Tesis de Licenciatura:
Algoritmos de Reconocimiento de Grafos Arco Circulares

Ivo Koch
Director: Dr. Min Chih Lin - UBA
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

28 de Julio de 2006

Resumen: Los grafos arco circulares son grafos intersección de arcos alrededor de un círculo. En esta tesis repasamos los principales resultados conocidos sobre esta clase de grafos y algunas de sus subclases. Describimos las caracterizaciones matemáticas de cada clase, así como los algoritmos que reconocen si un grafo G pertenece a alguna de las subclases. Se implementaron además algoritmos para el reconocimiento de dos subclases importantes, los grafos arco circulares unitarios y arco circulares Helly. Se desarrolla además una sugerencia de Spinrad para mejorar a orden lineal uno de los algoritmos.
Palabras clave: algoritmo, grafo arco circular, grafo arco circular propio, grafo arco circular unitario, grafo arco circular Helly.

Índice general

Capítulo 1

Introducción

El grafo intersección de una familia de n conjuntos finitos no vacíos es el grafo donde cada vértice representa un conjunto y cada arista indica un par de conjuntos que se intersecan. Los grafos intersección han sido ampliamente estudiados. Podemos citar entre sus subclases los grafos triangulados, los grafos de intervalos, los grafos de comparabilidad y los grafos arco circulares. Estos últimos serán objeto de estudio de esta tesis.

Los grafos arco circulares son grafos intersección de arcos alrededor de un círculo. Fueron mencionados por primera vez en 1964 y Alan Tucker aportó la primera caracterización de éstos y algunas de sus subclases en la década del '70.

Llamamos reconocimiento de una clase de grafos al problema de decidir si un grafo G pertenece o no a esa clase. El objetivo de esta tesis es describir de manera organizada los algoritmos de reconocimiento más importantes para los grafos arco circulares y sus subclases más comunes, así como implementar los reconocimientos de dos de estas subclases: arco circular unitarios y arco circular Helly.

A continuación describimos la estructura del trabajo.

El capítulo 1 presenta la clase de grafos arco circulares. Revisamos los principales resultados conocidos sobre esta clase y sobre la subclase de los grafos arco circulares propios. Desarrollamos una sugerencia de Spinrad para reducir la complejidad cúbica del algoritmo de Alan Tucker para reconocer esta subclase a orden lineal.

En el capítulo 2 estudiamos los grafos arco circulares unitarios. Se reseñan los resultados más importantes y presentamos la primera implementación realizada enteramente sobre el algoritmo de reconocimiento de Durán, Gravano, McConnell, Spinrad y Tucker [7].

El capítulo 3 está destinado a los grafos arco circulares Helly. Repasamos los algoritmos de reconocimiento existentes y describimos aquí también la primera implementación del algoritmo de Lin y Szwarcfiter [19].

En los capítulos 2 y 3 presentamos aplicaciones originales para la clase tratada.

Finalmente desarrollamos en el capítulo 4 las conclusiones del trabajo.

Definiciones básicas y clases de grafos

Sea G un grafo, con $E(G)$ su conjunto de aristas y $V(G)$ su conjunto de vértices, $|V(G)| = n$ y $|E(G)| = m$. Dados $v, w \in V(G)$ notamos a la arista $e \in E(G)$ incidente a v y w como $e = (v, w)$. Si G es un digrafo, notamos al conjunto de aristas como $D(G)$ y cada arista $e = (v, w) \in D(G)$ está orientado de v a w. La vecindad de v será notada como $N(v), N(v) = \{w \in V(G), (v, w) \in E(G)\}$ y $N[v] = N(v) \cup \{v\}$. En el caso de digrafos, los vértices $w \in N(v), (w, v) \in D(G)$ constituyen el *conjunto de entrada* de v, y los vértices $u \in N(v), (v, u) \in D(G)$ constituyen el *conjunto de salida* de v.

Una *clique* es un conjunto maximal de vértices adyacentes de a pares.

Un ciclo es *inducido* si no posee cuerdas. Notamos C_k al ciclo inducido por k vértices.

$M(G)$ es la *matriz de adyacencia* de G, con $m_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E(G)$ y $m_{ij} = 0$ en otro caso. $M^*(G)$ es la matriz de adyacencia *aumentada*, $M^*(G) = M(G) + I$ con I la matriz identidad.

Una matriz M de ceros y unos tiene la propiedad de *unos consecutivos por columnas (filas)* si sus filas (columnas) pueden permutarse de manera tal que los unos en cada columna (fila) aparezcan de forma consecutiva. M tiene *unos circulares* si sus filas pueden permutarse de manera tal que los unos en cada columna aparezcan de forma consecutiva circular (como si la matriz estuviera escrita sobre un cilindro). Nos referiremos siempre a unos consecutivos y unos circulares por columnas, salvo expresa mención.

G es un grafo *cordal* si no contiene ciclos inducidos de longitud mayor que tres.

G es un grafo *de comparabilidad* si sus aristas pueden orientarse para producir un digrafo G' tal que $(x, y) \in D(G'), (y, z) \in D(G') \Rightarrow (x, z) \in D(G')$. Esta orientación se llama *transitiva*.

Sea $\pi = (\pi_1, \ldots, \pi_n)$ una permutación de números de 1 a n. Sea $G[\pi]$ el grafo de n vértices tal que dos vértices son adyacentes si y sólo si el de índice mayor está a la derecha del de índice menor. Decimos que G es un grafo *de permutación* si existe una permutación π tal que G es isomorfo a $G[\pi]$. Un grafo G es de permutación si y sólo si G y su complemento $\overline{G}$ son grafos de comparabilidad.

Un grafo G es *denso* si el número de aristas es cuadrático con respecto a su conjunto de vértices. Los algoritmos sobre grafos que insumen $O(n^2)$ se consideran lineales sobre grafos densos, ya que si m es cuadrático con respecto a n, entonces $O(n^2) = O(m + n)$.

Un grafo G es *arco circular(CA)* si es el grafo intersección de un conjunto de arcos alrededor de un círculo. En otras palabras, si se lo puede representar como un conjunto de arcos A alrededor de un círculo C, de manera tal que los vértices sean modelados como arcos y las aristas como pares de arcos que se intersecan. Al par (C, A) se lo llama *representación* o *modelo* de G. Definimos que los arcos deben ser abiertos y asumimos, sin pérdida de generalidad, que ningún par de arcos de A tiene un extremo común y ningún arco cubre el perímetro total de C. Escribimos cada arco A_i como $A_i = (s_i, t_i)$, donde s_i y t_i son los *extremos* de A_i; s_i es el *extremo inicial* y t_i el *extremo final*. Los

extremos de A son los extremos de todos los arcos de A. Cuando recorremos C en sentido horario, obtenemos un ordenamiento circular de los extremos de A. Además, consideraremos el ordenamiento circular de los arcos $A_1, \ldots, A_n$ que resulta del ordenamiento circular de sus sus extremos iniciales $s_1, \ldots, s_n$. Las sumas y restas de índices en estos ordenamientos deben entenderse módulo n.

Un grafo G es *de intervalos* si es el grafo intersección de un conjunto de segmentos sobre una recta.

Un *reconocimiento* es la posibilidad de poder determinar si un grafo G pertenece a una determinada clase. En ocasiones es importante contar además con una representación de la clase arco circular a la que pertenece el grafo porque el modelo será input para otros algoritmos que resuelven otros problemas y porque el modelo constituye una certificación de que el grafo es miembro de esa clase. Entonces, los algoritmos de reconocimiento pueden contestar TRUE o FALSE simplemente o además generar el modelo en el caso TRUE. Algunos algoritmos producen además un certificado en el caso FALSE, esto es, un subgrafo prohibido o cualquier propiedad en el grafo (demostrada previamente) que exhibe que no pertenece a la clase testeada. De aquí en más, aclararemos en todos los algoritmos cuáles de estos puntos cumplen.

Capítulo 2

Grafos arco circulares y arco circulares propios

La caracterización y el reconocimiento de grafos arco circulares constituían problemas abiertos desde 1964, cuando fueron planteados en [11]. Alan Tucker propuso una caracterización para estos grafos en el año 1970 [30]. Diez años después, Tucker encontró el primer algoritmo de reconocimiento de complejidad $O(n^3)$ [34]. Desde entonces, se ha dedicado mucho interés a esta clase de grafos, así como a las subclases que trataremos en este trabajo. El algoritmo de Tucker ha sido sucesivamente mejorado: Spinrad y Eschen reducen en [26] la complejidad a $O(n^2)$, atacando los cuellos de botella de la implementación cúbica de Tucker. Hsu [14] publicó un algoritmo $O(nm)$. McConnell encontró un algoritmo lineal de reconocimiento [22]. Más recientemente, Kaplan y Nussbaum [18] encontraron otro algoritmo de orden lineal más simple que el de McConnell, refinando el algoritmo cuadrático de Spinrad y Eschen. Salvo el algoritmo de Hsu, los demás siguen una misma dirección de trabajo; cada uno toma los resultados de sus antecesores y produce algoritmos más eficientes con las mismas ideas.

2.1. Aplicaciones

Los grafos arco circulares tienen aplicaciones en genética [27], control de tráfico [29], diseño de compiladores [33], estadística [17], y problemas de almacenamiento [2]. Más recientemente, podemos citar aplicaciones en modelado de redes de fibra óptica que utilizan WDM (Wave Division Multiplexing) [28] e inteligencia artificial [3].

2.2. Grafos arco circulares

2.2.1. Caracterizaciones

Revisamos aquí las tres caracterizaciones conocidas para este tipo de grafos. Las primeras dos son de Tucker y la tercera es de Gavril.

Teorema 2.1 *[30] Un grafo G es arco circular si y sólo si sus vértices pueden ser indexados circularmente $v_1, \ldots, v_n$ tal que para cada $i < j$, si $(v_i, v_j) \in E(G)$, entonces $v_{i+1}, \ldots, v_j \in N(v_i)$ o $v_{j+1}, \ldots, v_n, v_1, \ldots, v_i \in N(v_j)$.*

Sea ahora M una matriz de ceros y unos simétrica, con unos en la diagonal. Sea U_i el conjunto circular de 1s en la columna i, comenzando desde el uno de la diagonal y avanzando hacia abajo (y circularmente) tanto como sea posible. Sea V_i el conjunto análogo de unos en la fila i, comenzando desde el uno de la diagonal y avanzando hacia la derecha (y circularmente) tanto como sea posible. M tiene *unos cuasi circulares* si los U_is y V_is contienen todos los unos de la matriz.

Teorema 2.2 *[31] G es arco circular si y sólo si los vértices de G pueden numerarse de forma tal que $M^*(G)$ tiene unos cuasi circulares.*

Finalmente, veamos la caracterización de Gavril. Llamamos *sistema de cubrimiento* a un conjunto $C_1, C_2, \ldots, C_k$ de subgrafos completos de un grafo G si satisfacen:

- $V(G) = \bigcup_{i=1}^{k} V(C_i)$
- Si $i \neq j$ entonces $V(C_i)$ no está incluido en $V(C_j)$
- Para cada par de vértices adyacentes u, v, existe un conjunto C_i que los contiene.

Teorema 2.3 *[9] Un grafo G es arco circular si y sólo si tiene un sistema de cubrimiento $C_1, C_2, \ldots, C_k$ tal que la matriz M de $n \times k$, con $m_{ij} = 1 \Leftrightarrow v_i \in C_k$ y 0 en otro caso tiene la propiedad de unos circulares.*

Hasta el momento no ha podido encontrarse una caracterización de grafos arco circulares a través de subgrafos prohibidos.

2.2.2. Algoritmo de Tucker

Este fue el primer algoritmo de reconocimiento para esta clase de grafos, y tiene una complejidad de $O(n^3)$. Es reputado como particularmente difícil de implementar, en parte porque el algoritmo y las demostraciones son complejos y en parte porque ambas se presentan en varias partes del paper juntas. Tucker mismo reconoció la necesidad de simplificar su algoritmo. Textualmente del paper: *"The real difficulty in our algorithm is not its speed but its length"*. Presentamos a continuación el algoritmo:

Algoritmo 2.1 *Algoritmo de reconocimiento de grafos CA de Tucker*

1. *Preprocesar el grafo el grafo input G de n vértices de manera tal de eliminar los vértices v, w de G tales que $N[v] = N[w]$ y los vértices u de grado $n-1$. Estos vértices pueden agregarse luego trivialmente al modelo si G es CA.*

2. *Clasificar G en alguno de los siguientes casos:*

- *Caso I. $\overline{G}$ es bipartito, o equivalentemente, G puede cubrirse con dos cliques*
- *Caso II. $\overline{G}$ tiene un ciclo inducido de longitud impar.*

  Subcaso IIa. $\overline{G}$ contiene un triángulo, o equivalentemente, G tiene 3 o más vértices no adyacentes de a pares.

  Subcaso IIb. $\overline{G}$ tiene un odd hole*, un ciclo inducido de longitud* ≥ 5.

  */**Nota: Estos subcasos del caso II no son excluyentes.*/*

3. *Ejecutar ahora para G los siguientes tres procedimientos. La forma en que se implementa cada uno dependerá del caso en que se ubique G en el paso anterior.*

   - *Procedimiento 1: Encontrar un subconjunto especial de m vértices cuyos arcos asociados pueden ser posicionados fácilmente en el círculo. Los extremos de estos m arcos dividen el círculo en 2m sectores, tales que ninguno de los arcos que faltan ubicar pueda tener ambos extremos en el mismo sector.*
   - *Procedimiento 2: Los extremos de los arcos restantes son ubicados en sectores adecuados.*
   - *Procedimiento 3: El conjunto de extremos en cada sector es ordenado correctamente. Con esto termina la generación del modelo CA de G.*

Si G no es arco circular, o bien la construcción del modelo termina anticipadamente o bien el modelo que se obtiene como output no representa al grafo G.

Uno de los cuellos de botella de la implementación que da Tucker es la fase de preprocesamiento donde se identifican las inclusiones de vecindades de los vértices en el paso 1 del algoritmo. Además, Tucker aplica su algoritmo recursivamente sobre un tipo particular de grafos, en el Procedimiento 2 y 3 de los Subcasos IIa y IIb, y esta recursión también lleva a un orden cúbico.

Complejidad y alcance del algoritmo

La complejidad es $O(n^3)$. El algoritmo genera un modelo arco circular del grafo G, pero no un certificado en caso de que G no sea arco circular.

2.2.3. Algoritmo de Eschen y Spinrad

Este algoritmo se concentra específicamente en las dos secciones más ineficientes del algoritmo de Tucker:

Relaciones de inclusión: Los autores muestran cómo pueden computarse las relaciones de inclusión de vecindades en $O(n^2)$. Para ello, construyen a partir de G cuatro grafos tales que si G es CA, entonces cada uno de estos grafos es de intervalos o bipartito cordal. Estos grafos se construyen de manera tal que $N(v)$ contiene a $N(w)$ en G sí y sólo si $N(v)$ contiene a $N(w)$ en cada uno

de estos grafos. Como es posible computar la inclusión de vecindades en grafos de intervalos o bipartitos cordales en $O(n^2)$, se logra el orden cuadrático [26].

Estructura recursiva: En el Caso I del algoritmo (G puede cubrirse con dos cliques) podemos usar el mismo procedimiento del caso anterior para determinar en $O(n^2)$ todos los pares de arcos que pueden cubrir el círculo en un modelo de G. Esto mejora a $O(n^2)$ un resultado anterior de Spinrad [25], que a su vez simplifica el Caso I del algoritmo. Con esto se obtiene una implementación cuadrática (no recursiva) del Caso I.
Para implementar los casos IIa y IIb en $O(n^2)$, Eschen y Spinrad modifican la estructura recursiva del algoritmo de Tucker. Los autores muestran cómo implementar el algoritmo de manera tal que cada llamado recursivo sea sobre un grafo que se puede cubrir con dos cliques (Caso I), con lo cual no se producen más llamados recursivos. Se explica cómo la complejidad cuadrática sigue a partir de que la suma de los tamaños de los grafos es proporcional al tamaño de G.

Complejidad y alcance del algoritmo

La complejidad es $O(n^2)$. El algoritmo genera un modelo CA del grafo G, pero no un certificado en caso de que G no sea arco circular.

2.2.4. Algoritmo de McConnell

El algoritmo de McConnell toma elementos de los papers de Tucker, Hsu y Eschen y Spinrad. McConnell demuestra que se puede determinar en tiempo lineal si un modelo r es la representación de un grafo G. El algoritmo tiene como precondición que el grafo sea arco circular y se comporta de la siguiente manera:

1. Produce un modelo r para G, tal que r es una representación de G sii G es CA.

2. Si G no es CA está garantizado que el algoritmo de todas maneras se detiene, en tiempo lineal [1]

Entonces tenemos un algoritmo de reconocimiento: verificamos el output r. Si el output es vacío el grafo no es CA. En caso de que exista determinamos en tiempo lineal si r es modelo de G.

