k-conexidad

Un problema de especial interés en Teoría de Gráficas es: dada una gráfica k-conexa, encontrar una subgráfica generadora k-conexa con un número mínimo de aristas. La solución al caso $k = 1$ fue publicada por primera vez en 1926 por Otakar Borůvka [42]

[1] La gráfica $G - S$ se define como la subgráfica inducida en G por $V(G) \setminus S$ en caso de que S sea un subconjunto de vértices o, en otro caso, como la gráfica $(V(G), E(G) \setminus S)$.

como un método para construir eficientemente una red eléctrica en Moravia. Su solución corresponde con los árboles generadores de peso mínimo.

Sin embargo, se ha demostrado que este problema es NP-duro si $k \geq 2$, ya que encontrar una solución con el mínimo número de aristas sería equivalente a encontrar un ciclo hamiltoniano en una gráfica hamiltoniana [34]. A pesar de esto, es posible encontrar aproximaciones a este problema para todo $k \geq 2$. Una de ellas es proponer algoritmos que reciban como entrada una gráfica k-conexa y entreguen como salida una subgráfica *rala*[2] generadora que preserve la k-conexidad de la gráfica original. A esta subgráfica la llamaremos **certificado de k-conexidad**.

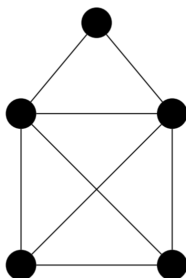


Figura 1.1: Gráfica G 2-conexa

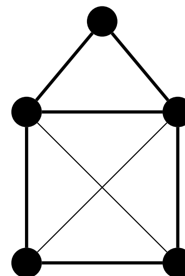


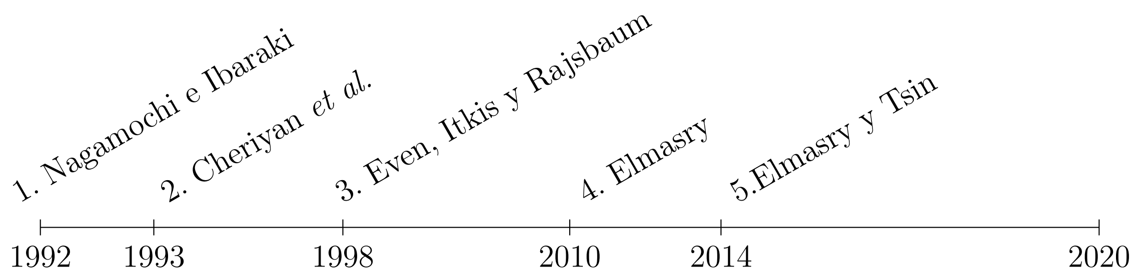
Figura 1.2: Certificado de 2-conexidad de G en negritas

Los algoritmos que producen certificados de k-conexidad, los cuales generan gráficas con un menor número de aristas que la gráfica original, pueden ser utilizados como preprocesamiento con el fin de disminuir la complejidad de tiempo de otros algoritmos. Por ejemplo,

- Mejoran la complejidad en algoritmos que prueban la k-conexidad de gráficas no dirigidas [9] y [41].
- Mejoran el tiempo de ejecución para encontrar tres árboles generadores independientes [10].
- Mejoran la complejidad de algoritmos dinámicos en gráficas con o sin pesos, algoritmos para determinar si una gráfica es bipartita, problemas de intersección de matroides, calcular los mejores k árboles generadores, etc. [17].

A continuación haremos un breve resumen, de manera cronológica, de algoritmos que se han inventado para resolver esta aproximación al problema. En adelante, consideramos la k-conexidad tanto por vértices y aristas a menos que se especifique otra cosa.

[2] Una gráfica G es **rala** si $|E(G)| = O(kn)$ con $k, n \in \mathbb{N}$ y $k < n$.



1. Nagamochi e Ibaraki presentaron un algoritmo que en tiempo lineal $O(|E|)$ genera una subgráfica rala generadora k-conexa [41], ya sea por vértices o por aristas, a partir de una gráfica k-conexa dada. Su algoritmo consiste en encontrar secuencias de bosques $F_1, F_2, ...$ en la gráfica de entrada. Cada uno de estos bosques F_j es maximal en la subgráfica resultante $G - \cup_{j-1}^{i=1} F_i$. Esta maximalidad es suficiente para probar que la subgráfica $G_k = \cup_{k}^{i=1} F_i$ es k-conexa por aristas.

2. Cheriyan, Kao y Thurimella publicaron un nuevo algoritmo llamado *Scan-First Search* [9], el cual al ser ejecutado k veces en secuencias de subgráficas de una gráfica G genera certificados con a lo más $k(n-1)$ aristas. Además en el mismo artículo, propusieron una versión en paralelo del mismo.

3. Even, Itkis y Rajsbaum generalizaron estos resultados, sobre k-conexidad, a conexidad mixta [21], donde la conexidad por vértices y aristas son casos especiales de conexidad mixta. Además, en este mismo artículo presentan un algoritmo distribuido para este nuevo enfoque.

4. Elmasry publica un artículo en el que se presentan dos algoritmos enfocados en los casos $k = 2, 3$ [15]. Estos algoritmos son, esencialmente, un algoritmo DFS modificado para generar eficientemente, en tiempo lineal, las subgráficas ralas generadoras 2 y 3-conexas.

5. Elmasry y Tsin publican un nuevo artículo [16], donde proponen un algoritmo más simple que resuelve el caso de la 3-conexidad del artículo de Elmasry de 2010. En él, logran simplificar el algoritmo al reducir el tiempo de ejecución. Esto se logró al evitar el ordenamiento por cubetas y la construcción de la lista de adyacencias; dos procesos que sí aparecen en el artículo del 2010. Además, este nuevo algoritmo usado como una prerrutina permite mejorar la eficiencia de otros algoritmos, así como los problemas relacionados que tratan la 3-conexidad.

1.1.3. Gráficas con pesos

Al igual que sucede con el tema de conexidad, las gráficas con pesos han encontrado múltiples aplicaciones en diferentes campos de estudio. En un modelo representado por medio de una gráfica con pesos, como podría ser una red de información o un circuito eléctrico, la reducción del flujo entre algún par de vértices es más relevante y común que la interrupción total del flujo o la desconexión de toda la red. Así, el problema al que

nos enfrentamos ahora, es extender los algoritmos que generan certificados de k-conexidad para utilizarlos en gráficas con pesos.

Como antecedentes tenemos una versión generalizada del teorema de Menger para gráficas con pesos [37], publicada en 2011 por Mathew y Sunitha, donde las gráficas simples se consideran como gráficas cuyas aristas tienen todas peso 1. En un artículo posterior, estos mismos autores generalizaron conceptos clásicos de conexidad por vértices y aristas para gráficas con pesos, además de presentar una generalización del teorema de Whitney[3]

1.2. Resultados

En el presente trabajo presentamos de forma unificada, con una misma notación, los siguientes algoritmos que generan certificados de k-conexidad:

- El algoritmo secuencial de Nagamochi e Ibaraki, presentado en su artículo *A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph* [41].
- `Scan-First Search` de Cheriyan, Kao y Thurimella, en su versión secuencial, presentado en su artículo *Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity* [9].
- El algoritmo distribuido de Even, Itkis y Rajsbaum presentado en su artículo *On mixed connectivity certificates* [21].

Además, explicamos las relaciones que tienen estos tres algoritmos en términos teóricos y prácticos. Luego, hacemos su implementación así como su modelación en el lenguaje de programación `Python`.

En segundo lugar, buscamos extender los conceptos de conexidad, para construir un algoritmo capaz de generar certificados de k-conexidad en gráficas con pesos.

1.3. Organización de la tesis

La organización de la tesis es la siguiente:

- Capítulo 2. Revisamos conceptos de conexidad relacionados con los certificados, así como el teorema de Menger.
- Capítulo 3. Formalizamos los conceptos de gráficas con pesos y presentamos algunos resultados importantes que utilizaremos más adelante.
- Capítulo 4. Revisamos a detalle el caso de los certificados de 2-conexidad, lo anterior con el fin de ilustrar de manera más clara los algoritmos antes mencionados.
- Capítulo 5. Revisamos a detalle los algoritmos que generan certificados de k-conexidad.

[3]Una gráfica G de orden $n \geq 3$ es 2-conexa si y solo si para cualesquiera u, v vértices de G existen dos (u, v)-trayectorias internamente ajenas.

- Capítulo 6. Revisamos la implementación y visualización de los algoritmos.
- Capítulo 7. Presentamos nuestras conclusiones.

Capítulo 2

Sobre el Teorema de Menger

> "... one of the most important results in graph theory ... the fundamental theorem on connectivity in graphs."
>
> Frank Harary

Como ya vimos en la introducción, podemos estudiar la conexidad de una gráfica G ya sea por sus vértices o por sus aristas. Así mismo, podemos estudiar la conexidad de una gráfica de manera local o global. En primer lugar revisaremos los conceptos de conexidad de tal suerte que, eventualmente lleguemos a revisar el teorema que da nombre a este capítulo. En adelante consideramos G una gráfica no dirigida sin aristas múltiples y sin lazos.

2.1. Conceptos de Conexidad

2.1.1. Conexidad por vértices (local y global)

Revisemos primero algunas definiciones y resultados clásicos de conexidad por vértices. Como referencia utilizamos el libro de G. Chartrand y P. Zhang [8]. Así, salvo que se indique otra cosa, las definiciones y resultados siguientes provienen de esta fuente.

Un **conjunto de corte** S entre dos vértices u y v es un conjunto de vértices que al ser removidos de G causan que u y v queden desconectados. Si $|S| = 1$ decimos que este único vértice es un **vértice de corte**. La **conexidad local** por vértice entre un par de vértices u, v, denotada como $\kappa_G(u, v)$, es el tamaño de un conjunto mínimo de corte que separa a u de v. Exceptuando las gráficas completas[1] K_n, $\kappa(G) = \text{mín}\,\{\kappa_G(u, v)\}$, con u, v no adyacentes; $\kappa(G)$ denota la **conexidad global** por vértices de la gráfica G. Así, $\kappa(G)$ es la cantidad mínima de vértices que al ser removidos de G desconectan a la gráfica, o bien, generan una gráfica trivial[2]. Luego, tenemos el siguiente resultado:

Proposición 2.1 (G. Chartrand y P. Zhang [8])**.** *Sea G una gráfica de orden n, entonces* $0 \leq \kappa(G) \leq n - 1$.

[1] Ya que las gráficas completas K_n no tienen vértices de corte, por convención se define $\kappa(K_n) = n - 1$.

[2] La gráfica trivial corresponde con K_1, es decir, un solo vértice.

Demostración. Sea G como en el enunciado. Es claro, que no podemos quitar más de n vértices de G y si eliminamos los n vértices de G obtenemos como resultado una gráfica vacía.[3] Así, $\kappa(G)$ queda acotada por $n-1$. □

Notemos que $\kappa(G) = 0$ si y solo si $G = K_1$, o bien, G es inconexa; $\kappa(G) = 1$ si y solo si $G = K_2$, o bien, G es conexa y tiene vértices de corte. Como ya lo discutimos en la introducción, a veces es más útil y económico, en términos computacionales, saber si una gráfica G puede ser desconectada al remover cierto número de vértices, en vez de conocer el número exacto de $\kappa(G)$. Así, decimos que G es **k-conexa**, con $k \geq 1$, si $\kappa(G) \geq k$. Es decir, G es k-conexa si al remover una cantidad k' de vértices, con $k' < k$, G se mantiene conexa, o bien, no se convierte en una gráfica trivial[4].

2.1.2. Conexidad por aristas (local y global)

La conexidad de una gráfica G también puede medirse en términos del número de aristas, que al ser removidas de G, desconectan a la gráfica, tal como lo mencionamos en la introducción. Igual que en la subsección anterior, utilizamos como referencia [8]. Un **conjunto de corte por aristas** S' entre dos vértices u y v es un conjunto de aristas que al ser removidas de G causan que u y v queden desconectados. Si $|S'| = 1$ decimos que S es un **puente**. La **conexidad local por aristas** entre un par de vértices u, v, denotada como $\lambda_G(u, v)$, es el tamaño de un conjunto mínimo de corte por aristas que separa a u de v. La **conexidad global** por aristas de una gráfica G está definida como $\lambda(G) = \text{mín}\{\lambda_G(u, v)\}$. La gráfica trivial K_1 no contiene aristas, sin embargo por convención, definimos $\lambda(K_1) = 0$.

Como ya vimos, $\lambda(G)$ denota la cantidad mínima de aristas que al ser removidas de G la desconectan, o bien, generan una gráfica trivial. Así, tenemos el siguiente resultado:

Proposición 2.2 (G. Chartrand y P. Zhang [8])**.** *Para todo $n \in \mathbb{N}$, $\lambda(K_n) = n - 1$.*

Demostración. Sea G como en el enunciado. Recordemos que $\lambda(K_1) = 0$, así que supondremos que $n \geq 2$. Si removemos las $n-1$ aristas incidentes a cualquiera de los vértices de K_n obtenemos una gráfica inconexa, luego $\lambda(G) \leq n - 1$. Sea X un corte por aristas mínimo de K_n, por lo tanto, $|X| = \lambda(K_n)$ y $G - X$ está formada por dos componentes G_1 y G_2. Supongamos que $|G_1| = k$, luego $|G_2| = n - k$. Así, $|X| = k(n-k)$. Como $k \geq 1$ y $n - k \geq 1$ tenemos que $(k-1)(n-k-1) \geq 0$ y por lo tanto:

$$(k-1)(n-k-1) = k(n-k) - (n-1) \geq 0,$$

lo que implica que

$$\lambda(K_n) = |X| = k(n-k) \geq n - 1.$$

Así, $\lambda(K_n) = n - 1$. □

Corolario 1 (G. Chartrand y P. Zhang [8])**.** *Sea G una gráfica de orden n, entonces $0 \leq \lambda(G) \leq n - 1$.*

[3] Definimos una gráfica vácia como una gráfica sin vértices ni aristas.

[4] Notemos que esta definición de k-conexidad por vértices es igual a la que utilizamos en la introducción.

En el caso de los árboles enunciamos los siguientes resultados básicos, sin demostración, sobre su conexidad.

Proposición 2.3. *Sea T un árbol. T se desconecta al eliminar cualquier arista de T, es decir, $\lambda(T) = 1$.*

Proposición 2.4. *Sean T un árbol y $v \in V(T)$ un vértice interior*[5]*. Si eliminamos v, desconectamos a T es decir, $\kappa(T) = 1$.*

Como vemos, todas las aristas de un árbol T son puentes y cualquier vértice interior de T es un vértice de corte. Para más ejemplos de conexidad por vértices $\kappa(G)$ y aristas $\lambda(G)$, véanse las figuras 2.1 y 2.2.

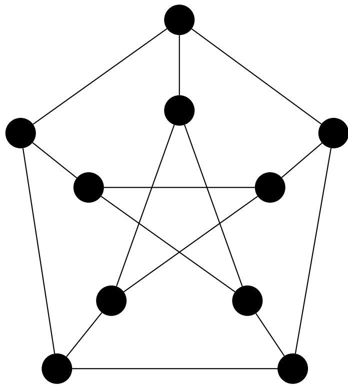


Figura 2.1: Gráfica de Petersen, $\kappa(G) = 3$ y $\lambda(G) = 3$

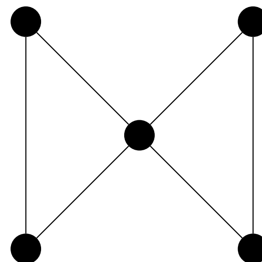


Figura 2.2: Gráfica H, $\kappa(H) = 1$ y $\lambda(H) = 2$

Notemos que una gráfica k-conexa, ya sea por vértices o aristas, es también $(k-1)$-conexa. En general, una gráfica k-conexa es k'-conexa con $1 \leq k' \leq k$. La conexidad por vértices y aristas de una gráfica G satisfacen la siguiente desigualdad, donde $\delta(G)$ denota el grado mínimo de G.

Teorema 2.5 (G. Chartrand y P. Zhang [8])**.** *Sea G una gráfica, entonces $\kappa(G) \leq \lambda(G) \leq \delta(G)$.*

De esta manera relacionamos las dos *métricas* de conexidad, que ya revisamos en las subsecciones anteriores, con el grado mínimo de la gráfica. Por otro lado, notemos que en el caso de la gráfica H de la figura 2.2 se cumple que: $\kappa(H) = 1 < \lambda(H) = 2 \leq \delta(H) = 2$, es decir, la desigualdad es estricta. Sin embargo, en el caso de la gráfica de Petersen tenemos que $\kappa(G) = \lambda(G) = 3$. Esto sucede ya que esta gráfica es cúbica[6].

Teorema 2.6 (G. Chartrand y P. Zhang [8])**.** *Para toda gráfica G cúbica se cumple que: $\kappa(G) = \lambda(G)$.*

Los teoremas 2.5 y 2.6 aparecen como teorema 2.16[7] y 2.17[8] respectivamente en [8]. Las demostraciones de ambos teoremas son largas y desviarían el foco de atención del presente trabajo, razón por la cual no las incluimos.

[5]Decimos que v es vértice interior de un árbol T si $d(v) \geq 2$, es decir, v no es una hoja del árbol T.

[6]Una gráfica G es cúbica si para todo $v \in V(G)$, $d(v) = 3$.

[7]Ibid. p. 62

[8]Ibid. p. 63

2.1.3. Resultados de 2-conexidad

En esta subsección trataremos el caso de 2-conexidad, tanto por vértices como por aristas. Tal como lo mencionamos en la introducción, dedicaremos un capítulo del presente trabajo para tratar detalladamente el caso de certificados de 2-conexidad. Así, que los resultados, en su mayoría clásicos, que veremos a continuación, estarán relacionados con los temas que revisaremos en el capítulo 4. Primero veamos el siguiente teorema que caracteriza a las gráficas 2-conexas por vértices.

Teorema 2.7. *Sea G de orden $n \geq 3$, luego los siguientes enunciados son equivalentes:*

1. *G es 2-conexa por vértices.*
2. *G es conexa y no tiene vértices de corte.*
3. *Para cualquiera $u, v, w \in V(G)$ distintos, existe una (u, v)-trayectoria que no contiene al vértice w.*

Demostración. **1→3.** Como G es 2-conexa, $G' = G - w$ es conexa. Así, existe en G' una (u, v)-trayectoria P. Así es que P es una (u, v)-trayectoria en G que no contiene a w.

3→2. Sea G como en el enunciado 3. Así, G es conexa. Supongamos que w es un vértice de corte en G tal que $G' = G - w$ es inconexa. Sean $u, v \in V(G)$ distintos que viven cada uno diferentes componentes conexas en G', así, no existe una (u, v)-trayectoria en G'. Por lo tanto, toda (u, v)-trayectoria en G contiene a w, lo que contradice la hipótesis.

2→1. Sea G como en el enunciado 2. Como G es de orden $n \geq 3$ y no tiene vértices de corte entonces, $\kappa(G) \geq 2$. Por lo que G es 2-conexa por vértices por definición.

□

Revisemos ahora el siguiente resultado, que de igual manera trata la conexidad por vértices. Este resultado, así como su corolario los tomamos de [5].

Teorema 2.8 (H. Whitney [46])**.** *Una gráfica G de orden $n \geq 3$ es 2-conexa por vértices si y solo si para cualquier par de vértices $u, v \in V(G)$ existen dos (u, v)-trayectorias ajenas por vértices.*

El teorema de Whitney fue publicado por primera vez en 1932. La demostración del mismo es larga lo que desviaría el foco de atención del presente trabajo, por lo cual no la incluimos[9]. Como corolario del teorema 2.8 tenemos:

Corolario 2. *Si G es 2-conexa por vértices, entonces cualquiera dos vértices de G viven en un mismo ciclo.*

Notemos que para desconectar un ciclo C_n, con $n > 3$, necesitamos remover de C_n ya sea dos vértices no consecutivos, o bien, dos o más aristas no consecutivas. Es decir, un ciclo C_n no tiene puentes ni vértices de corte, o bien, $\lambda(C_N) = 2$ y $\kappa(C_n) = 2$. El teorema 2.8 y su corolario son un caso especial del ubicuo teorema de Menger. Para finalizar de revisar los resultados que tratan la 2-conexidad por vértices, enunciamos sin demostración el siguiente teorema clásico publicado por G. A. Dirac en 1960 en [12], el cual es una generalización del corolario anterior.

[9]Si gusta revisar la demostración, puede consultarla en [5] en la página 45.

Teorema 2.9 (G. A. Dirac [12]). *Sean G una gráfica k-conexa, con $k \geq 2$, y $S \subset V(G)$ de tamaño k. Entonces, existe un ciclo C en G tal que C contiene a todos los vértices de S.*

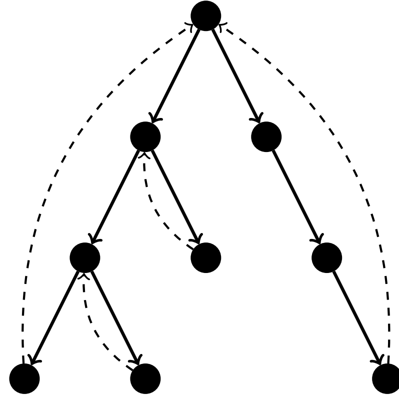


Figura 2.3: Orientación de una gráfica 2-conexa por aristas a partir de `DFS`.

Revisemos ahora el caso de la 2-conexidad por aristas. El siguiente resultado es conocido también como teorema de Robbins, ya que fue publicado por el matemático estadounidense Herbert Ellis Robbins en *A theorem on graphs, with an application to a problem on traffic control* en el año de 1939. La demostración del teorema 2.10 es una adaptación de la demostración del teorema 2.1.1 que aparece en [44].

Teorema 2.10. *Una gráfica G puede ser orientada para obtener una digráfica fuertemente conexa si y solo si G es 2-conexa por aristas.*

Demostración. Demostremos la necesidad por contrapositiva. Si G tuviera un puente, es decir no es 2-conexa por aristas, entonces no existe una orientación fuerte de G.[10]

Ahora bien para demostrar la suficiencia, sea G una gráfica 2-conexa por aristas y encontremos una orientación fuerte de G. Fijamos un vértice $v \in V(G)$ y corremos `DFS`[11] a partir de v para obtener un árbol T (Véase la figura 2.3). Dirigimos todas las aristas de T del padre al hijo, aristas de árbol en negritas. Y dirigimos las aristas de los descendientes a sus ancestros[12], *back edges* en flechas punteadas. Así, obtenemos la siguiente digráfica G':

- $V(G') = V(G)$
- $E(G') = \{(u, v)\colon (u, u') \in E(T) \text{ o } (v, u') \in E(G) \backslash E(T) \text{ tales que } u \text{ es ancestro de } u'\}$

[10]Notemos que $\delta^+(D) \geq 1$ y $\delta^-(D) \geq 1$ es una condición necesaria para que D se fuertemente conexa, pero no es suficiente.

[11]La versión de `DFS` a la que hacemos mención corresponde con la versión publicada por R. Tarjan en [45].

[12]Todas las aristas $e \in E(G) \setminus E(T)$ conectan a los descendientes con sus ancestros.

Veamos ahora que G' es fuertemente conexa. Para ello tenemos que demostrar que para cualesquiera $u, v \in V(G')$ existen una (u, v)-trayectoria y una (v, u)-trayectoria dirigidas. Notemos que para toda $u' \in V(G') - v$ existe una (v, u')-trayectoria dirigida, recordemos que v es la raíz del árbol T. Supongamos ahora que existe un vértice u, con la menor distancia entre este y v, tal que no existe una (u, v)-trayectoria dirigida en G' y lleguemos a una contradicción. Llamemos P a la (v, u)-trayectoria dirigida y $T' \subset T$ un subárbol con raíz en u. Tenemos así los siguientes dos casos:

- Si existe una flecha de $u' \in V(T')$ hacia algún $v' \in V(P) - \{u\}$, entonces existe una (u, v)-trayectoria dirigida. Lo anterior, ya que existe una (u, u')-trayectoria dirigida P_1 y existe la (u', v') flecha. Además, supusimos que $d_T(v, v') < d_T(v, u)$, así existe una (v', v)-trayectoria en G', llamémosla P_2. Así, de $P_1 \cup (u', v') \cup P_2$ tenemos una (u, v)-trayectoria dirigida.
- Si no existe una flecha de $u' \in V(T')$ hacia algún $v' \in V(P) - \{u\}$, entonces la flecha $(p(u), u)$ que conecta al padre de u con u es un puente. Esto es una contradicción.

Así, de ambos casos concluimos que G es fuertemente conexa, pues para todo $u, v \in V(G)$ existen (v, u)-trayectoria y (u, v)-trayectoria dirigidas. □

Como veremos más adelante, basta correr `DFS` sobre una gráfica 2-conexa por aristas para obtener un certificado de 2-conexidad por aristas.

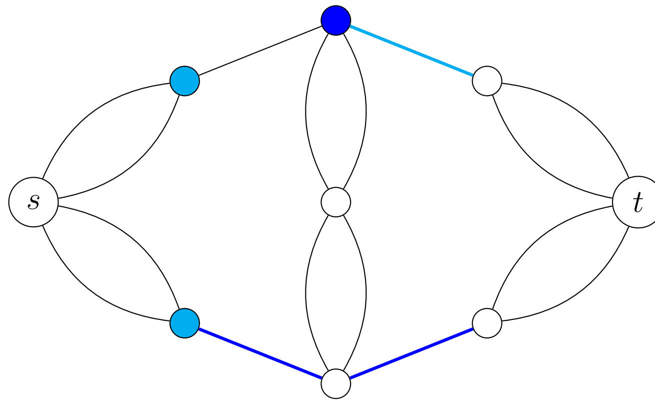


Figura 2.4: Ejemplo de un (1,2)-conjunto de corte mixto que desconecta a s y t en azul y un (2,1)-conjunto de corte mixto que desconecta a s y t en cian.

2.1.4. Conexidad mixta

Como ya vimos en la introducción, existe una generalización de estos dos tipos de conexidad, llamado **conexidad mixta**. En este tipo de conexidad removemos vértices y aristas simultáneamente. Para definir un (a, b)**-conjunto de corte mixto** consideremos los conjuntos $W \subset V(G)$ y $F \subset E(G)$ tales que $|W| \leq a$ y $|F| \leq b$ y $(G - F) - W$ es inconexa. Decimos que G es (k, l)**-conexa mixta** para u, v, con $k, l \in \mathbb{N}$ si G tiene un

(k,l)-conjunto de corte mixto mínimo que separa a u de v. Esta definición corresponde con la conexidad local para el caso de conexidad mixta. Por otro lado, decimos que G es (k,l)-conexa mixta, con $k,l \in \mathbb{N}$ si G tiene un (k,l)-conjunto de corte mixto mínimo, esta definición corresponde con la conexidad global de G.

Como vemos un (a,b)-conjunto de corte mixto está formado por vértices y aristas. Notemos que la k-conexidad por vértices es equivalente a la ausencia de un $(k-1,0)$-conjunto de corte mixto, mientras que la k-conexidad por aristas es equivalente a la ausencia de un $(0,k-1)$-conjunto de corte mixto en G. Las definiciones para los (a,b)-conjuntos de corte mixto entre $u,v \in V(G)$, es decir, la conexidad local, son análogas. Todas las definiciones mencionadas hasta el momento en estos dos párrados corresponden con las dadas por E. Bonnet y S. Cabello en [6].

2.2. Teorema de Menger: una prueba corta

Uno de los resultados más importantes sobre conexidad es el teorema de Menger, el cual fue publicado en 1927 por Karl Menger [40]. Su autor mostró este resultado a Dénes Kőnig, el cual incluyó el teorema en su libro *Theorie der endlichen und unendliehen Graphen* publicado en 1936. Este fue el primer libro escrito sobre teoría de gráficas [8]. Este teorema caracteriza la conexidad por vértices y aristas en términos del número de trayectorias independientes entre los vértices o aristas de la gráfica. Así, este teorema nos permite reformular la conexidad de una gráfica no solo en términos de conjuntos de corte, sino también, en términos de las trayectorias ajenas entre pares de vértices.

En esta tesis presentaremos una demostración corta de este teorema, para el caso de conexidad por vértices[13], basada en la prueba realizada por F. Göring en [28].

Veamos algunas definiciones que usaremos en la prueba, para $A, B \subseteq V(G)$, no necesariamente ajenos:

- Una AB-**trayectoria** es una trayectoria P que comienza en algún vértice $a \in A$ y termina en algún vértice $b \in B$ tal que P no tiene vértices interiores en A o en B.
- Un AB-**separador** de tamaño k en G es un conjunto $S \subseteq V(G)$ tal que $G - S$ no contiene ninguna AB-trayectoria. S puede intersectar a A y B.
- Un AB-**conector** de tamaño k es la unión de k AB-trayectorias ajenas por vértices.

Teorema 2.11 (Teorema de Menger [28]). *El tamaño mínimo de un AB-separador de una gráfica G es igual al tamaño máximo de un AB-conector de G.*

Demostración. Haremos la demostración por inducción sobre el tamaño m de la gráfica G. Para la base inductiva: sea $m = 0$, así, G no tiene aristas. Por lo que el tamaño de un AB-separador mínimo es 0, al igual que el tamaño de un AB-conector mínimo que es también 0.