El algoritmo reduce el problema al reconocimiento de grafos de intervalos, especificando tipos particulares de intersección entre los intervalos. Se utiliza la misma fase de preprocesamiento de Eschen y Spinrad para computar las relaciones de inclusión entre vecindades. Para lograr el orden lineal, McConnell refina el análisis de Eschen y Spinrad: muestra que el preprocesamiento puede implementarse en orden lineal porque solamente interesan las relaciones de inclusión entre vértices adyacentes, y los grafos cordales bipartitos asociados (del algoritmo de Eschen y Spinrad) no pueden ser entonces muy grandes.

[1] Existe una constante c tal que el algoritmo se detiene antes de $c(n + m)$ operaciones.

La parte más complicada del algoritmo de McConnell es encontrar una partición de V(G) en conjuntos de vértices llamados Δ módulos. Estos se usan para convertir el grafo input en un grafo de intervalos con tipos específicos de intersecciones entre los intervalos, y para encontrar una representación para este grafo de intervalos. McConnell presenta primero una implementación que es $O(m + n\mathbf{log}n)$. Se alcanza el orden lineal usando una adaptación del algoritmo lineal para hallar una orientación transitiva [23].

Complejidad y alcance del algoritmo

El algoritmo tiene una complejidad de $O(n + m)$. En el caso de que G sea arco circular genera un modelo. No se producen certificados de que el grafo no es arco circular. Notar que en general los certificados se basan en subgrafos prohibidos, y para grafos arco circulares no existe aún una caracterización en base a estos subgrafos.

2.2.5. Algoritmo de Kaplan y Nussbaum

Kaplan y Nussbaum dan una implementación lineal del algoritmo de Eschen y Spinrad, más sencilla que la de McConnell. El algoritmo primero busca un conjunto independiente de tamaño 3. Si lo encuentra, se aplica el Subcaso IIa del algoritmo. Si no lo encuentra, concluye que el grafo tiene $\Theta(n^2)$ aristas, con lo cual el algoritmo de Eschen y Spinrad corre en tiempo lineal.

Eschen y Spinrad encuentran en el Subcaso IIa un conjunto independiente maximal y ubican los arcos correspondientes en el círculo. Se muestra cómo hacer esto en tiempo lineal. La implementación sigue luego a grandes rasgos la de Eschen y Spinrad, donde el aporte realmente novedoso es ajustar la cota de la complejidad, mostrando que el orden para esta parte es lineal, en lugar de cuadrático como propusieron estos autores. Se demuestra que cada subgrafo considerado por el algoritmo al ubicar y ordenar los arcos sobre el círculo es denso, con lo cual podemos afirmar que el algoritmo es lineal. Además, la suma de los tamaños de estos subgrafos es lineal con respecto al tamaño del grafo input.

Se mantiene la propuesta de McConnell con respecto al cálculo lineal de las relaciones de inclusión. A su vez, también se mantiene el paso final del algoritmo de McConnell, la verificación lineal del output para comprobar si efectivamente es un modelo del grafo input.

Complejidad y alcance del algoritmo

El algoritmo tiene una complejidad de $O(n + m)$. En el caso de que G sea arco circular genera un modelo. No se producen certificados de que el grafo no es arco circular.

2.3. Grafos arco circulares propios

Un grafo G es *arco circular propio(PCA)* si existe una representación arco circular de G tal que ningún arco esté contenido en forma propia en otro.
Tucker caracterizó por primera vez estos grafos en [31] y propuso un algoritmo de reconocimiento de complejidad $O(n^3)$ usando matrices de representación. Este algoritmo plantea dos casos, dependiendo de si los vértices de G pueden cubrirse con dos cliques o no.

Un grafo G es *de intervalos propio(PIG)* si existe una conjunto de segmentos sobre una recta que representa a G tal que ningún segmento está contenido en forma propia en otro. Citamos finalmente el algoritmo lineal hallado por Deng, Hell y Huang [5], que se aparta de la estructura del algoritmo de Tucker. Este algoritmo reconoce primero si G es un grafo de intervalos propio. Si G no es PIG, el algoritmo efectúa el reconocimiento usando una caracterización de los grafos PCA en base a la posibilidad de orientar sus aristas de una manera particular.

2.3.1. Caracterizaciones matemáticas

Una matriz M de $n \times n$ simétrica es de *unos compatibles* si los unos en cada columna son circulares y si invertimos o permutamos cíclicamente el orden de las filas y correspondientes columnas, el último 1 en orden circular descendente en la segunda columna siempre está a igual altura o por debajo del último 1 en la primera columna (salvo que en alguna columna todos los elementos sean unos o ceros).

El siguiente teorema es la primera caracterización de grafos PCA de Tucker:

Teorema 2.4 *[31] Un grafo G es arco-circular propio si y sólo si existe un arreglo de $M^*(G)$ con unos compatibles.*

Dado un grafo G, llamamos $G^* = G \cup \{v\}$, donde v es un vértice aislado que se agrega al grafo G.

Tucker también presentó otra caracterización en base a subgrafos prohibidos:

Teorema 2.5 *[32] Un grafo G es arco-circular propio si y sólo si:*

- *No contiene como subgrafos inducidos a H_1^* ni a $C_k^*, k \geq 4$ y*
- *$\overline{G}$ no contiene como subgrafos inducidos a H_1, H_2, H_3, H_4, H_5 ni a $C_{2j}, j \geq 3$ ni a $C_{2j+1}^*, j \geq 1$*

Una *enumeración circular* de un grafo orientado D es un orden circular $v_1, v_2, \ldots, v_n$ de sus vértices tal que para cada i existen enteros no negativos k, l tal que el vértice v_i tiene conjunto de entrada $v_{i-1}, v_{i-2}, \ldots, v_{i-k}$ y conjunto de salida $v_{i+1}, v_{i+2}, \ldots, v_{i+l}$. Las sumas y restas de índices deben entenderse módulo n. Un grafo orientado que admite una enumeración circular se llama *grafo ronda*. Se dice que un grafo no dirigido G tiene una *orientación circular* si admite una orientación que lo convierte en un grafo dirigido ronda.

Finalmente, mencionamos la caracterización de Deng et al.

Teorema 2.6 *[5] Un grafo G es arco-circular propio si y sólo si posee una orientación circular.*

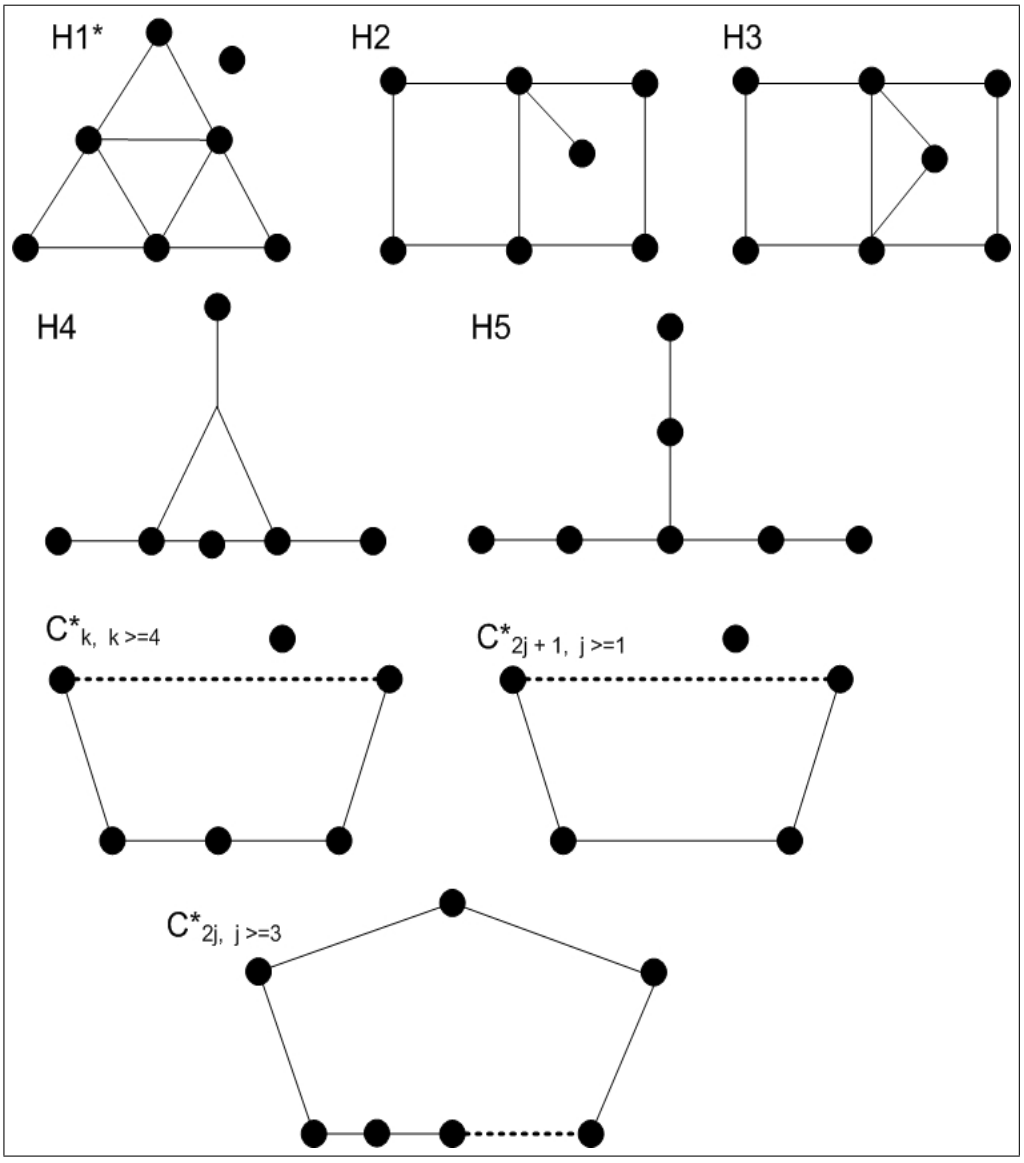

Figura 2.1: Subgrafos prohibidos de la caracterización de Tucker de grafos PCA

2.3.2. Algoritmo de Tucker

El algoritmo de Tucker para reconocimiento de grafos PCA utiliza una rutina que verifica si una matriz M de unos y ceros tiene la propiedad de unos consecutivos, y en este caso devuelve la matriz permutada de forma que los unos sean consecutivos. Esta rutina se debe a Fulkerson y Gross y fue publicado en [8]. El reconocimiento de la propiedad de unos circulares puede reducirse al reconocimiento de unos consecutivos, como demuestra Tucker en el siguiente teorema:

Teorema 2.7 *[31] Sea M_1 una matriz de ceros y unos. Formar la matriz M_2 intercambiando los ceros por unos y viceversa de aquellas columnas con un 1 en la primera fila de M_1. Entonces M_1 tiene la propiedad de unos circulares sii M_2 tiene la propiedad de unos consecutivos.*

Presentamos a continuación el algoritmo.

Algoritmo 2.2 *Algoritmo de reconocimiento de grafos PCA de Tucker*

1. *Si el grafo input G se puede cubrir con dos cliques, entonces*
2. *Obtener la matriz complemento M^c intercambiando 0s y 1s de $M^*(G)$ y verificar utilizando el algoritmo de Fulkerson Gross si las filas de M^c se pueden permutar de forma de obtener una matriz con unos consecutivos y unos compatibles simultáneamente. G es PCA sii esto se cumple.*
3. *Sino testear si $M^*(G)$ posee la propiedad de unos circulares. G es PCA sii esto se cumple.*

El paso 1 es equivalente a verificar que $\overline{G}$ sea bipartito.

En el paso 2 se debe verificar la propiedad de unos compatibles en la matriz. Para ello, Tucker menciona que basta con una modificación al algoritmo de Fulkerson y Gross.

El paso 3 puede resolverse más fácilmente y se justifica con el Teorema 2.4 y con el siguiente lema, también de Tucker.

Lema 2.1 *Sean los vértices de G indexados de tal forma que $M^*(G)$ tenga 1-circulares. Si $M^*(G)$* no *es de 0s-consecutivos o G* no *puede cubrirse con dos cliques, entonces este arreglo tiene unos compatibles.*

Nostros buscamos comprobar si $M^*(G)$ tiene la propiedad de unos compatibles (porque con esto sabemos si es G es PCA). Como estamos en el caso que G *no* se puede cubrir con dos cliques, forzosamente si $M^*(G)$ tiene unos circulares este arreglo tendrá también unos compatibles por el lema anterior. Si $M^*(G)$ no tiene la propiedad de unos circulares, tampoco tendrá la de unos compatibles, por definición de unos compatibles. La propiedad de unos circulares se verifica con el algoritmo de Fulkerson y Gross.

Complejidad y alcance del algoritmo

El algoritmo de Tucker es $O(n^3)$, y el procedimiento de Fulkerson y Gross es el cuello de botella del algoritmo. El algoritmo genera modelo pero no certificados en caso de que el grafo no sea PCA.

Desarrollo teórico de mejoras al algoritmo de Tucker

Es interesante notar las formas en que puede mejorarse este algoritmo. Primero, el procedimiento de Fulkerson y Gross es $O(n^3)$, si conseguimos mejorar esto todo el algoritmo será más eficiente. Segundo, en el paper no se explica claramente cómo modificar el algoritmo de Fulkerson y Gross para que verifique la propiedad de unos compatibles. Tercero, en los últimos años el problema de los unos consecutivos ha recibido cierta atención y han aparecido nuevas formas de resolverlo. Y finalmente, los dos casos del algoritmo pueden desacoplarse y resolverse por separado. Con las sucesivas mejoras que describiremos a continuación, puede obtenerse una implementación en orden lineal del algoritmo PCA de Tucker.

Es posible verificar si G se puede cubrir con dos cliques en orden lineal, en el paso 1 [21].

Para el paso 2, podemos utilizar algoritmos $O(n^2)$, porque el grafo es denso (dado que puede cubrirse con dos cliques). Spinrad sugiere un algoritmo en [24], que terminamos de desarrollar en este trabajo.

Para el paso 3, pueden utilizarse tres resultados que resuelven el problema de unos consecutivos en forma lineal: Una estructura de datos denominada PQ-Tree, publicada en [1], otra estructura que resulta de una simplificación de esta última llamada PC-Tree [16] y finalmente un paper [15] que resuelve el problema usando matrices y estructuras más convencionales.

Las siguientes dos subsecciones describen brevemente estas mejoras y nuevos resultados.

El problema de los unos consecutivos

Los PQ-Trees de Booth y Lueker fueron inventados para representar todas las posibles permutaciones de un conjunto U que sean consistentes con una restricción dada por una colección $C_1, C_2, \ldots, C_k$ de subconjuntos de U. La restricción consiste en los elementos de cada subconjunto C_i deben ocurrir consecutivos en las permutaciones.

Un PQ-Tree es un árbol T con dos tipos de nodos internos: P y Q, representados respectivamente por círculos y rectángulos. Las hojas de T se corresponden uno a uno con las filas de la matriz, en nuestro problema.

La Fig. 2.2 muestra un ejemplo de una matriz que satisface la propiedad de unos consecutivos, representada como un PQ-Tree. El algoritmo de Booth y Lueker considera cada columna como una restricción y las agrega de a una por vez. En cada iteración, se intenta agregar una columna. Llamamos *permutación consistente* a la permutación de filas en la matriz que se obtiene de ordenar las hojas del árbol de izquierda a derecha. El algoritmo modifica el PQ-Tree actual T para satisfacer la restricción de unos consecutivos para cada columna u que

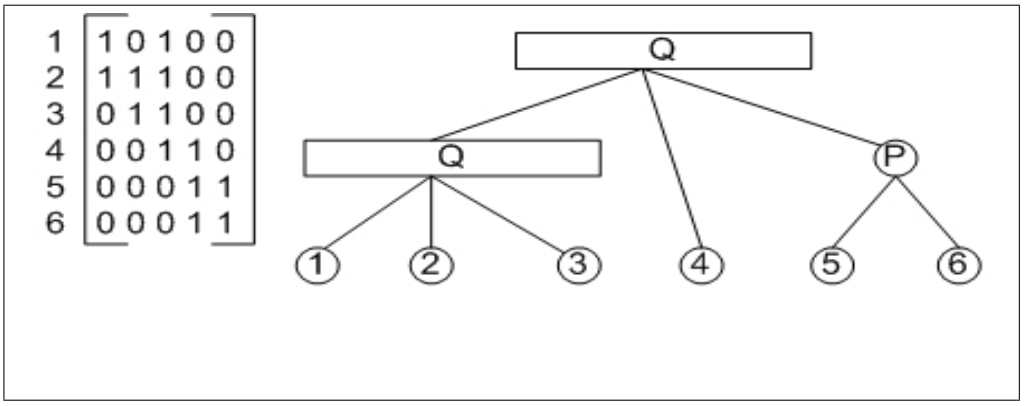

Figura 2.2: PQTree que representa un arreglo de filas con unos consecutivos

se agrega, de forma tal que las permutaciones consistentes del nuevo PQ-Tree T' sean exactamente las permutaciones consistentes de T en que las hojas (filas) que tienen un 1 en u ocurran de manera consecutiva. Finalmente, el PQ-Tree resultante representa todos los posibles ordenamientos de unos consecutivos que tiene la matriz, efectuando las siguientes dos operaciones sobre el árbol:

- Permutar arbitrariamente los hijos de un nodo P.
- Invertir el orden de los hijos de un nodo Q.

El procedimiento de actualización se basa en aplicar *templates* a los nodos de T, donde cada template se compone de un *patrón* y un *reemplazo*. Cuando un nodo cumple la condición de un patrón, se reemplaza por la estructura definida por el reemplazo. Existen nueve templates posibles, que se verifican nodo por nodo de forma bottom up. En la Fig. 2.3 pueden apreciarse dos templates con sus correspondientes patrones y reemplazos.

Este algoritmo se considera notoriamente difícil de programar.

La estructura análoga llamada PC-Tree de Hsu y McConnell representa todos los posibles ordenamientos de unos circulares (ya no consecutivos) de una matriz de 0s y 1s. Estos árboles no tienen raíz, a diferencia de los PQ-Trees, y se componen de nodos P (nodos internos, en negro) y nodos C (nodos con doble círculo).

La Fig.2.4 ilustra un PC-Tree y la forma en que representa los arreglos de unos circulares de una matriz. Las hojas son las filas de la matriz, y están dispuestas sobre el círculo. Los nodos C tienen un orden cíclico de sus hijos que puede invertirse, de sentido horario a antihorario o viceversa. A esta operación de inversión la llamamos *flip*. Se pueden pensar como monedas con aristas adosadas a los costados, y pueden colocarse en el árbol de cualquiera de los dos lados de la moneda. Los hijos de los nodos P no tienen ningún orden cíclico en particular y se pueden permutar arbitrariamente. Modificando el PC-Tree según estas reglas, pueden generarse todas los arreglos de unos circulares de la matriz, que se leerán del orden en que figuran las hojas (filas) en el círculo externo. La construcción del árbol se realiza de manera incremental, asegurando que el paso i del algoritmo modifica el árbol para que sea correcto (represente

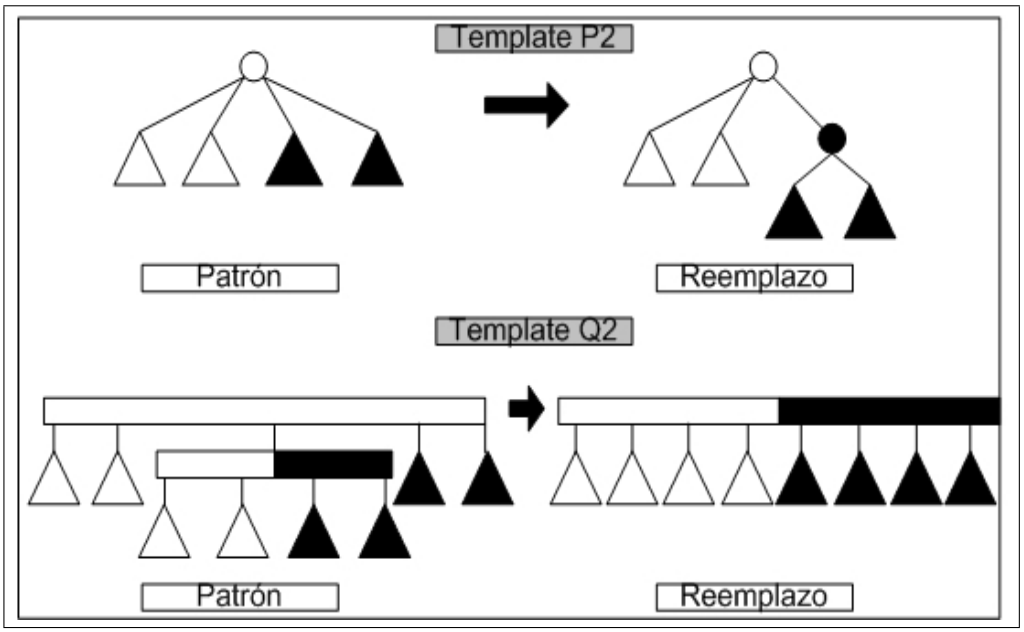

Figura 2.3: Dos templates del algoritmo de PQTrees

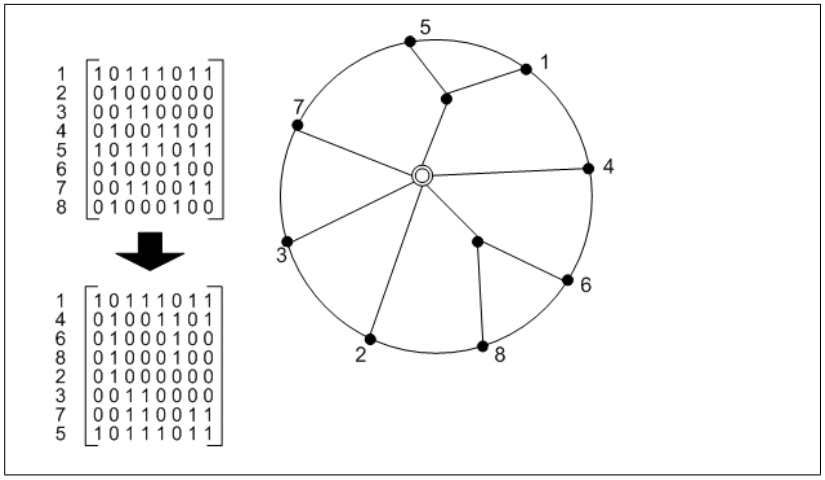

Figura 2.4: PCTree con la matriz antes y después de aplicar uno de los ordenamientos que surge del PCTree. Notar que la segunda matriz tiene unos circulares.

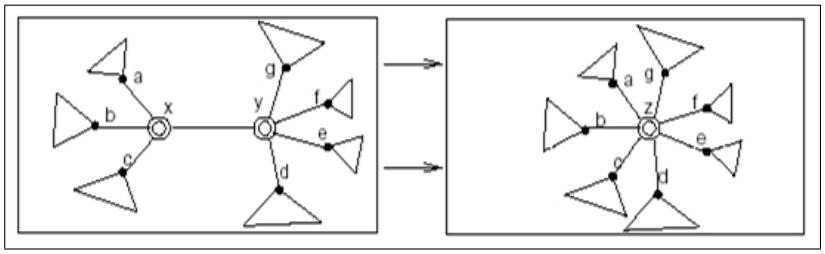

Figura 2.5: Operación de contracción de una arista en un PCTree. La idea es preservar el orden cíclico de los vecinos de x e y alrededor de la arista.

ordenamientos de unos circulares) para la submatriz que representan las primeras i columnas de la matriz input. El algoritmo se basa en la observación clave de que en una matriz con unos circulares, las aristas que deben actualizarse en cada iteración se hallan sobre un camino, que llamamos *camino terminal*. Llamamos hojas *llenas* con respecto a i a las hojas que corresponden a filas que tienen unos en la columna i (la que estamos procesando en ese momento) y *vacías* a las que corresponden a filas con ceros en la columna i. Una operación de *contracción* sobre una arista se muestra en la Fig. 2.5.

El algoritmo puede resumirse de la siguiente manera:

Algoritmo 2.3 *Construcción del PC-Tree. El PC-Tree inicial es un nodo P que es adyacente a todas las hojas, que son las filas de la matriz, que permite todas las permutaciones posibles.*
Para cada columna i hacer:

1. *Encontrar el camino terminal*

2. *Para los nodos del camino terminal, realizar flips sobre los nodos C y modificar el orden de los hijos de los nodos P tal que todos los vecinos de los nodos del camino terminal que llevan a hojas llenas estén de un mismo lado del camino terminal.* [2]

3. *Dividir cada nodo k en el camino en dos nodos, uno adyacente a las aristas de k que llevaban a hojas llenas y otro adyacente a las aristas de k que llevaban a hojas vacías.*

4. *Eliminar las aristas del camino y reemplazarlos por un nuevo nodo C llamado x cuyo orden cíclico preserve el orden de los nodos en el camino terminal.*

5. *Contraer las aristas de x a sus vecinos que sean nodos C, y cualquier nodo que tenga solamente dos vecinos.*

La Fig.2.6 ilustra cada paso del algoritmo.

Presenta sobre los PQ-Trees las siguientes ventajas:

- No utiliza templates complicados

[2] Esto implica claramente que los vecinos de los nodos del camino terminal que llevan a hojas vacías estarán del otro lado del camino terminal.

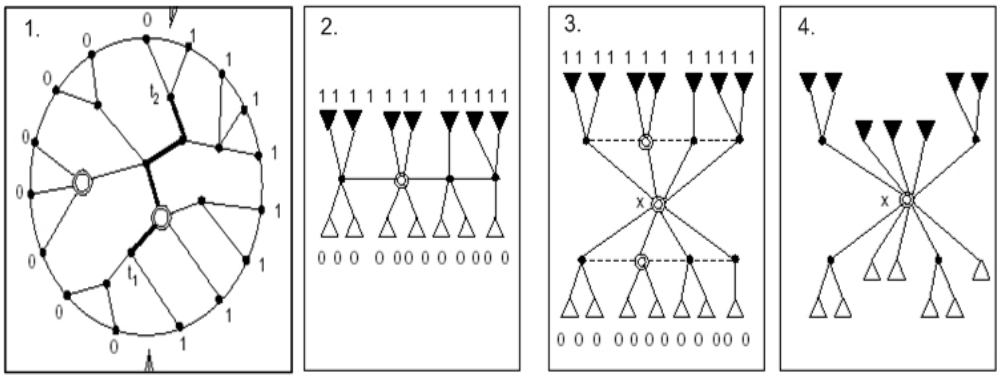

Figura 2.6: Algoritmo de unos circulares utilizando PCTrees. Los nodos que llevan a hojas llenas están marcados con unos y pintados de negro y los que llevan a hojas vacías están marcados con ceros, pintados de blanco.

- En cada iteración, modifica en un paso el árbol, sin necesidad de actualizaciones que se basan en un approach de comparación con estructuras patrón, nodo por nodo y bottom up.
- Es una estructura más natural para la verificación de unos circulares en matrices.

Citamos finalmente el resultado de Hsu [15], que resuelve el problema directamente utilizando la matriz de ceros y unos. Conceptualmente, el algoritmo se basa en una descomposición de la matriz en subconjuntos especiales de columnas, cada uno de los cuales admite esencialmente un único ordenamiento de filas que realiza la propiedad de unos consecutivos. Si cada uno de estos subconjuntos admite la propiedad de unos consecutivos, entonces la matriz también la admite. La implementación es complicada, pero lo interesante del paper es que solamente se utilizan estructuras de datos convencionales, a diferencia de los resultados anteriores.

Estos tres algoritmos para el cómputo de unos consecutivos pueden implementarse en orden lineal.

El problema de las dos cliques

Recordemos que el caso en que G se puede cubrir con dos cliques es el más complicado del algoritmo PCA de Tucker. Además, Tucker menciona que para procesar este caso basta una modificación al algoritmo de Fulkerson Gross, pero no da los detalles en su paper. Como el grafo es denso, la solución puede ser de orden cuadrático. Spinrad demostró en [24] que el problema en este caso es equivalente a verificar si $\overline{G}$ bipartito es un grafo de permutación. Utilizaremos para ello un resultado de Hell y Huang, que demostraron en [12] que los grafos bipartitos de permutación coinciden exactamente con otra clase de grafos, los bigrafos propios de intervalos, para cuyo reconocimiento publicó además un algoritmo en [13]. Un grafo bipartito G, con partición (X, Y) es un *bigrafo de*

intervalos propio si existe una familia F de intervalos $I_v, v \in (X \cup Y)$ tal que para todo $x \in X, y \in Y, xy$ son adyacentes en G sii I_x, I_y se intersecan y tal que ningún intervalo contiene a otro.

El algoritmo se presenta a continuación:

Algoritmo 2.4 *Algoritmo de reconocimiento de grafos PCA para el caso de que G se cubra con dos cliques*

1. *Calcular el complemento $\overline{G}$ de G.*
2. *Verificar utilizando el algoritmo 2.7 si $\overline{G}$ es bigrafo de intervalos propio.*
3. *Si lo es, devolver que G es PCA.*
   Sino, devolver que G no es PCA.

El algoritmo de Hell y Huang es lineal. El cuello de botella entonces para el reconocimiento PCA en este caso estriba en computar el complemento de G. De todas formas, como el grafo es denso, podemos utilizar un orden cuadrático sin comprometer el orden lineal total del reconocimiento PCA.

Describimos aquí finalmente el algoritmo de reconocimiento de bigrafos propios de intervalos de Hell y Huang. Se basa en un resultado anterior de Corneil, quien utilizó por primera vez el recorrido LexBFS. Ilustramos primero el recorrido LexBFS.

Un orden lexicográfico es el orden del diccionario. Esto quiere decir que por ejemplo $786 < 79$ y $658 < 6582$. La idea del LexBFS es ir numerando los vértices y elegir siempre el de número lexicográficamente mayor como siguiente. Si hay más de un vértice con el mismo número se elige uno al azar.

Algoritmo 2.5 *Procedimiento LexBFS(G, u):*
Input: un grafo conexo $G = (V, E)$ y un vértice distinguido u
Output: un ordenamiento de los vértices de G, determinado por los números de $\sigma(v)$

1. *label*$(u) \leftarrow |V|$
2. *Para cada vértice $v \in V - \{u\}$ hacer*

   Poner label$(v) \leftarrow \Lambda$ *(cadena vacía)*
3. *Para $i = |V|$ hasta 1 hacer*

   Elegir un vértice no numerado v con la mayor etiqueta en orden lexicográfico.($\star$)
   Poner $\sigma(v) \leftarrow |V| + 1 - i$ (numerar a v como $|V| + 1 - i$)
   Para cada vértice no numerado $w \in N(v)$ hacer
   Concatenar i al final de label(w)

El siguiente procedimiento fija una regla para resolver el caso de que varios vértices tengan el mismo número ("empaten"en la elección), en lugar de que sea una elección arbitraria como antes.

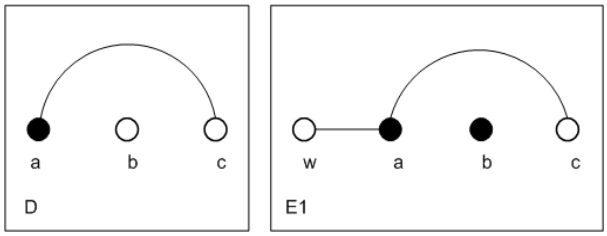

Figura 2.7: Estructuras prohibidas D y E1. Sea un coloreo del grafo bipartito input. Los círculos negros y blancos representan vértices coloreados. En D, los vértices b y c están a la misma distancia d del vértice distinguido del que partimos en el recorrido, mientras que a está a una distancia $d-1$. En E1, los vértices a y b están a la misma distancia e del vértice distinguido y c y w están a una distancia $d+1$.

Algoritmo 2.6 *Procedimiento LexBFS+(G, σ) Sea σ un ordenamiento de vértices obtenido de un recorrido anterior de LexBFS. En este procedimiento, en el paso $\star$, sea S el conjunto de vértices con la mayor etiqueta en orden lexicográfico. Escoger v como el vértice en S que aparece último en σ.*

Se muestra ahora el algoritmo de reconocimiento en sí:

Algoritmo 2.7 *Algoritmo de reconocimiento de bigrafos de intervalos propios. El input es un grafo bipartito G.*

1. *Realizar un recorrido LexBFS arbitrario, resultando en un arreglo σ.*
2. *Realizar un recorrido LexBFS+(G, σ), resultando en un arreglo σ.*
3. *Testear si σ^+ posee dos estructuras prohibidas llamadas D o $E1$ (Fig. 2.7). Si las posee, devolver que G no es bigrafo de intervalos propio. Sino seguir.*
4. *Realizar un recorrido LexBFS+(G, σ^+), resultando en un arreglo σ^{++}.*
5. *Testear si σ^{++} posee las estructuras prohibidas D o $E1$. Si las posee, devolver que G no es bigrafo de intervalos propio. Sino devolver que G sí lo es.*

2.3.3. Algoritmo de Deng et al.

El algoritmo se basa totalmente en caracterizaciones nuevas de grafos PCA y PIG basadas en la posibilidad de orientar a estos grafos de alguna forma particular. El algoritmo consiste en orientar determinados subgrafos de G con características especiales, para finalmente consolidar todas esas orientaciones y obtener una orientación D de G que sea un grafo ronda. Sabemos que G es PCA sii admite una orientación que lo transforma en un grafo dirigido ronda (Teorema 2.6). Si G es PCA, se ejecutan todos los pasos del algoritmo y se utiliza un procedimiento para convertir a D en un modelo arco circular de G.