Supongamos ahora que el teorema es válido para $G - \{e\}$. Es decir, si $G - \{e\}$ tiene un AB-separador mínimo de tamaño k (que no contiene ninguno de los dos vértices incidentes

[13] El caso de conexidad por aristas es análogo.

a la arista e), entonces $G - \{e\}$ tiene un AB-conector máximo de tamaño k y por lo tanto, también en G.

En otro caso, sea S un AB-separador mínimo en $G - \{e\}$, con $|S| < k$, tal que toda AB-trayectoria en G contiene un vértice en S o en la arista $e = (v_1, v_2)$. Notemos que $|S| = k - 1$, ya que si $|S| < k - 1$, $S \cup \{v_1\}$ o $S \cup \{v_2\}$ sería un separador más pequeño en G. Sabemos que en $G - S$ existe una AB-trayectoria P que pasa por la arista e. Lo anterior, ya que S no es un AB-separador en G, pues un separador en G es de tamaño al menos k. Ya que si tenemos un número k de AB-trayectorias en G, necesitamos eliminar un vértice en cada una de estas k trayectorias para desconectar A y B.

Por lo anterior, tenemos la trayectoria $P = (a, \ldots, v_1, v_2, \ldots, b)$, con $a \in A$ y $b \in B$, que es también una AB-trayectoria. Veamos que v_1 es alcanzable desde A, pero no desde B en $(G-S)-\{e\}$. Por otro lado, v_2 es alcanzable desde B, pero no desde A en $(G-S)-\{e\}$ (véase la figura 2.5).

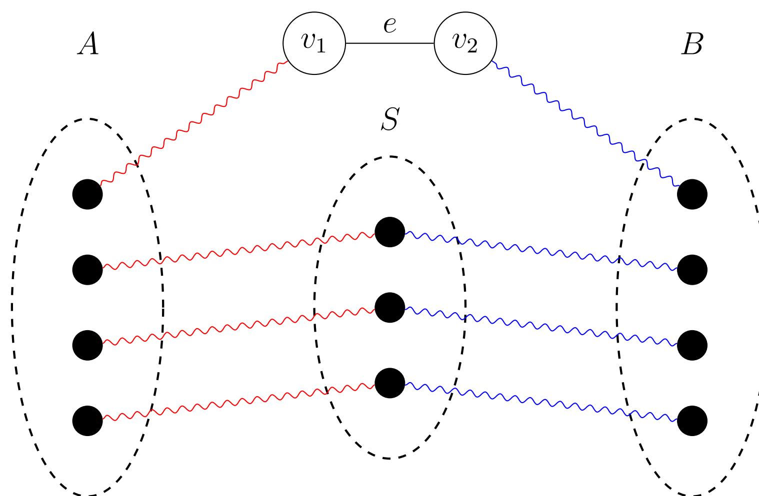


Figura 2.5: Tenemos los conjuntos A, B y S, las aristas rizadas representan trayectorias en rojo si pertenecen a C_1, o bien, en azul si pertenecen a C_2.

Sea $S_1 = S \cup \{v_1\}$ y consideremos un AS_1-separador mínimo T en $G - \{e\}$. Como v_2 no es alcanzable desde A en $G - S_1$, T es también un AS_1-separador en G. Luego, T es también un AB-separador en G ya que toda AB-trayectoria intersecta a los vértices en S_1. Así, $|T|$ es al menos k. Por hipótesis de inducción, $G - \{e\}$ contiene un AS_1-conector C_1 de tamaño k. Como su tamaño es k, los vértices finales de estas trayectorias corresponden con los vértices de S_1.

Similarmente, $S_2 = S \cup \{v_2\}$ tiene un S_2B-separador de tamaño k y un S_2B-conector C_2 de tamaño k con vértices iniciales en S_2. Además, como S_1 desconecta a G, toda trayectoria en C_1 es internamente ajena de toda trayectoria en C_2. Así, podemos definir un AB-conector de tamaño k en G al concatenar las trayectorias en C_1 y C_2 (Véase la figura 2.5). Tenemos $k - 1$ trayectorias que pasan por S y una trayectoria que pasa por $e = (v_1, v_2)$. $\square$

Existen diferentes versiones y extensiones del teorema de Menger. Por ejemplo, R. Aha-

roni y E. Berger demostraron en 2009 una versión de este teorema para gráficas infinitas [1], R. Borndörfer y M. Karbstein demostraron en 2012 una versión para hipergráficas [7]. Y, como veremos en el próximo capítulo, existe una versión de este mismo teorema para gráficas con pesos. Sin embargo, ¿qué sucede para el caso de conexidad mixta?

En 1967, L. Beineke y F. Harary publicaron en [4] su versión del teorema de Menger para el caso de conexidad mixta, donde presumían tener una demostración de la misma. Lamentablemente, W. Mader probó en [36] que esta demostración era incorrecta, así que hasta el día de hoy este problema sigue abierto [6]. A continuación enunciamos esta versión.

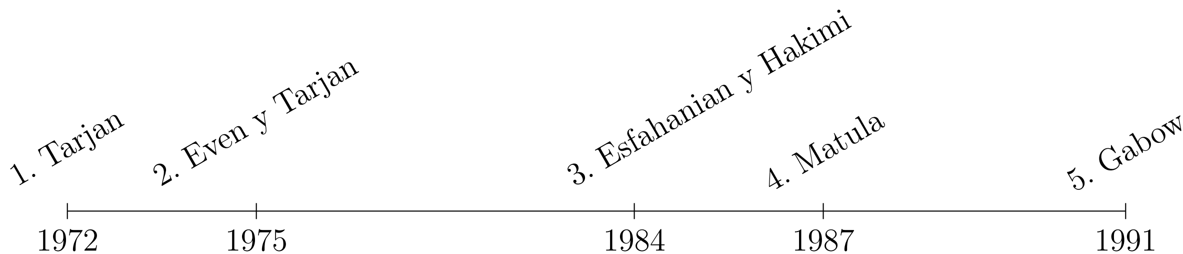
Conjetura 1 (Beineke y Harary [4]). Sean G una gráfica no dirigida, $s, t \in V(G)$ y $k, l \in \mathbb{N}$ tales que $l \geq 1$. Si G es (k, l)-conexa mixta para los vértices s y t, entonces existen $k + l$ (s, t)-trayectorias ajenas por aristas, de las cuales $k + 1$ son internamente disjuntas.

A pesar de que la verificación de esta versión general sigue abierta, en 1991 Y. Egawa, A. Kaneko y M. Matsumoto publicaron en [14] una versión acotada del teorema de Menger para conexidad mixta, la cual enunciamos a continuación sin demostración.

Teorema 2.12 (Egawa, Kaneko y Matsumoto [14])**.** *Sean $u, v \in V(G)$. Existen λ uniones ajenas por aristas de k (u, v)-trayectorias internamente ajenas si y solo si para cada conjunto S de $0 \leq r \leq \text{mín}\{k-1, |V(G)|-2\}$ vértices, la subgráfica $G - S$ contiene $\lambda(k - r)$ (u, v)-trayectorias ajenas por aristas.*

2.3. Algoritmos que calculan la conexidad

Como mencionamos en la introducción, los algoritmos que generan certificados de k-conexidad mejoran la complejidad de tiempo de varios algoritmos. Así que, para finalizar este capítulo, hablaremos primero sobre la complejidad que tienen algunos algoritmos para calcular la conexidad por aristas y vértices de una gráfica simple. Para ello seguiremos la cronología presentada por A.H. Esfahanian en [19]. Revisemos primero la conexidad por aristas. Sean $n = |V(G)|$ y $m = |E(G)|$.



1. R. E. Tarjan, en el artículo titulado *Depth-first search and linear graph algorithms* [45], aplica `DFS` para determinar si una gráfica G cumple que $\lambda(G) = 2$, o bien, $\lambda(G) = 3$. Lo anterior le toma tiempo $O(m+n)$, que corresponde con la complejidad de tiempo de `DFS`. Revisaremos esta versión del algoritmo más adelante en el capítulo dedicado a la 2-conexidad.

2. Recordemos que $\lambda(G) = \text{mín}\{\lambda(u, v; G) : u, v \in V(G)\}$, así que uno puede calcular $\lambda(G)$ haciendo n llamadas a un algoritmo que calcule $\lambda(u, v; G)$ para u, v arbitrarios.

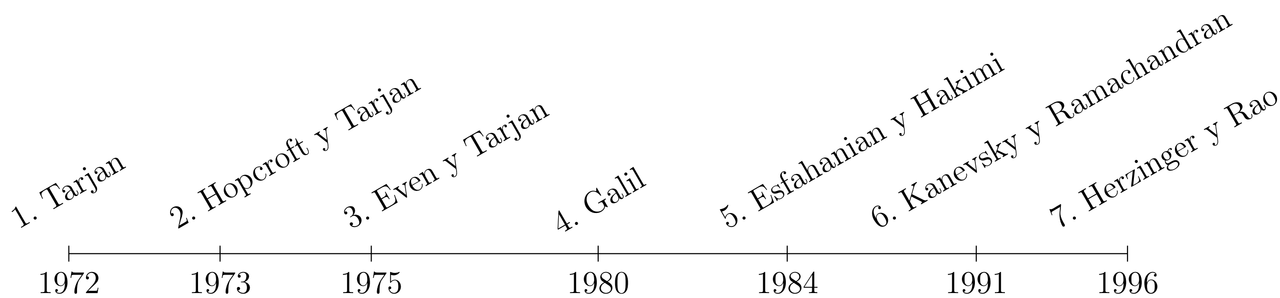
S. Even y R. E. Tarjan publicaron un algoritmo que calcula el valor de $\lambda(G)$ en el artículo titulado *Network flow and testing graph connectivity* [22]. Como su nombre lo indica, utilizaron técnicas de flujos en una gráfica. Este algoritmo tiene una complejidad de $O(mn \times \text{mín}\{m^{1/2}, n^{2/3}\})$, donde n denota la cardinalidad de $V(G)$ y m el número de aristas de la gráfica.

3. A. H. Esfahanian y S. L. Hakimi publicaron un algoritmo en el artículo titulado *On computing the connectivities of graphs and digraphs* [20]. Este algoritmo reduce las llamadas al algoritmo de flujo máximo menor que o igual a $n/2$, por lo que reduce la complejidad de tiempo a $O(\lambda mn)$[14].

4. D. W. Matula publica un algoritmo en el artículo titulado *Determining edge connectivity in O(nm)* [39] que utiliza conjuntos dominantes[15] para calcular $\lambda(G)$ en tiempo $O(mn)$.

5. H. N. Gabow presentó, en *A Matroid Approach to Finding Edge Connectivity and Packing Arborescences* [25], un algoritmo con una complejidad de tiempo $O(m + \lambda^2 n \log(n/\lambda))$ para gráficas simples. Este algoritmo encuentra arborescencias disjuntas por aristas utilizando matroides tal como el título del artículo lo indica.

Revisemos ahora el caso de conexidad por vértices.



1. R. E. Tarjan, en el citadísimo artículo titulado *Depth-first search and linear graph algorithms* [45], aplica `DFS` para determinar si una gráfica G cumple que $\kappa(G) = 2$. Lo anterior le toma tiempo $O(m+n)$, que corresponde con la complejidad de tiempo de `DFS`.

2. Posteriormente junto con J. Hopcroft, en el artículo titulado *Dividing a graph into triconnected components* [30], Tarjan publicó un algoritmo que determina si una gráfica G cumple que $\kappa(G) = 3$. Este algoritmo divide la gráfica en componentes 3-conexas y tiene una complejidad de $O(m + n)$, que igualmente corresponde con la complejidad de tiempo de `DFS`.

[14] λ denota a $\lambda(G)$.

[15] Un conjunto dominante D de una gráfica G es un conjunto $D \subseteq V(G)$ tal que para cualquier $v \in V(G)$ se cumple que $v \in D$, o bien, v es incidente a un vértice u que vive en D.

3. S. Even y R. E. Tarjan publicaron un algoritmo que calcula el valor de $\kappa(G)$, en el artículo titulado *Network flow and testing graph connectivity* [22]. Como su nombre lo indica utilizaron técnicas de flujos en una gráfica para calcular este valor. Este algoritmo tiene una complejidad de $O(\kappa(n-\delta-1)mn^{1/3})$.

4. Z. Galil presentó una nueva implementación del algoritmo de Even y Tarjan, en el artículo titulado *Finding the Vertex Connectivity of Graphs* [26] que calcula $\kappa(G)$ con una complejidad de tiempo de $O(\text{mín}\{\kappa, n^{2/3}\}mn)$, o bien, determina si $\kappa(G) = k$ con una complejidad de tiempo de $O(\text{mín}\{k, n^{2/3}\}kmn)$.

5. A. H. Esfahanian y S. L. Hakimi publicaron un algoritmo en el artículo titulado *On computing the connectivities of graphs and digraphs* [20] que determina si $\kappa(G)$ es al menos k. Tiene una complejidad de $O((n-\delta-1+\delta(\delta-1)/2)mn^{2/3})$.

6. A. Kanevsky y V. Ramachandran [33] presentaron un algoritmo basado en la descomposición por orejas de una gráfica G para verificar si $\kappa(G) = 4$, todo esto con una complejidad de $O(n^2)$.

7. M. Herzinger y S. Rao [29] presentaron un algoritmo secuencial que calcula $\kappa(G)$ con una complejidad $O(\text{mín}\{\kappa^3 + n, n\})$. Además, en este mismo artículo presentaron un algoritmo probabilista para calcular $\kappa(G)$, con una probabilidad de error de $1/2$, todo esto en tiempo $O(nm)$.

Como ya vimos, $\lambda(G)$ y $\kappa(G)$ pueden ser calculadas en tiempo polinomial por cualquiera de los algoritmos anteriores. Revisemos ahora el caso de conexidad mixta de una gráfica G. A continuación presentamos las dos versiones de este problema del corte mixto:

- **Corte Mixto Global.** Dados una gráfica G y dos enteros a y b. La pregunta es: ¿Podemos desconectar G al remover a lo más a vértices y b aristas?

- **Corte Mixto Local.** Dados una gráfica G, dos vértices distintos $u, v \in V(G)$ y dos enteros a y b, la pregunta es: ¿Al remover a lo más a vértices de $V(G) - \{s, t\}$ y a lo más b aristas hacemos que s y t vivan en dos componentes conexas distintas?

El problema de corte mixto local es NP-duro tal como S. Johann *et al.* demostraron en [32]. Notemos que el resultado anterior no implica que el problema del corte mixto global sea también NP-duro, ya que un corte mixto global bien podría desconectar otro par de vértices distintos de s y t. Sin embargo, E. Bonnet y S. Cabello demostraron en [6] que este problema también es NP-duro. Así, podemos concluir que el problema de encontrar un (a, b)-conjunto de corte mixto mínimo, tanto en su versión global como local, es NP-duro.

Capítulo 3

Una generalización del teorema de Menger

En el capítulo anterior revisamos el teorema de Menger para el caso de conexidad por vértices para gráficas simples, así como sus versiones para conexidad por aristas, mixta, entre otras. En este capítulo presentaremos una versión de este teorema para gráficas con pesos en las aristas. Por ello comenzamos presentando definiciones y ejemplos, para posteriormente revisar algunos resultados que se ocupan en la demostración del mismo. El teorema en su versión para gráficas con pesos, así como su demostración, serán revisados al final de este capítulo.

3.1. Gráficas con pesos

Las gráficas con pesos en las aristas tienen múltiples aplicaciones en varios campos, por ejemplo, en la investigación de operaciones, bases de datos, análisis de redes, teoría de la información, redes complejas, etc. Además, la conectividad juega un papel fundamental en estas aplicaciones. Así que en este capítulo presentaremos algunas definiciones de conexidad en gráficas con pesos, los cuales fueron hechas por S. Mathew y M. S. Sunitha en [37]. En este artículo los autores presentan versiones generalizadas, para gráficas con pesos, del famoso teorema de Menger, tanto para conexidad por vértices como por aristas. De aquí en adelante, los resultados y demostraciones fueron tomadas de [37] salvo que se indique otra cosa.

Nota. Una gráfica G sin pesos puede ser vista como un caso especial de una gráfica con pesos, en la que cada arista $e \in E(G)$ tiene un peso $w(e) = 1$.

Definición 1. *Una **gráfica con pesos** G es una gráfica con una función*

$$w\colon E(G) \to \mathbb{R}^+$$

$$w(e) \mapsto x$$

*llamaremos a x el **peso** de la arista e.*

Definición 2 (S. Mathew y M. S. Sunitha [37]). *Sea G una gráfica con pesos. La* ***fuerza*** *de una trayectoria P de n aristas e_i, con $1 \leq i \leq n$, denotada por $s(P)$, es igual a $\text{mín}_{1\leq i\leq n}\{w(e_i)\}$. La definición es análoga para la fuerza de un ciclo C.*

Definición 3 (S. Mathew y M. S. Sunitha [37]). *Sea G una gráfica con pesos. La* ***fuerza de conexidad*** *de un par de vértices $u, v \in V(G)$, denotada por $CONN_G(u,v)$, está definida como:*

$$CONN_G(u,v) = \text{máx}\{s(P) : P \textit{ es una } (u,v)\textit{-trayectoria en } G\}.$$

Si u y v viven en distintas componentes de G, entonces $CONN_G(u,v) = 0$. Además, si G no es dirigida tenemos que $CONN_G(u,v) = CONN_G(v,u)$.

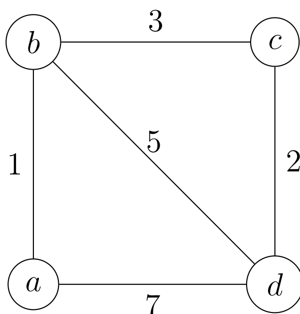


Figura 3.1: Fuerza de conexidad en una gráfica con pesos G

Ejemplo 1. *Sea G como en la figura 3.1. Tenemos que: $CONN_G(a,b) = 5$ ya que entre a y b tenemos 3 (a,b)-trayectorias: $P_1 = (a,b)$ tal que $s(P_1) = 1$, $P_2 = (a,d,b)$ tal que $s(P_2) = 5$ y $P_3 = (a,d,c,b)$ tal que $s(P_3) = 2$. Para los demás vértices tenemos, $CONN_G(a,d) = 7$, $CONN_G(a,c) = 3$, $CONN_G(b,c) = 3$, $CONN_G(b,d) = 5$ y $CONN_G(c,d) = 3$.*

Definición 4 (S. Mathew y M. S. Sunitha [37]). *A una (u,v)-trayectoria P en una gráfica con pesos G le decimos* ***fuerte*** *si $s(P) = CONN_G(u,v)$.*

Definición 5 (S. Mathew y M. S. Sunitha [37]). *Sea G una gráfica con pesos. Un vértice w es un* ***p-vértice de corte*** *de G si existen un par de vértices $u, v \in V(G)$ tales que $u \neq v \neq w$ y $CONN_{G-w}(u,v) < CONN_G(u,v)$.*

Ejemplo 2. *Sea G como en la figura 3.1. El vértice b es un p-vértice de corte ya que $CONN_{G-b}(a,c) = 2 < CONN_G(a,c)$.*

Definición 6 (S. Mathew y M. S. Sunitha [37]). *Sea G una gráfica con pesos. Una arista $e = (u,v)$ es un* ***p-puente*** *de G si $CONN_{G-\{e\}}(u,v) < CONN_G(u,v)$.*

Ejemplo 3. *Sea G como en la figura 3.1. La arista (b,d) es un p-puente ya que*

$$CONN_{G-\{(b,d)\}}(b,d) = 2 < CONN_G(b,d) = 5.$$

El teorema 3.2 fue enunciado en [37] sin demostración. Nosotros planteamos una demostración y, para ella, proponemos el siguiente lema.

Lema 3.1. *Sea C un ciclo en una gráfica G con pesos. Si el peso de una arista $e \in E(C)$ es menor que el peso de cada una de las otras aristas en C, entonces e no puede pertenecer a ningún árbol generador de peso máximo de G.*

Demostración. Sea G como en el enunciado y T un árbol generador de peso máximo de G. Supongamos que T contiene una arista e como en la hipótesis y lleguemos a una contradicción. Consideremos ahora un árbol generador de peso máximo T' construido a partir de $(T - \{e\}) \cup \{e'\}$, con $e' \neq e$ y $e' \in E(C)$ con C el ciclo de G que contiene a la arista e. Notemos que $w(e) < w(e')$. Así, tenemos que:

$$w(T) = w(T - \{e\}) + w(e) < w(T - \{e\}) + w(e') = w(T').$$

Hemos encontrado un árbol generador de peso máximo T' más pesado que T, lo que es una contradicción. □

Teorema 3.2 (S. Mathew y M. S. Sunitha [37])**.** *Sea G una gráfica con pesos. $r \in V(G)$ es un p-vértice de corte si y solo si r es vértice interior de todo árbol generador de peso máximo de G.*

Demostración. Sea G como en la hipótesis y r es vértice interior de todo árbol generador de peso máximo T de G. Así, r tiene al menos dos vecinos, u, v, en T. Luego tenemos los siguientes dos casos.

Caso 1. Al eliminar r de G desconectamos a u de v. Así, tenemos que:

$$0 = CONN_{G-r}(u, v) < CONN_G(u, v).$$

Recordemos que los pesos de la gráfica son todos positivos. Por lo tanto, r es p-vertice de corte en este caso.

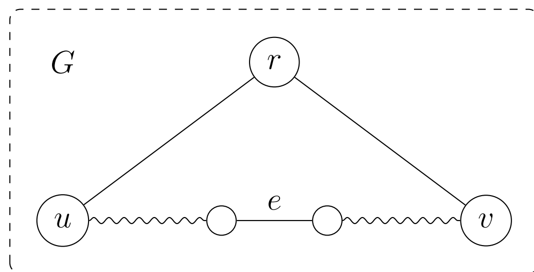


Figura 3.2: Ciclo C

Caso 2. Al remover r de G, u y v permanecen conectados. Esto sucede si existe una (u, v)-trayectoria en $G - r$ que llamaremos P (figura 3.2). Consideremos ahora el ciclo:

$$C = P \cup \{(u, r), (r, v)\}.$$

Por el lema 3.1, sabemos que las aristas del ciclo C que pertenecen a T tienen un peso mayor que aquellas que no pertenecen a T. Más aún, sabemos que existe al menos una arista e en el ciclo C que no pertenece a T. Así, $w(e) < w(e')$ con $e' \in E(T) \cap E(C)$. En específico, tanto (u, r) como (r, v) cumplen esta desigualdad: $w(e) < w(u, v)$ y $w(e) < w(r, v)$. Así al tomar la fuerza de cada una de las trayectorias anteriores tenemos que:

$$w(e) = \text{mín}\{w(e') \colon e' \in E(P)\} = s(P) < \text{mín}\{w((u, r)), w((r, v))\} = s(u, v, r).$$

Así, al tomar la fuerza de conexidad entre u y v en $G - \{r\}$, tenemos como resultado que:

$$s(P) = CONN_{G-r}(u, v) < CONN_G(u, v).$$

Por lo que r es vértice de corte en G.

De manera análoga si r es p-vértice de corte podemos demostrar que r es vértice interior de todo árbol generador de peso máximo T de la gráfica G.

□

Definición 7 (S. Mathew y M. S. Sunitha [37]). *Un (u, v)-**conjunto reductor de fuerza de vértices** en una gráfica con pesos G es un conjunto de vértices $S \subseteq V(G)$ tal que para algún par de vértices $u, v \in G - S$ o $G - S$ es trivial, tenemos que $CONN_{G-S}(u, v) < CONN_G(u, v)$. Si $S = \{w\}$, entonces w es un p-vértice de corte.*

Definición 8 (S. Mathew y M. S. Sunitha [37]). *Un (u, v)-**conjunto reductor de fuerza de aristas** en una gráfica con pesos G es un conjunto de aristas $F \subseteq E(G)$ tal que $CONN_{G-F}(u, v) < CONN_G(u, v)$ para algún par de vértices u, v con al menos uno de ellos no incidente a las aristas en F.*

Definición 9 (S. Mathew y M. S. Sunitha [37]). *Un (u, v)-conjunto reductor de fuerza con n elementos es un (u, v)-**conjunto reductor de fuerza mínimo** si no existe un (u, v)-conjunto reductor de fuerza con menos de n elementos.*

Definición 10 (S. Mathew y M. S. Sunitha [37]). *Sea G una gráfica con pesos. Una arista $e = (x, y)$ es **fuerte** si el peso de e es igual que la fuerza de conexidad entre sus vértices x y y en G. Una arista $e = (x, y)$ es:*

- *α-fuerte si $CONN_{G-\{e\}}(x, y) < w(e)$*
- *β-fuerte si $CONN_{G-\{e\}}(x, y) = w(e)$*
- *δ-fuerte si $CONN_{G-\{e\}}(x, y) > w(e)$*

*Así, una arista es **fuerte** si y solo si la arista es α-fuerte o β-fuerte.*

El siguiente teorema es enunciado sin demostración en este mismo artículo [37].

Teorema 3.3 (S. Mathew y M. S. Sunitha [37]). *Sea G conexa y con pesos y sean $u, v \in V(G)$ tales que (u, v) no es una arista fuerte. Un conjunto $S \subseteq V(G)$ es un (u, v)-conjunto reductor de fuerza de vértices en G si y solo si toda (u, v)-trayectoria fuerte contiene al menos un vértice en S.*

Demostración. Sean G y $u, v \in V(G)$ como en el enunciado. Supongamos que S es un (u, v)-conjunto reductor de fuerza de vértices en G, pero existe una (u, v)-trayectoria fuerte P tal que P no contiene ningún vértice en S. Así, en $G - S$ tendríamos que $CONN_{G-S}(u, v) = s(P)$, pues P es fuerte. Por lo tanto:

$$CONN_{G-S}(u, v) = s(P) = CONN_G(u, v).$$

Lo que contradice que S sea un conjunto reductor de fuerza. □

Teorema 3.4 (S. Mathew y M. S. Sunitha [37])**.** *Sea G conexa y con pesos y sean $u, v \in V(G)$. Entonces un conjunto $F \subseteq E(G)$ es un (u, v)-conjunto reductor de fuerza de aristas en G si y solo si toda (u, v)-trayectoria fuerte contiene al menos una arista en F.*

La demostración del teorema anterior es análoga a la del teorema 3.3.

Definición 11 (S. Mathew y M. S. Sunitha [37])**.** *El número de aristas fuertes en una gráfica con pesos G es el* ***tamaño fuerte*** *de G. Lo denotamos como $ss(G)$.*

Lema 3.5. *Sea G con pesos, si v es un vértice en G con al menos una arista fuerte incidente a v, entonces $ss(G - v) < ss(G)$*

Demostración. Sean G y x como en el enunciado y sea (u, v) una arista fuerte incidente a v. Supongamos que $ss(G) = n$, para $n \in \mathbb{N}$. Así, tenemos que al remover v de $V(G)$ la arista (u, v) desaparece. Así, tenemos que:

$$ss(G - v) \leq n - 1 < n = ss(G)$$

□

Nota. Si E es un conjunto de aristas fuertes de cardinalidad n en G, al remover E de G reducimos el tamaño fuerte de G por n. Es decir, tenemos que $ss(G - E) = ss(G) - n$.

Nota. Para definir la fuerza de conexidad entre un par de vértices u, v en una gráfica G, es necesario que exista al menos una (u, v)-trayectoria en G. Si no existe una (u, v)-trayectoria en G, diremos que $CONN_G(u, v) = 0$.

Lema 3.6. *G no tiene aristas si y solo si $ss(G) = 0$.*

Demostración. Si G tiene al menos una arista $e = (u, v)$, con $w(e) > 0$, esta arista e es fuerte. Ya que la única (u, v)-trayectoria en G es e con peso $w(e)$ por lo tanto, trivialmente tenemos que $CONN_G(u, v) = w(e)$.

Notemos que si G no tiene aristas, entonces no podemos encontrar una arista fuerte. □

El teorema 3.3, así como el lema 3.5, aparecen en este mismo artículo [37] y sirven para la demostración del teorema 3.7.

3.2. Teorema de Menger generalizado

El siguiente teorema es la versión generalizada del teorema de Menger para el caso de conexidad por vértices. La demostración es personal, pero está basada en la prueba que los autores presentan en [37]. Ya que, descubrimos que la prueba original contenía algunos errores.

Teorema 3.7 (S. Mathew y M. S. Sunitha [37])**.** *Sea G una gráfica con pesos. Para cualesquiera $u, v \in V(G)$ tales que (u, v) no es fuerte, el máximo número de (u, v)-trayectorias fuertes internamente ajenas en G es igual al número de vértices en un (u, v)-conjunto reductor de fuerza minimal.*

Demostración. La demostración será por inducción sobre el tamaño fuerte $ss(G)$ de G. Cuando $ss(G) = 0$, por el lema 3.6 tenemos que la única posibilidad es que G no tenga aristas, así que para cualquier par de vértices u y v, no existe una (u, v)-trayectoria en G. Luego, ambos parámetros dados en el enunciado se reducen a cero y por lo tanto el resultado es trivialmente cierto para cualquier par de vértices $u, v \in V(G)$.