Si G no es PCA, cuando no cumple alguna de las condiciones necesarias que se testean en cada paso, el algoritmo termina. El algoritmo utiliza además como subrutina un procedimiento lineal para reconocer grafos PIG, también basado en orientaciones.

Un *torneo* es una orientación de un grafo completo.

Un grafo dirigido es *transitivo* si admite una orientación transitiva.

Un grafo no dirigido G en el que solamente algunos de sus aristas han sido orientadas se denomina *grafo mezclado*.

Sea la relación de equivalencia en $V(G)$ en la que v y w están relacionados sii $N[v] = N[w]$. Las clases de equivalencia de esta relación se llaman *bloques* de G.

Una *enumeración lineal* de un grafo orientado D es un orden lineal $v_1, v_2, \ldots, v_n$ de sus vértices tal que para cada i existen enteros no negativos k, l tal que el vértice v_i tiene conjunto de entrada $v_{i-1}, v_{i-2}, \ldots, v_{i-k}$ y conjunto de salida $v_{i+1}, v_{i+2}, \ldots, v_{i+l}$. Un grafo orientado que admite una enumeración lineal se llama *grafo lineal*. Se dice que un grafo no dirigido G tiene una *orientación lineal* si admite una orientación que lo convierte en un grafo lineal.

Un *grafo lineal mezclado* se obtiene a partir de un grafo lineal D reemplazando todos los arcos (v, w) tales que v y w pertenecen al mismo bloque por aristas no dirigidas.

Una *enumeración lineal* de los bloques de G es un ordenamiento de los bloques en el orden de la enumeración lineal de los vértices correspondientes de S.

Una *inversión completa* de un grafo dirigido D es la operación que consiste en invertir la orientación de todos las aristas de D.

Un *grafo ronda mezclado* se obtiene a partir de un grafo ronda D reemplazando todos los arcos (v, w) tales que v y w pertenecen al mismo bloque por aristas no dirigidas.

A continuación resumimos el algoritmo:

Algoritmo 2.8 *Algoritmo de Deng et al. de reconocimiento de grafos PCA*

1. *Verificar si G es PIG. Si lo es, devolver su representación de intervalos como un caso especial de grafos arco circulares. Sino devolver que G no es PCA.*

2. *Elegir un vértice x de mínimo grado en G. Sea A el grafo inducido por $N[x]$ y sea $B = G - A$. Si B es una clique, encontrar una representación de G por medio del algoritmo de Tucker. Sino seguir.*

3. *Si B no es PIG, devolver que G no es PCA. Sino orientar a B como un grafo mezclado lineal B', donde $B_1, \ldots, B_q$ es una enumeración lineal de los bloques de B'.*

4. *Sea L el conjunto de vértices de A adyacentes a un vértice de B_q y sea R el conjunto de vértices de A adyacentes a un vértices de B_1. Sea $C = G - L, D = G - R$ y sea E el subgrafo de G inducido por $A \cup B_q$. Si*

alguno de los grafos C, D, E no son PIG, devolver que G no es PCA. Sino orientar C, D, E como grafos mezclados lineales C', D', E'.

5. *Si no es posible hacer inversiones completas sobre C', D', E', tal que todas las aristas estén orientadas de manera consistente (es decir, si una arista está orientada en más de un grafo, debe estar orientada de la misma manera en cada uno), devolver que G no es PCA. Sino encontrar orientaciones consistentes C', D', E' y construir H de G orientando cualquier arista uv tal que $u \to v$, si $u \to v$ existe en alguno de los grafos mezclados C', D', E'.*

6. *Orientar todas las aristas no dirigidas que queden en H tal que cada bloque de H se convierta en un torneo transitivo. Sea D el grafo dirigido resultante.*

7. *Transformar D en una representación arco circular propia.*

En el paso 2, se utiliza el algoritmo de Tucker (mejorado con las estructuras que explicamos antes) porque si B es completo y x de mínimo grado, entonces m es cuadrático con respecto a n, entonces $O(n^2) = O(m + n)$ y el algoritmo cuadrático de Tucker es en realidad lineal.

El paso 3 es posible porque Deng et al. demuestran que los grafos PIG son orientables como grafos mezclados lineales. Primero se verifica si B es PIG porque A y B deben ser PIG si G es PCA.

Para el paso 4, se demuestra que los grafos C, D, E deben ser PIG, si G es PCA. Deng et al. demuestran que para un grafo PIG existe una única orientación lineal (salvo inversiones completas). Además, se muestra un algoritmo para hallar estas orientaciones. Esta rutina se usa para obtener los grafos mezclados C', D', E', que son las orientaciones mencionadas.

En el paso 5, la razón por la nos interesa el grafo H a partir de C', D', E' orientados de manera consistente es porque se demuestra que H es un grafo mezclado ronda. Esto es, ya casi hemos llegado a obtener una orientación D de G que sea ronda. Faltaría únicamente orientar las aristas faltantes en H.

El paso 6 se justifica porque Deng et al. muestran que si cada bloque de H es un torneo transitivo, el grafo resultante D es ronda.

Deng et al. describen un procedimiento para realizar el paso 7 en orden lineal.

Complejidad y alcance del algoritmo

Deng et al. reconocen y generan un modelo arco circular propio en orden $O(n+m)$. No se generan certificados en caso de que el grafo input no sea PCA.

Capítulo 3

Grafos arco circulares unitarios

Un grafo G es *arco circular unitario (UCA)* si existe un modelo arco circular de G tal que todos sus arcos tienen la misma longitud.

3.1. Aplicaciones

Mostramos aquí una aplicación a la planificación de turnos del personal operario en una fábrica. Supongamos que se deben organizar múltiples turnos de trabajo, de manera tal de respetar que algunos turnos no pueden solaparse con otros, porque utilizan el mismo recurso compartido (p.ej. la misma maquinaria) o porque deben realizarse tareas entre que finaliza un turno y comienza el otro (p. ej. planillas de horas, inventario de productos terminados), etc. Los turnos deben ser de la misma duración. Buscamos una representación arco circular unitaria, tal que cada turno esté modelado por un arco alrededor del círculo, que representa una unidad de tiempo para los turnos que se repetirá cíclicamente. La longitud del arco es la duración del turno. Pedimos que el modelo sea unitario para que todos los turnos tengan la misma duración. Partiremos entonces del grafo de G de compatibilidades entre turnos. Cada vértice es un turno, y cada arista (v, w) indica que el turno v y el w son compatibles. Buscamos entonces un subgrafo generador H de G tal que H sea UCA, y el modelo UCA resultante será la planificación que queremos obtener. La Fig. 3.1 ilustra estas ideas. En el gráfico (a) vemos cada uno de los turnos que buscamos organizar. Debemos tener en cuenta las siguientes restricciones: Los turnos de producción no pueden solaparse porque utilizan el mismo equipamiento, así como tampoco las actividades de QA y empaque, porque esta última depende de los resultados de la anterior. Así, el grafo de compatibilidades G que responde a estos criterios es el del gráfico (b). En el gráfico (c) se muestra un modelo arco circular unitario posible para un subgrafo generador de G.

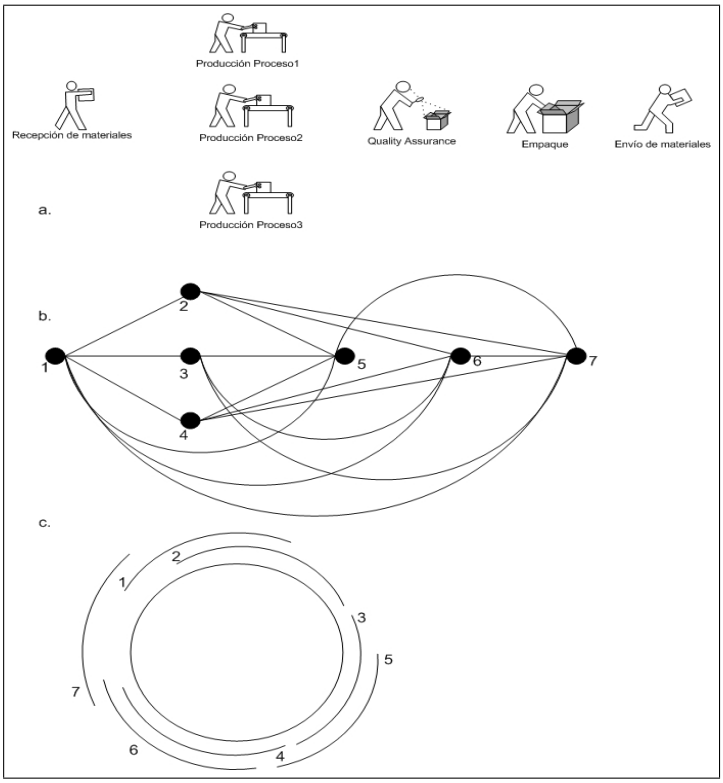

Figura 3.1: Aplicación de grafos UCA a la planificación de turnos de una fábrica

3.2. Caracterizaciones

Llamamos $CI(j,k)(j > k)$ al grafo arco-circular cuya representación en arcos circulares se construye de la siguiente forma: sea ϵ un número real positivo pequeño y $r = 1$ el radio de la circunferencia. Se dibujan j arcos $A_0, A_1, \ldots, A_{j-1}$ de longitud $l_1 = \frac{2\pi k}{j} + \epsilon$, tal que cada arco A_i comienza en $\frac{2\pi i}{j}$. Esto es, cada arco A_i será de la forma $(\frac{2\pi i}{j}, \frac{2\pi(i+k)}{j} + \epsilon)$. Ahora dibujamos otros j arcos $B_0, B_1, \ldots, B_{j-1}$ de longitud $l_2 = \frac{2\pi k}{j} - \epsilon$, tal que cada arco B_i comienza en $\frac{2\pi i+\pi}{j}$. Esto es, cada arco B_i será de la forma $(\frac{2\pi i+\pi}{j}, \frac{2\pi(i+k)+\pi}{j} - \epsilon)$.

La primera caracterización se la debemos a Tucker:

Teorema 3.1 *[32] Sea G un grafo arco-circular propio. G es arco-circular unitario si y sólo si G no contiene a $CI(j,k)$ como subgrafo inducido, para j, k coprimos y $j > 2k$.*

3.3. Implementación del algoritmo de Durán et al.

El algoritmo de Durán et al. se basa en la caracterización de Tucker del Teorema 3.1. La caracterización en base a las estructuras prohibidas CI no lleva trivialmente a un algoritmo de reconocimiento, pero el procedimiento está implícito en la demostración de este teorema, y sobre ésta trabajan los autores

para desarrollar sus resultados. El algoritmo parte de un modelo PCA para un grafo G [1], elimina los pares de arcos que cubren todo el círculo y luego determina si G es UCA.

3.3.1. Descripción del algoritmo

Un *circuito* (n, k) con respecto a un modelo (C, A) es un conjunto $\{v_1, v_2, \ldots, v_n\}$ de vértices $(n \geq 1)$tal que v_i y v_{i+1} son adyacentes $(1 \leq i \leq n-1)$, v_n es adyacente a v_1, el arco A_{i+1} comienza después del extremo inicial de A_i y antes del extremo final de A_i, recorriendo a C en sentido horario y el conjunto de arcos recorre k veces al círculo C. Para contar el número de vueltas alrededor de C, comenzamos en el extremo final de A_1, saltando de A_1 a A_2, de A_2 a A_3, y así sucesivamente; contamos un nuevo giro cada vez que pasamos por el punto de partida.

Un *conjunto independiente* (m, l) *de* G con respecto a un modelo (C, A) es un conjunto $\{v_1, v_2, \ldots, v_m\}$ de vértices $(m \geq 1)$ tal que v_i y v_{i+1} son no adyacentes $(1 \leq i \leq m-1)$, v_m es no adyacente con v_1, el arco A_{i+1} comienza después del extremo final de A_i, recorriendo a C en sentido horario y el conjunto de arcos recorre l veces al círculo C. Para contar el número de vueltas alrededor de C de la misma manera que antes, solamente que ahora consideramos el último giro como completo (es decir, sumamos 1 al número de giros).

Un circuito (n, k) de C es *minimal* si no existen circuitos (n', k'), con $n'/k' < n/k$, que puedan formarse con los vértices de C, (quizás en distinto orden). Si $k = 0$, asumimos que el cociente es igual a un M grande. Un conjunto independiente (m, l) de C *maximal* se define de manera análoga.

Un modelo es *normal* si no existen pares de arcos que cubran todo el círculo. Esto es equivalente a un modelo sin circuitos (2, 1).

Describiremos primero el algoritmo en sí y luego el fundamento teórico que garantiza su correctitud. Primero, utilizaremos el algoritmo de Deng et al. para obtener un modelo o representación r del grafo G que sea PCA. Si este algoritmo devuelve que G no es PCA, sabemos que tampoco es UCA y el algoritmo termina. Si G es PCA, modificaremos el modelo r de forma tal de obtener un modelo equivalente pero sin pares de arcos que cubran el círculo. Se demuestra que si G es PCA esto siempre es posible. La razón de este paso es que sino los teoremas que garantizan la correctitud del algoritmo tienen como precondición que no existan pares de arcos que cubran el círculo. Ahora buscaremos cotas superiores e inferiores para la circunferencia de un modelo UCA de G. El rango entre la cota inferior y superior es vacío si y sólo si no existe modelo UCA para G. La cota superior para la longitud de la circunferencia es el valor más chico m de los n/k entre todos los circuitos (n, k) minimales, y la cota inferior es el valor más grande M de los m/l entre todos los conjuntos independientes (m, l) maximales. Esto quiere decir que en un modelo UCA de G la circunferencia debe medir menos que M y más que m. Si esto es imposible, G no es UCA.

[1] recordemos que los grafos UCA son una subclase de los grafos PCA

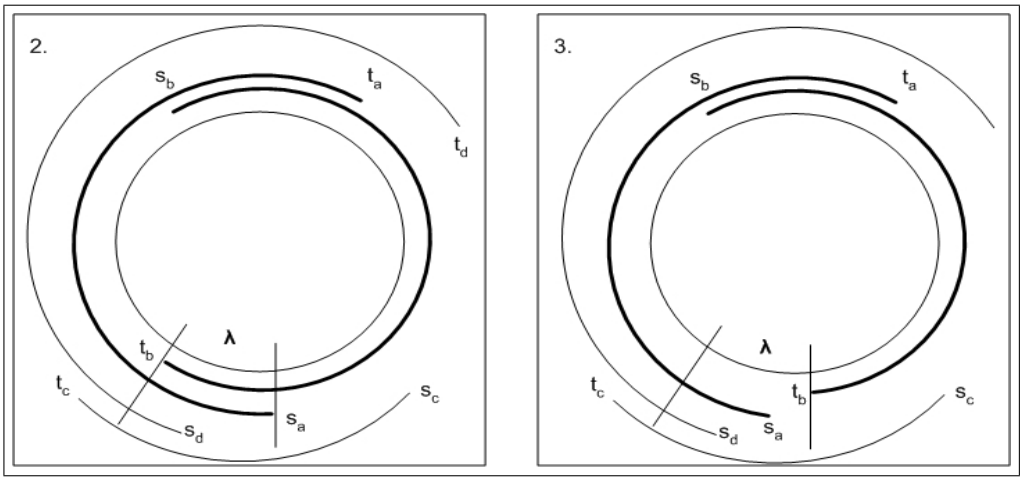

Figura 3.2: Se muestra aquí cómo el algoritmo elimina los circuitos (2 ,1), manteniendo las adyacencias

Algoritmo 3.1 *Algoritmo de reconocimiento UCA de Durán et al.*

1. *Utilizar el algoritmo de Deng et al. para obtener una representación r de G que sea PCA. Si G no es PCA, devolver que G no es UCA.*
2. *Transformar el modelo r en uno equivalente sin pares de arcos que cubran todo el círculo.*
3. *Encontrar el valor más chico m de los n/k entre todos los circuitos (n, k).*
4. *Encontrar el valor más grande M de los m/l entre todos los conjuntos independientes (m, l).*
5. *Si $m = M$ devolver que G no es UCA, sino devolver que sí lo es.*

Algoritmo 3.2 *Generación de modelo normal*

1. *Generar a partir del modelo PCA una secuencia σ de letras, donde cada letra representa un extremo de un arco. Entonces, cada arco A_x tiene dos letras relacionadas, s_x y t_x, que representan el extremo inicial y final de A respectivamente.*
2. *Encontrar una subsecuencia circular en σ de la forma $\underbrace{s_a \ldots tb}_{\tau} \underbrace{\ldots s_b \ldots t_a}_{\rho}$*
3. *Reemplazar τ con $\tau_1\tau_2$, donde τ_1 son las letras t en τ y τ_2 son las letras s en τ, preservando su orden relativo.*
4. *Ir a 2, hasta que no existan más de estas subsecuencias.*

La Fig. 3.2 ilustra el algoritmo.