Supongamos ahora que el enunciado es verdadero para cualquier gráfica con pesos G de tamaño fuerte menor a m, con $m \geq 1$. Sea G tal que $ss(G) = m$. Sean $u, v \in V(G)$ tal que (u, v) no es fuerte. Si u y v viven en diferentes componentes conexas, el resultado es claramente cierto. Así, supongamos que u, v pertenecen a la misma componente de G. Así, u, v no son adyacentes, o bien, (u, v) es una δ-arista. En ambos casos, existen (u, v)-conjuntos reductores de fuerza en G, que veremos más adelante. Notemos que si $(u, v) \in E(G)$ fuera una arista fuerte, entonces al eliminar cualquier número de vértices no se reduciría la fuerza de conexidad entre u y v, por lo tanto, no existiría ningún conjunto reductor de fuerza entre este par de vértices.

Supongamos que $S_G(u, v)$ es un (u, v)-conjunto reductor de fuerza minimal en G tal que $|S_G(u, v)| = k \geq 1$. Sabemos por el teorema 3.3 que cada (u, v)-trayectoria fuerte debe contener al menos un elemento de $S_G(u, v)$. Así, cualquier (u, v)-conjunto reductor de fuerza debe contener al menos tantos vértices como el número de (u, v)-trayectorias fuertes internamente ajenas. Es decir, existen a lo más k (u, v)-trayectorias fuertes internamente ajenas.

Si $k = 1$, entonces $|S_G(u, v)| = 1$. Sea $S_G(u, v) = \{w\}$, como $S_G(u, v)$ es un (u, v)-conjunto reductor de fuerza, entonces $CONN_{G-w}(u, v) < CONN_G(u, v)$. Así, w es un p-vértice de corte de G. Así que toda (u, v)-trayectoria fuerte tiene que pasar por w. Por lo que, el número de (u, v)-trayectorias fuertes internamente ajenas es una, luego el enunciado es cierto. Ahora supongamos que $k \geq 2$. Así, tenemos los siguientes casos:

Caso 1. El (u, v)-conjunto reductor de fuerza minimal de G contiene un vértice x tal que (u, x) es α-fuerte y (x, v) es fuerte, o bien, viceversa.

Sea $S_G(u, v)$ un (u, v)-conjunto reductor de fuerza como en el enunciado. Tenemos que $S_G(u, v) - \{x\}$ es un (u, v)-conjunto reductor de fuerza minimal en $G - x$ con $k - 1$ vértices; ya que si hubiera otro (u, v)-conjunto reductor de fuerza en $G - x$ menor a $S_G(u, v) - \{x\}$ significa que $S_G(u, v)$ no es un (u, v)-conjunto reductor de fuerza minimal de G. Como (u, x) y (x, v) son fuertes luego, $ss(G - x) < ss(G)$. Por hipótesis de inducción, tenemos que $G - x$ contiene $k - 1$ (u, v)-trayectorias fuertes internamente ajenas.

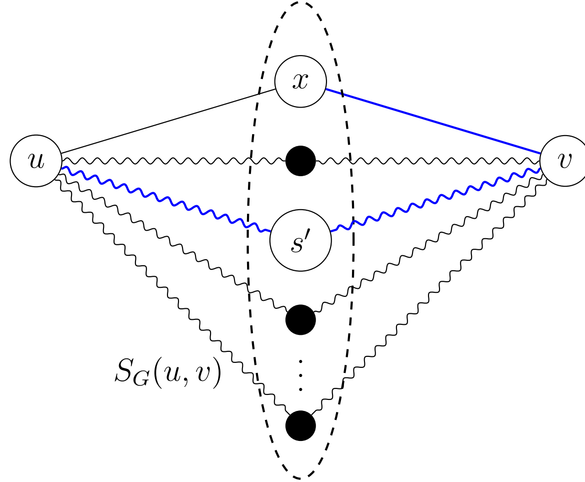


Figura 3.3: Caso 1. Los vértices dentro del óvalo representan a los vértices del conjunto $S_G(u, v)$. Las aristas serpenteadas representan las trayectorias fuertes entre los vértices. La (u, x)-trayectoria P' aparece en azul.

Veamos ahora que $P = (u, x, v)$ es una (u, v)-trayectoria fuerte. Tenemos que el valor de $CONN_G(u, v) = \rho$, con $\rho \in \mathbb{R}^+$. Veamos que:

$$s(P) = \text{mín}\{w((u, x)), w((x, v))\} = \rho.$$

Supongamos que no y lleguemos a una contradicción. Así, por tricotomía tenemos los siguientes dos subcasos:

1. $s(P) = \text{mín}\{w((u, x)), w((x, v))\} < \rho$. En este subcaso, una o ambas aristas tienen un peso menor a ρ. Primero, supongamos que $w((u, x)) = w((x, v)) < \rho$. Esto contradice que (u, x) es α-fuerte, pues tendríamos que la (u, x)-trayectoria $P' = (u, \ldots, s', \ldots, v, x)$, con $s' \in S_G(u, v)$, es tal que, $s(P') = \rho$.

   Supongamos ahora que $w((x, v)) \neq w((u, x))$. Así, $w((x, v)) < \rho$. Sin embargo, esto contradice que la arista (x, v) es fuerte, ya que la (x, v)-trayectoria $P'' = (x, u, \ldots, s' \ldots, v)$ es tal que, $s(P'') = \rho > w((u, x))$.

2. $s(P) = \text{mín}\{w((u, x)), w((x, v))\} > \rho$. En este subcaso, ambas aristas tienen un peso mayor a ρ. Sin embargo, esto contradice que $CONN_G(u, v) = \rho$, pues tendríamos que la (u, v)-trayectoria P es tal que $s(P) = \text{mín}\{w((u, x)), w((x, v))\} > \rho$.

De ambos casos tenemos que $s(P) = \rho = CONN_G(u, v)$. Así, P es una (u, v)-trayectoria fuerte en G. Luego, tenemos $k - 1 + 1 = k$ (u, v)-trayectorias fuertes internamente ajenas.

Caso 2. El (u, v)-conjunto reductor de fuerza minimal de G, $S_G(u, v)$, es tal que todo vértice en él es vecino α-fuerte de u pero no de v, o bien, todo vértice en él es vecino α-fuerte de v pero no de u.

Supongamos, sin pérdida de generalidad, que todo vértice en $S_G(u, v)$ es vecino α-fuerte de u pero no de v y que $CONN_G(u, v) = \rho$, con $\rho \in \mathbb{R}^+$. Consideremos una

(u, v)-trayectoria fuerte P en G con la menor cantidad de aristas (geodésica). Sea x el primer vértice de P tal que $x \in S_G(u, v)$. Así, por hipótesis (u, x) es una arista α-fuerte. Supongamos que $(x, v) \in E(G)$. Si la arista (x, v) es fuerte, caemos en el caso 1. Por otro lado, si (x, v) no es fuerte, estaríamos contradiciendo que P es una (u, v)-trayectoria fuerte.

De cualquier manera no puede existir la arista $(x, v) \in E(G)$, por lo tanto existe un vértice $y \in V(G)$ tal que $(x, y) \in E(G)$ y $(x, y) \in P$; denotamos a (x, y) como e. Así, tenemos que $P = (u, x, y, \ldots, v)$ es la (u, v)-trayectoria fuerte con la menor cantidad de aristas.

Veamos ahora que (x, y) es una arista fuerte. Supongamos que no y lleguemos a una contradicción. Tenemos que $CONN_G(u, v) = \rho$ y $e = (x, y)$ es δ-fuerte. Es decir, $\sigma = w((x, y)) < CONN_G(x, y) = \rho$, con $\sigma \in \mathbb{R}^+$. Sin embargo, esto es absurdo, pues tendríamos que $s(P) = \text{mín}_{1 \leq i \leq n}\{w(e_i)\} = \rho > \sigma = w((x, y))$, con $(x, y) \in E(P)$.

Veamos ahora que todo (u, v)-conjunto reductor de fuerza minimal en $G - \{e\}$ tiene exactamente k vértices. Supongamos que no y lleguemos a una contradiccion. Supongamos que existe un (u, v)-conjunto reductor de fuerza minimal de $G - \{e\}$ con $k - 1$ vértices, digamos $Z = \{z_1, z_2, \ldots, z_{k-1}\}$. Luego, $Z \cup \{x\}$ es un (u, v)-conjunto reductor de fuerza minimal en G. Pero, también $Z \cup \{y\}$ es un (u, v)-conjunto reductor de fuerza minimal en G, así que y es también un vecino α-fuerte de u.

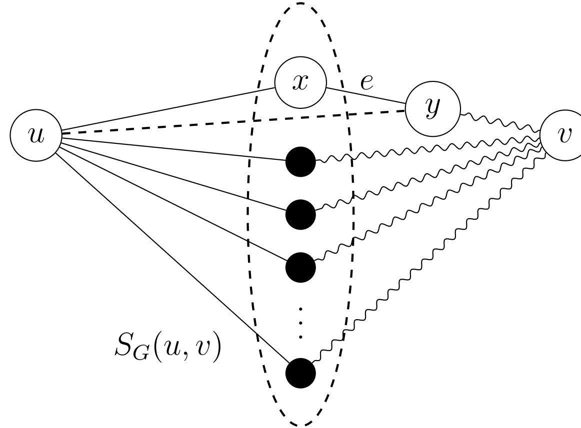


Figura 3.4: Caso 2. Los vértices dentro del óvalo representan a los vértices del conjunto $S_G(u, v)$. Las aristas serpenteadas representan parte de las trayectorias fuertes entre los vértices de $S_G(u, v)$ y v.

Si y es un vecino α-fuerte de u, tenemos que (u, y) es una arista α-fuerte de G (véase la figura 3.4.). Consideremos ahora el triángulo generado por las aristas α-fuertes: (u, x) y (u, y) y la arista fuerte (x, y). Si (x, y) es una arista α-fuerte, entonces $w((x, y)) = w((u, x)) = w((u, y))$. Así, hemos encontrado una (u, v)-trayectoria $P' = (u, y, \ldots, v)$ con una menor cantidad de aristas que P. Lo anterior es una contradicción, pues P es una geodésica. Por otro lado, si $e = (x, y)$ es una arista β-fuerte tendríamos que e es la arista más débil (con menor peso) del triángulo, esto contradice que e sea fuerte, pues la arista más débil (con menor peso) en un ciclo es δ-fuerte.

Así, k es el mínimo número de vértices en una (u, v)-conjunto reductor de fuerza en $G - \{e\}$. Y como $ss(G - \{e\}) < ss(G)$, entonces por hipótesis de inducción tenemos que

existen k (u, v)-trayectorias fuertes internamente ajenas en $G - \{e\}$ y por lo tanto, también en G.

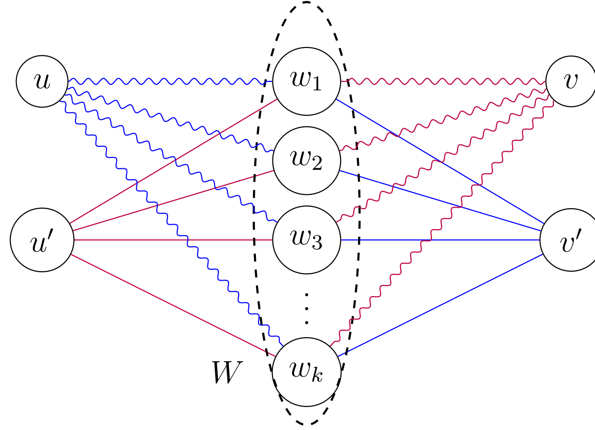


Figura 3.5: Caso 3. Dentro del óvalo los vértices $w_i \in W$. En azul las aristas correspondientes a G'_u y en morado las aristas correspondientes a G'_v.

Caso 3. Existe un (u, v)-conjunto reductor de fuerza W en G tal que ningún elemento de W es un vecino α-fuerte de u y w al mismo tiempo y W contiene al menos un vértice que no es un vecino α-fuerte de u y al menos un vértice que no es un vecino α-fuerte de v.

Sea W un (u, v)-conjunto reductor de fuerza minimal con k elementos que cumple las propiedades del enunciado. Sea $W = \{w_1, w_2, \ldots, w_k\}$ (véase figura 3.5). Consideremos todas las trayectorias fuertes entre u y v. Como W es minimal, cada uno de los vértices w_i, con $i = 1, 2, \ldots, k$, deben pertenecer a al menos a una de estas trayectorias. Sea G_u la subgráfica de G que consiste de todas las (u, w_i)-subtrayectorias fuertes de entre todas las (u, v)-trayectorias fuertes, donde $w_i \in W$ es el único vértice de la trayectoria que pertenece a W. Notemos que si bien pueden existir aristas entre los w_i en G, estas aristas no son parte de G_u ya que consideramos las subtrayectorias de u hasta el primer $w_i \in W$.

Sea G'_u la gráfica construida a partir de G_u al añadir un nuevo vértice v' y unir este vértice con todos los w_i, con $i = 1, 2, \ldots, k$ por medio de aristas de la forma (w_i, v'). Notemos que estas aristas son α-fuertes, pues estas aristas son las únicas (w_i, v')-trayectorias en G'_u. De manera análoga definimos G'_v (véase la figura 3.5).

Como W contiene al menos un vértice que no es vecino α-fuerte de u y al menos un vértice que no es vecino α-fuerte de v, tenemos que $ss(G'_u) < ss(G)$ y $ss(G'_v) < ss(G)$. Además, tenemos que $|S_{G'_u}(u, v')| = k$ y $|S_{G'_v}(u, v')| = k$. Así, por hipótesis de inducción, G'_u contiene k (u, v')-trayectorias fuertes internamente ajenas digamos A_i, $i = 1, 2, \ldots, k$, donde cada A_i contiene al vértice w_i correspondiente. También, por hipótesis de inducción, G'_v contiene k (u', v)-trayectorias fuertes internamente ajenas digamos B_i, $i = 1, 2, \ldots, k$, donde cada B_i contiene al vértice w_i correspondiente.

Sea A'_i las (u, w_i)-subtrayectorias de A_i y B'_i las (w_i, v)-subtrayectorias de B_i para $1 \leq i \leq k$. Consideremos las k (u, v)-trayectorias formadas por la unión de una (u, w_i)-trayectoria y una (w_i, v)-trayectoria respectivamente. Denotaremos al conjunto de estas trayectorias como $\mathcal{P} = \cup P_i$, donde cada P_i representa la trayectoria que surge al concatenar

la (u, w_i)-trayectoria y la (w_i, v)-trayectoria para ese preciso valor de i. Notemos que cada una de estas (u, v)-trayectorias es internamente ajena, pues las trayectorias en A_i y en B_i son ajenas entre ellas y solo comparten el vértice w_i. Además, cada una de ellas contiene un vértice interior w_i, el cual pertenece a un (u, v)-conjunto reductor de fuerza W. Así, tenemos los siguientes subcasos:

- Al quitar W desconectamos a u de v, es decir, $CONN_{G-W}(u, v) = 0$. En este subcaso las trayectorias en P_i son trivialmente (u, v)-trayectorias fuertes.

- Al quitar W tenemos que $0 < CONN_{G-W}(u, v) < CONN_G(u, v)$. Esto sucede porque cada w_i es adyacente a una arista e_i, tal que $w(e_i) = CONN_G(u, v)$, respectivamente. Ya que si esta e_i no fuera adyacente a w_i, podríamos encontrar una (u, v)-trayectoria P' en $G - W$ que no contiene a w_i y $s(P') = w(e_i) = CONN_G(u, v)$. Lo anterior contradice que W sea un (u, v)-conjunto reductor de fuerza de G. Por lo tanto, las (u, v)-trayectorias que pertenecen a $\mathcal{P}$ son (u, v)-trayectorias fuertes.

Así, de ambos subcasos tenemos que en G existen k (u, v)-trayectorias fuertes internamente ajenas. Y así, de los tres casos hemos probado el teorema por inducción sobre el tamaño fuerte de G.

□

Por último, presentamos el enunciado correspondiente al caso de conexidad por aristas en gráficas con pesos. Su demostración es análoga a la del teorema anterior.

Teorema 3.8. *Sean G una gráfica con pesos y $u, v \in V(G)$. Entonces el número de (u, v)-trayectorias fuertes ajenas por aristas en G es igual al número de aristas en un (u, v)-conjunto reductor de fuerza minimal.*

Capítulo 4

Certificados de 2-conexidad

> "Ahora bien, ven, hagamos un pacto tú y yo y que sirva de testimonio entre los dos."
>
> *Génesis 31:44*

A lo largo de los dos capítulos anteriores hemos revisado conceptos y resultados de conexidad por aristas, vértices y mixta. Con esto en mente, en este capítulo revisaremos cronológicamente los algoritmos mencionados en la introducción para el caso específico de 2-conexidad. Por último, veremos como se comparan cada uno de estos algoritmos con respecto a su entrada, su salida, su complejidad y el tipo de conexidad que manejan.

Un **certificado de 2-conexidad por aristas** de una gráfica G es una subgráfica $G' \subseteq G$, tal que G' es 2-conexa por aristas. Análogamente, un **certificado de 2-conexidad por vértices** de una gráfica G es una subgráfica $G' \subseteq G$, tal que G' es 2-conexa. Como mencionamos en la introducción, los certificados de conexidad pueden ser utilizados como preprocesamiento con el fin de disminuir la complejidad de tiempo de otros algoritmos. Así que, estas subgráficas G' se caracterizan por tener una menor cantidad de aristas que la gráfica original G. Más en específico, decimos que son subgráficas **ralas** ya que la cantidad de aristas es $O(n)$, donde n es el número de vértices de la gráfica[1]. De igual manera que en los capítulos anteriores, las proposiciones, así como las demostraciones del presente son personales salvo que se indique otra cosa. En adelante, G es una gráfica que puede tener aristas múltiples, pero no tiene lazos, a menos que se indique otra cosa.

4.1. `NI-Search`

Revisemos primero el algoritmo publicado por Nagamochi e Ibaraki en [41]. A este algoritmo lo llamaremos `NI-search` (véase la figura 5.1). `NI-Search` etiqueta a todas las aristas de G con valores que van desde 1 hasta $|E(G)|$. Esto por medio de la función $rank(e)$, cuyo valor nos indica el bosque maximal al cual la arista e pertenece. Así que en este caso, para generar un certificado de 2-conexidad nos quedaramos unicamente con las aristas $e \in E(G)$ tales que $rank(e) \leq 2$. En la proposición 4.1 veremos que basta con estos dos bosques para generar un certificado de 2-conexidad por aristas.

[1] Tengamos en cuenta que el tamaño de una gráfica completa es $O(n^2)$.

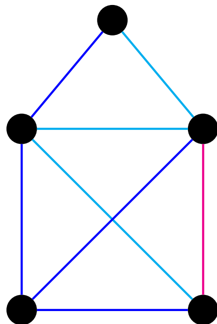


Figura 4.1: Resultado de la ejecución de `NI-Search` con $rank(e) = 1$ en azul, $rank(e) = 2$ en cian y $rank(e) = 3$ en magenta.

Este algoritmo comienza por un vértice y genera primero un bosque maximal[2] de G formado por las aristas tales que $rank(e) = 1$, al que llamaremos F_1 y luego genera un bosque maximal de la subgráfica $G - E(F_1)$ formado por las aristas tales que $rank(e) = 2$. Obteniendo así una subgráfica $F_1 \cup F_2$, tal que esta es un certificado de 2-conexidad por aristas.

Ejemplo 4. *Para ver una ejemplo de la ejecución de este algoritmo, véase el apéndice A.*

Proposición 4.1. *Sean G una gráfica y $x, y \in V(G)$, tales que $\lambda_G(x, y) \geq 2$. Sean F_1 un bosque maximal en G y F_2 un bosque maximal en $G - E(F_1)$ generado por `NI-search`, entonces $F_1 \cup F_2$ es un certificado de 2-conexidad por aristas de G.*

Demostración. Sean F_1 y F_2 como en la hipótesis, así tenemos que F_1 es un bosque maximal en G, por lo tanto, tenemos que:

$$\lambda_{F_1}(x, y) \geq \text{mín}\{\lambda_G(x, y), 1\}\ \forall x, y \in V(G) \tag{4.1}$$

$\lambda_{F_1}(x, y)$ denota la conexidad por aristas en la subgráfica F_1. Supongamos que $F_1 \cup F_2$ no es un certificado de 2-conexidad por aristas de G. Así,

$$\lambda_{F_1 \cup F_2}(x, y) < \text{mín}\{\lambda_G(x, y), 2\} \tag{4.2}$$

Notemos que F_1 y F_2 son subgráficas de G y $F_1 \cup F_2$ surge de agregar aristas a F_1, por lo que,

$$\lambda_{F_1}(x, y) \leq \lambda_{F_1 \cup F_2}(x, y) \tag{4.3}$$

Luego, de las desigualdades 4.1,4.2 y 4.3 tenemos que:

$$1 = \lambda_{F_1}(x, y) = \lambda_{F_1 \cup F_2}(x, y) < 2 \leq \lambda_G(x, y).$$

Así, $F_1 \cup F_2$ tiene una arista de corte e tal que $F_1 \cup F_2 - \{e\}$ tiene dos componentes conexas X y Y tal que $x \in X$ y $y \in Y$. Notemos que $\{e\} \cap E(F_2) = \varnothing$ ya que $\lambda_{F_1}(x, y) = 1$, es decir, $e \in E(F_1)$ y $E(F_1) \cap E(F_2) = \varnothing$. Sin embargo, existe $e' \in E(G) \setminus (E(F_1) \cup E(F_2))$ que conecta a x y y, pues $\lambda_G(x, y) \geq 2$ por la desigualdad 4.3. Esto contradice la maximalidad de F_2. Por lo tanto, $F_1 \cup F_2$ es un certificado de 2-conexidad de G. $\square$

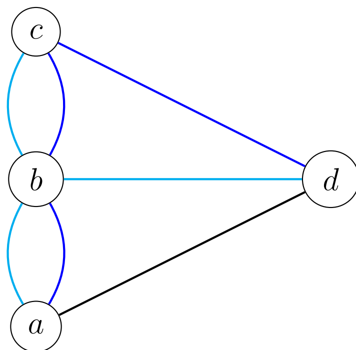


Figura 4.2: F_1 en azul y F_2 en cyan, $F_1 \cup F_2$ no es un certificado de 2-conexidad por vértices

Así, `NI-Search` genera certificados de 2-conexidad por aristas. Sin embargo, notemos que en el caso de conexidad por vértices, $F_1 \cup F_2$ generado por `NI-search`, no siempre es un certificado de 2-conexidad por vértices(véase figura 4.2). Sin embargo, si G es simple, es decir que no tiene aristas múltiples, entonces $F_1 \cup F_2$ sí es un certificado de 2-conexidad por vértices [41].

4.1.1. Propiedades de conexidad mixta

Además de generar certificados de k-conexidad por aristas y por vértices (en gráficas simples), `NI-search` genera también certificados que tienen cierta propiedad de conexidad mixta tal como lo hicieron notar A. Frank, T. Ibaraki y H. Nagamochi en [23]. Lo que ellos demostraron es que, si la gráfica G es S-simple, entonces la subgráfica F_k generada por `NI-search` es un certificado de k-conexidad S-mixta local de G.[3] Antes de revisar este resultado, revisaremos algunas definiciones y enunciados previos. Las siguientes son algunas definiciones de conexidad mixta que tomamos de este mismo artículo.

Definición 12 (A. Frank, T. Ibaraki y H. Nagamochi [23])**.** *Sean G una gráfica conexa y $S \subseteq V(G)$. Decimos que G es S-**simple** si el conjunto de aristas paralelas conecta a los vértices en $V(G) - S$.*

Definición 13 (A. Frank, T. Ibaraki y H. Nagamochi [23])**.** *Sean G una gráfica conexa y $S \subseteq V(G)$. Decimos que una familia $\mathcal{F}$ de (x, y)-trayectorias es S-**independiente** si las trayectorias en $\mathcal{F}$ son disjuntas por aristas y todo elemento de $S \subseteq V(G)$ aparece como vértice interior en a lo más una de estas trayectorias.*

Definición 14. *La **conexidad** S**-mixta** de x y y en G, denotada por $\lambda_S(x, y; G)$, es el máximo número de (x, y)-trayectorias S-independientes.*

Así, $S = \varnothing$ corresponde con la conexidad local por aristas entre x y y y $S = V(G)$ corresponde con la conexidad local por vértices entre estos mismos dos vértices.

[2]Definimos que una gráfica F es un bosque si F es acíclica. Así, los bosques consisten unicamente de la unión disjunta de uno o más árboles.

[3]Cada F_i, con $i = 1, \ldots, k$, es un bosque maximal en $G - E(F_1) - \cdots - E(F_{i-1})$

Definición 15 (A. Frank, T. Ibaraki y H. Nagamochi [23])**.** *Sean G una gráfica conexa y $S \subseteq V(G)$ y sea $\{Z, A, B\}$ una partición de $V(G)$ tal que $Z \subseteq S$ (Z puede ser vacío), $A \neq \emptyset$ y $B \neq \emptyset$. Decimos que el par $C = (Z, E(A, B))$ es un **corte S-mixto**; donde C, al ser eliminado de la gráfica G, separa a dos vértices x y y si $x \in A$ y $y \in B$, o bien, viceversa. La cardinalidad de $|C| = |Z| + |E(A, B)|$.*[4][5]

Con esto en mente los autores enunciaron la siguiente versión del ubicuo teorema de Menger, el cual aparece como teorema 1.2 en [23]. Además, ocupan el famoso teorema de Flujo Máximo y Corte Mínimo en la demostración del mismo.

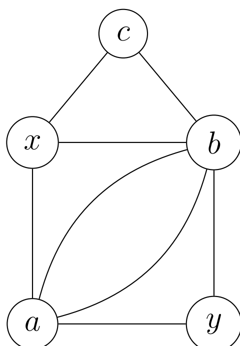


Figura 4.3: Gráfica G S-simple con $S = \{x, y, c\}$

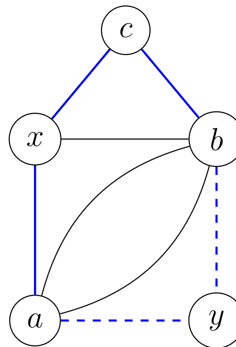


Figura 4.4: 2 Trayectorias S-independientes en azul. Nótese que al quitar las 2 aristas punteadas desconectamos a x y y.

Teorema 4.2 (A. Frank, T. Ibaraki y H. Nagamochi [23])**.** *Sea G una gráfica S-simple entonces $\lambda_S(x, y; G)$ es igual a la cardinalidad mínima de un corte S-mixto que separa a x de y.*

Demostración. Sea G una gráfica S-simple. A partir de G construiremos la digráfica G' de la siguiente manera:

- Reemplazamos toda $(u, v) \in E(G)$ por dos flechas (u, v) y (v, u) en $A(G')$.
- Separamos cada vértice $v \in S \setminus \{x, y\}$ en dos vértices v' y v'' y agregamos la flecha (v', v'') a $A(G')$.
- Reemplazamos cada flecha (x, y) en $A(G')$ por flechas de la forma (x'', y') donde x'' denota x si $x \in V(G) - S$ y y' denota a y si $y \in V(G) - S$.

Aplicamos el teorema de Flujo Máximo y Corte Mínimo[6] a la digráfica G' y así finalizamos la prueba. □

[4] $E(X, Y)$ denota al conjunto de aristas con un extremo en X y otro en Y.

[5] Observemos que para una gráfica G se cumple que $\lambda_S = \lambda(G)$ si $S = \emptyset$ y $\lambda_S = \kappa(G)$ si $S = V(G)$

[6] Para revisar el enunciado y la demostración de este teorema, recomendamos el texto de L.R. Ford y D.R. Fulkerson, Flows in Networks. Princeton University Press, Estados Unidos (1962).

El siguiente resultado es el teorema principal del artículo [23].

Teorema 4.3 (A. Frank, T. Ibaraki y H. Nagamochi [23]). *Si G es S-simple, entonces:*

$$\lambda_S(x, y; F_k) \geq \min\{\lambda_S(x, y; G), k\} \ \forall x, y \in V(G).$$

El resultado anterior nos dice que si la gráfica G es S-simple, entonces la subgráfica F_k generada por `NI-search` es un certificado de k-conexidad S-mixta entre los vértices x y y de G ya que, $\lambda_S(x, y; F_k)$ queda acotado por debajo por k o bien por $\lambda_S(x, y; G)$ de la gráfica original. Es decir, la cantidad de (x, y)-trayectorias S-independientes en F_k es igual al mínimo entre k, o bien, la cantidad de (x, y)-trayectorias S-independientes en la gráfica G original. La demostración del teorema anterior es larga y desviaría el foco de atención del presente trabajo, razón por la cual no la incluimos. Retomaremos este teorema en el capítulo 5 dedicado a los certificados de k-conexidad cuando revisemos el caso de conexidad mixta.