Algoritmo 3.3 *Encontrar la cota superior más pequeña m de los n/k.*
Sea (C, A) un modelo PCA normal de G

1. *Para cada arco A_i en A hacer*

   $c \leftarrow A_i$

   Marcar c.

   Mientras haya arcos sin marcar adyacentes a c hacer

   Encontrar el arco sin marcar que sea adyacente a c y se extienda más lejos en dirección horaria. Marcarlo y guardarlo en c.

   Si completamos un nuevo giro alrededor de C, calcular n/k, donde n es el número de arcos marcados y k es el número de veces que recorrimos el círculo (calculado como dice la definición). Si este valor es la cota superior más pequeña hasta el momento, guardarlo.

Algoritmo 3.4 *Encontrar la cota inferior más grande M de los m/l.*
Sea (C, A) un modelo PCA normal de G

1. *Para cada arco A_i en A hacer*

   $c \leftarrow A_i$

   Marcar c.

   Mientras haya arcos sin marcar no adyacentes a c hacer

   Encontrar el arco sin marcar no adyacente a c más próximo en dirección horaria. Marcarlo y guardarlo en c.

   Calcular m/l, donde m es el número de arcos marcados y l es el número de veces que recorrimos el círculo (calculado como dice la definición). Si este valor es la cota inferior más grande hasta el momento, guardarlo.

Explicaremos ahora los fundamentos teóricos que garantizan la correctitud del algoritmo. Primero presentaremos un teorema que relaciona los circuitos y conjuntos independientes con la caracterización 3.1

Teorema 3.2 *[7] Sea G un grafo PCA. Entonces las siguientes afirmaciones son equivalentes:*

1. *G no es un grafo UCA*
2. *G contiene un subgrafo $CI(n, k)$, con n, k coprimos y $n > 2k$*
3. *G tiene un circuito minimal (n, k) y un conjunto independiente (n, k) maximal con respecto a cualquier modelo PCA, con n, k coprimos y $n > 2k$.*

Esto justifica en el algoritmo por qué se buscan circuitos y conjuntos independientes. Sabemos también que los procedimientos 3.3 y 3.4 visitan todos los circuitos maximales y conjuntos independientes gracias a los Lemas 3.1 y 3.2:

Lema 3.1 *[7] Sea $C = x_1, \ldots, x_n$ un circuito minimal, con el primer vértice fijo. Entonces podemos ordenar los otros vértices de C de forma tal que para cada i, A_{i+1} es el arco adyacente a A_i que se extiende más lejos en dirección horaria.*

Lema 3.2 *[7] Sea $C = x_1, \ldots, x_m$ un conjunto independiente maximal, con el primer vértice fijo. Entonces podemos ordenar los otros vértices de C de forma tal que para cada i, A_{i+1} es el arco no adyacente a A_i más próximo, recorriendo el círculo en dirección horaria.*

En efecto, como ambos procedimientos parten sucesivamente de cada arco (i.e lo fijan) y recorren los arcos tal y como dicen los lemas, seguro visitarán cada circuito minimal y conjunto independiente maximal, respectivamente. Faltaría ver únicamente el paso 5 del algoritmo, la razón por la cual $m = M$ determina que el grafo sea o no UCA. Esto está justificado en el siguiente teorema.

Teorema 3.3 *[7] Sean m/l y n/k las cotas inferiores m y M halladas por el algoritmo. Entonces G no es UCA sii $m = M$*

Demostración:

Tucker demostró dos lemas que se utilizan para demostrar esta propiedad. El primero de ellos es el Lema 3.3:

Lema 3.3 *[32] Sea G un grafo PCA y sea (C, A) un modelo de G. Entonces para cualquier circuito (n, k) y conjunto independiente (m, l) de (C, A), se cumple que $m/l \leq n/k$.*

Este lema es útil porque nos permite restringir a los casos $m/l = n/k$ y $m/l < n/k$ la demostración del teorema:
Caso $m/l < n/k$: Como el algoritmo recorre todos los circuitos minimales y conjuntos independientes maximales, en particular recorre el mínimo y el máximo, respectivamente. Los valores de m/l y n/k del mínimo y máximo son los que el procedimiento devuelve en M y m. Entonces no puede ocurrir que exista un circuito minimal (n', k') y un conjunto independiente maximal (m', l') tal que $m'/l' = n'/k'$. Entonces tampoco puede ocurrir que exista un circuito minimal (n'', k'') y un conjunto independiente maximal (n'', k''). Por el teorema 3.2, entonces G es UCA.
Caso $m/l = n/k$: Para esta parte de la demostración, usaremos el segundo lema de Tucker:

Lema 3.4 *[32] Sea G un grafo PCA. Para cuaquier n, k coprimos y $n > 2k$, cualquier modelo de G contiene un circuito minimal (n, k) y un conjunto independiente maximal (n, k) si y sólo si G contiene como subgrafo inducido a $CI(n, k)$. Para $n < 2k$, o n, k no coprimos, no existe este conjunto independiente maximal. Los circuitos $(2, 1)$ pueden ser eliminados alterando el modelo PCA.*

En este caso, existe un circuito minimal (n, k) y un conjunto independiente maximal (n, k). Además, n, k son coprimos y $n < 2k$, por el lema anterior (sino el conjunto independiente maximal no podría existir) y porque nos aseguramos que no existieran circuitos $(2, 1)$ al aplicar el procedimiento 3.2 en el algoritmo. Entonces G no es UCA.
Con esto queda demostrada este teorema y la correctitud del algoritmo.△

Complejidad y alcance del algoritmo

El algoritmo parte de un grafo PCA, y comprueba si G es UCA, pero no genera modelo ni produce certificados en caso que G no sea UCA. Se describe cómo llegar a la mejor complejidad teórica posible, dado que el algoritmo genera y chequea 2N secuencias de arcos de longitud N, que es $O(N^2)$. El paper refiere a un algoritmo para generar modelo contenido en la demostración del Teorema 3.1 de Tucker. Sin embargo, la complejidad de este algoritmo implícito en el teorema de Tucker no es conocida, porque involucra la manipulación de enteros grandes, donde no puede asumirse que operar con ellos insuma tiempo constante.

3.3.2. Descripción de la implementación

Describiremos aquí la implementación de cada uno de los procedimientos que conforman el algoritmo, en dos niveles de abstracción. Primero se realiza una descripción conceptual y luego se explican las decisiones que se tomaron, junto con los problemas encontrados en la implementación en un lenguaje de programación.

Descripción conceptual

- *Algoritmo 3.2 (Generación de modelo normal)*

Estructuras de datos
Utilizaremos un arreglo L de longitud $2N$ para implementar este algoritmo, que representará la secuencia de letras σ. Cada elemento de L se compone de tres valores: una letra s o t con su correspondiente número de arco y el índice en L de su extremo opuesto.

```
1. Para i = 0 a 2N - 1 hacer
2.    Si L[i] es un extremo inicial
3.       Si i < L[i].opposite
4.          Encontrar el mayor j tal que
            i < j < L[j].opposite < L[i].opposite
5.       Sino
6.          Encontrar el mayor j tal que
            L[i].opposite < i < j < L[j].opposite
```

```
7.    Si se encontro un j como se pide
8.        Rearreglar L[i...j]: Poner las letras t
          antes que las letras s, preservando su
          orden relativo y actualizando los indices
          de sus extremos opuestos.
9.        Incrementar i para que apunte nuevamente
          al mismo elemento de L.
10.Fin para
```

Explicaremos los pasos 3 a 6 del algoritmo, donde se encuentran posibles arcos que junto con el actual A_i cubren el círculo. Supongamos que el arco A_i tiene k arcos adyacentes $A_{i_1}, \ldots, A_{i_k}$ cada uno de los cuales cubre junto con A_i el círculo. La secuencia sería entonces $s_i, \ldots, t_{i_1}, \ldots, t_{i_2}, \ldots, t_{i_k}, \ldots, s_{i_1}, \ldots, s_{i_2}, \ldots, s_{i_k}, \ldots, t_i$. En los pasos 3 a 6, el algoritmo busca el *último* de estos k arcos, es decir, L[j] corresponde a s_{i_k}, y j a t_{i_k}(notar que no puede ser que L[j] corresponda a t_{i_k} y j a s_{i_k} porque suponemos que el modelo es PCA). Ahora, cuando reordenamos los extremos en el paso 8, no solamente eliminamos el circuito (2,1) que formaba A_i con A_k, sino *también* los circuitos de los demás arcos. De ahí la conveniencia de buscar j como se describe.

- *Algoritmos 3.3 y 3.4 (Encontrar la cota superior más pequeña M de los n/k y encontrar la cota inferior más grande m de los m/l)*

Como puede verse en los algoritmos de la página 25, la idea es generar y calcular las cotas de 2N secuencias de arcos, cada una de longitud N como máximo. Esto quiere decir que para lograr una implementación $O(N^2)$ necesitamos poder obtener el siguiente arco de la secuencia a marcar en orden constante amortizado. En 3.3, este arco es el arco sin marcar que sea adyacente al actual y se extienda más lejos en dirección horaria y en 3.4 es el arco sin marcar no adyacente al actual más próximo en dirección horaria.

Estructuras de datos

Utilizaremos para estos requerimientos anteriores una estructura de datos denominada *union find* que explicamos debajo, siguiendo la implementación sugerida en [4].

El tipo union find es una estructura que mantiene una colección dinámica $S = \{S_1, S_2, \ldots, S_k\}$ de conjuntos disjuntos. Cada conjunto estará identificado por un *representante*, que es un elemento del conjunto. El tipo soporta las siguientes operaciones:

- *makeset(x)* Crea el nuevo conjunto cuyo único elemento (y por ende su representante) es x. Como los conjuntos son disjuntos, requerimos que x no esté ya en un S_i.

- *union(x, y)* Une los conjuntos (disjuntos) que contienen a x e y, S_x y S_y respectivamente, para formar el nuevo conjunto unión. El representante del conjunto resultado es algún miembro de $S_x \cup S_y$. Elimina finalmente los conjuntos S_x y S_y de la colección.

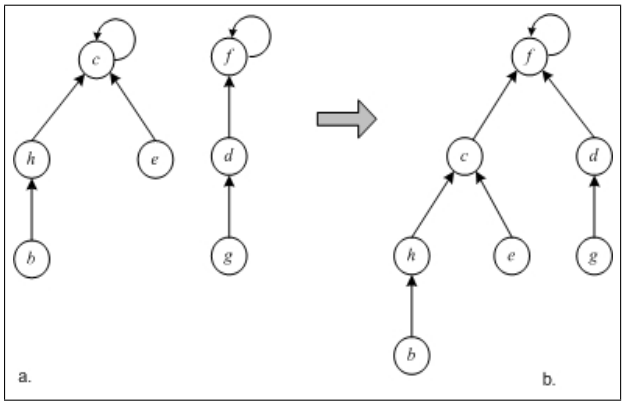

Figura 3.3: Operación de union entre dos árboles

- *findset(x)* Devuelve un puntero al representante del (único) conjunto que contiene a x.

Implementamos el union find utilizando un *bosque*, un conjunto de árboles donde cada árbol representa un conjunto y cada nodo contiene un elemento del conjunto. Cada nodo apunta únicamente a su padre. La raíz de cada árbol contiene al representante y es su propio padre. Cormen et al. introducen en los procedimientos directos para implementar las operaciones dos heurísticas *unión por rango* y *compresión de caminos* que resultan en la estructura asintóticamente más rápida que se conoce para conjuntos disjuntos. La implementación es como sigue:

- *makeset* Crear un árbol con un sólo nodo
- *findset* Seguir punteros a los padres hasta encontrar la raíz del árbol.
- *union* Hacer que la raíz de un árbol apunte a la raíz del otro, como muestra la Fig. 3.3

La idea de la primera heurística, la unión por nivel, es hacer que la raíz del árbol con menos nodos apunten a la raíz del árbol con más nodos. Para cada nodo, mantenemos una variable *nivel*, que es una cota superior para la altura de ese nodo. En la operación union, la raíz con menor valor de nivel apuntará a la raíz con mayor valor de nivel.

La heurística de compresión de caminos se aplica durante la operación de findset(x). La idea es hacer que cada nodo ubicado en el camino desde x hasta la raíz apunte directamente a la raíz. La Fig. 3.4 ilustra este procedimiento.

Mostramos a continuación los pseudocódigos de cada método. Designamos al padre del nodo x como $p[x]$:

Algoritmo 3.5 *Procedimientos del tipo union find*

- *makeset(x)*

  1. $p[x] \leftarrow x$

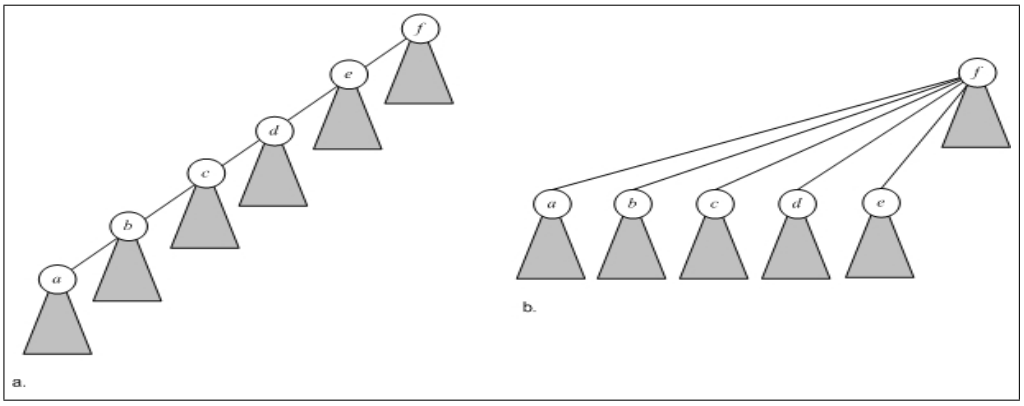

Figura 3.4: Heurística de compresión de caminos

2. $nivel[x] \leftarrow 0$

- *findset(x)*

  1. *Si* $x \neq p[x]$
  2. $p[x] \leftarrow findset(p[x])$
  3. *Devolver* $p[x]$

- *union(x, y)*

  1. $x' \leftarrow findset(x)$
  2. $y' \leftarrow findset(y)$
  3. *Si* $nivel[x'] > nivel[y']$
  4. $p[y'] \leftarrow x'$
  5. *Sino*
  6. $p[x'] \leftarrow y'$
     Si $nivel(x') = nivel(y')$
     $nivel(y') = nivel(y') + 1$

Cuando un conjunto es creado por makeset, el nivel inicial del único nodo del árbol es 0. Cuando se aplica union a dos árboles, ponemos a la raíz de mayor nivel como padre de la raíz de menor nivel. En caso de tener los mismos valores de nivel, elegimos arbitrariamente una raíz como padre e incrementamos su nivel. El procedimiento findset(x) recorre dos veces el camino entre x y la raíz su árbol. La primera vez sube desde x para encontrar la raíz, y luego baja nuevamente por este camino y actualiza cada nodo para que apunte directamente a la raíz. Esto funciona así: El caso base de la recursión es cuando x es la raíz. En este caso $x = p[x]$ y el algoritmo devuelve x, que es correcto. Si x no es la raíz se ejecuta la línea 2. Aquí la llamada a findset retorna la raíz del árbol (utilizando ahora al padre de x como parámetro). Esta raíz hallada se asigna en la línea 2 a x como padre y se devuelve en la línea 3. Este procedimiento implementa la heurística de compresión de caminos.