4.2. `Scan-first Search`

Siguiendo el orden cronológico, revisemos ahora el algoritmo publicado por Cheriyan, Kao y Thurimella en [9], el cual ellos nombraron `Scan-first Search` (véase figura 5.2.). Este algoritmo, para $k = 2$, genera un bosque maximal en G, al que llamaremos F_1 y un bosque maximal en $G - E(F_1)$, al que llamaremos F_2. $F_1 \cup F_2$ es un certificado de 2-conexidad por vértices de una gráfica G simple.

Ejemplo 5. *Para ver un ejemplo de la ejecución de este algoritmo, véase el apéndice B.*

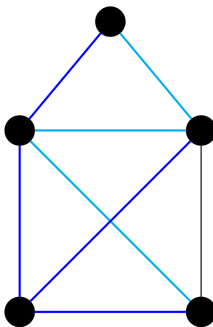


Figura 4.5: Resultado de la ejecución de `Scan-first Search` con F_1 en azul y F_2 en cian.

Proposición 4.4. *Sean F_1 un bosque maximal en G y F_2 un bosque maximal en $G - E(F_1)$ generado por `Scan-first Search`, entonces $F_1 \cup F_2$ es un certificado de 2-conexidad por vértices de G.*

Demostración. Sean F_1 y F_2 como en la hipótesis, así F_1 es un bosque maximal en G, por lo tanto, tenemos que:

$$\kappa_{F_1}(x, y) \geq \min\{\kappa_G(x, y), 1\} \ \forall x, y \in V(G) \tag{4.4}$$

$\kappa_{F_1}(x, y)$ denota la conexidad por aristas de la subgráfica F_1. Supongamos que $F_1 \cup F_2$ no es un certificado de 2-conexidad por vértices de G. Así,

$$\kappa_{F_1 \cup F_2}(x, y) < \text{mín}\{\kappa_G(x, y), 2\} \tag{4.5}$$

Notemos que $F_1 \cup F_2$ es una subgráfica de G que surge de agregar aristas a F_1, por lo que,

$$\kappa_{F_1}(x, y) \leq \kappa_{F_1 \cup F_2}(x, y) \tag{4.6}$$

Así, por las desigualdades 4.4, 4.5 y 4.6, tenemos que:

$$1 = \kappa_{F_1}(x, y) = \kappa_{F_1 \cup F_2}(x, y) < 2 \leq \kappa_G(x, y).$$

Así, $F_1 \cup F_2$ tiene un vértice de corte w tal que $F_1 \cup F_2 - \{w\}$ tiene dos componentes conexas X y Y tales que $x \in X$ y $y \in Y$. Notemos que $w \in V(F_1)$ ya que $\kappa_{F_1}(x, y) = 1$. Es decir, existe una (x, y)-trayectoria P en $F_1 \cup F_2$ tal que $w \in V(P)$.

Sin embargo, existe una (x, y)-trayectoria P' ajena por vértices en $G - E(F_1 \cup F_2)$ que conecta a x y y pues $\kappa_G(x, y) \geq 2$ por la desigualdad 4.6. Así, existe un vértice $w' \in P'$, claramente $w' \neq w$, tal que $w' \in V(F_1)$ pero $w' \notin V(F_2)$. Esto contradice la maximalidad de F_2. □

Una característica interesante de los árboles generados por `Scan-first Search` es que son una generalización de los árboles generados por `BFS`. Ya que todo árbol generado por `BFS` es un árbol `Scan-first Search`, sin embargo, existen árboles `Scan-first Search` que no son `BFS` [9].

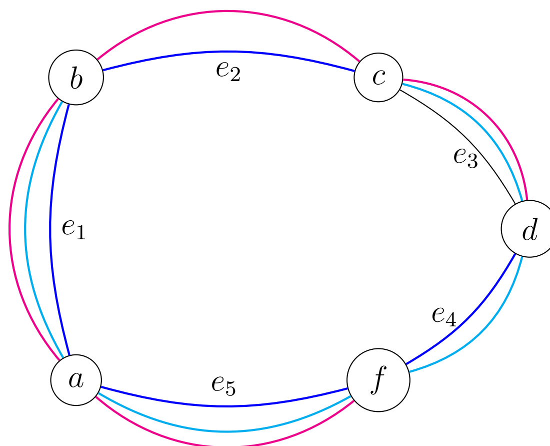


Figura 4.6: Sea $C = \{a, b, c, d, f\}$ un ciclo. Tenemos que los árboles generadores $C - \{e_2\}$ en cian, $C \setminus \{e_3\}$ en azul y $C - \{e_4\}$ en magenta son árboles `Scan-first search` con raíz en a. Sin embargo, el árbol generador $C - \{e_3\}$ es el único árbol `BFS` con raíz en a.

4.3. Algoritmo distribuido: `Distributed-NI`

Revisemos ahora el algoritmo publicado por S. Even, G. Itkis y S. Rajsbaum en [21]. A este lo llamaremos `Distributed_NI` (véase figura 5.3). Este algoritmo implementa `NI-Search` de manera distribuida, por eso el nombre que recibe. Este algoritmo recibe como entrada una gráfica G y nos regresa un certificado de 2-conexidad, de la misma manera que `NI-Search` lo hace.

Antes de explicar los detalles de la ejecución de este algoritmo para el caso de 2-conexidad, revisaremos que es un sistema distribuido. Definimos un **sistema distribuido** como una colección de dispositivos computacionales individuales (procesadores) que pueden comunicarse entre sí. Estos sistemas *"son omnipresentes hoy en día en los negocios, el mundo académico, el gobierno y el hogar"* [3]. Uno de los ejemplos más importantes de un sistema distribuido es Internet.

En un sistema distribuido, los procesos se comunican mediante el envío de mensajes por medio de canales de comunicación. Cada uno de estos canales provee una conexión bidireccional entre dos procesos específicos. El patrón de conexiones entre estos canales describe la *topología* del sistema. La *topología* es representada por medio de una gráfica G no dirigida en la que cada vértice representa un procesador, por otro lado, una arista conecta a dos vértices dentro de la gráfica si y solo si existe un canal de comunicación entre ellos dos. Así, un **algoritmo distribuido** es un algoritmo que corre de manera local en cada procesador dentro del sistema. Cada algoritmo local permite que el procesador realice cómputos locales y que envíe y reciba mensajes de sus vecinos en la red.[7]

`Distributed-NI` se ejecuta de forma restringida[8]. Es decir, tenemos un solo centro de actividad, al que llamaremos *servidor*, que se mueve de un vértice a otro vértice alrededor de la red. Cuando el *servidor* se encuentra en un vertice v, los mensajes son enviados exclusivamente entre v y sus vecinos. Los mensajes usados por el algoritmo son: `VISIT`, `RETURN`, `RANK_EDGE`, `EDGE_RANKED(i)` con $1 \leq i \leq |E(G)|$. Cada vértice u tiene las siguientes variables:

- $label(u)$, que funciona de la misma manera que en `Busqueda-NI`.
- $first_time$, una variable booleana inicializada en `True`.
- *unvisited*, que es una lista que inicialmente incluye todas las aristas incidentes al vértice u.

Ejemplo 6. *Para ver un ejemplo de la ejecución de* `Distributed_NI`*, consúltese el apéndice C.*

Como veremos en la siguiente sección, `Distributed-NI` es una implementación distribuida de `NI-Search`. De lo anterior podemos concluir que `Distributed_NI` genera un certificado de 2-conexidad por aristas en la gráfica G por la proposición 4.1; también genera un certificado de 2-conexidad por vértices si G es S-simple; así como, un certificado de k-conexidad S-mixta local de G por el teorema 4.3. Este último resultado lo revisaremos igualmente en el próximo capítulo.

[7] Para profundizar más en las definiciones que aquí damos, recomendamos consultar el texto de H. Attiya y J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics.* Estados Unidos, 2004

[8] Si no fuera restringido, el algoritmo correría simultáneamente en más de un vértice de la gráfica.

4.4. `Sparsify-2`

Para finalizar este capítulo, revisaremos el algoritmo `Sparsify-2` de A. Elmasry que aparece en [15]. Este algoritmo genera certificados de 2-conexidad y está basado en el clásico algoritmo `DFS` de búsqueda y exploración de gráficas en su versión recursiva. La versión de `DFS` utilizada por Elmasry corresponde con la versión publicada por Tarjan en [45] en 1972, este algoritmo toma como entrada una gráfica G y nos regresa una palmera P.

Nota. Recordemos que en el capítulo 2 (teorema 2.10) vimos que podemos dar una orientación de una gráfica G con `DFS`, de tal suerte que esta sea fuertemente conexa si y solo si G es 2-conexa por aristas.

Definición 16. *Una* ***palmera*** *P es una subgráfica dirigida generada por la ejecución de `DFS` en una gráfica G. Esta subgráfica tiene dos tipos de flechas:*

- *Flecha de árbol: generadas entre u y $parent[u]$ cuando u es visitado por primera vez en `DFS` y*
- *Back edge: Se llama así porque toda arista que no pertenece al árbol conecta a un vértice de regreso, «back», con uno de sus ancestros.*

`Sparsify-2` recibe como entrada una gráfica G y un vértice donde comienza a correr el algoritmo. Al terminar, nos regresa una subgráfica P', la cual es 2-conexa por vértices y aristas. Esta P' es una subgráfica de la **palmera** generada por `DFS`, la 2-conexidad se preserva asociando a cada vértice de P' a lo más una *back edge*.

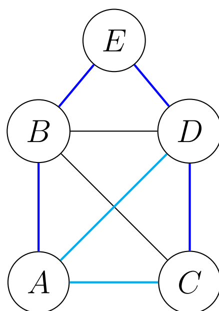


Figura 4.7: Resultado de la ejecución de `Sparsify-2` comenzando en el vértice A, en azul tenemos las flechas de árbol y en cian las *back edges*

Ejemplo 7. *Para ver un ejemplo de la ejecución de este algoritmo, véase el apéndice D.*

Nota. Definimos las siguientes variables del algoritmo `Sparsify-2`:

- $num[u]$: es un entero que representa el orden en el que el vértice u fue visitado por primera vez por `DFS`.
- $parent[u]$: es el padre del vértice u en el árbol P.

- $low[u]$: es un entero que guarda el valor $num[w]$, donde w es el vértice que tiene el menor número entre todos los vértices v tales que $u \hookrightarrow v$[9] en P.

Lema 4.5 (A. Elmasry [15]). *Sea $(x, y) \in E(G)$ tales que $num(y) < num(x)$ pero no existen la arista de árbol (y, x) ni la back edge (x, y) en $E(P')$. Entonces existe $w \in V(G)$ tal que $x \hookrightarrow w \in E(P')$ y $num(w) < num(y)$. (Véase figura 4.8)*

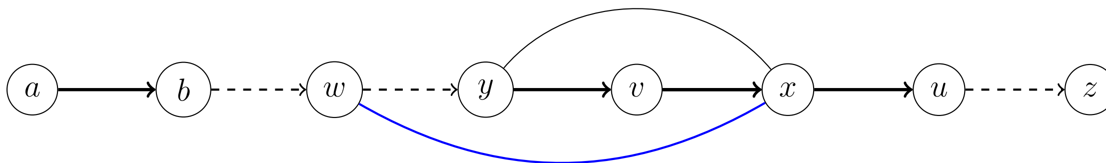


Figura 4.8: Recorremos el árbol de a a z. Así, $num(a) = 1$ y $num(z) = |V(G)|$. Las flechas de árbol se representan con flechas negritas y *back edge* de x a w en azul.

Proposición 4.6 (A. Elmasry [15]). *P' es 2-conexa si y solo si G es 2-conexa*

Demostración. Es claro que G es 2-conexa si P' es 2-conexa, ya que P' es una subgráfica de G.

Para probar la suficiencia, supongamos que G es 2-conexa pero P' no es 2-conexa y lleguemos a una contradicción. Como P' no es 2-conexa, entonces existe un vértice de corte $a \in V(P')$. Sean X y Y las dos componentes conexas en $P' - \{a\}$. Como G es 2-conexa, entonces existe $e = (x, y)$ tal que $e \in E(G)$ y $x \in X$ y $y \in Y$. Supongamos, sin pérdida de generalidad, que $num(y) < num(x)$. Es decir, y fue visitado antes que x. Así, tenemos que existe una (y, x)-trayectoria en P', así, $num(y) < num(a) < num(x)$.

Por el lema anterior, existe una $x \hookrightarrow w$ en P' tal que $num(w) < num(y)$. Como la *back edge* $x \hookrightarrow w$ está en P', tenemos que $w \in X$. Así, existe una (w, x)-trayectoria en P'. Luego, tenemos que $num(w) < num(x) < num(y)$, por lo tanto, existe una (w, y)-trayectoria en P'.

Como $num(w) < num(y) < num(a)$, una (w, y)-trayectoria no puede contener al vértice a. Así, aunque $w \in X$ y $y \in Y$, la (w, y)-trayectoria no contiene al vértice a. Así, la (w, y)-trayectoria no está en P'; lo que es una contradicción. Por lo tanto, P' es 2-conexa. □

Proposición 4.7 (A. Elmasry [15]). *P' tiene a lo más $2|V| - 3$ aristas*

Demostración. Existen a lo más $|V| - 1$ aristas de árbol en P'. Por otro lado, cada vértice en P' tiene a lo más una *back edge* que conecta a este vértice con uno de sus ancestros. Además, el primer vértice u y el segundo vértice u' visitados por `Sparsify-2` no tienen *back edges*. Ya que no existen vértices w tales que $num[w] < num[u]$ y $parent[u] \neq w$, o bien, tales que $num[w] < num[u']$ y $parent[u'] \neq w$. □

[9] $u \hookrightarrow v$ denota a una *back edge* de u a v

Algoritmo 1: Sparsify-2

```
    Datos: G = (V, E)
    Resultado: Subgráfica P' 2-conexa
 1  c = c + 1
 2  low[u] = num[u] = c
 3  para w ∈ lista de adyacencia de u hacer
 4  |   si num[w] == 0 entonces
 5  |   |   parent[w] = u
 6  |   |   Sparsify(w)
 7  |   |   agrega (u, w) como arista de árbol en P'
 8  |   fin
 9  |   en otro caso
10  |   |   si num[w] < num[u] && w ≠ parent[w] entonces
11  |   |   |   si num[w] < low[u] entonces
12  |   |   |   |   low[u] = num[w]
13  |   |   |   |   l = w
14  |   |   |   fin
15  |   |   fin
16  |   fin
17  |   si low[u] ≠ num[u] entonces
18  |   |   agrega u ↪ l como back edge en P'
19  |   fin
20  fin
```

Figura 4.9: Algoritmo `Sparsify-2`

Algoritmo 2: main()

```
    Datos: G = (V, E)
    Resultado: Subgráfica P' 2-conexa
 1  para i = 1 hasta n hacer
 2  |   num[i] = parent[i] = 0 // Inicializamos todos en 0
 3  fin
 4  c = 0
 5  SFS(s) // s es un vértice arbitrario de G
```

Figura 4.10: función `main()` que inicializa las variables y llama a `Sparsify-2`

Capítulo 5

Certificados de k-conexidad

> "If you like, an effective procedure is a set of rules telling you, moment by moment, what to do to achieve a particular end; it is an algorithm."
>
> Richard Feynman, *Lectures on Computation*

La siguiente tabla resume el contenido de este capítulo, en el cual expondremos cada uno de los algoritmos mencionados en la introducción, pero esta vez para cualquier $k \in \mathbb{N}$. Consideremos que $|E| = m$ y $|V| = n$.

	Tipo de conexidad	**# de aristas**	**Tiempo**
`NI-search` [41]	aristas	$m \leq kn - k(k+1)/2$	$O(m+n)$
	vértices y mixta	$m \leq k(n-1)$	$O(m+n)$
`Scan-first Search` [9]	vértices	$m \leq k(n-1)$	$O(k(m+n))$
`Distributed-NI` [21]	mixta	$m \leq kn - 1$	$4m$
`Sparsify-2` [15]	$k=2$: vértices y aristas	$m \leq 2n-3$	$O(m+n)$

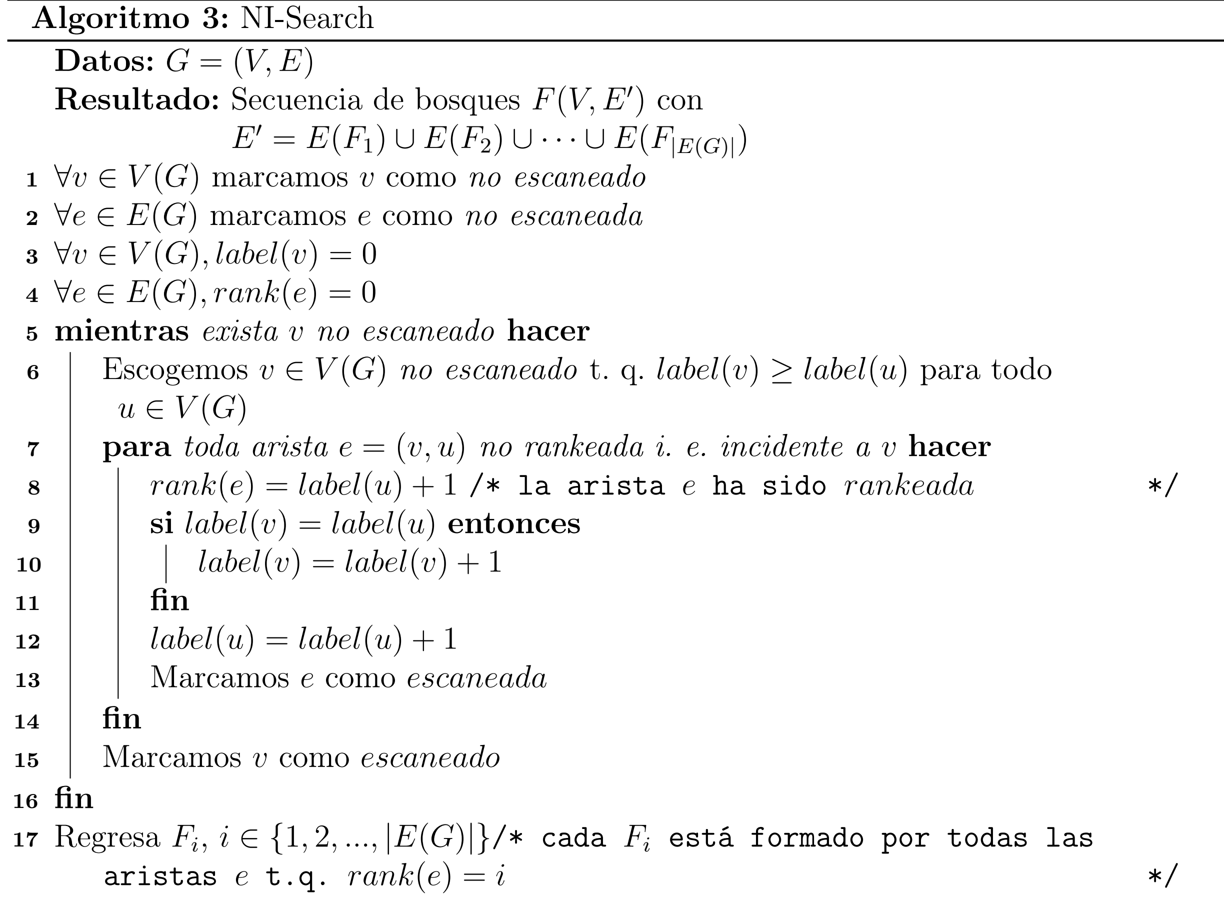
5.1. `NI-Search`

`NI-Search` (figura 5.1) recibe como entrada una gráfica G y nos regresa como salida una familia F de bosques maximales tales que:

$$F = \bigcup_{i=1}^{|E(G)|} F_i,$$

donde cada F_i es un bosque maximal en la subgráfica $G - \{E(F_1) \cup \cdots \cup E(F_{i-1})\}$, donde podría pasar que $E(F_i) = \varnothing$ para algunos $i \in k$. Como veremos más adelante, el lema 5.1 nos asegura que la gráfica con vértices $V(G)$ y aristas $E'(G) = E(F_1) \cup E(F_2) \cup \cdots \cup E(F_k)$ es k-conexa por aristas si $k \leq \lambda(G)$ y $|E'(G)| \leq k(|V(G)|-1)$, pues tenemos que:

$$|E(F_i)| \leq |V(G)| - 1 \text{ para toda } i \in k \tag{5.1}$$



Algoritmo 3: NI-Search

Datos: $G = (V, E)$

Resultado: Secuencia de bosques $F(V, E')$ con $E' = E(F_1) \cup E(F_2) \cup \cdots \cup E(F_{|E(G)|})$

1 $\forall v \in V(G)$ marcamos v como *no escaneado*

2 $\forall e \in E(G)$ marcamos e como *no escaneada*

3 $\forall v \in V(G), label(v) = 0$

4 $\forall e \in E(G), rank(e) = 0$

5 **mientras** *exista v no escaneado* **hacer**

6 Escogemos $v \in V(G)$ *no escaneado* t. q. $label(v) \geq label(u)$ para todo $u \in V(G)$

7 **para** *toda arista* $e = (v, u)$ *no rankeada i. e. incidente a v* **hacer**

8 $rank(e) = label(u) + 1$ `/* la arista` e `ha sido` *rankeada* `*/`

9 **si** $label(v) = label(u)$ **entonces**

10 $label(v) = label(v) + 1$

11 **fin**

12 $label(u) = label(u) + 1$

13 Marcamos e como *escaneada*

14 **fin**

15 Marcamos v como *escaneado*

16 **fin**

17 Regresa F_i, $i \in \{1, 2, ..., |E(G)|\}$`/* cada` F_i `está formado por todas las aristas` e `t.q.` $rank(e) = i$ `*/`

Figura 5.1: Algoritmo `NI-Search`

La igualdad en la ecuación 5.1 ocurre cuando F_i es un árbol generador de la gráfica G, lo cual siempre es el caso para F_1. Por otro lado, la desigualdad es estricta en el caso de que F_i sea un bosque en G.

Pasemos ahora a describir al algoritmo `NI-Search`. En las líneas 1-4 (véase la figura 5.1) inicializamos vértices y aristas. Los vértices tienen las siguientes variables: *escaneado*, la cual es una variable booleana inicializada en `False` y *label(v)* de tipo entero que guarda el valor de *label*. Las aristas tienen las variables *escaneado* la cual es una variable booleana inicializada en `False` y $rank(e)$ de tipo entero que guarda el valor de $rank$.

Elegimos un vértice v no escaneado, *i. e. escaneado* = `False`, tal que $label(v) \geq label(u)$ para todo $u \in V(G)$, línea 6. Notemos que la elección del primer vértices es arbitraria. Luego, para toda arista *no rankeada*, *i. e.* $rank(e) = 0$ asignamos el valor $rank(e) = label(u) + 1$, línea 8. Posteriormente si se cumple que $label(v) = label(u)$ hacemos $label(v) = label(v) + 1$, líneas 9-10.

Notemos que `NI-Search` puede funcionar perfectamente sin las líneas 9-10, ya que una vez que un vértice v es elegido en la línea 6, todas las aristas incidentes a v que no han sido *escaneadas* son *escaneadas* y v no volverá a ser elegido nuevamente. Sin embargo, tanto para la prueba como para su implementación es útil mantener estas líneas. Repetimos las líneas anteriores hasta que no existan vértices que no hayan sido *escaneados*, línea 5.

Los siguientes resultados aparecen como lema 2.1 y como teorema 2.1 en [41], respectivamente.

Lema 5.1 (H. Nagamochi y T. Ibaraki [41]). *Sean $G = (V, E)$, simple o múltiple, y $F_i = (V, E(F_i))$ un bosque generador maximal en $G - \{E(F_1) \cup E(F_2) \cup \cdots \cup E(F_{i-1})\}$, con $i = 1, 2, \ldots, |E(G)|$, donde posiblemente $E(F_i) = E(F_{i+1}) = \cdots = E(F_{|E(G)|}) = \varnothing$ para algún i. Luego, cada subgráfica generadora $G_i = (V, E(F_1) \cup \cdots \cup E(F_i))$ cumple que:*

$$\lambda_{G_i}(x, y) \geq \text{mín}\,\{\lambda_G(x, y), i\} \text{ para toda } x, y \in V(G). \tag{5.2}$$

Demostración. Sean G, F_i y $E(F_i)$ como en el enunciado. Haremos la demostración por inducción sobre i. Para $i = 1$, la ecuación 5.2 se cumple ya que el bosque F_1 es maximal, véase la proposición 4.1. Supongamos que existen $u, v \in V(G)$ e $i > 1$ tales que $\lambda_{G_i}(u, v) \geq \text{mín}\{\lambda_G(u, v), i\}$, pero $\lambda_{G_{i+1}}(u, v) < \text{mín}\{\lambda_G(u, v), i + 1\}$.

Como $\lambda_{G_i}(x, y) \geq \text{mín}\{\lambda_G(u, v), i\}$, esto significa que $\lambda_G(u, v) \geq i+1$ y que $\lambda_{G_i}(u, v) = \lambda_{G_{i+1}}(u, v) = i$. Así, G_{i+1} tiene un conjunto de corte mínimo por aristas $W \subseteq E(G)$ con $|W| = i$ tal que existen dos componentes conexas A y B en la subgráfica $G_{i+1} - W$. Donde tenemos que $u \in A$ y $v \in B$.

Como $\lambda_{G_i}(u, v) = i$, entonces $|W - E(F_{i+1})| \geq i$. Luego, $W \cap E(F_{i+1}) = \varnothing$. Sin embargo, existe una arista $e \in E(G) \setminus \{E(F_1) \cup E(F_2) \cup \cdots \cup E(F_{i+1})\}$ que conecta a un vértice en A con un vértice en B, ya que $\lambda_G(x, y) \geq i + 1$. Lo anterior contradice la maximalidad de la subgráfica $F_{i+1} = (V(G), E(F_{i+1}))$. □

Teorema 5.2 (H. Nagamochi y T. Ibaraki [41]). *Dada una gráfica $G = (V, E)$, una partición E_i, con $i = 1, 2, \ldots, |E(G)|$, de $E(G)$ que satisface el lema 5.1 es encontrado por `NI-Search` en un tiempo $O(|V(G)| + |E(G)|)$, donde $|E_i| \leq |E(G)| - i$ con $i = 1, 2, \ldots, |V(G)| - 1$. Si G es simple tenemos que $|E_i| = 0$, con $i = |V(G)|, \ldots, |E(G)|$, y si G es múltiple $|E_i| \leq |V(G)| - 1$, con $i = 1, \ldots, |E(G)|$.*

En la subsección 5.4 presentaremos una generalización del teorema anterior así como la demostración de la misma. De ambos resultados, podemos concluir que podemos encontrar un certificado de k-conexidad por aristas $G_k = (V, E' = E_1 \cup E_2 \cup \cdots \cup E_k)$ de una gráfica G k-conexa por aristas en un tiempo $O(|V(G)| + |E(G)|) = O(|E(G)|)$. Si G es simple tenemos que $|E'| \leq k|V(G)| - k(k + 1)/2$ y si G tiene aristas múltiples cuando $|E'|k(|V(G)| - 1)$. Notemos además que estas cotas son fuertes cuando $|E(G)| = |E'| = k(k + 1) - \frac{k(k+1)}{2} = \frac{k(k+1)}{2}$ si $G = K_{k+1}$, una gráfica completa con $k+1$ vértices y $|E(G)| = |E'| = k(|V(G)|-1)$ si G es un árbol k-múltiple.

Consideremos ahora el caso de conexidad por vértices. La subgráfica $G_k = (V(G), E(F_1) \cup E(F_2) \cup \cdots \cup E(F_k))$ obtenida de una gráfica simple G por `NI-Search` es k-conexa por vértices para toda $k \leq \kappa(G)$. En adelante, G es una gráfica simple, es decir, no admite aristas múltiples. Lo anterior fue probado por Nagamochi e Ibaraki en [41], aparece como el teorema 3.1, que enunciamos a continuación.

Teorema 5.3 (H. Nagamochi y T. Ibaraki [41]). *Sean $G = (V, E)$ y E_i, con $i = 1, 2, \ldots, |E(G)|$, obtenida por la ejecución de `NI-Search` sobre G. Luego, cada subgráfica generadora $G_i = (V, E(F_1) \cup E(F_2) \cup \cdots \cup E(F_i))$ cumple que:*

$$\kappa_{G_i}(x, y) \geq \text{mín}\,\{\kappa_G(x, y), i\} \text{ para todo } x, y \in V(G).$$

En la subsección 5.4 presentaremos una generalización del teorema anterior así como la demostración de la misma. Las propiedades de conexidad mixta las revisaremos más adelante cuando estudiemos el algoritmo distribuido.

5.2. Scan-First Search

Este algoritmo recibe como entrada una gráfica simple G, un vértice $v \in V(G)$, por el que comenzaremos la ejecución, y un $k \in \mathbb{N}$. `Scan-first Search` fue concebido originalmente como un algoritmo de exploración, pues comenzamos con todos los vértices sin *escanear* (explorar), línea 1 (véase figura 5.2), y vamos *escaneando* (explorando) iterativamente cada vértice de la gráfica G hasta que todos los vértices hayan sido *escaneados* (explorados), línea 6. *Escanear* un vértice significa primero marcarlo, y posteriormente marcar todos sus vecinos no marcados. Decimos que un vértice v está marcado si $label(v) > 0$. Por último, obtenemos como resultado final un árbol generador T de G en cada iteración del ciclo `para` de la línea 4. En la figura 5.2 mostramos una versión personal de manera secuencial de este algoritmo basada en [9].