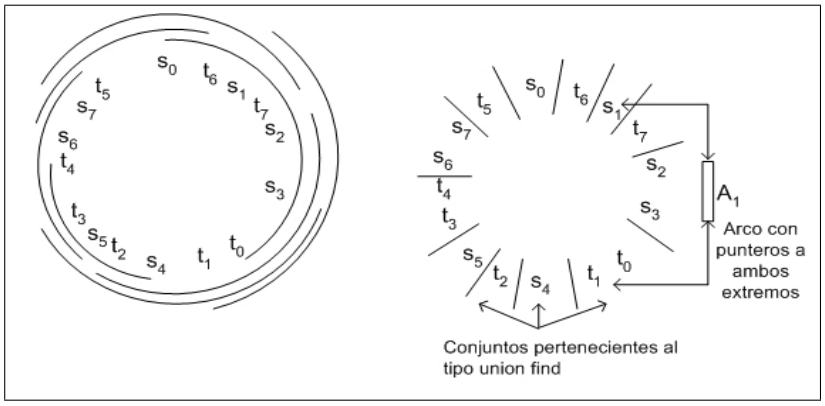

Figura 3.5: Instancia de union find inicial para un modelo PCA

Volviendo a la aplicación sobre nuestro algoritmo, imaginemos inicialmente los extremos del modelo organizados en bloques como muestra la Fig 3.5. El union find consistirá en conjuntos o bloques consecutivos de letras marcadas y sin marcar alrededor del círculo. La idea será trabajar con estos bloques marcando sucesivamente arcos y fusionando bloques a medida que éstos tienen todos sus elementos marcados.

Aumentamos el unionfind de Cormen et al. con los siguientes campos asociados a cada conjunto S_i:

- Una lista doblemente linkeada *unmarked*, que contendrá los elementos sin marcar de S_i
- Un puntero *next*, que apunta al siguiente conjunto S_{i+1} en sentido horario.
- Un puntero *previous*, que apunta al siguiente conjunto S_{i-1} en sentido antihorario.

Nuestro union find mantendrá los siguientes invariantes de representación:

1. Cada conjunto contiene por lo menos una letra sin marcar. Es decir, la lista unmarked asociada a cada conjunto es no vacía.
2. Un conjunto puede contener una mezcla de elementos s y t, pero los elementos sin marcar deben ser todos s o todos t. Es decir, la lista unmarked de cada conjunto tiene solamente elementos s o elementos t.
3. Cada conjunto es un bloque de letras (marcadas y sin marcar) que representan extremos alrededor del círculo.
4. Los bloques con elementos s sin marcar se alternan con bloques con elementos t sin marcar. Si C es un conjunto, entonces C.next y C.previous contienen elementos t si y sólo si C contiene elementos s.
5. La lista de elementos sin marcar en un bloque están ordenadas en sentido horario, comenzando por el principio del bloque de letras que ocupa ese conjunto.

Describiremos ahora los ciclos principales con sus subrutinas correspondientes a los algoritmos 3.3 y 3.4. Cuando nos referimos a inicializar una instancia del tipo unionFind, queremos decir comenzar con los conjuntos del tipo almacenados como se muestra en la Fig. 3.5.

Procedimiento1 Encontrar la cota superior más pequeña m de los n/k.

```
1. Para cada arco Ai en A hacer
2.    Inicializar una instancia de unionFind
3.    c = Ai
4.    Marcar(c)
5.    Mientras noMoreArcs sea false hacer
6.       c = MarcarSiguienteAdyacente(c, unionFind)
7.       Si c es vacio
8.          Poner noMoreArcs = true
9.       Sino
10.         Si completamos un nuevo giro alrededor de C
11.            Calcular n/k, con n el numero de arcos
               marcados y k el numero de veces
               que recorrimos el circulo. Guardar en m el
               valor si es el minimo n/k hallado hasta
               el momento.
12.   Fin Mientras
13. Fin Para
```

Procedimiento2 Encontrar la cota inferior más grande M de los m/l

```
1. Para cada arco Ai en A hacer
2.    Inicializar una instancia de unionFind
3.    c = Ai
4.    Marcar(c)
5.    Mientras noMoreArcs sea false hacer
6.       c = MarcarSiguienteNoAdyacente(c, unionFind)
7.       Si c es vacio
8.          Poner noMoreArcs = true
9.       Sino
10.         Calcular m/l, con m el numero de arcos
            marcados y l el numero de veces
            que recorrimos el circulo. Guardar en M el
            valor si es el maximo m/l hallado hasta
            el momento.
11.   Fin Mientras
12. Fin Para
```

Procedimiento3 Marcar(arco Ai)

```
1. Encontrar el conjunto S al que pertenece si
   haciendo unionFind.find(Ai.si)
2. Encontrar el conjunto E al que pertenece ti
```

```
   haciendo unionFind.find(Ai.ti)
3. Borrar a si y ti de las listas S.unmarked y E.unmarked.
4. Si algunas de estas listas se vacia,
   llamar al procedimiento merge(S) o merge(E)
   segun corresponda
```

Procedimiento4 MarcarSiguienteAdyacente(arco Ai)

```
1. Encontrar el conjunto E al que pertenece ti
   haciendo unionFind.find(Ai.ti)
2. Obtener el conjunto anterior a E en sentido
   horario, D, con el puntero E.previous
3. Sea bj el ultimo elemento de la lista F.unmarked
4. bj es el extremo inicial del siguiente arco
   adyacente. Marcarlo usando el procedimiento Marcar.
5. Si no se encontro un bj sin marcar en F,
   devolver 'vacio'
```

Procedimiento5 MarcarSiguienteNoAdyacente(arco Ai)

```
1. Encontrar el conjunto E al que pertenece ti
   haciendo unionFind.find(Ai.ti)
2. Obtener el conjunto siguiente a E en sentido
   horario, F, con el puntero E.next
3. Sea bj el primer elemento de la lista F.unmarked
4. bj es el extremo inicial del siguiente arco
   no adyacente. Marcarlo usando el
   procedimiento Marcar.
5. Si no se encotro un bj sin marcar en F,
   devolver 'vacio'
```

Procedimiento6 Merge(conjunto C)

```
D sera el conjunto resultado del merge.
1. Hacer D.unmarked = C.previous.unmarked,
   concatenandole al final C.next.unmarked
2. Hacer D = unionFind.union(C, C.previous)
   /*se actualizan los punteros a next y
   previous*/
3. Hacer unionFind.union(D, C.next)
   /*se actualizan los punteros a next y
   previous*/
```

El orden de complejidad de la implementación que se muestra aquí es de $O(n^2\alpha(n))$, donde $\alpha()$ es la inversa de la función de Ackermann [2], de crecimien-

[2]La función de Ackermann A se define así:

$$\begin{aligned} A(1,j) &= 2^j, j \geq 1 \\ A(i,1) &= A(i-1,2), i \geq 2 \\ A(i,j) &= A(i-1, A(i,j-1)), i,j \geq 2 \end{aligned}$$

to extremadamente lento, pero no acotada. Esto es porque la implementación de Cormen et al. del tipo union find, que se realizó en este trabajo, permite efectuar $O(n)$ operaciones *union* y *find* en un conjunto de tamaño $O(n)$ tenga un costo de $O(n\alpha(n))$. Como el ciclo de marcado de arcos en 3.4 y 3.3 se ejecuta n veces, el costo total será de $O(n^2\alpha(n))$.
Durán et al. describen además una forma de reducir este orden a $O(n^2)$. Esta es la mejor implementación posible para el algoritmo, que generar y chequea $\Theta(n)$ listas de $\Theta(n)$ elementos cada una.

Detalles de la implementación en máquina

La implementación fue realizada en el lenguaje de programación Java [3]. Para las estructuras de datos se utilizaron las librerías standard (java.util.*). No fue necesario recurrir a librerías especializadas en grafos u otras. Explicaremos algunos detalles de la implementación, detallando las decisiones tomadas.

- En el paso 2 de los procedimientos 1 y 2 del apartado anterior, observar que necesitamos en total $2n$ instancias del tipo unionFind para ambos ciclos, porque cada iteración parte de un union find inicial, como el que se muestra en la Fig. 3.5. Es por esto que antes de ejecutar estos procedimientos ocupamos un tiempo de $O(n^2)$ en inicializar estas 2n instancias de unionFind y dejarlas en un arreglo para ser utilizadas por ellos.

- El algoritmo parte de la base de que los arcos y extremos en el modelo pueden accederse en el orden en que figuran en el círculo, en sentido horario. Como buscamos una implementación de orden cuadrático, no hay dificultad en ordenar en $O(n^2)$ los arcos en el modelo PCA input y presentar luego arcos y extremos ordenados al algoritmo.

- El paso 10 de los procedimientos 1 y 2, que es cómo saber si se ha cumplido un nuevo ciclo. Para ello, demostraremos el siguiente:

  Lema 3.5 *Sea (C,A) un modelo PCA sin circuitos (2, 1) y sea un punto cualquiera de C que denominamos s. Sean A_1 y A_2 dos arcos de C. Entonces al saltar de A_1 a A_2 se ha cumplido un nuevo giro alrededor de C, comenzado en s si y sólo si los extremos de A_1 y A_2 aparecen en C en alguno de los siguientes ordenamientos circulares:*

  1. $s, \ldots, s_2, \ldots, t_2, \ldots, s_1, \ldots, t_1$
  2. $s, \ldots, s_2, \ldots, s_1, \ldots, t_2, \ldots, t_1$
  3. $s, \ldots, t_2, \ldots, s_1, \ldots, s_2, \ldots, t_1$
  4. $s, \ldots, t_2, \ldots, s_1, \ldots, t_1, \ldots, s_2$

[3] Entorno de desarrollo Java 2 SDK, Standard Edition, Version 1.4.2. Entorno de ejecución Java(TM) 2 Runtime Environment, Standard Edition Version 1.4.2. La plataforma de desarrollo utilizada fue Eclipse Version: 3.0.2, con el plugin Cloudgarden's Jigloo GUI Builder Version 3.5 para diseño de GUIs

5. $s, \ldots, t_2, \ldots, t_1, \ldots, s_2, \ldots, s_1$
6. $s, \ldots, s_1, \ldots, t_2, \ldots, t_1, \ldots, s_2$

Demostración Puede verificarse fácilmente que los ordenamientos expuestos marcan un nuevo giro alrededor de C. La Fig. 3.6 ilustra los ordenamientos. Demostraremos el lema por exclusión del resto de los casos. Fijemos s y sea L la secuencia de extremos de C ordenados en sentido horario a partir de s. Si a continuación de s tuviéramos t_1, entonces acabamos de cruzar s con el arco A_1. Para que A_2 marque un nuevo giro, s_2 debería comenzar antes de s_1 en L, y como a continuación de s tenemos t_1, deberá terminar luego de t_1 en L. Pero esto es imposible porque A_2 incluiría a A_1.
Supongamos entonces que a continuación de s fijamos t_2. Los items 2 a 5 son válidos. Quedarían entonces para este caso únicamente las permutaciones $s, \ldots, t_2, \ldots, s_2, \ldots, s_1, \ldots, t_1$ y $s, \ldots, t_2, \ldots, t_1, \ldots, s_1, \ldots, s_2$ por revisar. Pero estos dos casos son imposibles porque en ambos casos A_2 incluye a A_1. El caso $s, \ldots, t_2, \ldots, s_2, \ldots, t_1, \ldots, s_1$ genera un circuito (2, 1), que dijimos estaba excluido.
Supongamos ahora que a continuación de s fijamos s_1. Para que A_2 marque un nuevo giro, s_2 debería comenzar después de s_1 en L y t_2 debería aparecer antes de s_2 en L. Pero la única forma de colocar estos extremos sin generar un cubrimiento de C es como se indica en el punto 6.
Queda por ver el caso en que a continuación de s queda s_2. Para que esto quiera decir que A_2 haya marcado un nuevo giro, debemos colocar los extremos de A_1 ordenados (primero s_1 y luego t_1) en L. Como a continuación de s fijamos s_2, hay sólo dos formas de hacerlo, como se muestra en los ordenamientos 1 y 2. Con esto queda demostrado el lema.
Este lema nos proporciona un procedimiento para verificar si hemos cumplido un nuevo giro alrededor de C, alcanza con ver si los extremos del arco marcado y el anterior están ordenados como pide el lema en el círculo.

- En varios puntos de la implementación fue necesario prescindir de las listas que provee Java (tipo LinkedList), por razones de eficiencia. En su lugar, se agregaron en los objetos que debían colocarse en la lista referencias al objeto anterior y al siguiente, y esto constituye la lista. Esto es necesario porque dado un objeto presente la lista, para modificarlo o borrarlo el tipo LinkedList recorre la lista hasta encontrarlo, y luego realiza la operación. A los fines del algoritmo, dado un objeto de la lista, necesitamos modificarlo y borrarlo en $O(1)$. $\triangle$

3.3.3. Resultados computacionales

Antes de comprobar la eficiencia del algoritmo, debemos resolver otro problema anterior que es el testing. El problema es difícil porque no hay algoritmos de fuerza bruta triviales para reconocer grafos UCA. La idea fue entonces generar modelos PCA donde sabemos de antemano si son UCA o no lo son, por la forma en que lo construimos. Más específicamente, cuando

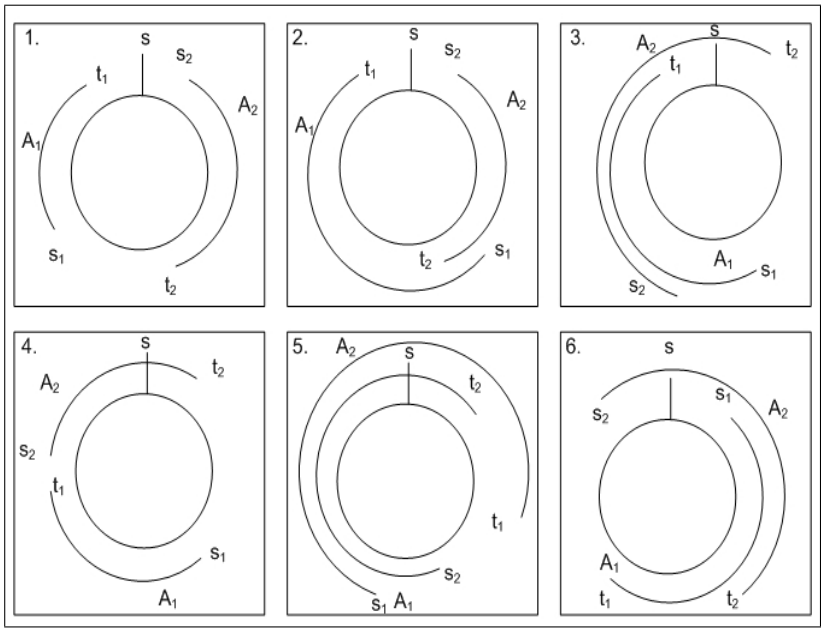

Figura 3.6: Casos del lema. Se ilustra como en cada caso se cruza el punto s al saltar de A_1 a A_2

buscamos modelos PCA no UCA introducimos en el modelo la estructura prohibida $CI(n, k)$, con n, k coprimos y $n > 2k$. Cuando queremos generar modelos UCA, construimos un modelo random PCA con arcos de la misma longitud y luego lo "deformamos", alteramos las longitudes de los arcos para que sea más general, preservando las adyacencias de los arcos. Entonces, siempre presentamos al algoritmo modelos cuya respuesta correcta sabemos de antemano, así podemos constatar si el algoritmo dice la verdad.

Veamos ahora los resultados. Fueron contadas las operaciones de suma, resta, comparación, multiplicación, división y las operaciones sobre listas (agregar, quitar, acceder a un elemento de una lista). Nos interesa discriminar esta última porque el uso de listas es intensivo, en el algoritmo, por la cantidad de veces que se utilizan en cada capa del programa(bosque de árboles, union find, ciclo principal).La Fig. 3.7(a) muestra la cantidad total de operaciones en función de n, con n la cantidad de arcos. Para tener una idea de la aproximación a los resultados teóricos, graficamos en (b) una comparación con la función $f(n) = 4n^2$. Esto es porque Cormen et al. mencionan en [4] que el valor de la función de Ackermann $\alpha(n)$ de la cota $O(n\alpha(n))$ en la eficiencia del tipo union find no debería superar nunca el valor de 4 en las aplicaciones prácticas. Efectivamente, vemos que en el gráfico se verifica esta afirmación, los valores de la cantidad de operaciones están por debajo de la curva. La línea roja del gráfico es la función cuadrática que mejor aproxima a los puntos dados, calculada en base al método numérico de mínimos cuadrados.