Algoritmo 4: k-SFS

Datos: $G = (V, E)$
Resultado: Bosque generador F de G con $F = T_1 \cup T_2 \cup \cdots \cup T_K$ donde cada T_i es un árbol generador de G_i

1. $\forall v \in V(G)$ marcamos v como *no escaneado*
2. $\forall v \in V(G), label(v) = 0$
3. $T_0 \to \varnothing$
4. **para** $i = 0$ *hasta* k **hacer**
5. $\quad G' = G - E(T_i)$ `/* Removemos las aristas del árbol` T_i `*/`
6. $\quad$ **mientras** *exista* v *no escaneado en* G' **hacer**
7. $\quad\quad$ Escogemos $v \in V(G)$ *no escaneado* t. q. $label(v) \geq label(u)$ para todo $u \in V(G)$
8. $\quad\quad$ **para** *todo vecino* u *de* v *con* $label(u) = 0$ *i. e.* u *no ha sido explorado* **hacer**
9. $\quad\quad\quad$ Agrega (v, u) a T_i
10. $\quad\quad$ **fin**
11. $\quad\quad$ **si** $label(v) = label(u)$ **entonces**
12. $\quad\quad\quad$ $label(v) = label(v) + 1$
13. $\quad\quad$ **fin**
14. $\quad\quad$ $label(u) = label(u) + 1$
15. $\quad\quad$ Marcamos v como *escaneado*
16. $\quad$ **fin**
17. $\quad$ Regresa T_i
18. $\quad$ $i = i + 1$
19. **fin**

Figura 5.2: Algoritmo `Scan-first Search`

El árbol T comienza siendo vacío (línea 3). Luego para cada vértice $v \in V(G)$, cuando v es *escaneado*, todas las aristas entre v y sus vecinos no marcados son agregadas a T, líneas 8-9. Las aristas entre V y sus vecinos marcados no se agregan a T.

Para una gráfica G inconexa podemos aplicar `Scan-first Search` en cada una de sus componentes conexas lo que produce un árbol generador en cada componente cuya unión es un bosque generador de G. Para generar un certificado de k-conexidad, aplicamos `Scan-first Search` a la gráfica G y obtenemos T_1. Retiramos las aristas pertenecientes a T_1 y aplicamos nuevamente el algoritmo sobre la subgráfica $G - E(T_1)$ y obtenemos T_2. Repetimos este procedimiento hasta obtener el árbol T_k. T_k es un árbol `Scan-first Search` de la subgráfica $G' = G - \{E(T_1), E(T_2), \ldots, E(T_{k-1})\}$. $\cup_{i=1}^{k} T_i$ es un certificado de k-conexidad por vértices de G.

El siguiente enunciado, que aparece como teorema 2.4 en [9] y es el resultado principal de este mismo artículo. El cual *grosso modo* nos dice que para generar un certificado de k-conexidad por vértices de una gráfica G, basta con ejecutar k veces `Scan-first Search`.

Teorema 5.4 ("Main Certificate Theorem", J. Cheriyan, M. Y. Kao y R. Thurimella [9]). *Sean G una gráfica simple, $|V(G)| = n$ y $k \in \mathbb{N}$. Para $i = 1, 2, \ldots, k$, sea E_i el conjunto de aristas de un bosque `Scan-first Search` en la gráfica $G_i = G - \{E_1 \cup E_2 \cup \cdots \cup E_{i-1}\}$. Entonces $E_1 \cup E_2 \cup \cdots \cup E_k$ es un certificado de k-conexidad por vértices de G y este certificado tiene a lo más $k(n-1)$ aristas.*

En la subsección 5.4 presentaremos una generalización del teorema anterior así como la demostración de la misma. Revisemos ahora la complejidad de tiempo de este algoritmo. El siguiente enunciado aparece como teorema 2.8 en [9], la demostración del mismo es personal.

Teorema 5.5 (J. Cheriyan, M. Y. Kao y R. Thurimella [9]). *Sea G una gráfica simple, $|E(G)| = m$ y $|V(G)| = n$, un certificado de k-conexidad con a lo más $k(n-1)$ aristas se encuentran en tiempo $O(k(m+n))$ con `Scan-first Search`.*

Demostración. Sea G como en el enunciado. Veamos primero que para $k = 1$, `Scan-first Search` toma tiempo $O(m+n)$. Al inicio, los vértices comienzan no *escaneados* y $label(u) = 0$. Notemos que el algoritmo termina una vez que *escaneamos* cada uno de los vértices de G, línea 6. Esto ocurre una vez marcamos todos los vecinos de cada vértice y $label(u) > 0$ para todo vértice $u \in V(G)$; lo que toma tiempo $O(n)$. Para marcar estos vértices tenemos que revisar todas las aristas de la forma (u, v) donde u ya ha sido marcado y $label(v) = 0$, luego agregamos estas aristas a T, línea 8-10. De lo anterior concluimos, que para $k = 1$ `Scan-first Search` toma tiempo $O(n + m)$.

Para generar un certificado de k-conexidad tenemos que ejecutar `Scan-first Search` k veces. Así, tenemos que esto toma tiempo $O(k(m + n))$.

Por otro lado, como cada una de las k iteraciones el algoritmo nos regresa un árbol generador de G, o bien, un árbol generador de $G' \subset G$. Entonces, en cada iteración obtenemos un certificado con a lo más $n - 1$ aristas y como se repite k veces, obtenemos que el certificado tiene a lo más $k(n-1)$ aristas. □

Es necesario mencionar que existen versiones no secuenciales de `Scan-first Search` que arrojan como resultado un árbol generador de G, pero que se comportan de dife-

rente manera. Estas versiones también son tratadas en este mismo artículo [9] las cuales enunciamos a continuación:

- **Cómputo en paralelo.** `Scan-first Search` corre en tiempo $O(\log(n))$ utilizando $C(n, m)$ procesadores en un `CRCW PRAM`[1], donde $C(n, m)$ es el número de procesadores usados para computar un árbol generador en cada componente conexa en tiempo $O(log(n))$.
- **Cómputo distribuido.** `Scan-first Search` corre en tiempo $O(d \log^3(n))$ enviando $O(m + n \log^3(n))$ mensajes, donde d es el diámetro de la gráfica que recibe como entrada.

Y de la misma manera que en su versión secuencial, podemos generar certificados de k-conexidad por vértices corriendo k veces estas versiones sobre una gráfica [9].

5.3. `Distributed-NI`

Este algoritmo se ejecuta sobre una gráfica G que es representada por una red, donde cada vértice $v \in V(G)$ corresponde con un nodo de la red y cada arista $e \in E(G)$ corresponde con un enlace de comunicación. Como ya vimos en el capítulo 4, cada vértice v tiene las siguientes variables: $label(v)$, $first_time$ y $unvisited$ (Véase la figura 5.3.). Además, el algoritmo ocupa los siguientes mensajes: `VISIT`, `RETURN`, `RANK_EDGE` y `EDGE_RANKED(i)` con $i = 1, 2, \ldots, |E(G)|$. `Distributed-NI` (véase la figura 5.3.) comienza cuando se envía un mensaje `VISIT` a algún vértice a través de una **arista nula**[2] y termina su ejecución cuando esta misma arista nula recibe un mensaje `RETURN`. Un vértice v completa su trabajo una vez envía `RETURN` en una arista e tal que $rank(e) > 0$, o bien, en la arista nula si es el vértice por el que comenzó la ejecución. Una vez que todos los vértices terminan su trabajo, $rank(e) > 0$ para toda $e \in E(G)$.

Como ya lo habíamos mencionado, `Distributed-NI` es de forma restringida, ya que tenemos un solo centro de actividad. A este le llamamos *servidor* y viaja de vértice en vértice alrededor de la red. Cuando el *servidor* se encuentra en un vértice v, los mensajes son enviados exclusivamente entre v y sus vecinos. El *servidor* viaja por medio de los mensajes `VISIT` y `RETURN`. Las aristas no *rankeadas* incidentes a un vértice v son *rankeadas* una vez que el *servidor* llega a v por primera vez. Decimos que un vértice v ha sido visitado si $first_time =$ `False`. Así, un vértice visitado v ha recibido al menos un mensaje `VISIT`, y una vez que recibe su primer mensaje `VISIT`, el *servidor* pasa a otro vértice hasta que v no tenga aristas incidentes sin *rankear*.

Veamos ahora por qué es que `Distributed-NI` genera certificados de k-conexidad. El siguiente enunciado aparece como teorema 4 en [21], la prueba del mismo está basada en los resultados expuestos en este mismo artículo.

[1] CRCW PRAM: Concurrent Read Concurrent Write Parallel Random Access Machine. Para una definición y ejemplo de este modelo de cómputo paralelo recomendamos la entrada de Joseph F. Jaja sobre PRAM (Parallel Random Access Machines) en *Encyclopedia of Parallel Computing* [31].

[2] Una **arista nula** es una arista vacía

Algoritmo 5: `Distributed-NI`

Datos: $G = (V, E)$

Resultado: Bosque generador $F(V, E')$ con $E' = E(F_1) \cup E(F_2) \cup \cdots \cup E(F_{|E(G)|})$

1 $\forall v \in V(G), label(v) = 0$
2 $\forall e \in E(G), rank(e) = 0$
3 $U = E(G)$ `/* Conjunto de aristas no visitadas */`
4 **para** *mensaje* ***RANK_ARISTA*** *que llega a v por la arista e* **hacer**
5     $label(v), rank(e) = label(v) + 1$
6   envía **ARISTA_RANKEADA(rank(e))** por la arista e
7 **fin**
8 **para** *mensaje* ***VISITA*** *que llega a v por la arista e* **hacer**
9   Retira e del conjunto U
10   **si** $first_time = true$ **entonces**
11         $first_time = false$
12   **para** *todas las aristas* e', $e' \neq nil$, *t.q.* $rank(e) = 0$ **hacer**
13   Envía **RANK_ARISTA** por e'
14   Espera a que llegue el mensaje **ARISTA_RANKEADA(rank(i))** por e'
15             $rank(e') = i$
16   **fin**
17   **para** *cada* $e' \in U$ *con* $rank(e') \geq rank(e)$ *en orden decreciente de* $rank(e')$ **hacer**
18   Retira e' de U
19   Envía el mensaje **VISIT** por e'
20   Espera a que llegue el mensaje **RETURN** por e'
21   **fin**
22   Envía el mensaje **RETURN** por e
23   **fin**
24 **fin**
25 Regresa F_i, $i \in \{1, 2, ..., |E(G)|\}$

Figura 5.3: Algoritmo `Distributed-NI`

Teorema 5.6 (S. Even, G. Itkis y S. Rajsbaum [21])**.** *El algoritmo `Distributed-NI` implementa `NI-Search`, corre en tiempo $4|E(G)|$ y envía $4|E(G)|$ mensajes.*

Demostración. Para demostrar que `Distributed-NI` en efecto es una implementación distribuida de `NI-Search` tenemos que verificar que todas las aristas de la gráfica G, que recibe como entrada, son *rankeadas* y que este *ranking* genera bosques maximales F_i tales que cada F_i, $i = 1, ..., |E(G)|$, es un bosque maximal en la subgráfica $G - \{E(F_1) \cup ... \cup E(F_{i-1})\}$.

Notemos que cada $e \in E(G)$ solo puede ser *rankeada* una vez, líneas 5 y 15. Cuando el mensaje `VISIT` llega a un vértice, línea 8, o es enviado, línea 19, por una arista e, e es retirada de la lista `UNVISITED`. Así, `VISIT` no se vuelve a enviar por la misma arista. Por lo tanto, `RETURN` solo puede ser enviado por cada arista e a lo más una vez.

Veamos ahora que F_i es una subgráfica acíclica de G. Supongamos que no y lleguemos a una contradicción. Así, supongamos que existe un ciclo C en G tal que sus aristas tienen $rank(e) = i$. Sea $e = (a, b)$ la última arista de C en ser *rankeada*. Así, el *servidor* está en a o en b cuando esto sucede. Justo antes de *rankear* a e, como a y b ya tienen un arista incidente e' con $rank(e') = i$, tenemos que $label(a) = label(b) = i$. Sin embargo, por la línea 5, e debería tener un $rank(e) > i$, lo que es una contradicción.

Revisemos ahora que una vez que un vértice v envía `VISIT` en alguna arista e', la próxima vez que el *servidor*[3] vuelve a estar en v es por un mensaje `RETURN` que llega por e', en la dirección opuesta tal como se indica en la línea 20. Para esto, daremos una orientación de las aristas en dirección en el que el mensaje `VISIT` las ha atravesado. Como ya lo habíamos visto antes, `VISIT` solo puede pasar una vez por la misma arista. Así, que no existe ambigüedad en la asignación de la dirección. Diremos que una arista e está **abierta** si el mensaje `RETURN` no la ha atravesado aún. Denotemos con $\vec{G}$ la digráfica que contiene a todas las aristas abiertas.

Lema 5.7 (S.Even, G. Itkis y S. Rajsbaum [21])**.** *$\vec{G}$ es una trayectoria simple y dirigida.*

Demostración. Veamos primero que $\vec{G}$ es acíclica. Durante la ejecución de `Distributed-NI`, $\vec{G}$ va cambiando. Supongamos que la digráfica $\vec{G}$ tiene un ciclo y lleguemos a una contradicción. Sea $\vec{C}$ el primer ciclo que aparece en $\vec{G}$.

$$\vec{C} = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \xrightarrow{e_3} \cdots \xrightarrow{e_{i-1}} v_{i-1} \xrightarrow{e_i} v_0$$

Por la línea 17, tenemos que $rank(e_1) \leq rank(e_2) \leq \cdots \leq rank(e_i)$. Sin embargo, como F_i es acíclica, entonces el valor de $rank(e)$ no puede ser el mismo para todas las aristas de $\vec{C}$. Así, $rank(e_1) < rank(e_i)$. Además, cuando e_1 es elegido por el vértice v_0 como en la línea 17, e_i ya ha sido *rankeada*. Por lo tanto tenía que elegirse e_i en vez de e_1. Así, $\vec{G}$ es acíclica.

Ahora consideremos la posición del *servidor*. Como el *servidor* se mueve únicamente con los mensajes `VISIT` y `RETURN`, la gráfica G sobre la que corre el algoritmo es conexa. `Distributed-NI` nunca envía un segundo mensaje `VISIT` por el mismo vértice, antes que `RETURN` arrive de la arista anterior. Así, el exgrado de cada vértice de $\vec{G}$ es lo más uno. Por otro lado, el vértice v_0 es el único vértice con exgrado uno e ingrado cero, ya que es sobre

[3]Recordemos que solo existe un único centro de actividad, llamado *servidor*, que se mueve de un vértice a otro por medio de la aristas que los conectan.

este vértice donde comenzamos la ejecución de `Distributed-NI`. Así, $\vec{G}$ es una trayectoria simple y dirigida. □

Sea T la componente de F_1 que contiene al vértice raíz r, el cual es el vértice con la arista nula por la que comienza la ejecución del algoritmo, línea 8. Como ya lo vimos con anterioridad, T es acíclica. Veamos que T es un árbol generador de G.

Veamos primero que todo vértice en T es visitado. Como el mensaje `RETURN` es enviado desde r en la arista nula, todo vecino de r en T ha recibido un mensaje `VISIT` y por el lema 5.7, ha envíado el mensaje `RETURN`. Así, por inducción sobre la distancia desde r, todo vértice de T ha recibido el mensaje `VISIT` y ha envíado de vuelta `RETURN`. Como la gráfica G es conexa y finita, sabemos que la distancia entre cualquier par de vértices en G, así como de cualquier subgráfica de G, es finita.

Lema 5.8 (S. Even, G. Itkis y S. Rajsbaum [21]). *T es una subgráfica generadora de G.*

Demostración. Sea $S = V(T)$, T es la componente de F_1 que contiene al vértice raíz r. Si $S \neq V(G)$, consideremos el conjunto de aristas $E(S, \bar{S})$, tienen un extremo en S y el otro en el complemento de S denotado por $\bar{S}$. Este conjunto no es vacío, pues G es conexa. El *servidor* comienza en $r \in S$ y la única manera en que pueda llegar a algún vértice en $\bar{S}$ cuando pase por alguna arista de $E(S, \bar{S})$. Así, cuando el *servidor* visita por primera vez un vértice con una arista $e \in E(S, \bar{S})$, ninguno de los vértices en $\bar{S}$ ha sido etiquetado, es decir $label(v) > 0$, y ninguna de las aristas de $E(S, \bar{S})$ ha sido *rankeada*. Así, por las líneas 4 y 12 del algoritmo 5.3, tenemos que $rank(e) = 1$, lo que es una contradicción. □

Hasta el momento hemos visto que una vez finaliza la ejecución de `NI-Search`, todo vértice de G ha sido visitado por el *servidor*. Además, cada arista ha sido *rankeada*, es decir para todo $e \in E(G)$, tenemos que $rank(e) > 0$. Probemos ahora que cuando un vértice v ha sido visitado por primera vez, se tiene que $label(v) > label(u)$, donde $u \in U$, donde la lista U guarda a los vértices que no han sido visitados.

Lema 5.9 (S. Even, G. Itkis y S. Rajsbaum [21]). *Supongamos que en algun punto de la ejecución de `Distributed-NI`, v va a enviar el mensaje `VISIT` en una arista e tal que $rank(e) = i$. Entonces no existe una arista abierta e' tal que $rank(e') > i$.*

Demostración. Por el lema 5.7, v debe ser el último vértice en la trayectoria dirigida $\vec{G}$. Si e está abierta, entonces pertenece a una trayectoria dirigida, luego $rank(e) \leq i$. □

Lema 5.10 (S. Even, G. Itkis y S. Rajsbaum [21]). *Supongamos que en algún punto de la ejecución de `Distributed-NI`, v ha enviado `RANK_EDGE` por la arista $e' = (v, w)$, pero v no ha enviado `VISIT` por e'. Entonces existe un arista abierta e'' tal que $rank(e'') \geq rank(e')$.*

Demostración. Consideremos el instante en el que v envía `RANK_EDGE` a través de la arista e' y sea $rank(e') = \rho$. Esto sucede en la instrucción `si` de la línea 10, después de que v recibió `VISIT` por primera vez, digamos que fue por la arista e. Mientras ejecuta el ciclo `para` de la línea 12 tenemos dos casos:

1. Si $\rho \geq rank(e)$, la línea 17 implica que v ha enviado `VISIT` por una arista e'' así, $rank(e'') \geq rank(e')$, pero no ha recibido aun el mensaje `RETURN`. Es decir, e'' está abierta.

2. En otro caso, $\rho < rank(e)$. Cuando v recibe el mensaje `VISIT`, $label(v) \geq rank(e)$. Así, v ya tiene un arista incidente $e_1 = (v, v_1)$ tal que $rank(e_1) = \rho$. Como recibió `VISIT` por primera vez a través de e, ningún mensaje `VISIT` ha llegado por la arista e_1; aunque v_1 ya envió `RANK_EDGE` por e_1. Si repetimos el argumento anterior encontramos una trayectoria $P = (v, e_1, v_1, e_2, \ldots, v_j, \ldots)$, tal que v_j ha enviado `RANK_EDGE` por la arista e_j, pero aun no ha recibido el mensaje `VISIT` por esta. El $rank$ de todas las aristas de P es ρ. Como la gráfica G es finita y como F_ρ es acíclica, P tiene que terminar en una arista que satisfaga el primer caso[4].

□

Hasta el momento hemos demostrado que todos los vértices de G son visitados y una vez que un vértice es visitado por primera vez, sus aristas no *rankeadas*(*i. e.* $rank(e) = 0$) son *rankeadas* como lo hace `NI-Search`. Para demostrar que `Distributed-NI` implementa `NI-Search` falta probar que cuando un vértice v es visitado por primera vez, $label(v)$ es la mayor de entre todos los vértices en la lista U de no visitados. Probemos esto por contradicción.

Supongamos que v va a enviar `VISIT` hacia un vértice no visitado u, `first_time(u) = True`, tal que $label(u) = i$. Pero existe un vértcie no visitado w tal que $label(w) = i' > i$, $i, i' \in \{1, 2, \ldots, |E(G)|\}$. Entonces existe una arista $e' = (w', w)$, tal que $rank(e') = i'$ y w' es un vértice ya visitado, es decir $first_time =$ `False`. El vértice w' y la arista e' satisfacen las hipótesis del lema 5.10. Así, existe una arista abierta e'' tal que $rank(e'') \geq rank(e')$. Esto contradice el lema 5.9. Así, `Distributed_NI` es en efecto una implementación distribuida de `NI-Search`.

Por último, notemos que cada arista no nula recibe y envía los cuatro mensajes: `VISIT`, `RANK_EDGE`, `EDGE_RANKED(i)` y `RETURN`. Todos estos mensajes son de tamaño constante a excepción de `EDGE_RANKED(i)`, que tiene $\lceil \log(i) \rceil$ bits, $i < |E(G)|$. Así el número total de mensajes enviados por el algoritmo es a lo más $4m$, $m = |E(G)|$. Y como en cada paso se envía un mensaje, entonces la complejidad de tiempo del algoritmo es a lo mas $4m$, $O(m)$.

□

Así, por el teorema anterior, `Distributed-NI` hereda las propiedades de conexidad local y global, así como por aristas y por vértices de `NI-Search`; las cuales ya revisamos con anterioridad.

5.4. Certificados para conexidad mixta

En la subsección 4.1.1 del capítulo anterior revisamos algunas definiciones referentes al caso de conexidad mixta. Recordemos que $\lambda_S(u, v; G)$ denota el máximo numero de (u, v)-trayectorias S-independientes en la gráfica G, véase la definición 14. Decimos que $u, v \in V(G)$ están k-conexos S-mixtos si $\lambda_S(u, v; G) \geq k$. Veamos ahora el caso de conexidad mixta global.

[4]No podemos extender la trayectoria P *ad infinitum*.

Definición 17 (A. Frank, T. Ibaraki y H. Nagamochi [23]). *La* ***conexidad global*** S-***mixta*** *de una gráfica* G *está definida por:*

$$\lambda_S(G) = \min_{u,v \in V(G)} \lambda_S(u, v; G).$$

G *es* S*-mixta* k*-conexa si* $\lambda_S(G) \geq k$.

Definición 18 (O. Fülöp [24]). *Un* ***certificado de*** k***-conexidad*** S***-mixta local*** *de* G *es una subgráfica* $G' = (V, E')$, $E' \subseteq E(G)$, *tal que* G' *tiene* $O(kn)$ *aristas y para cualquier par de vértices* $u, v \in V(G)$ *se cumple que:*

$$\lambda_S(u, v; G') \geq \min\{\lambda_S(u, v; G), k\}.$$

Similarmente, G' *es un* ***certificado de*** k***-conexidad*** S***-mixta global*** *de* G *si* G' *tiene* $O(kn)$ *aristas y cumple que:*

$$\lambda_S(G') \geq \min\{\lambda_S(G), k\}.$$

También en el capítulo anterior revisamos una versión del omnipresente teorema de Menger para este caso especial de conexidad (Véase teorema 4.2). Este nos indica que $\lambda_S(u, v; G)$ es igual a la cardinalidad del corte mínimo S-mixto que separa a u de v. A continuación presentamos un procedimiento de búsqueda no determinista que produce un bosque maximal F de una gráfica G. Notemos que todos los algoritmos que hemos revisado hasta ahora son deterministas.

Los siguientes resultados y definiciones fueron tomadas de [21]. Para producir este bosque maximal F, comenzamos con $F = \varnothing$. Durante el procedimiento cada vértice es visitado al menos una vez. Las aristas que se agregan a F son incidentes al vértice visitado. Los vértices de G son visitados de la siguiente manera:

- El primer vértice visitado es elegido de manera arbitraria.
- Una vez que termina la visita de un vértice, el siguiente vértice a ser visitado puede ser elegido de entre los vértices pertenecientes al bosque F, o bien, puede ser algún vértice de alguna componente de G que aún no tiene vértices en F.

Las aristas son añadidas a F, una a la vez, de la siguiente manera:

- Durante la primera visita de un vértice $v \in S$, para todo vecino x de v, $x \notin V(F)$, agrega la arista $(v, x) \in E(G)$ al bosque F. La gráfica G es S-simple lo que implica que existe una única arista que conecta a v con x en este caso.
- Cuando visitamos $v \notin S$, si $(v, x) \in E(G)$ y si $x \notin V(F)$, podemos agregar esta arista (v, x). Si (v, x) es añadida a F entonces, sus aristas paralelas, si es que las hay, no pueden ser jamás agregadas a F.

Así, ninguna arista incidente a $v \in S$ se agrega después de la primera visita. Si $v \notin S$, las aristas incidentes a este vértice v pueden ser agregadas durante varias visitas. Si bien los autores no especifican un mecanismo para la terminación de este procedimiento, basta ver

que una vez que F contenga a todos los vértices de G, podemos afirmar que F es ya un bosque maximal. Los bosques generados por este procedimiento son llamados S-**greedy**.

Definamos ahora a los bosques *greedy* sin referirnos a algoritmo alguno de la siguiente manera: Sea F un bosque maximal de G y sea $t : V(G) \to \{1, 2, \ldots, |V(G)|\}$ una numeración uno a uno de los vértices. Esta numeración t induce una orientación de las aristas:

$$\vec{F}(t) = \{u \to v \colon (u, v) \in F \ \&\ t(u) < t(v)\}.^5$$

Si $\vec{T}$ es un árbol con raíz r, dirigido de la raíz hacia las hojas, entonces, para cada $v \neq r$, $padre(v)$ es el único vértice u tal que $u \to v \in E(\vec{T})$. Además, tenemos que $padre(r) = r$ por definición.

Definición 19 (S. Even, G. Itkis y S. Rajsbaum [21]). *Un bosque maximal F de una gráfica G es S-**greedy** si existe una numeración t de los vértices tal que:*

- *Para todo árbol (maximal) T de F, $\vec{T}(t)$ es un árbol enraizado.*
- *Si $(w, v) \in E(G)$ y $w \in S$ entonces $t(padre(v)) \leq t(w)$.*

Es justo en este mismo artículo [21] donde se presenta una generalización de los resultados principales de los artículos de H. Nagamochi y T. Ibaraki [41] y de J. Cheriyan, M. Y. Kao y R. Thurimella [9]. Los resultados de Nagamochi e Ibaraki aparecen como teorema 5.2 (conexidad por aristas) y teorema 5.3, mientras que el resultado de Cheriyan, Kao y Thurimella aparecen como teorema 5.4. Esta generalización está basada en los bosques S-greedy. Veamos primero los siguientes lemas que tomamos de [24], así como sus demostraciones.

Lema 5.11 (O. Fülöp [24]). *Si u y v son dos vértices de la misma componente de bosque F_k entonces, $\lambda_S(u, v; G_k) \geq k$.*[6]

Demostración. Haremos la demostración por inducción sobre k. Para $k = 1$, se cumple ya que F_1 es un bosque maximal de G. Sean $k \geq 2$ y $C = (Z, E_G(A, B))$ (véase la definición 15) un corte S-mixto tal que $u \in A$ y $v \in B$. El corte C separa a los vértices u y v si uno de ellos pertenece a la componente A y el otro a la componente B. Para la demostración, ocuparemos la versión del teorema de Menger que presentamos en el teorema 4.2 del capítulo anterior, el cual nos dice que $\lambda_S(u, v; G)$, la cantidad de (u, v)-trayectorias S-independientes, es igual a la cardinalidad de un conjunto mínimo S-mixto de corte que separa a u de v.

Como cada F_i es maximal, tenemos que u y v viven en la misma componente conexa de F_i, $i \leq k$. Consideremos las subgráficas $G' = G - F_1$ y $J = F_2 \cup \cdots \cup F_k$. Supongamos que el enunciado es cierto para $n < k$. Luego, tenemos los siguientes dos casos:

- Si F_1 contiene alguna arista de $E_k(A, B)$ entonces, aplicamos la hipótesis de inducción para G' y los bosques $F_1, F_2, \ldots, F_k$. Tenemos:

$$|Z| + |E_{G_k}(A, B)| \geq |Z| + |E_J(A, B)| + 1 \geq (k - 1) + 1 = k,$$

[5] $u \to v$ denota a la flecha (u, v).

[6] Recordemos que G_k es el certificado tal que $G_k = (V(G), E_1 \cup E_2 \cup \cdots \cup E_k)$.

así que $\lambda_S(u, v; G_k) \geq k$. $E_J(A, B)$ denota las aristas entre las partes A y B en la subgráfica J y $J = G_k - F_1$.

- Supongamos ahora que F_1 no contiene una arista de $E_{G_k}(A, B)$. Sea H_1 la componente que contiene a u y v en F_1. Sin pérdida de generalidad, supongamos que el primer vértice visitado r de F_1 no pertenece a B. Sea z_1 el primer vértice visitado de $Z \cap V(H_1)$ durante la construcción del bosque F_1. Como z_1 pertenece al conjunto S, pues $Z \subseteq S$, en su primera visita todos las aristas que lo conectan a vértices no visitados se agregan a F_1. Luego, en J no hay aristas entre z_1 y B, es decir, $E_J(A, B) = E_J(A', B)$, con $A' = A \cup \{z_1\}$. Si $Z' = Z \setminus \{z_1\}$ la hipótesis de inducción aplicada para G' y los bosques $F_2, \ldots, F_k$ nos da como resultado:

  $$|Z| + |E_{G_k}(A, B)| = |Z'| + 1 + |E_J(A, B)| = |Z'| + 1 + |E_J(A', B)| \geq (k - 1) + 1 = k,$$

  así que $\lambda_S(u, v; G_k) \geq k$.

□

Lema 5.12 (O. Fülöp [24]). *Si $F_1 \cup F_2 \cup \cdots \cup F_k$ no contiene alguna arista de un corte S-mixto $(Z, E(A, B))$, entonces la cardinalidad de un corte S-mixto $(Z, E_k(A, B))$ es al menos k.*

Demostración. Sea $e = (u, v) \in E_G(A, B) \setminus \{E(F_1), E(F_2), \ldots, E(F_k)\}$. F_k es maximal por lo que u y v viven en la misma componente de F_k. Así que ocupando el lema anterior y el teorema 4.2, tenemos que la cardinalidad de un conjunto de corte S-mixto $(Z, E_k(A, B))$ es al menos k. □

El siguiente enunciado es la generalización de los resultados sobre certificados que anunciamos con anterioridad. La demostración es obra de O. Fülöp [24].

Teorema 5.13 ("Mixed Connectivity Certificate Theorem ´´, S. Even, G. Itkis y S. Rajsbaum [21]). *Sea G una gráfica de orden n. Sea $k \in \mathbb{N}$ y F_1 un bosque S-greedy de G. Para $i = 1, 2, \ldots, k$, sea F_i un bosque S-greedy de la subgráfica $G' = G - \{E(F_1) \cup E(F_2) \cup \cdots \cup E(F_{i-1})\}$, denotamos $E_k = E(F_1) \cup E(F_2) \cup \cdots \cup E(F_k)$, $G_k = (V(G), E_k)$. Entonces $\lambda_S(u, v; G_k) \geq \text{mín}\{\lambda_S(u, v; G), k\}$ para todo $u, v \in V(G)$, es decir, $F_1 \cup F_2 \cup \cdots \cup F_k$ es un certificado de k-conexidad S-mixta local de G, y este certificado tiene a lo más $k(n-1)$ aristas.*

Demostración. Es suficiente probar que $\lambda_S(u, v; G) \geq k$. Sean $u, v \in V(G)$ y G_k, G' y G como en el enunciado. Supongamos que lo anterior no es cierto y lleguemos a una contradicción. Así, $\lambda_S(u, v; G_k) < \text{mín}\{k, \lambda_S(u, v; G)\}$, es decir, $\lambda_S(u, v; G_k) < k$. Luego existe un conjunto de corte S-mixto $C = (Z, E_{G_k}(A, B))$ que separa a u de v en G_k, tal que $|C| \leq k - 1$. Sin embarg, por el lema 5.12, no puede existir este corte C de cardinalidad a lo más $k - 1$ en G. Por lo que la cardinalidad de un conjunto de corte S-mixto C' es al menos k. Esto significa que existe una arista $e \in E_G(A, B)$ que no pertenece a $F_1 \cup F_2 \cup \cdots \cup F_k$. Así, por el lema 5.12, tenemos que la cardinalidad de un conjunto de corte S-mixto C' es tal que $|C'| = |Z| + |E_{G_k}(A, B)| \geq k$. Lo anterior es una contradicción de nuestra suposición inicial. □

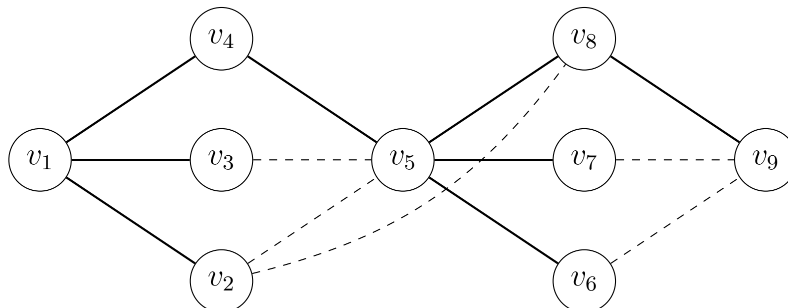


Figura 5.4: Gráfica G con 9 vértices y 13 aristas

Nota. Si u y v son dos vértices que viven en la misma componente de F_k entonces, las k (u, v)-trayectorias determinadas por los bosques $F_1, F_2, \ldots, F_k$ no son necesariamente S-independientes. Por ejemplo, sea $S = V(G)$ y $k = 2$ y consideremos la gráfica de 9 vértices y 13 aristas como en la figura 5.4. Y consideremos $u = v_3$ y $y = v_8$. Las aristas en negritas representan al árbol F_1 y las punteadas al árbol F_2.

Tal como Even, Itkis y Rajsbaum notaron en [21], si $S = V(G)$ en el teorema 5.13, obtenemos el resultado de Cheriyan *et al.* que genera certificados ralos de k-conexidad por vértices; el cual aparece como teorema 5.4 en esta tesis. Así, como el caso de conexidad por vértices generado por `NI-Search`, que aparece como teorema 5.3 en esta tesis.

Por otro lado, si $S = \varnothing$ en *Mixed Connectivity Certificate Theorem*, obtenemos el resultado de Nagamochi e Ibaraki para el caso de certificados ralos de k-conexidad por aristas que aparece como teorema 5.2 en esta tesis.

5.5. Comparación de los algoritmos

Para finalizar este capítulo, presentamos una comparación de los tres algoritmos que hemos revisado en esta tesis. Esta comparación ocupa los resultados presentados anteriormente. Así que nos servirá como un resumen de los conceptos y resultados que hemos estudiado hasta el momento.

Tal como lo reconocen los autores de [9] y [21], `NI-Search` fue el primer algoritmo publicado para generar certificados ralos de k-conexidad. Posteriormente, Cheriyan *et al.* publicaron un nuevo algoritmo para el caso de conexidad por vértices `Scan-first Search`. Como ya lo vimos estos algoritmos comparten las siguientes características:

- Ambos corren sobre una gráfica no dirigida G y nos entregan como salidas bosques maximales F_i, de cuya unión obtenemos los certificados ralos.
- Ambos corren en tiempo lineal, con k una constante oculta en el caso de `Scan-first Search`.
- Ambos son algoritmos secuenciales.

Sin embargo, a pesar de estas similitudes, estos algoritmos presentan diferencias muy definidas. Primero, notemos que `NI-Search` genera certificados de conexidad tanto por

vértices como por aristas, en este último caso de hecho admite gráficas con aristas múltiples. Mientras que `Scan-first Search` solo corre sobre gráficas simples y solo cubre el caso de conexidad por vértices. Además, tal como lo notaron A. Frank *et al.* en [23] y Even *et al.* en [21], `NI-Search` genera certificados que tienen una propiedad especial de conexidad mixta, llamada k-conexidad S-mixta (véanse definición 18 y teorema 4.3).

En segundo lugar, `NI-Search` *rankea* todas las aristas de G y tiene una complejidad de tiempo de $O(m+n)$. Mientras que `Scan-first Search` en una primera iteración genera un árbol `Scan-first Search` T_1 de la gráfica G en tiempo $O(m+n)$. En una segunda iteración, retiramos las aristas pertenecientes a T_1 y corremos nuevamente `Scan-first Search` sobre la gráfica $G \setminus T_1$. Así, para generar un certificado de k-conexidad necesitamos repetir k veces este procedimiento.[7] Esta es la razón de que la complejidad de tiempo de este algoritmo sea mayor, ya que es de $O(k(m+n))$.

Revisemos ahora la cantidad de aristas que tienen los certificados generados por ambos algoritmos. En el caso de conexidad por aristas para una gráfica con aristas múltiples, el certificado generado por `NI-Search` tiene una cantidad de aristas $m \leq kn - k(k+1)/2$. Mientras que en el caso de conexidad por vértices, ambos algoritmos generan un certificado con $m \leq k(n-1)$ aristas.

Después de revisar estas diferencias, surge la siguiente pregunta: ¿Por qué usar `Scan-first Search` para generar certificados, si `NI-Search` es más general y más eficiente? Si bien en su versión secuencial esto es cierto, `Scan-first Search` tiene otras implementaciones en otros enfoques. Como ya lo habíamos revisado antes en la sección de dedicada a este algoritmo, `Scan-first Search` tiene también una versión en paralelo así como una versión distribuida. Cada uno con complejidad $O(\log(n))$ y $O(d \cdot \log^3(n))$ respectivamente. Así, si queremos generar certificados de k-conexidad tenemos que correr cada uno de estos algoritmos k veces, obteniendo así una complejidad de tiempo $O(k \cdot \log(n))$ en paralelo y $O(k \cdot n \cdot \log^3(n))$ en su versión distribuida. Las cuales son implementaciones eficientes tal como lo mencionan sus autores.

Por otro lado, `Scan-first Search` tiene una ventaja sobre los dos famosos algoritmos de búsqueda en gráficas. Ya que si bien `Depth-first Search` corre en tiempo $O(m+n)$ en su versión secuencial, no se conoce una implementación en paralelo eficiente del mismo [9]. `Breadth-first Search` corre en tiempo $O(m+n)$ en su versión secuencial, y en tiempo $O(m + n \cdot \text{polylog}(n))$ en su versión distribuida[8], sin embargo, al momento de la publicación de este algoritmo, la implementación paralela más rápida no es más rápida que la multiplicación de matrices [27][9].

En el caso de `Distributed-NI`, al ser una implementación distribuida de `NI-Search`, como lo vimos en el teorema 5.6, hereda las propiedades de este. Sin embargo, tiene una complejidad de tiempo menor: $O(4m)$.

Por último, nos gustaría mencionar que los algoritmos que hemos revisado hasta el

[7] Si en algún momento de la ejecución encontramos una subgráfica $G' \subset G$ inconexa, corremos `Scan-first Search` en cada una de sus componentes conexas. Obteniendo así como resultado un bosque maximal de G'.

[8] La versión que presentamos corresponde con el artículo de B. Awerbuch y D. Peleg titulado *Network synchronization with polylogarithmic overhead.*

[9] En este artículo, los autores hacen una reducción del `BFS` paralelo al problema de la multiplicación de matrices.

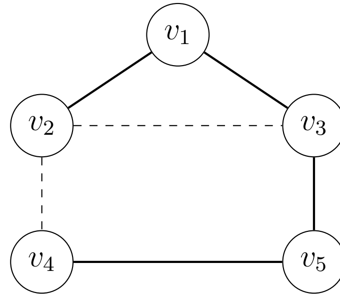


Figura 5.5: Un certificado formado por bosques V-greedy que no puede ser generado por `NI-Search`. Las aristas negritas representan a F_1 y las punteadas a F_2. Notemos que `NI-Search` no puede comenzar sobre los vértices incidentes a las aristas en F_2, v_2, v_3 y v_4. Por otro lado, si comenzamos en v_1, la arista (v_2, v_4) debería pertenecer a F_1 y la arista (v_4, v_5) a F_2 y si comenzamos en v_5, (v_2, v_4) pertenecería a F_1 y (v_1, v_2) a F_2.

momento son todos deterministas. En contraste, en la sección anterior revisamos un algoritmo no determinista para generar bosques maximales de una gráfica G, llamados bosques S-greedy. Estos bosques son una generalización de los bosques generados por `NI-Search` y `Scan-first Search`, cuya unión genera certificados de k-conexidad, por vértices, por aristas, o bien, S-mixtos, tal como lo vimos con anterioridad. Todo esto se resume en *Mixed Connectivity Certificate Theorem*. Sin embargo, tal como Even *et al.* notaron en [21], existen certificados compuestos por bosques S-greedy que no pueden ser producidos por `NI-Search`, véase la figura 5.5. Por eso, los resultados de A. Frank *et al.* en [23], sobre conexidad S-mixta no implican que en general los bosques S-greedy generen certificados de conexidad mixta.

Capítulo 6

Implementación y visualización

Beautiful is better than ugly.
...
Sparse is better than dense.
...
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
...

The Zen of Python

Una vez que hemos revisado a nivel teórico los algoritmos que generan certificados de k-conexidad, exponemos ahora su implementación y visualización. Comenzaremos revisando la implementación de gráficas en el lenguaje de programación `Python` para posteriormente revisar cada uno de los algoritmos estudiados a lo largo de esta tesis.

Elegimos `Python`[1][2], ya que es un lenguaje de alto nivel, es de propósito general y multiparadigma. El hecho de que sea de propósito general nos permitió no solo generar la implementación desde cero, sino que también nos permitió generar una interfaz de usuario (GUI por sus siglas en inglés) para hacer la visualización de los algoritmos y conectar nuestro código con algunas bibliotecas de uso común para el análisis y visualización de gráficas y redes. Por otro lado, explotamos el hecho de que sea multiparadigma al utilizar en nuestro código conceptos de programación orientada objetos, programación estructural, así como de programación funcional.

Además de lo mencionado anteriormente, escogimos este lenguaje ya que es muy popular y de uso común tanto en la academia como en la industria. Es por esto que tenemos a nuestra disposición múltiples bibliotecas (*libraries*), que podemos usar para ahorrar tiempo y esfuerzo en el desarrollo de nuestros programas. Y al ser un lenguaje tan popular en diversas áres, esperamos que este código sirva para múltiples aplicaciones a futuro; algunas de las cuales ya comentamos previamente.

[1]https://www.python.org
[2]La versión que utilizamos es `Python` 3.9.4 de 64 bits que corre en Conda de manera local.

6.1. Implementación de gráficas en Python

Existen varias formas de representar a una gráfica G dentro de una computadora, ya sea como una matriz de adyacencias, o bien, como una lista de adyacencias. La elección de la representación depende de la situación específica, en nuestro caso elegimos representar a las gráficas como una lista de adyacencias. Aunado a lo anterior, ocupamos el paradigma de programación orientada a objetos. Así, definimos primero la clase `Vertice` que representa a los vértices de nuestra gráfica.

```
1  class Vertice:
2       ### Constructor
3      def __init__(self,i):
4          self.id = i # El id para cada vertice es unico
5          self.scan = False
6          self.label = 0
7          self.vecinos = []
8          self.padre = None
9
10     def agregar_vecino(self,v,p):
11         if v not in self.vecinos:
12             self.vecinos.append([v,p])
13
14     def quitar_vecino(self,v,p):
15         if v in self.vecinos:
16             self.vecinos.remove([v,p])
```

Notemos que el constructor tiene las mismas variables que el algoritmo `NI-Search`, además de las variables *id* y *padre*. Por otro lado, tiene dos funciones:

- `agregar_vecinos`: esta recibe como entrada un vértice v y un entero p que guarda el valor de $rank(e)$ inicializado en cero.
- `quitar_vecinos`: recibe los mismos valores que la función anterior.

Veamos ahora como construimos una gráfica a partir de la clase `Vertice`. Para esto, definimos otra clase llamada `Grafica` que es la encargada de crear las gráficas a partir de una lista de aristas y de correr los algoritmos de `Busqueda_NI` y `Scan-first Search` sobre las mismas. Razón por la cual hemos elegido reprentar a G como una lista de adyacencias. Como ejemplo, véase la figura 6.1. La lista de adyacencias del vértice 0 es `[[1, 0], [2, 0], [3, 0]]`. Donde cada elemento de la forma `[a,b]` representa a un vecino de 0 y cuya primera entrada corresponde al *id* de este, por ejemplo 1, y la segunda guarda el valor $rank(e)$ de la arista formada por 0 y este vecino, por ejemplo la arista (0,1) con $rank(e) = 0$.

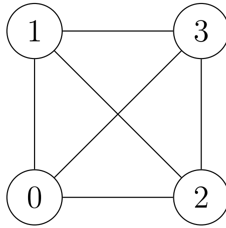


Figura 6.1: Sea $G = K_4$ y su lista de adyacencias

Vértice	Lista de adyacencias
0	[[1, 0], [2, 0], [3, 0]]
1	[[0, 0], [2, 0], [3, 0]]
2	[[0, 0], [1, 0], [3, 0]]
3	[[0, 0], [1, 0], [2, 0]]

Ahora bien, para construir la gráfica G ocupamos las siguientes funciones.

```
class Grafica:
    ### Constructor
    def __init__(self):
        self.vertices={} ###diccionario

    def agregar_vertice(self,v):
        if v not in self.vertices:
            self.vertices[v] = Vertice(v)

    def agregar_arista(self,a,b,p):
        if a in self.vertices and b in self.vertices:
            self.vertices[a].agregar_vecino(b,p)
            self.vertices[b].agregar_vecino(a,p)
```

Guardamos en un diccionario los vértices que vamos agregando a la gráfica, líneas 3-4. Y para agregar vértices a G ocupamos la función `agregar_vertice` que recibe como entrada el *id* del vértice, el cual recordemos es único, y si v no está en el diccionario `vertices`, pues crea un objeto `Vertice` y lo agrega, líneas 6-8. Por otro lado, para agregar aristas a G ocupamos la función `agregar_arista` que recibe como entrada los *id* de ambos vértices que forman la arista así como un entero p el cual debe ser cero.[3] Luego, si ambos vértices viven en el diccionario `vertices` pues los agrega a cada uno como vecino del otro. Para esto ocupa la función `agregar_vecino` de la clase `Vertice`. Así, para generar K_4 como en la figura 6.1, ocupamos el siguiente código:

```
G = Grafica() # Creamos un objeto de tipo Grafica
# Agregamos vertices
G.agregar_vertice(0)
G.agregar_vertice(1)
G.agregar_vertice(2)
G.agregar_vertice(3)
# Agregamos aristas
G.agregar_arista(0,1,0)
G.agregar_arista(0,2,0)
```

[3]Recordemos que este entero p guardará el valor de $rank(e)$

```
10 G.agregar_arista(0,3,0)
11 G.agregar_arista(1,2,0)
12 G.agregar_arista(1,3,0)
13 G.agregar_arista(2,3,0)
```

6.2. Implementación de NI-Search

Para implementar este algoritmo definimos la función `ni_search` en la clase `Grafica`, que como su nombre lo indica, es la encargada de ejecutar `Busqueda_NI` sobre la gráfica G. Esta función recibe como entrada el vértice r que actuará como raíz, es decir, el vértice sobre el cual iniciaremos la ejecución.

```
1  def ni_search(self,r):
2
3          if r in self.vertices and len(self.vertices[r].vecinos) > 0:
4              no_scan_vertices = []
5              actual = r
6
7          for v in self.vertices:
8                  no_scan_vertices.append(v)
9
10         while len(no_scan_vertices) > 0:
11             for vecino in self.vertices[actual].vecinos:
12                 if self.vertices[vecino[0]].scan == False:
13                     vecino[1] = (self.vertices[vecino[0]].label) + 1
14
15                 if self.vertices[actual].label == self.vertices[vecino[0]].label:
16                     self.vertices[actual].label += 1
17
18                 self.vertices[vecino[0]].label += 1
19
20             self.vertices[actual].scan = True
21             no_scan_vertices.remove(actual)
22
23             actual = self.maximo(no_scan_vertices)
24
25         else:
26             False
```

En la línea 3 pregunta si r forma parte de los vértices de G y si r tiene al menos un vecino en G. De ser así, crea una lista vacía `no_scan_vertices` y asigna a la variable *actual* al vértice r, líneas 3-4. Posteriormente agrega a todos los vértices de G a esta lista, línea 8. `no_scan_vertices` representa a los vértices que aun no han sido *escaneados*. En la línea 10, comenzamos un ciclo while que se ejecuta hasta que todos los vértices hayan sido *escaneados*; tal como sucede en la línea 5 del algoritmo 5.1.

Luego, comienzo un ciclo `for`, línea 11, que recorre a todos los vecinos del vértice *actual*. Si el vecino u no ha sido *escaneado*, asigna a $rank(e) = label(u)+1$ con $e = (actual, u)$; tal como sucede en la línea 8 del algoritmo `Busqueda_NI`. Luego, si $label(actual) = label(u)$ asignamos $label(actual) = label(actual) + 1$. como en las líneas 9 y 10 del algoritmo 5.1. En la línea 18, hacemos $label(u) = label(u) + 1$ tal como en la línea 12 de `Busqueda_NI`. Por último, marcamos al vértice *actual* como *escaneado*. En la línea 23 elegimos al vértice u tal que $label(u) \geq label(w)$ para todo $w \in no_scan_vertices$, como sucede en la línea 6 del algoritmo original, para ello ocupamos la función auxiliar `maximo`. Y así, seguimos en el ciclo `while` hasta que hayamos vaciado la lista $no_scan_vertices$.

Al finalizar la ejecución de la función `busqueda_ni`, ya hemos *rankeado* todas las aristas de G. Por último, ocupamos la función `ni_edges` que nos regresa una lista donde cada elemento es una sublista de la siguiente forma: `[[a,b],r]` donde `[a,b]` representa a la arista (a,b) y `r` representa al *rank* de esta misma arista.

Nota. Las funciones `ni_edges` así como la función `maximo`, que es usada por `ni_search` en la línea 23, son funciones de la clase `Grafica` cuyo código incluimos en el apéndice E.

A partir de esta lista de las aristas y el valor de *rank*, podemos crear una visualización de los bosques maximales generados por `busqueda_ni`, que no es más que la implementación del algoritmo `Busqueda_NI` de Nagamochi e Ibaraki. Esto lo hacemos con ayuda del módulo `xlsxwriter` que permite generar archivos de excel, estos contienen la lista de aristas con sus respectivos *ranks*. Este archivo se lo damos de comer a `Gephi`, que es un software *open-source* de análisis y visualización de redes escrito en `Java` (véase figura 6.2).

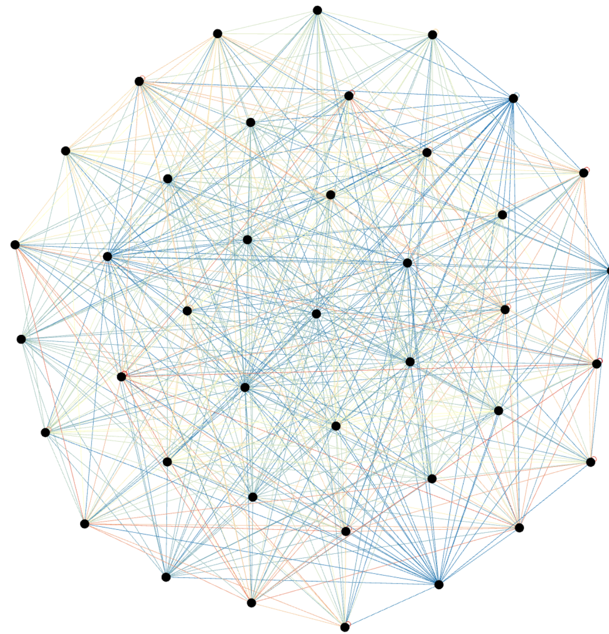


Figura 6.2: Ejemplo de la ejecución de `busqueda_ni` en una gráfica aleatoria de 40 vértices y 550 aristas con un grado medio de 27.5 y diámetro de 2. En este ejemplo el mayor $k = 27$. Imagen generada con Gephi, los diferentes colores representan los distintos valores de $rank(e)$.

6.3. Implementación de `Scan-first Search`

Para la implementación de este algoritmo definimos la función `sfs` dentro de la clase `Grafica`.

```
def sfs(self,r):

      if r in self.vertices and len(self.vertices[r].vecinos) > 0:

         no_scan = []
         actual = r

         for v in self.vertices:
            self.vertices[v].padre = None
            no_scan.append(v)

         while len(no_scan) > 0:
            for vecino in self.vertices[actual].vecinos:
               if self.vertices[vecino[0]].label == 0:
                  self.vertices[vecino[0]].padre = actual
            for vecino in self.vertices[actual].vecinos:
               if self.vertices[actual].label == self.vertices[vecino[0]].label:
                  self.vertices[actual].label += 1

               self.vertices[vecino[0]].label += 1

            self.vertices[actual].scan = True
            no_scan.remove(actual)

            actual = self.maximo(no_scan)
      else:

         return False
```

Esta función recibe como argumento el vértice r sobre el cual comenzará la ejecución. Así, si r es parte de la gráfica y tiene al menos un vecino comienza la ejecución, línea 3. Creamos una lista vacía `no_scan` y asignamos la variable actual al vértice r. Posteriormente agregamos a todo vertice de G a esta lista, así indicamos que no han sido *escaneados*. Además inicializamos la variable *padre* a `None` para todo v, líneas 8-10.

Mientras existan vértices que no hayan sido *escaneados* se ejecutará el ciclo `while` de la línea 12 a la 25. Para todos los vecinos de r, si $label(v) = 0$, designamos a actual como el *padre* de estos vértices, líneas 14 y 15. Luego, si $label(actual) = label(v)$ alguno de sus vecinos, pues sumamos 1 a $label(actual)$. Luego, a todo vecino v hacemos que $label(v) = label(v){+}1$. Por último, marcamos a r como *escaneado*, línea 22, y lo removemos de la lista de los no *escaneados*. Elegimos ahora como `actual` al vértice que tenga el mayor *label* en la lista `no_scan`. Al finalizar esta función obtenemos como resultado un árbol `Scan-first Search` de la gráfica G original.

Para generar certificados de k-conexidad por vértices de una gráfica G, utilizando `Scan-first Search` es necesario ejecutarlo k veces (Véase el teorema 5.4.), sobre la subgráfica $G - \{E(T_1) \cup \cdots \cup E(T_i)\}$ con $i = 1, 2, \ldots, k-1$. Para esto definimos la función `sfs_tree`. Esta función recibe como entrada una gráfica G, sobre la cual ya se corrió `sfs()` y nos regresa como salida las aristas que pertenecen a este árbol `Scan-first Search`. De esta manera podemos visualizar el árbol y generar una lista con las aristas que quitaremos para la siguiente iteración. Para quitar este árbol utilizamos la función `quitar_arbol`. Esta función recibe la lista de las aristas pertenecientes a un árbol `Scan-first Search` T_i y nos regresa como salida la gráfica $G' = G - E(T_i)$. Así, podemos volver a correr `sfs()` sobre la gráfica G_i para obtener un árbol `Scan-first Search` T_{i+1} de esta misma subgráfica. Repetimos este proceso k veces para obtener un certificado de k-conexidad por vértices de G.

Nota. Las funciones `sfs_tree` así como la función `quitar_arbol`, son funciones de la clase `Grafica` cuyo código incluimos en el apéndice E.

De igual manera que lo hicimos con `NI-Search`, a partir de la lista de las aristas generada por `sfs_tree`, podemos crear una visualización de los bosques maximales generados por `sfs`, que no es más que la implementación del algoritmo `Scan-first Search`. Consideremos la gráfica aleatoria G, véase la figura 6.3. En una primera iteración de `sfs` generamos el árbol `sfs`, comenzando por el vértice r_1, como en la figura 6.4. En rojo se destaca el vértice sobre el cual iniciamos la ejecución del algoritmo. Y de la misma manera que el algoritmo anterior, ocupamos `xlswriter` y Gephi para generar la imagen.

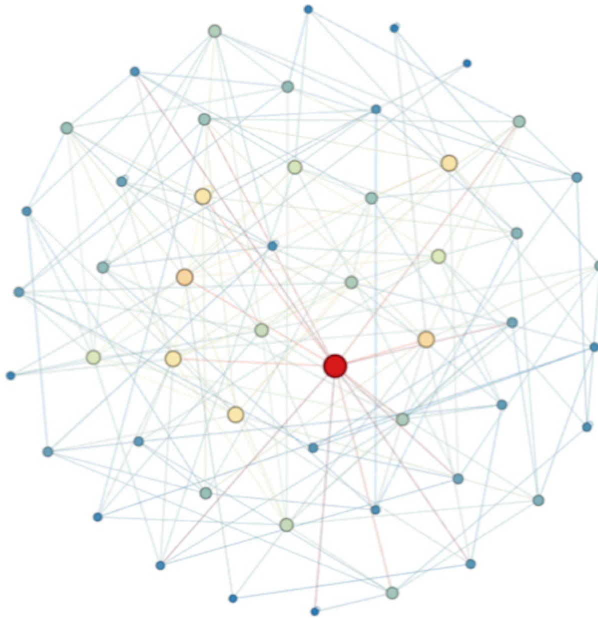


Figura 6.3: Gráfica aleatoria de 50 vértices y 189 aristas.