Veremos ahora cómo se comporta la curva para un n dado, a medida que se incrementa la cantidad de arcos adyacentes (lo que equivale a las aristas

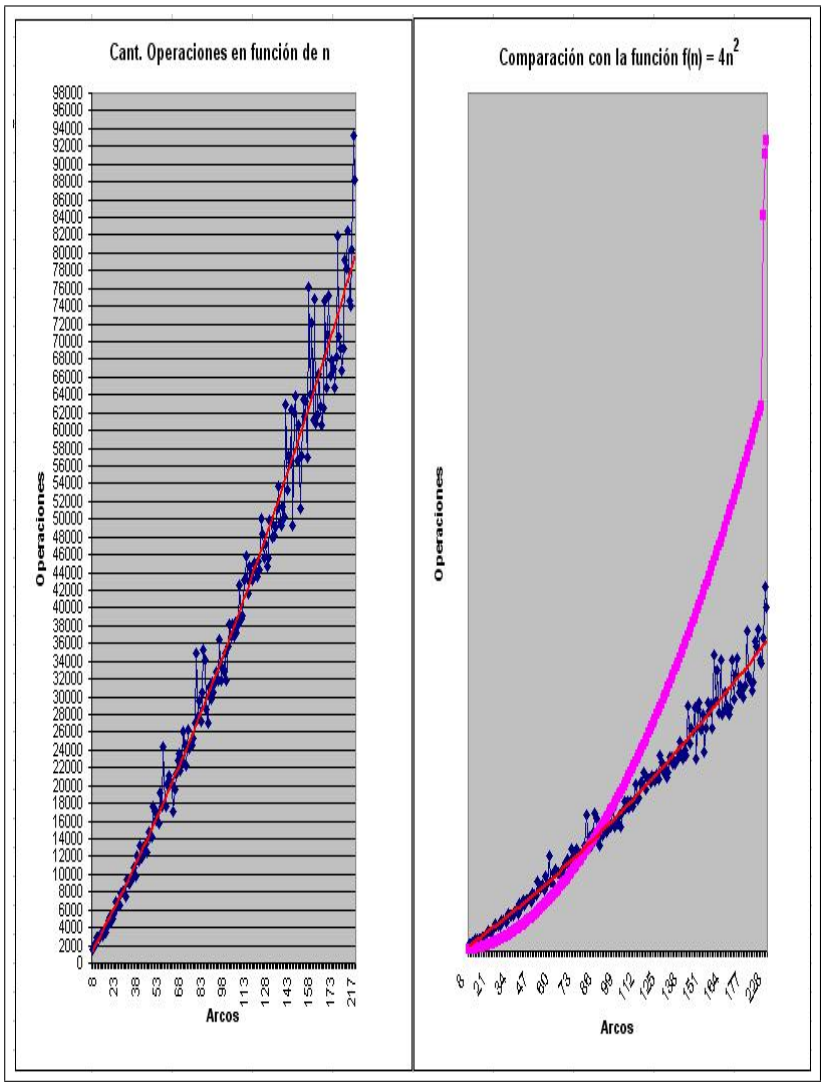

Figura 3.7: Resultados de cantidad de operaciones en función de n

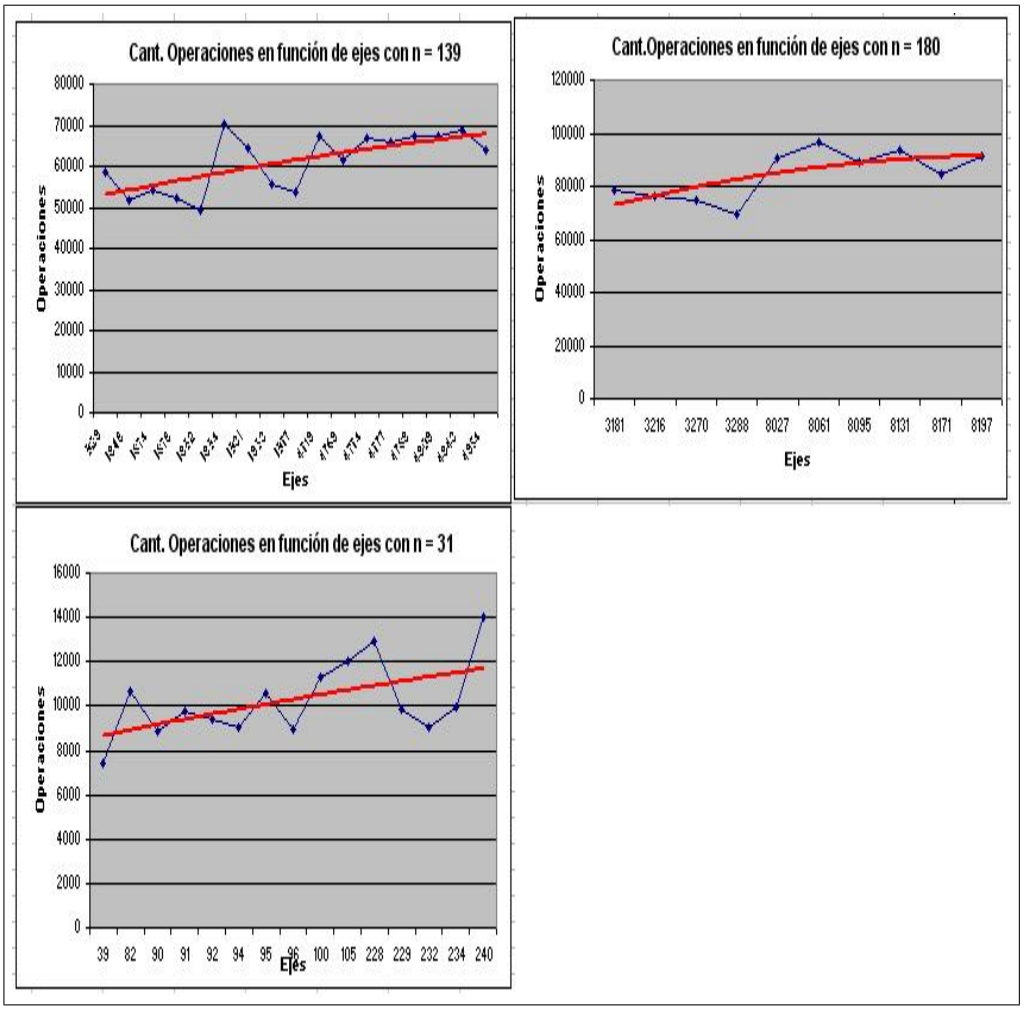

Figura 3.8: Resultados de cantidad de operaciones en a medida que aumentan las adyacencias

del grafo intersección). La Fig. 3.8 muestra la evolución a medida que aumenta la cantidad de aristas, para algunos valores de n fijos. Notar que si bien la tendencia es creciente en todos los casos, no es uniformemente creciente, sino que presenta fluctuaciones. Esto puede atribuirse a que existe otro factor importante que incide en la cantidad de operaciones, además de la cantidad de arcos y sus adyacencias, que es *cómo están dispuestos* los arcos en el círculo. Es decir, la ubicación de los arcos puede determinar que existan más o menos iteraciones, según si se encuentra o no un siguiente arco que marcar. En la Fig. 3.9 ilustramos esta idea. Notar que ambos modelos tienen la misma cantidad de arcos y adyacencias, pero el algoritmo realiza menos iteraciones en el modelo (b) que en el (a).

La Fig. 3.10 muestra cómo inciden los tipos de operación en el promedio de la cantidad de operaciones. Se agruparon las sumas, restas y comparaciones en la primera columna, las operaciones sobre listas en la segunda y las multiplicaciones y divisiones en la tercera. La razón de este agrupamiento es que

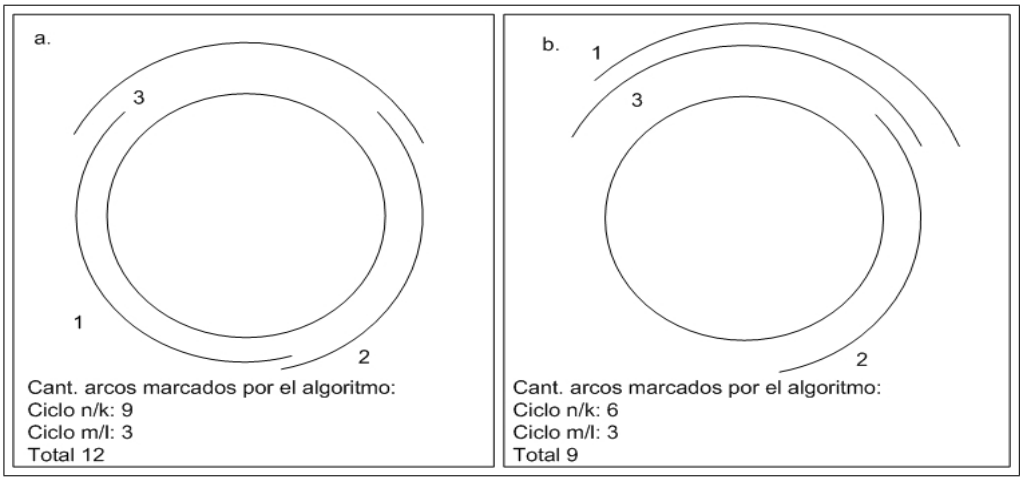

Figura 3.9: Diferencias en la cantidad de iteraciones para la misma cantidad de nodos y adyacencias

las multiplicaciones y divisiones son mucho más costosas. Además, buscamos analizar por separado las operaciones sobre listas, como se menciona arriba. Notar que en la primera columna las operaciones aritméticas están equilibradas entre las dedicadas al tipo union find y el ciclo principal del algoritmo. En la segunda, hay un claro predominio de union find en las operaciones de listas. Esto es razonable, porque este tipo es la parte más compleja del algoritmo en el manejo de listas, dada su implementación sobre árboles representados con listas. Las operaciones sobre listas insumen un 45 % del total. Notar además la escasa incidencia de las multiplicaciones y divisiones. Esto se justifica en que la única vez que intervienen estas operaciones en el algoritmo es en el cálculo de los n/k y m/l en el ciclo principal, y ni siquieran se computan en cada iteración del mismo.

3.4. Algoritmo de Lin y Szwarcfiter

Mencionaremos finalmente que recientemente fue publicado [20] por Lin y Szwarcfiter un algoritmo lineal de reconocimiento para grafos UCA, que además genera un modelo arco circular unitario del grafo. Este algoritmo se basa en nuevas caracterizaciones para grafos UCA, y se resume como sigue: Recordamos que un modelo normal es un modelo sin pares de arcos que cubran todo el círculo.
Dado un modelo (C, A), un arco de C definido por dos extremos consecutivos de A es un *segmento*.
Sea G un grafo CA y (C, A) un modelo de G. Sean $S_1, \ldots, S_{2n}$ los segmentos de (C, A) en orden circular, donde el inicio de S_1 coincide con el de A_1. Sea l_j la longitud de S_j. Cada arco puede descomponerse en los segmentos que lo forman. La longitud de $A_1 = \sum_{S_j \subseteq A_i} l_j$. La condición de igualdad entre la longitud de los arcos de A que se requiere en el modelo UCA puede expresarse con un sistema de $n-1$ ecuaciones lineales q_i, junto con dos $2n$ desigualdades,

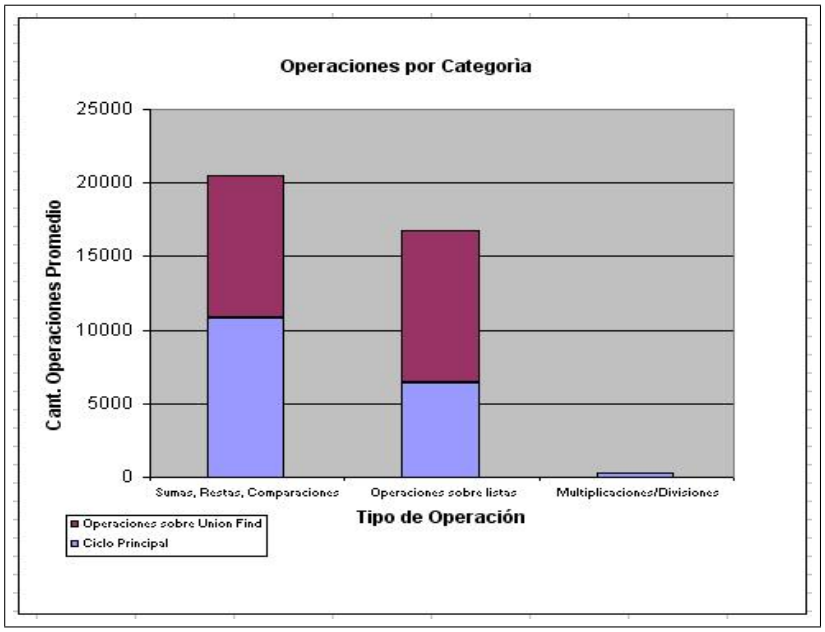

Figura 3.10: Cantidad de operaciones por categorías

que llamamos el *sistema completo* de (C, A):

$$q_i : \sum_{S_j \subseteq A_i} l_j = \sum_{S_j \subseteq A_{i+1}} l_j, i = 1, 2, \ldots, n-1 \text{ con } l_j > 0, j = 1, 2, \ldots, 2n$$

Cada equación q_i corresponde a la condición que la longitud del arco A_i sea igual a la de A_{i+1}, y los l_j son las variables de la ecuación. Buscamos resolver este sistema de ecuaciones, porque entonces los valores de l_j nos permiten construir un modelo UCA de G. Llamamos *lado izquierdo* de q_i al término $\sum_{S_j \subseteq A_i} l_j$ y *lado derecho* de q_i al término $\sum_{S_j \subseteq A_{i+1}} l_j$. Las ecuaciones q_i pueden simplificarse en caso de que aparezcan los mismos elementos l_j a ambos lados de la ecuación, restando de ambos lados. El sistema que se obtiene luego de estas operaciones se llama *sistema reducido* de (C, A). Para resolver eficientemente este sistema, se define el siguiente digrafo D, llamado *digrafo de segmentos*: Poner un vértice v_i para cada ecuación q_i del sistema reducido R de (C, A) y una arista e_j para cada segmento S_j de (C, A). Adicionalmente, poner un vértice distinguido v_0. Cada arista e_j se orienta de la siguiente manera:

- Si l_j aparece del lado izquierdo de alguna ecuación q_i de R, entonces e_j comienza en v_i, sino comienza en v_0
- Si l_j aparece del lado derecho de alguna ecuación q_k de R, entonces e_j termina en v_k, sino termina en v_0

Las Fig. 3.11, que se puede ver en [19] muestra un grafo G, su modelo normal, su sistema reducido y su digrafo de segmentos.

Los autores demuestran que el sistema tiene solución si y sólo si cada una de las componentes conexas de G es fuertemente conexa. Para la construcción del

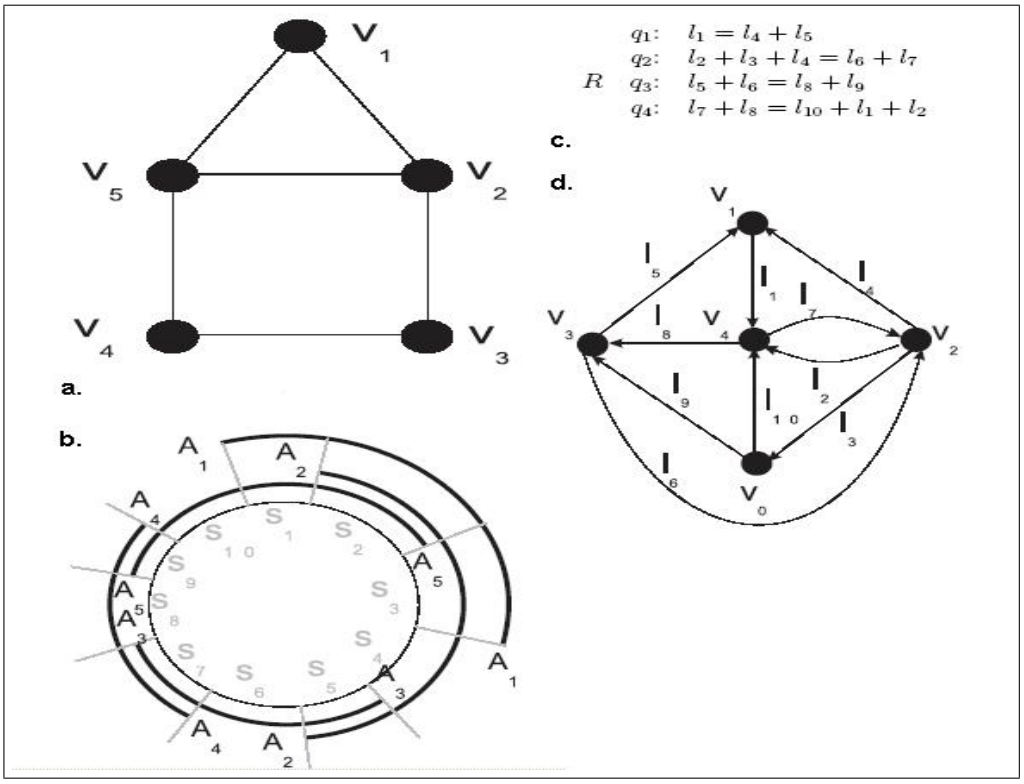

Figura 3.11: Completar

modelo UCA, la idea es encontrar una solución del sistema reducido de ecuaciones de un modelo PCA normal de G, y aplicarla para modificar las longitudes del modelo. Se trabaja sobre el modelo normal porque los autores demuestran los teoremas necesarios para resolver el sistema sobre modelos normales. El modelo UCA se obtiene del modelo normal, modificando las distancias entre los extremos, preservando el orden circular. No entraremos en detalles de la construcción del modelo.