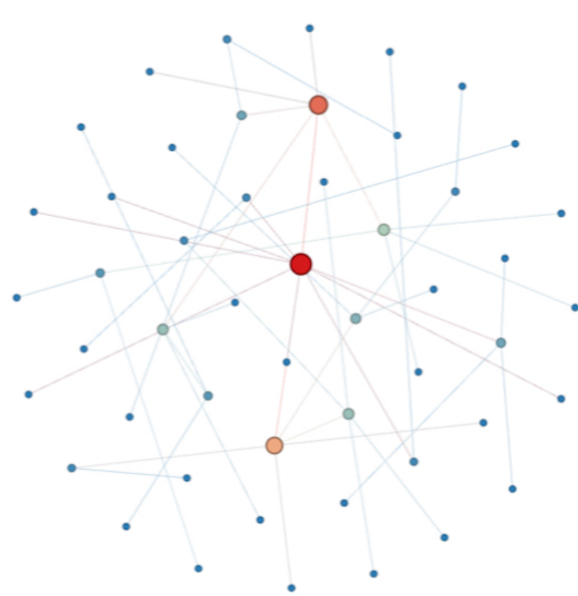


Figura 6.4: Árbol `sfs` T_1 de G con 50 vértices y 49 aristas.

Figura 6.5: Árbol `sfs` T_2 de $G \setminus T_1$ con 49 vértices y 48 aristas.

Para generar un árbol `sfs` en $G \setminus T_1$, quitamos el árbol T_1 de G con la función `quitar_arbol` y corremos de nuevo `sfs` sobre $G \setminus T_1$ comenzando con un vértice $r_2 \neq r_1$. De esta manera obtenemos el árbol `sfs` T_2, como en la figura 6.5 . Notemos que T_2 tiene un vértice menos que T_1, ya que el vértice raíz sobre el cual comenzamos la ejecución, queda completamente saturado. Es decir, todas sus aristas forman parte del árbol `sfs` T_1, por lo que al quitar este árbol de G este vérticé ya no forma parte de $G \setminus T_1$. Repetimos este procedimiento k veces, comenzando por un vértice distinto en cada iteración, para generar un certificado de k-conexidad por vértices.

6.4. Visualización de los algoritmos

Para generar la visualización ocupamos el programa `Grafica.py`, del que ya hablamos en las secciones anteriores, así como el programa `Certificates_tk.py`, el cual contiene todas las funciones de la GUI. Este programa utiliza las siguientes bibliotecas de `Python`:

- `Matplotlib`.[4] Es una biblioteca para graficado que proporciona una API orientada a objetos para incrustar gráficos en aplicaciones que utilizan kits de herramientas de GUI, entre ellas está `Tkinter`.

- `Tkinter`.[5] El paquete tkinter (ïnterfaz Tk") es la interfaz estándar de `Python` para generar GUI. Al igual que `Matplotlib`, utiliza como paradigma la programación orientada a objetos. Estas son las razones por la cual elegimos esta biblioteca para generar la visualización de los algoritmos.

[4]https://matplotlib.org
[5]https://docs.python.org/3/library/tkinter.html

- `Networkx`.[6] Esta es una biblioteca para la creación, manipulación y estudio de la estructura, dinámica y funciones de redes. Si bien, tiene varios métodos y algoritmos para gráficas, aunque no incluye los algoritmos que mencionamos en la tesis. A pesar de lo anterior, utilizamos esta biblioteca exclusivamente para la visualización tal como lo veremos más adelante.

A continuación describimos el funcionamiento y estructura de este programa. La pantalla de inicio, véase la figura 6.6, incluye las instrucciones que el usuario debe seguir para visualizar los algoritmos. Primero debe agregar las aristas de la gráfica de la forma (a, b) en los recuadros blancos y luego oprime el botón *Agregar aristas*. La lista de aristas agregadas se irán imprimiendo en pantalla. Una vez que ha agregado las aristas deseadas, el usuario debe oprimir el botón *Dibujar gráfica*. De esta manera la gráfica ingresada por el usuario se imprime en el espacio blanco al centro. En caso de que el usuario se equivoque puede presionar el botón *Limpiar*.

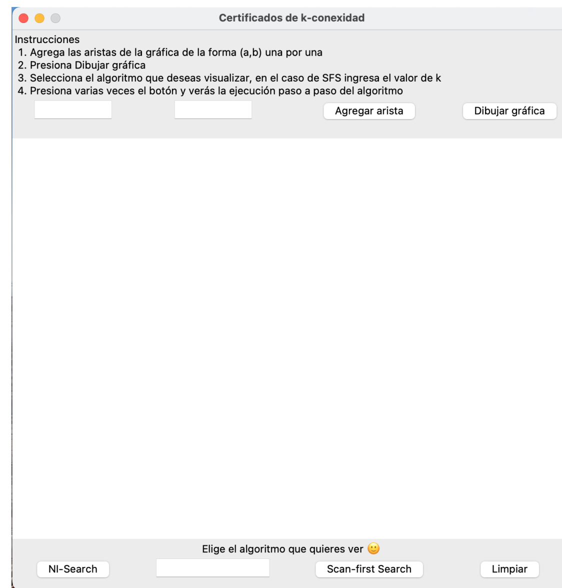


Figura 6.6: Imagen de la pantalla de inicio del programa

Para realizar lo anterior definimos las siguientes funciones:

- `add_edge()`. Esta función es utilizada para generar las aristas de G a partir de los pares que ingresa el usuario. Por ejemplo, el usuario ingresa 1 y 2 en los espacios en blanco y se genera la aristas $(1, 2)$ en la lista de aristas de la gráfica G. Es decir, generamos el objeto `Grafica` a partir de la lista de sus aristas.

- `crear_grafica()`. Una vez que se han ingreso correctamente todas las aristas de G, creamos un objeto `Graph` de `networkx` y generamos la imagen. Esta es la única vez que ocupamos esta biblioteca como parte de nuestro código.

[6] https://networkx.org

- `limpiar_ventana()`. Tal como su nombre lo indica, esta función borra la lista de aristas de G y limpia el espacio en blanco al centro. Lo anterior nos permite introducir nuevos datos, ya sea por que el usuario se equivocó o porque queremos generar una nueva gráfica o visualizar otro algoritmo.

Una vez que se tiene la gráfica deseada, el usuario debe elegir si ejecutará el algoritmo de `NI-Search`, o bien, `Scan-first Search`. A continuación describimos las funciones correspondientes a cada algoritmo:

- `ni_search_button()`. Como su nombre lo indica, esta función se encarga de hacer la visualización de `NI-Search`. Primero corre la función `ni_search(x)` de la clase `Grafica`, por default comienza por el vértice 1. Si volvemos a presionar el botón, se generará un certificado de 1-conexidad de G. Al volverlo a presionar aparecen las aristas correspondientes al bosque F_2, que junto con las del árbol F_1 generan un certificado de 2-conexidad. En cada i-ésima iteración aparecerá el bosque F_i de tal suerte que $F_1 \cup F_2 \cup ... \cup F_i$ es un certificado de i-conexidad. Este proceso para una vez que llegamos al valor de k más alto entre las aristas *rankeadas*.

- `k_sfs()`. Tal Como su nombre lo indica, esta función se encarga de hacer la visualización de `Scan-first Search`. En este caso el usuario debe ingresar el valor de j deseado, en caso de que este valor supere al valor k', el valor de k más alto entre las aristas *rankeadas*, pues nos imprime que no existe un certificado de j-conexidad. En caso de que si exista, en la primera iteración genera el árbol T_1 en color naranja.

  En una segunda iteración, imprime el árbol T_2 en naranja, los árboles anteriores se imprimen en color azul para distinguirlos del nuevo árbol generado. Esto se repite cada vez que oprimamos este botón hasta llegar al valor j ingresado por el usuario.

Por último, para generar otra gráfica o correr otro algoritmos, basta limpiar la ventana y volver a repetir los procedimientos antes mencionados.

Nota. Incluimos la totalidad del código de `Certificates_tk.py` en el apéndice F.

6.4.1. Vinculación con `networkx`

Como `networkx` es una de las bibliotecas más populares en gráficas y redes, decidimos añadir dos funciones a nuestro código para correr los algoritmos directamente sobre gráficas en este formato.

- `nx_ni_search(Gnx, v_0)`: esta función recibe una gráfica `Gnx` de `networkx` y un vértice `v_0` perteneciente a esta gráfica y nos regresa como salida una gráfica `H` de `networkx` con las aristas etiquetadas por `NI-Search`.

- `nx_sfs(Gnx, v_0)`: esta función recibe una gráfica `Gnx` de `networkx` y un vértice `v_0` perteneciente a esta gráfica y nos regresa como salida un árbol `Scan-first Search`.

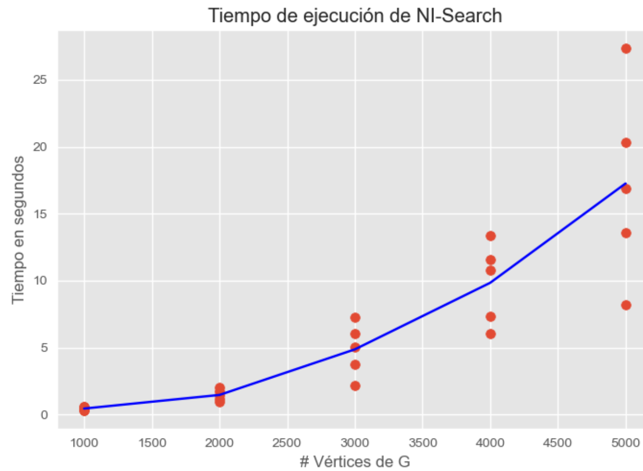


Figura 6.7: Los puntos en rojo representan el tiempo de ejecución en segundos y en azul el promedio de estos.

6.5. Evaluación y discusión

Como ya lo habíamos visto, `NI-Search` es un algoritmo que en tiempo lineal *rankea* todas las aristas de la gráfica que recibe como entrada, permitiendo generar bosques maximales con este *ranking*. Al ser más general y eficiente que `Scan-first Search`, además de presentar menos problemas con su ejecución. Lo anterior, ya que al eliminar el árbol `sfs` generado en alguna iteración, es posible que desconectemos la gráfica, o bien, advertir que vértices se quedarán sin vecinos en esta nueva subgráfica. Decidimos usar `NI-Search` como base de la visualización, pues ambos algoritmos generan bosques maximales de cuya unión obtenemos certificados de k-conexidad, de la manera en que ya lo hemos revisado a lo largo de este trabajo.

Para probar la eficiencia de la implementación realizada, generamos varias gráficas aleatorias y corrimos en cada una de ellas la función `ni_search()`. A continuación, exponemos los resultados:

Estas gráficas aleatorias corresponden con el famoso modelo de Erdős-Renyi. Generamos una gráfica aleatoria de $G(n, p)$ estableciendo un valor fijo para n, que será la cantidad de vértices y p la probabilidad de conexión entre cada par de ellos. Estas gráficas se caracterizan por ser densas y por tener una distribución de grados binomial.

Para generar estas gráficas escribimos la función `grafica_aleatoria(n,p)`, dentro del programa `Grafica.py`, que recibe estos mismos parámetros. Con ayuda de este código generamos veinticinco gráficas aleatorias con entre 1000 a 5000 vértices cada una y con valores de probabilidad de conexión de 0.2, 0.3, 0.4, 0,5 y 0.6 respectivamente, tal como se muestra en la figura 6.7.

Como lo podemos apreciar en las figuras 6.7, 6.8 y 6.9, el tiempo de ejecución crece linealmente respecto a la entrada, que es el tamaño de la gráfica.

Por último, nos gustaría comentar el caso de la implementación y visualización de `Distributed-NI`. Ya que se trata de un algoritmo distribuido, es claro que no podemos

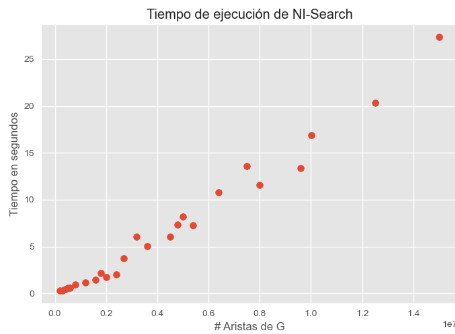


Figura 6.8: Tiempo de ejecución con respecto al tamaño de la gráfica.

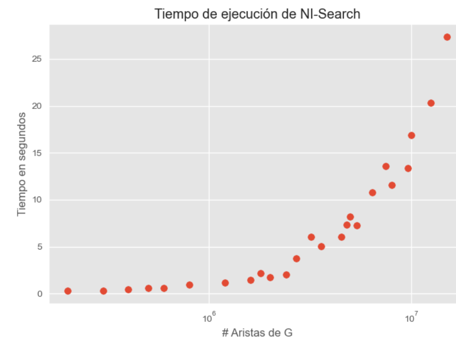


Figura 6.9: Tiempo de ejecución con respecto al tamaño de la gráfica en escala logarítmica.

programarlo de la misma manera que los otros algoritmos. Por otro lado, la mayoría de los simuladores contemporáneos, para el desarrollo de algoritmos distribuidos, requieren un tiempo de aprendizaje y experimentación, que no es trivial [18]. Aunado a esto, como este algoritmo no es más que la implementación distribuida de `NI-Search`, consideramos que la implementación y visualización de este segundo algoritmo ayudará también al lector a entender claramente como es la ejecución y resultado de `Distributed-NI`.

Capítulo 7

Conclusiones y trabajo futuro

> "An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it."
>
> Donald Knuth, *The Art of Computing Programming*

Comenzamos este trabajo revisando conceptos y resultados de conexidad en gráficas, con o sin pesos, algunos de los cuales fueron resultados propios; incluyendo la corrección de la prueba del teorema principal del artículo [37]. Posteriormente conectamos estos conceptos y resultados con los algoritmos anunciados desde la introducción. Presentamos tres algoritmos secuenciales: `NI-Search`, `Scan-first Search` y `Sparsify-2` y uno distribuido `Distributed-NI`, los cuales generan certificados ralos de conexidad, ya sea por vértices, por aristas o conexidad S-mixta. Revisamos primero el caso de 2-conexidad para luego estudiar el caso general de k-conexidad y tal como vimos, estos algoritmos corren en tiempo lineal. Estos algoritmos, salvo `Sparsify-2`, están basados en la descomposición de la gráfica en bosques maximales, de cuya unión obtenemos los certificados.

Consideramos que la aportación principal es la implementación y visualización de estos algoritmos, ya que es la primera vez que se lleva a cabo esta tarea, al menos de manera pública.[1] Ya que, como lo mencionamos en repetidas ocasiones, podemos ocupar estos algoritmos que generan certificados ralos de conexidad para reducir la complejidad de ejecución de otros algoritmos en gráficas. Aunado a esto, verificamos que la implementación es bastante eficiente ya que en cuestión de segundos es posible generar certificados ralos para gráficas con una gran cantidad de aristas. Además, agregamos funciones que permiten ocupar nuestro código junto con bibliotecas populares de `Python`. Por otro lado, con el incremento de la popularidad de la llamada *Ciencia de datos*, esperamos que estos algoritmos encuentren mútiples aplicaciones, más allá de las que mencionamos en este trabajo. Toda vez que es posible modelar problemas físicos, biológicos o sociales por medio de redes, que se traducen como gráficas en esta tesis.

Sobre el trabajo a futuro, un camino a seguir es extender la idea de descomponer la gráfica en bosques maximales al caso más general de gráficas con pesos. Lo anterior siguiendo la línea de investigación de S. Mathew y M. S. Sunitha en [37] y [38], que

[1] Puede consultar el código y la documentación completa en Github: https://github.com/VikSanz/connectivity_certificates

presentamos en este trabajo. Sobre este punto nos gustaría mencionar un algoritmo de reciente publicación para encontrar trayectorias pesadas[2] en gráficas *fuzzy*[3], el cual es obra de S. Mathew junto con S. Ali y J. N. Mordeson [2]. De hecho en este mismo artículo presentaron una aplicación particularmente interesante del estudio del tráfico de personas, identificando el origen, movimiento y destino de esta ominosa actividad.

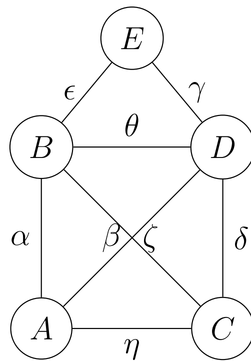
Otra dirección de trabajo sería hacer la implementación distribuida, así como su visualización, de `Distributed-NI`. Ya que esto permitiría el desarrollo de algún código o *framework* capaz de ser usado en cualquier otro algoritmo de gráficas que utilice el envío y recibo de mensajes para realizar la comunicación entre sus vértices durante la ejecución del mismo.

[2]Decimos que una (x, y)-trayectoria P es pesada si la suma de los pesos en sus aristas es la más alta de entre todas las trayectorias entre x y y.

[3]Una gráfica $fuzzy$ $G = (V, \mu, \rho)$ es un conjunto no vacío de vértices sobre el cual definimos un par de funciones $\mu : V \to [0, 1]$ y $\rho : V \times V \to [0, 1]$ tales que para todo $x, y \in V(G)$, $\rho(x, y) \leq \text{mín}\{\mu(x), \mu(y)\}$.

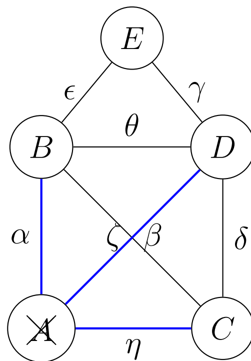
Apéndice A

Ejemplo de ejecución de `NI-Search`



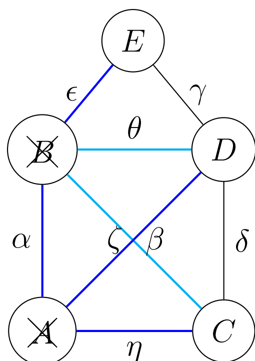
$V(G)$	$label(u)$	$E(G)$	$rank(e)$
A	0	α	0
B	0	β	0
C	0	δ	0
D	0	γ	0
E	0	ϵ	0
		ζ	0
		η	0
		θ	0

Figura A.1: Gráfica G de entrada, comenzamos en A



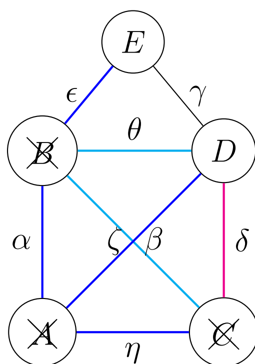
$V(G)$	$label(u)$	$E(G)$	$rank(e)$
A	1	α	1
B	1	β	0
C	1	δ	0
D	1	γ	0
E	0	ϵ	0
		ζ	1
		η	1
		θ	0

Figura A.2: Esto es lo que obtenemos tras escanear A



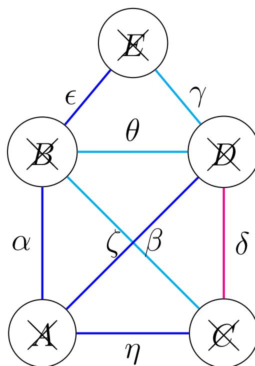
$V(G)$	$label(u)$	$E(G)$	$rank(e)$
A	1	α	1
B	2	β	2
C	2	δ	0
D	2	γ	0
E	1	ϵ	1
		ζ	1
		η	1
		θ	2

Figura A.3: Continuamos con B y lo escaneamos



$V(G)$	$label(u)$	$E(G)$	$rank(e)$
A	1	α	1
B	2	β	2
C	3	δ	3
D	3	γ	0
E	1	ϵ	1
		ζ	1
		η	1
		θ	2

Figura A.4: Continuamos con C y lo escaneamos

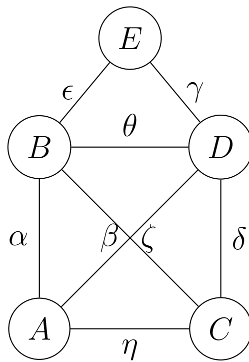


$V(G)$	$label(u)$	$E(G)$	$rank(e)$
A	1	α	1
B	2	β	2
C	3	δ	3
D	3	γ	2
E	2	ϵ	1
		ζ	1
		η	1
		θ	2

Figura A.5: Continuamos con D y lo escaneamos. Al terminar de escanearlo solo queda E cuyas aristas incidentes ya están todas *rankeadas* así que lo escaneamos y termina la ejecución.

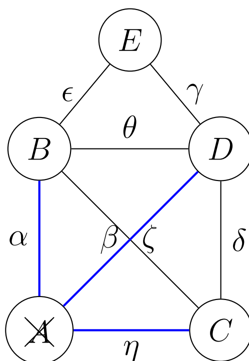
Apéndice B

Ejemplo de ejecución de `Scan-first Search`



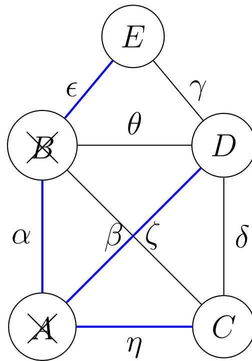
$V(G)$	$label(u)$	T_i	$E(T_i)$
A	0	T_0	$\varnothing$
B	0		
C	0		
D	0		
E	0		

Figura B.1: Gráfica G de entrada, comenzamos en A



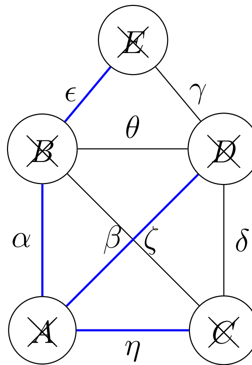
$V(G)$	$label(u)$	T_i	$E(T_i)$
A	1	T_0	$\varnothing$
B	1	T_1	α, β, η
C	1		
D	1		
E	0		

Figura B.2: Terminamos de escanear A



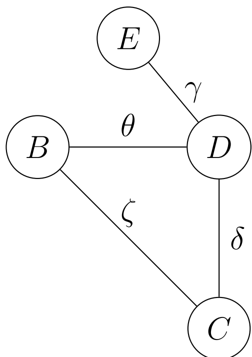
$V(G)$	$label(u)$	T_i	$E(T_i)$
A	1	T_0	$\varnothing$
B	1	T_1	$\alpha, \beta, \eta, \epsilon$
C	1		
D	1		
E	1		

Figura B.3: Continuamos con B



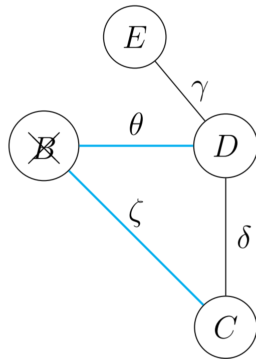
$V(G)$	$label(u)$	T_i	$E(T_i)$
A	1	T_0	$\varnothing$
B	1	T_1	$\alpha, \beta, \eta, \epsilon$
C	1		
D	1		
E	1		

Figura B.4: Notemos que todos los vértices cumplen que $label(u) > 0$, así que ya no agregamos más aristas a T_1 y terminamos de escanear todos los vértices de G



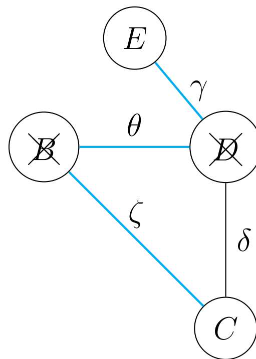
$V(G)$	$label(u)$	T_i	$E(T_i)$
A	0	T_0	$\varnothing$
B	0	T_1	$\alpha, \beta, \eta, \epsilon$
C	0		
D	0		
E	0		

Figura B.5: Gráfica $G \setminus T_1$, comenzamos en B



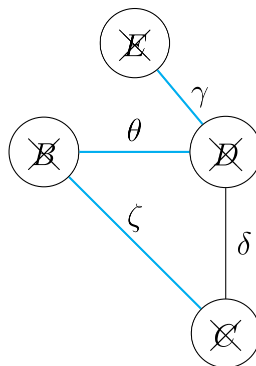
$V(G)$	$label(u)$	T_i	$E(T_i)$
A		T_0	$\varnothing$
B	1	T_1	$\alpha, \beta, \eta, \epsilon$
C	1	T_2	θ, ζ
D	1		
E	0		

Figura B.6: Terminamos de escanear B



$V(G)$	$label(u)$	T_i	$E(T_i)$
A		T_0	$\varnothing$
B	1	T_1	$\alpha, \beta, \eta, \epsilon$
C	1	T_2	θ, ζ, γ
D	1		
E	1		

Figura B.7: Continuamos con D



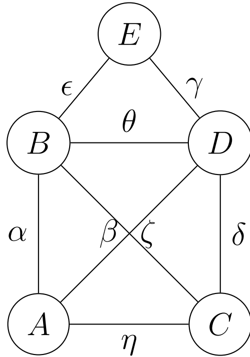
$V(G)$	$label(u)$	T_i	$E(T_i)$
A		T_0	$\varnothing$
B	1	T_1	$\alpha, \beta, \eta, \epsilon$
C	1	T_2	θ, ζ, γ
D	1		
E	1		

Figura B.8: Notemos que todo vértice cumple que $label(u) > 0$, así que ya no agregaremos más aristas a T_2 por lo que terminamos de escanear los vértices y termina la segunda iteración.

Apéndice C

Ejemplo de ejecución de Distributed_NI

El algoritmo comienza con el envío de un mensaje VISIT a uno de los vértices por medio de una arista nula (*nil edge*). Comenzemos por el vértice A, por lo que el *servidor* comienza en este vértice. Primero hacemos $first_time =$ False. Una vez que A envía el mensaje RANK_EDGE en sus aristas incidentes e', espera a que EDGE_RANKED(i) llegue por e' (véase figura C.1).



$V(G)$	$label(u)$	$first_time$	$unvisited$
A	0	False	$[\alpha, \beta, \eta]$
B	0	True	$[\alpha, \beta, \theta, \epsilon]$
C	0	True	$[\eta, \zeta, \delta]$
D	0	True	$[\theta, \delta, \zeta, \gamma]$
E	0	True	$[\epsilon, \gamma]$

Figura C.1: El vértice A envía RANK_EDGE en las aristas α, β y η

El mensaje RANK_EDGE llega al vértice B por la arista α. Así que $label(B) = rank(\alpha) = label(B) + 1 = 0 + 1$. Y envía EDGE_RANKED(rank(α)) por la arista α de vuelta a A. Lo mismo se repite para los vértices D y C, ya que la lista *unvisited* de A incluye a las aristas β y η. Notemos que ahora $label(B) = label(C) = label(D) = 1$ y $rank(\alpha) = rank(\beta) = rank(\eta) = 1$, tal como se aprecia en la figura C.2.

Una vez que EDGE_RANKED(i) llega por cada una de las aristas incidentes α, β y η, el vértice A *rankea* a las tres con el número 1. Por último tenemos $label(A) = 1$. Las aristas e tales que $rank(e) = 1$ son representadas en azul. Luego retiramos de la lista *unvisited* a todas las aristas e' tales que $rank(e') \geq rank(e)$ y enviamos VISIT en e' (Véase la figura C.3). Por último esperamos que el mensaje RETURN llegue por la arista e'. Notemos que el algoritmo termina una vez que RETURN es recibido por la arista nula incidente a A que comenzó la ejecución del algoritmo..

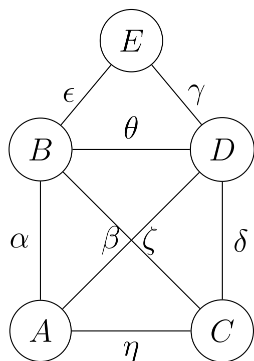


Figura C.2: Los vecinos de A reenvían EDGE_RANKED(rank(e)) por las aristas α, β y η

$V(G)$	$label(u)$	$first_time$	$unvisited$
A	0	False	$[\alpha, \beta, \eta\ ]$
B	1	True	$[\alpha, \beta, \theta, \epsilon\ ]$
C	1	True	$[\ \eta, \zeta, \delta\ ]$
D	1	True	$[\ \theta, \delta, \zeta, \gamma\ ]$
E	0	True	$[\ \epsilon, \gamma\ ]$

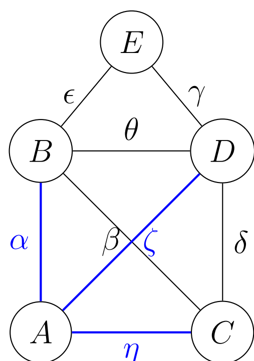


Figura C.3: α, β y η en azul

$V(G)$	$label(u)$	$first_time$	$unvisited$
A	1	False	$[\,]$
B	1	True	$[\alpha, \beta, \theta, \epsilon\ ]$
C	1	True	$[\ \eta, \zeta, \delta\ ]$
D	1	True	$[\ \theta, \delta, \zeta, \gamma\ ]$
E	0	True	$[\ \epsilon, \gamma\ ]$

Continuando con la ejecución, repetimos lo mismo que hicimos en A para el vértice B. El cual ya recibió un mensaje VISIT por la arista α (véase figura C.4). Así que $first_time$=False y B envía RANK_EDGE en las aristas incidentes ϵ, θ, η. Notemos que α es retirado de *unvisited*, pues es por esta arista por la que llegó el mensaje VISIT.

El mensaje RANK_EDGE llega ahora al vértice E por la arista ϵ. Así que $label(E) = rank(\epsilon) = label(E) + 1 = 0 + 1$. Y envía EDGE_RANKED(rank(ϵ)) por esta misma arista ϵ. Luego, el mensaje RANK_EDGE llega al vértice D por la arista θ. Así que $label(D) = rank(\theta) = label(E) + 1 = 1 + 1 = 2$. Y envía EDGE_RANKED(rank(θ)) al vértice B por esta misma arista θ. Lo mismo se repite para C y ζ (véase figura C.5).

Una vez que EDGE_RANKED(i) llega por cada una de las aristas β, θ, ϵ, *rankeamos* a las tres como en la figura C.5. Por último hacemos $label(B) = 2$. Las aristas e tales que $rank(e) = 1$ son representadas en azul y las aristas e' tales que $rank(e') = 2$ son representadas en cian. Por último retiramos de *unvisited* a todas las aristas e' tales que $rank(e') \geq rank(e)$ y enviamos VISIT en e' (véase la figura C.3). Y por último, esperamos que el mensaje RETURN llegue por la arista e'.

Seguimos ahora con el vértice D, que recibe el mensaje VISIT por la arista β. Repetimos lo que hicimos para los vértices A y B, al terminar obtenemos la figura C.6. Notemos que ya hemos *rankeado* todas las aristas de G. Sin embargo, el algoritmo aún no termina.

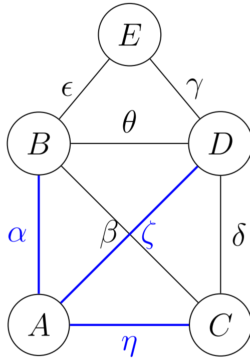


Figura C.4: Seguimos ahora con el vértice B

$V(G)$	$label(u)$	$first_time$	$unvisited$
A	1	False	[ ]
B	1	False	$[\beta, \theta, \epsilon\ ]$
C	1	True	$[\ \eta, \zeta, \delta\ ]$
D	1	True	$[\ \theta, \delta, \zeta, \gamma\ ]$
E	0	True	$[\ \epsilon, \gamma\ ]$

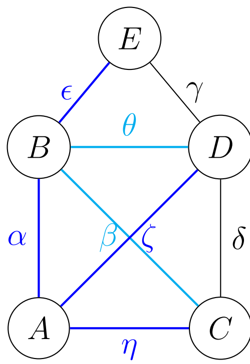


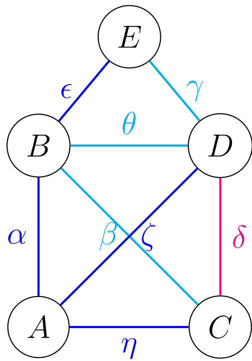
Figura C.5: $ranK(e) = 1$ en azul y $ranK(e) = 2$ en cian

$V(G)$	$label(u)$	$first_time$	$unvisited$
A	1	False	[ ]
B	2	False	[ ]
C	2	True	$[\ \eta, \zeta, \delta\ ]$
D	2	True	$[\ \theta, \delta, \zeta, \gamma\ ]$
E	1	True	$[\ \epsilon, \gamma\ ]$

Seguimos ahora con el vértice C que recibe el mensaje `VISIT` por la arista η. Repetimos lo que hicimos para los vértices A, B y D, al terminar obtenemos la figura C.7.

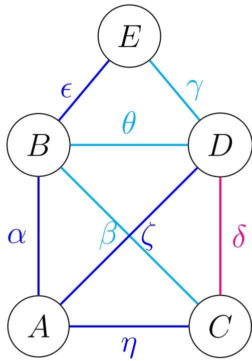
Seguimos ahora con el vértice D, que recibe el mensaje `VISIT` por la arista ϵ. Repetimos lo que hicimos para los vértices anteriores, al terminar obtenemos la figura C.8.

Por último enviamos el mensaje `RETURN` por cada una de las aristas, lo que ocurre una única vez en cada una de ellas. El algoritmo termina una vez que envíamos `RETURN` por la arista nula en el vértice A.



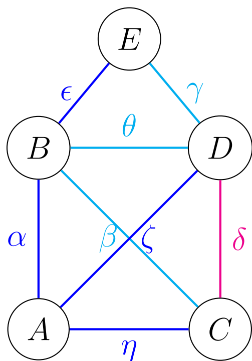
$V(G)$	$label(u)$	$first_time$	$unvisited$
A	1	`False`	[ ]
B	2	`False`	[ ]
C	3	`True`	[η, ζ, δ]
D	3	`False`	[ ]
E	1	`True`	[ϵ, γ]

Figura C.6: $rank(e) = 1$ en azul, $rank(e) = 2$ en cian y $rank(e) = 3$ en magenta



$V(G)$	$label(u)$	$first_time$	$unvisited$
A	1	`False`	[ ]
B	2	`False`	[ ]
C	3	`False`	[ ]
D	3	`False`	[ ]
E	1	`True`	[ϵ, γ]

Figura C.7: $rank(e) = 1$ en azul, $ranK(e) = 2$ en cian y $ranK(e) = 3$ en magenta



$V(G)$	$label(u)$	$first_time$	$unvisited$
A	1	`False`	[ ]
B	2	`False`	[ ]
C	3	`False`	[ ]
D	3	`False`	[ ]
E	2	`True`	[ ]

Figura C.8: $rank(e) = 1$ en azul, $ranK(e) = 2$ en cian y $ranK(e) = 3$ en magenta

Apéndice D

Ejemplo de ejecución de `Sparsify-2`

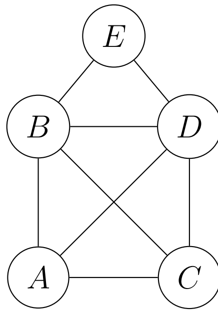


Figura D.1: Gráfica G de entrada

	$parent(u)$	$num(u)$	$low(u)$
A	0	0	0
B	0	0	0
C	0	0	0
D	0	0	0
E	0	0	0

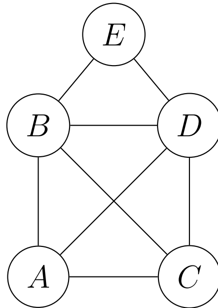
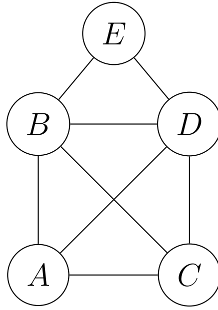


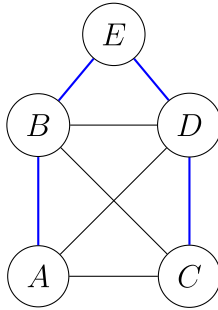
Figura D.2: Comenzamos en el vértice A

	$parent(u)$	$num(u)$	$low(u)$
A	0	1	1
B	0	0	0
C	0	0	0
D	0	0	0
E	0	0	0



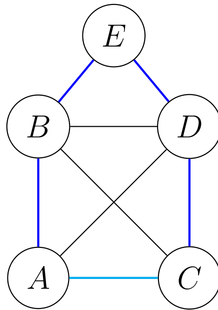
	$parent(u)$	$num(u)$	$low(u)$
A	0	1	1
B	A	2	2
C	0	0	0
D	0	0	0
E	0	0	0

Figura D.3: Tras la primera iteración



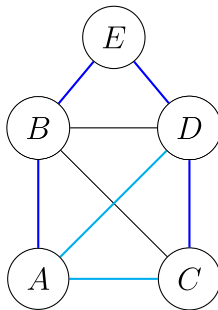
	$parent(u)$	$num(u)$	$low(u)$
A	0	1	1
B	A	2	2
C	D	5	5
D	E	4	4
E	B	3	3

Figura D.4: Primero se construye el árbol DFS



	$parent(u)$	$num(u)$	$low(u)$
A	0	1	1
B	A	2	2
C	D	5	1
D	E	4	4
E	B	3	3

Figura D.5: Ejecución de las líneas 12, 13 y 18 en C, agregamos la *backedge* de $C \hookrightarrow A$



	$parent(u)$	$num(u)$	$low(u)$
A	0	1	1
B	A	2	2
C	D	5	1
D	E	4	1
E	B	3	3

Figura D.6: Al final de la ejecución obtenemos P'

Apéndice E

Código de la clase `Gráfica`

```
class Vertice:
     ### Constructor
    def __init__(self,i):
        self.id = i
        self.scan = False
        self.label = 0
        self.vecinos = []
        self.padre = None

    def agregar_vecino(self,v,p):
        if v not in self.vecinos:
            self.vecinos.append([v,p])

    def quitar_vecino(self,v):
        if v in self.vecinos:
            self.vecinos.remove(v)


class Grafica:

    ### Constructor
    def __init__(self):
        self.vertices={} ###diccionario

    def aristas(self):
        edge_lst = []
        for u in self.vertices:
            for v in self.vertices[u].vecinos:
                    edge_lst.append([u,v[0]])

        edges = {tuple(sorted(e)) for e in edge_lst}
        return list(edges)

```

```
34      def info(self):
35          n = len(self.vertices)
36          edges = self.aristas()
37          m = (len(edges))//2
38          print("La grfica tiene "+str(n)+" vrtices y "+str(m)+" aristas.")
39  
40      def reiniciar_grafica(self):
41          self.vertices.clear()
42  
43          for u in self.vertices:
44            self.vertices[u].vecinos.clear()
45  
46      def agregar_vertice(self,v):
47          if v not in self.vertices:
48              self.vertices[v] = Vertice(v)
49  
50      def agregar_arista(self,a,b,p):
51          if a in self.vertices and b in self.vertices:
52              self.vertices[a].agregar_vecino(b,p)
53              self.vertices[b].agregar_vecino(a,p)
54              #e = self.Arista(a, b, x)
55  
56      def aristas_incidentes(self,v):
57          if v in self.vertices:
58              edges=[]
59              for vecino in self.vertices[v].vecinos:
60                  edges.append([v,vecino])
61          return edges
62  
63      def quitar_arista(self,a,b,p):
64          if a in self.vertices and b in self.vertices:
65              self.vertices[a].quitar_vecino([b,p])
66              self.vertices[b].quitar_vecino([a,p])
67  
68      ### Nos ayuda a quitar las aristas del rbol generadas por sfs_tree
69      def quitar_arbol(self,lista):
70          for i in range(0, len(lista)-1,3):
71              self.quitar_arista(lista[i],lista[i+1],lista[i+2])
72  
73      #### Solo ayuda para saber quien es el pap de los vrtices en sfs
74      def imprimir_grafica(self,r):
75          for v in self.vertices:
76              print("El pap del vrtice "+str(v)+ " es " + str(self.vertices[v].padre))
77  
78  
79      def maximo(self, lista):
80          if len(lista)>0:
```

```
81            m = self.vertices[lista[0]].label ### label del primer elemento de la lista
82
83            v = lista[0]
84
85            for e in lista:
86                if m < self.vertices[e].label:
87                    m = self.vertices[e].label
88                    v = e
89
90            return v
91    ### Puede ser reutilizada para k-sfs
92    def sfs(self,r):
93
94        if r in self.vertices and len(self.vertices[r].vecinos) > 0:
95
96            no_scan = []
97            actual = r
98
99            for v in self.vertices:
100               self.vertices[v].padre = None
101               no_scan.append(v)
102
103           while len(no_scan) > 0:
104               for vecino in self.vertices[actual].vecinos:
105                   if self.vertices[vecino[0]].label == 0:
106                       self.vertices[vecino[0]].padre = actual
107               for vecino in self.vertices[actual].vecinos:
108                   if self.vertices[actual].label == self.vertices[vecino[0]].label:
109                       self.vertices[actual].label += 1
110
111                   self.vertices[vecino[0]].label += 1
112
113               self.vertices[actual].scan = True
114               no_scan.remove(actual)
115
116               actual = self.maximo(no_scan)
117       else:
118
119           return False
120
121   def sfs_tree(self,g):
122       t = Grafica()
123       edge_list=[]
124       edge_plain=[]
125
126       for v in g.vertices:
127           t.agregar_vertice(v)
```

```
128         for v in t.vertices:
129             t.agregar_arista(v,g.vertices[v].padre,1)
130 
131         for u in t.vertices:
132             for v in t.vertices[u].vecinos:
133                 edge_list.append((u,v[0]))
134                 edge_list.append(1)
135         for u in t.vertices:
136             for v in t.vertices[u].vecinos:
137                 edge_plain.append(u)
138                 edge_plain.append(v[0])
139                 edge_plain.append(v[1])
140 
141         return edge_list,edge_plain
142 
143     def ni_search(self,r):
144 
145         if r in self.vertices and len(self.vertices[r].vecinos) > 0:
146 
147             no_scan_vertices = []
148             actual = r
149 
150         for v in self.vertices:
151                 no_scan_vertices.append(v)
152 
153 
154         while len(no_scan_vertices) > 0:
155             for vecino in self.vertices[actual].vecinos:
156                 if self.vertices[vecino[0]].scan == False:
157                     vecino[1] = (self.vertices[vecino[0]].label) + 1
158 
159                 if self.vertices[actual].label == self.vertices[vecino[0]].label:
160                     self.vertices[actual].label += 1
161 
162                 self.vertices[vecino[0]].label += 1
163 
164             self.vertices[actual].scan = True
165             no_scan_vertices.remove(actual)
166 
167             actual = self.maximo(no_scan_vertices)
168 
169         else:
170 
171             False
172 
173     def ni_edges(self):
174       edge_list=[]
```

```
175
176         for u in self.vertices:
177           for v in self.vertices[u].vecinos:
178               if v[1]>0:
179                   edge_list.append([u,v[0]])
180                   edge_list.append(v[1]) ### v[1] se guarda el rank de la arista [u,v]
181
182         return edge_list
183
184
185
186 def nueva_grafica(G):
187     H = Grafica()
188
189     for v in G.vertices:
190         H.agregar_vertice(v)
191
192     for u in G.vertices:
193         for v in G.vertices[u].vecinos:
194             H.agregar_arista(u,v[0],v[1])
195     return H
196
197 def grafica_aleatoria(n,p):
198     J = Grafica()
199
200     vertices=[]
201
202     for i in range(n):
203          vertices.append(i)
204
205     for v in vertices:
206          J.agregar_vertice(v)
207
208     edges = [(i,j) for i in range(n) for j in range(i) if random.random() < p]
209
210     for (i,j) in edges:
211         J.agregar_arista(i,j,0)
212
213     return J
```

Apéndice F

Código de Certificates_tk.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon May 5 11:38:45 2021
@author: Victor Sanchez
"""

import matplotlib
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt
import tkinter as tk
import networkx as nx
from tkinter import *
import time

import grafica
from grafica import *

#----- ROOT -----#
root = tk.Tk()
root.wm_title("Certificados de k-conexidad")
root.geometry('700x700')
root.resizable(False, False)

# Salir del programa cuando cerramos la ventana
root.wm_protocol('WM_DELETE_WINDOW', root.quit)

# Matplotlib
f = plt.figure(figsize=(7,5))
a = f.add_subplot(111)
plt.axis('off')


```

```
34 # Creamos la grafica G
35 G = Grafica()
36
37 #----- Variables -----#
38 edge_aVar=tk.StringVar()
39 edge_bVar=tk.StringVar()
40 bttn_clicks = 0
41 k_Var=tk.StringVar()
42 xlim=a.get_xlim()
43 ylim=a.get_ylim()
44
45 # Canvas
46 canvas = FigureCanvasTkAgg(f, master=root)
47 canvas.draw()
48 canvas.get_tk_widget().grid(row=3,columnspan=4)
49
50 #--- FUNCIONES ---#
51 def add_edge():
52
53     edge_a=int(edge_aVar.get())
54     edge_b=int(edge_bVar.get())
55
56     G.agregar_vertice(edge_a)
57     G.agregar_vertice(edge_b)
58
59     G.agregar_arista(edge_a,edge_b, 0)
60
61     edge_aVar.set("")
62     edge_bVar.set("")
63
64     lista_aristas.config(text=f'Las aristas de G: {G.aristas()}')
65
66
67 def crear_grafica():
68     # the networkx part
69     H = nx.Graph()
70     H.add_edges_from(G.aristas())
71     pos=nx.circular_layout(H)
72     nx.draw_networkx(H,pos=pos,ax=a)
73     xlim=a.get_xlim()
74     ylim=a.get_ylim()
75
76     # a tk.DrawingArea
77     canvas = FigureCanvasTkAgg(f, master=root)
78     canvas.draw()
79     canvas.get_tk_widget().grid(row=3,columnspan=4)
80
```

```python
def ni_search_button():

        global bttn_clicks
        a.cla()
        G.ni_search(1)
        busqueda_ni = G.ni_edges()

        J= nx.Graph()
        for i in range(0,len(busqueda_ni),2):
            J.add_edge(busqueda_ni[i][0], busqueda_ni[i][1], weight=busqueda_ni[i+1])

        pos1 = nx.circular_layout(J)

        edges,weights = zip(*nx.get_edge_attributes(J,'weight').items())

        lista1=[]

        for e,w in zip(edges,weights):

            if w <= bttn_clicks:

                a.cla()
                lista1.append(e)
                lista1.append(w)

                K=nx.Graph()
                for i in range(0,len(lista1),2):
                    K.add_edge(lista1[i][0], lista1[i][1], weight=lista1[i+1])

                edges1,weights1 = zip(*nx.get_edge_attributes(K,'weight').items())

                nx.draw_networkx(K, pos1, edgelist=edges1,
                edge_color=weights1, edge_cmap=plt.cm.cool,ax=a)
                #a.set_xlim(xlim)
                #a.set_ylim(ylim)
                plt.axis('off')

                canvas = FigureCanvasTkAgg(f, master=root)
                canvas.draw()
                canvas.get_tk_widget().grid(row=3,columnspan=4)

        bttn_clicks += 1
        if bttn_clicks <= max(weights) + 1:
            label1.config(text=f'Certificado ralo de {bttn_clicks-1}-conexidad')



```

```
128 def k_sfs():
129
130     global bttn_clicks
131     a.cla()
132     G.ni_search(1)
133     busqueda_ni = G.ni_edges()
134
135     k_cert = int(k_Var.get())
136
137     J= nx.Graph()
138     for i in range(0,len(busqueda_ni),2):
139         J.add_edge(busqueda_ni[i][0], busqueda_ni[i][1], weight=busqueda_ni[i+1])
140
141     pos1 = nx.circular_layout(J)
142     edges,weights = zip(*nx.get_edge_attributes(J,'weight').items())
143
144     lista1=[]
145
146
147     for e,w in zip(edges,weights):
148
149         if w <= k_cert:
150
151             a.cla()
152             lista1.append(e)
153             lista1.append(w)
154
155             K=nx.Graph()
156             for i in range(0,len(lista1),2):
157                 K.add_edge(lista1[i][0], lista1[i][1], weight=lista1[i+1])
158
159             edges1,weights1 = zip(*nx.get_edge_attributes(K,'weight').items())
160
161             print(K.edges(data='weight'))
162
163             edge_colors = ['black' if peso > bttn_clicks else 'orange'
164             if peso == bttn_clicks else 'blue' for u,v,peso in K.edges(data='weight')]
165             print(edge_colors)
166             nx.draw_networkx(K, pos1, edgelist=edges1, edge_color=edge_colors,ax=a)
167             #a.set_xlim(xlim)
168             #a.set_ylim(ylim)
169             plt.axis('off')
170
171             canvas = FigureCanvasTkAgg(f, master=root)
172             canvas.draw()
173             canvas.get_tk_widget().grid(row=3,columnspan=4)
174
```

```
175        bttn_clicks += 1
176        if bttn_clicks <= max(weights)+1 and bttn_clicks <= (k_cert + 1):
177            label1.config(text=f'Bosques SFS:
178            En naranja el nuevo bosque construido T_{bttn_clicks-1},
179            en azul los bosques anteriores.')
180
181    def limpiar_ventana():
182        #Clear canvas
183        a.cla()
184
185        plt.axis('off')
186        canvas = FigureCanvasTkAgg(f, master=root)
187        canvas.draw()
188        canvas.get_tk_widget().grid(row=3,columnspan=4)
189
190        G.reiniciar_grafica()
191
192        print("G tiene las siguientes aristas"+str(G.aristas()))
193
194    instrucciones= ("""Instrucciones \n 1. Agrega las aristas de la grafica
195    de la forma (a,b) una por una \n 2. Presiona Dibujar grafica \n
196    3. Selecciona el algoritmo que deseas visualizar, en el caso de SFS
197    ingresa el valor de k \n 4. Presiona varias veces el boton y veras
198    la ejecucion paso a paso del algoritmo""")
199
200    #----- WIDGETS -----#
201    edge_a = tk.Entry(root,textvariable = edge_aVar,width=10)
202    edge_b = tk.Entry(root,textvariable = edge_bVar,width=10)
203    add_edge_button=tk.Button(root,text = 'Agregar arista',command= add_edge)
204    lista_aristas= tk.Label(root, text='')
205
206    draw_graph=tk.Button(root,text = 'Dibujar grafica',command= crear_grafica)
207    instructions = tk.Label(root,text=instrucciones,justify= LEFT)
208
209    button_ni = tk.Button(root, text="NI-Search",command=ni_search_button)
210    label1 = tk.Label(root, text="Elige el algoritmo que quieres ver")
211    limpiar = tk.Button(root, text='Limpiar',command=limpiar_ventana)
212    button_sfs=tk.Button(root, text='Scan-first Search',command=k_sfs)
213    k_sfs_entry=tk.Entry(root,textvariable= k_Var, width=15)
214
215    #----- PANTALLA -----#
216    # Primer renglon
217    instructions.grid(row=0,columnspan=3)
218    # Segundo renglon
219    edge_a.grid(row=1,column=0)
220    edge_b.grid(row=1,column=1)
221    add_edge_button.grid(row=1,column=2)
```

```
222 draw_graph.grid(row=1,column=3)
223 #Tercer renglon
224 lista_aristas.grid(row=2,columnspan=4)
225 # Cuarto renglon ----> Canvas
226 # Quinto renglon
227 label1.grid(row=4,columnspan=4)
228 # Sexto renglon
229 button_ni.grid(row=5, column=0)
230 k_sfs_entry.grid(row=5,column=1)
231 button_sfs.grid(row=5,column=2)
232 limpiar.grid(row=5,column=3)
233
234 root.mainloop()
```

Bibliografía

[1] Aharoni, R. y Berger, E.: *Menger's theorem for infinite graphs.* Inventiones Mathematicae, 176(1):1–62, 2009.

[2] Ali, S., Mathew, S. y Mordeson, J. N.: *Hamiltonian fuzzy graphs with application to human trafficking.* Information Sciences, 550:268–284, 2021.

[3] Attiya, H. y Welch, J.: *Distributed computing: fundamentals, simulations, and advanced topics*, vol. 19. John Wiley & Sons, 2004.

[4] Beineke, L. W. y Harary, F.: *The connectivity function of a graph.* Mathematika, 14(2):197–202, 1967.

[5] Bondy, J. A. y Murty, U. S. R.: *Graph theory with applications*, vol. 290. Macmillan London, 1976.

[6] Bonnet, É. y Cabello, S.: *The Complexity of Mixed-Connectivity.* arXiv preprint arXiv:2010.04799, 2020.

[7] Borndörfer, R. y Karbstein, M.: *A Note on Menger's Theorem for Hypergraphs.* 2012.

[8] Chartrand, G. y Zhang, P.: *Chromatic graph theory.* CRC press, 2019.

[9] Cheriyan, J., Kao, M. Y. y Thurimella, R.: *Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity.* SIAM Journal on Computing, 22(1):157–174, 1993.

[10] Cheriyan, J. y Maheshwari, S.: *Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs.* Journal of Algorithms, 9(4):507–537, 1988.

[11] Chunaev, P.: *Community detection in node-attributed social networks: a survey.* Computer Science Review, 37:100286, 2020.

[12] Dirac, G. A.: *In abstrakten Graphen vorhandene vollständige 4-Graphen und ihre Unterteilungen.* Mathematische Nachrichten, 22(1-2):61–85, 1960.

[13] Dong-jua, F. y Zhan-guob, X.: *3D Modeling and Motion Simulation of Table Vice Based on Inventor.* Mechanical Engineer, pág. 12, 2012.

[14] Egawa, Y., Kaneko, A. y Matsumoto, M.: *A mixed version of Menger's theorem.* Combinatorica, 11(1):71–74, 1991.

[15] Elmasry, A.: *Why depth-first search efficiently identifies two and three-connected graphs*. En *International Symposium on Algorithms and Computation*, págs. 375–386. Springer, 2010.

[16] Elmasry, A. y Tsin, Y. H.: *ON FINDING SPARSE THREE-EDGE-CONNECTED AND THREE-VERTEX-CONNECTED SPANNING SUBGRAPHS*. International Journal of Foundations of Computer Science, 25(03):355–368, 2014.

[17] Eppstein, D., Galil, Z., Italiano, G. F. y Nissenzweig, A.: *Sparsification—a technique for speeding up dynamic graph algorithms*. Journal of the ACM (JACM), 44(5):669–696, 1997.

[18] Erciyes, K.: *Distributed graph algorithms for computer networks*. Springer Science & Business Media, 2013.

[19] Esfahanian, A. H.: *Connectivity algorithms*. En *Topics in structural graph theory*, págs. 268–281. Cambridge University Press, 2013.

[20] Esfahanian, A. H. y Louis Hakimi, S.: *On computing the connectivities of graphs and digraphs*. Networks, 14(2):355–366, 1984.

[21] Even, S., Itkis, G. y Rajsbaum, S.: *On mixed connectivity certificates*. Theoretical computer science, 203(2):253–269, 1998.

[22] Even, S. y Tarjan, R. E.: *Network flow and testing graph connectivity*. SIAM Journal on Computing, 4(4):507–518, 1975.

[23] Frank, A., Ibaraki, T. y Nagamochi, H.: *On sparse subgraphs preserving connectivity properties*. Journal of Graph Theory, 17(3):275–281, 1993.

[24] Fülöp, O.: *Sparse graph certificates for mixed connectivity*. Discrete mathematics, 294(3):285–290, 2005.

[25] Gabow, H. N.: *A matroid approach to finding edge connectivity and packing arborescences*. Journal of Computer and System Sciences, 50(2):259–273, 1995.

[26] Galil, Z.: *Finding the vertex connectivity of graphs*. SIAM Journal on Computing, 9(1):197–199, 1980.

[27] Gazit, H. y Miller, G. L.: *An improved parallel algorithm that computes the BFS numbering of a directed graph*. Information Processing Letters, 28(2):61–65, 1988.

[28] Göring, F.: *Short Proof of Menger's Theorem*. Discrete Mathematics, 219(1-3):295–296, 2000.

[29] Henzinger, M. R., Rao, S. y Gabow, H. N.: *Computing vertex connectivity: new bounds from old techniques*. En *Proceedings of 37th Conference on Foundations of Computer Science*, págs. 462–471. IEEE, 1996.

[30] Hopcroft, J. E. y Tarjan, R. E.: *Dividing a graph into triconnected components.* SIAM Journal on Computing, 2(3):135–158, 1973.

[31] Jájá, J.: *PRAM (Parallel Random Access Machines).*, 2011.

[32] Johann, S. S., Krumke, S. O. y Streicher, M.: *On the Mixed Connectivity Conjecture of Beineke and Harary.* arXiv preprint arXiv:1908.11621, 2019.

[33] Kanevsky, A. y Ramachandran, V.: *Improved algorithms for graph four-connectivity.* Journal of Computer and System Sciences, 42(3):288–306, 1991.

[34] Karp, R. M.: *Reducibility among combinatorial problems.* En *Complexity of computer computations*, págs. 85–103. Springer, 1972.

[35] Liu, Y., Tipper, D. y Siripongwutikorn, P.: *Approximating optimal spare capacity allocation by successive survivable routing.* IEEE/ACM Transactions on Networking, 13(1):198–211, 2005.

[36] Mader, W.: *Connectivity and edge-connectivity in finite graphs.* En *Surveys in Combinatorics (Proceedings of the Seventh British Combinatorial Conference), London Mathematical Society Lecture Note Series*, vol. 38, págs. 66–95, 1979.

[37] Mathew, S. y Sunitha, M.: *A generalization of Menger's Theorem.* Applied Mathematics Letters, 24(12):2059–2063, 2011.

[38] Mathew, S. y Sunitha, M.: *A generalization of classical connectivity parameters for graphs.* World Applied Sciences Journal, 22:18–22, 2013.

[39] Matula, D. W.: *Determining edge connectivity in 0 (nm).* En *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, págs. 249–251. IEEE, 1987.

[40] Menger, K.: *Zur allgemeinen kurventheorie.* Fundamenta Mathematicae, 10(1):96–115, 1927.

[41] Nagamochi, H. y Ibaraki, T.: *A linear-time algorithm for finding a sparsek-connected spanning subgraph of ak-connected graph.* Algorithmica, 7(1-6):583–596, 1992.

[42] Nešetřil, J., Milková, E. y Nešetřilová, H.: *Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history.* Discrete Mathematics, 233(1-3):3–36, 2001.

[43] Paten, B., Diekhans, M., Earl, D., John, J. S., Ma, J., Suh, B. y Haussler, D.: *Cactus graphs for genome comparisons.* Journal of Computational Biology, 18(3):469–481, 2011.

[44] Salia, N.: *Graphs with Isomorphic DFS Spanning Trees.* Tesis de Maestría, Central European University, 2015.

[45] Tarjan, R. E.: *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, 1:146–160, 1972.

[46] Whitney, H.: *Congruent graphs and the connectivity of graphs*. En *Hassler Whitney Collected Papers*, págs. 61–79. Springer, 1992.