El algoritmo se resume como sigue:

Algoritmo 3.6 *Reconocimiento de grafos UCA*

1. *Verificar si G es un grafo PCA, utilizando el algoritmo de Deng et al. [5]*
2. *Si G no es PCA, devolver que G no es UCA. Sino, sea (C,A) el modelo que genera el algoritmo anterior.*
3. *Transformar (C,A) en un modelo (C',A') sin circuitos (2, 1) (modelo normal)*
4. *Construir el digrafo de segmentos D de (C',A').*
5. *Encontrar las componentes conexas $C_1, \ldots, C_k$ de D.*
6. *Verificar si cada C_i es fuertemente conexo. En caso afirmativo G es UCA, sino no lo es.*

Capítulo 4

Grafos arco circulares Helly

Una familia de conjuntos $S_1, \ldots, S_n$ satisface la *propiedad de Helly* cuando cualquier subfamilia de S, $\{S_i, S_k, \ldots, S_m\}$ consistente en conjuntos que se intersecan de a pares tiene intersección no vacía.

Un grafo G es *arco-circular Helly* (HCA) si existe una representación arco-circular de G tal que los arcos satisfacen la propiedad de Helly.

Sean $C_1, \ldots, C_k$ las cliques de un grafo G. La *matriz clique* M de G es la matriz de ceros y unos de $n \times k$ donde $m_{i,j} = 1$ sii el vértice $v_i \in C_j$. En un modelo arco circular (C, A), el *complemento* de un arco $A_i = (s_i, t_i)$ es el arco $\overline{A_i} = (t_i, s_i)$. El *complemento* de un modelo (C, A) es el modelo $(C, \overline{A})$, con $\overline{A} = \{\overline{A_i} | A_i \in A\}$.

Una *s-secuencia (t-secuencia)* es una secuencia maximal de extremos iniciales (finales) de A, en el orden circular de C. Una s-secuencia o t-secuencia es una *secuencia extrema*. Para una secuencia extrema E, $NEXT(E)$ y $NEXT^{-1}(E)$ representan las secuencias extremas que anteceden y siguen a E en C, respectivamente. Dado un extremo $p \in A$, notamos $SEQUENCE(p)$ a la secuencia extrema que contiene a p, mientras $NEXT(p)$ se refiere a la secuencia $NEXT(SEQUENCE(p))$.

Sea s_i un extremo inicial de A y $S = SEQUENCE(s_i)$. s_i es *estable* cuando $i = j$ o $A_i \cap A_j = \emptyset$ para cada $t_j \in NEXT^{-1}(S)$. Un modelo (C, A) es *estable* cuando todos sus extremos iniciales lo son.

Un *obstáculo* es un grafo H que contiene una clique $K_t \subseteq V(H), t \geq 3$, cuyos vértices admiten un ordenamiento circular $v_1, \ldots, v_t$, tal que cada arista $(v_i, v_{i+1}), i = 1, \ldots, t$ satisface:

- $N(w_i) \cap K_t \backslash \{v_i, v_{i+1}\}$, para algún $w_i \in V(H) \backslash K_t$ o
- $N(u_i) \cap K_t \backslash \{v_i\}$ y $N(z_i) \cap K_t \backslash \{v_{i+1}\}$ para algunos vértices adyacentes $u_i, z_i \in V(H) \backslash K_t$

La Fig. 4.1 muestran algunos obstáculos.

4.1. Caracterizaciones

La primera caracterización, que debemos a Gavril, dice lo siguiente:

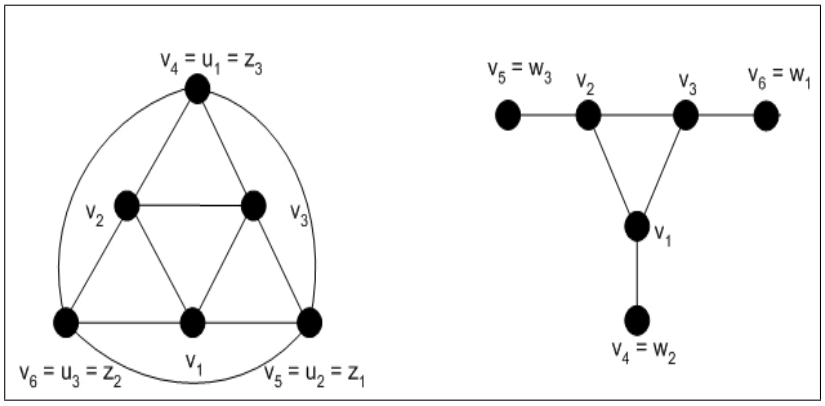

Figura 4.1: Algunos obstáculos

Teorema 4.1 *[9] Un grafo G es arco-circular Helly si y sólo si su matriz clique tiene unos circulares.*

Recientemente, Lin y Szwarcfiter dieron otras caracterizaciones, en base a subgrafos prohibidos y a determinadas propiedades que debe cumplir un modelo arco circular del grafo G:

Teorema 4.2 *[19] Sea G un grafo arco circular. Las siguientes afirmaciones son equivalentes:*

1. *G es HCA.*
2. *G no contiene obstáculos como subgrafos inducidos.*
3. *Todos los modelos estables de G son HCA.*
4. *Un modelo estable de G es HCA.*

La cantidad de obstáculos que puede tener un grafo arco circular que no sea Helly es exponencial en la cantidad de vértices, como mostramos en el siguiente ejemplo. Sea $H(2,n)$ el grafo n - partito con dos vértices en cada parte (los dos vértices en cada parte no son adyacentes, pero lo son a todos los dems vértices). Es claro que $H(2,n)$ tiene 2^n cliques. Por cada vértice, crear ahora un nodo gemelo (v' es *gemelo* de v sii $N[v] = N[v']$). Entonces ahora podemos formar un obstáculo que cumpla la condición ii de la definición eligiendo un nodo de cada partición como parte de la clique, y despus elegimos otro vértice de la misma particin entre los dos nodos no gemelos del nodo elegido previamente en el rol de u_i o z_i. Contemos ahora la cantidad de obstáculos así formados. En cada partición tenemos cuatro combinaciones posibles de vértices que pueden ser parte de un obstáculo. Si v y v' son gemelos en una partición y w y w' también lo son vértices gemelos de la misma, las combinaciones válidas son $\{v,w\},\{v',w\},\{v,w'\},\{v',w'\}$. Por lo tanto hay 4^n obstáculos distintos (en el sentido que son subgrafos inducidos distintos), siendo la cantidad de nodos son $4n$. Y si llamamos $N = 4n$ quedaría $2^{\frac{N}{2}} = \sqrt{2^N}$ que es exponencial en función de la cantidad de nodos.

4.2. Aplicaciones

Se muestra a continuación una aplicación a una empresa de logística. Supongamos que la empresa tiene una sucursal adonde continuamente llegan camiones con mercaderías. La mercancía se carga en otros camiones rumbo a otros destinos. A su vez los primeros toman nuevas cargas y retoman su viaje. La empresa pide a sus proveedores que aseguren que en un período dado, que se repite constantemente, debe llegar un camión con mercadería (por ejemplo, de 6 a 8hs). La flota de la empresa también funciona así. Está garantizado que existen períodos donde un camión de la empresa está disponible para llevar o traer mercaderías. Esta red de proveedores, distribuidores y clientes está definida en un grafo G, donde por cada vehículo que opera en esa sucursal tenemos un nodo, y por cada par v, w de vehículos cuyas cargas se transportan de v a w o viceversa, tenemos una arista en el grafo. La empresa busca minimizar el lapso de tiempo que se ocupa en operaciones de carga y descarga. Para ello se intentará realizar la máxima cantidad posible de operaciones simultáneamente. Esto permitirá optimizar los turnos del personal dedicado a estas tareas.
Para resolver este problema, buscamos un modelo arco circular de G que sea HCA. Los arcos representan vehículos, y su longitud representa el tiempo en que el vehículo permanece en la planta. Pensamos a la circunferencia como una unidad de tiempo que se repite constantemente. Como el modelo es HCA, está garantizado que todos aquellos vehículos que están mutuamente vinculados por una operación de carga/descarga tienen en el modelo un punto común de intersección en sus arcos. Este será el momento de realizar la operación. La Fig. 4.2 ilustra la situación. En el gráfico (a), se muestra cómo se realizan tres operaciones de carga y descarga, y se grafica debajo el modelo arco circular que representa la operación. En el gráfico (b) se ilustra el esquema deseado, que se produzca una única operación de carga y descarga para los tres camiones. El modelo arco circular HCA debajo muestra que existe un punto de encuentro en los períodos de tiempo donde esta única operación es posible.

4.3. Algoritmo de Gavril

El algoritmo de Gavril se basa en dos observaciones. Primero, demuestra que en un grafo HCA el número de cliques es a lo sumo n y segundo, que la propiedad de Helly es hereditaria, de manera que cualquier subgrafo de G HCA también será HCA. Entonces, un subgrafo de G de k vértices tendrá a lo sumo k cliques. El algoritmo se resume así:

Algoritmo 4.1 *Algoritmo de Gavril para reconocimiento de grafos HCA*

1. *Para* $i = 1, \ldots n$ *hacer*

   */*Construimos el conjunto* P_i *de todas las cliques del subgrafo* G^i *formado por los vértices* $v_1, \ldots, v_i$*, de la siguiente manera:*/*

   Si $i = 1, P_1 = \{\{v_1\}\}$

   Sino

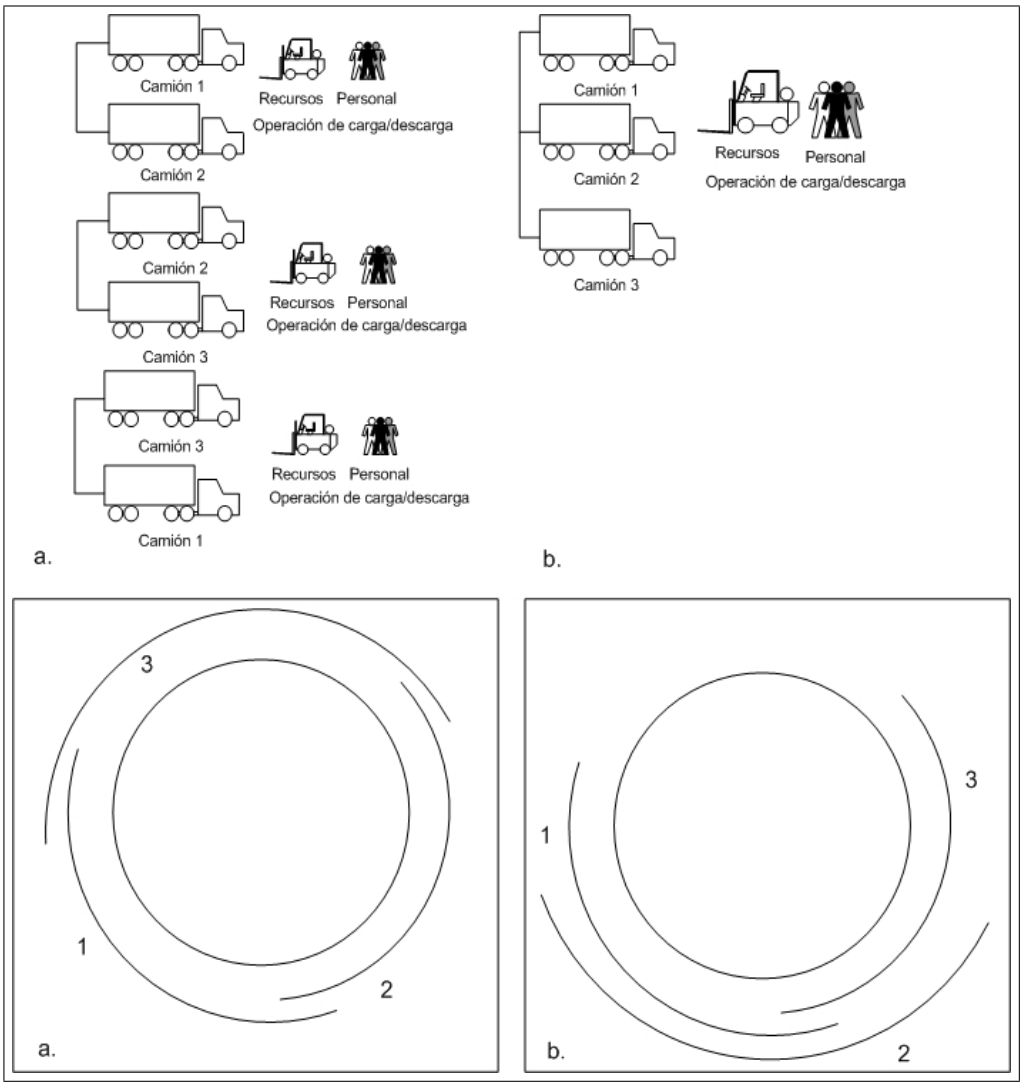

Figura 4.2: Aplicación de grafos PCA a una empresa de logística.

$P'_i = \{\{v_i\} \cup (C \cap N(v_i)))|$*para cada*$C \in P_{i-1}\}$
P_i = *el conjunto de elementos maximales de* $P'_i \cup P_{i-1}$
Si P_i *en supera los* i *elementos, devolver que* G *no es HCA.*

2. $P_n = C_1, \ldots, C_k$ *es el conjunto de cliques de* $G, k \leq n$. *Construir la matriz clique* M *de* G.

3. *Verificar si* M *tiene la propiedad de unos circulares usando el algoritmo de Fulkerson y Gross* [1]

4. *Devolver que* G *es HCA si* M *tiene unos circulares y que* G *no es HCA sino.*

Este algoritmo es $O(n^3)$, donde la parte más costosa es el cómputo de las cliques de G.

4.4. Implementación del algoritmo de Lin y Szwarcfiter

El algoritmo de Lin y Szwarcfiter presenta el primer algoritmo lineal de reconocimiento de grafos HCA. El algoritmo $O(n^3)$ de Gavril de la sección anterior era el mejor resultado hasta el momento. Lin y Szwarcfiter describen nuevas caracterizaciones para grafos HCA en base a subgrafos prohibidos, que describimos en 4.2. Estas caracterizaciones llevan al algoritmo lineal de reconocimiento.

4.4.1. Descripción del algoritmo

El algoritmo parte de un modelo CA del grafo G. Para ello encontrar este modelo cita el algoritmo de McConnell; si G no es arco circular tampoco será HCA y el algoritmo termina. Ahora, el teorema 4.2 permite restringir el trabajo a un modelo estable de G. La idea entonces es convertir el modelo CA de G en un modelo estable y luego comprobar si este modelo es HCA. Describimos el algoritmo a continuación, para luego mostrar la correctitud del mismo usando los fundamentos teóricos del paper.

Algoritmo 4.2 *Algoritmo de Lin y Szwarcfiter de reconocimiento de grafos HCA*

1. *Aplicar el algoritmo de Kaplan y Nussbaum para reconocer si G es un grafo arco circular. Si lo es, sea (C, A) el modelo de G que es el output del algoritmo citado. Si no lo es, terminar el algoritmo (G no es HCA)*

2. *Transformar a G en un modelo estable, utilizando el algoritmo 4.3*

3. *Verificar si (C, A) es un modelo HCA, utilizando el algoritmo 4.4.*

[1] Gavril cita este algoritmo, pero puede utilizarse alguno de los algoritmos posteriores, más eficientes, de la sección 2.3.2 para verificar esta propiedad.

4. *Devolver que G es HCA si la comprobación anterior es afirmativa y que G no es HCA en otro caso.*

Algoritmo 4.3 *Algoritmo de conversión de un modelo (C, A) en un modelo estable.*

1. *Para* $j = 1, \ldots, n$ *hacer*

   Examinar los extremos de A, comenzando por t_j*, en dirección horaria, y elegir el extremo inicial* s_i *más cercano que satisfaga* $i = j$ *o* $A_i \cap A_j = \emptyset$

   Mover t_j *de manera que se convierta en el extremo que preceda inmediatamente a* s_i *en el modelo.*

   */*En el paper se llama* expansion(t_j) *a esta operación*/*

2. *Para* $i = 1, \ldots, n$ *hacer*

   Examinar las t-secuencias, comenzando por S = SEQUENCE(si) avanzando en dirección antihoraria y eligiendo la t-secuencia T, lo más cerca posible de S*, que satisfaga* $i = j$ *o* $A_i \cap A_j = \emptyset$ *para algún* $t_j \in T$*.*

   Mover s_i *en sentido antihorario hacia T, transformando a T en la secuencia* $T' s_i T''$*, con* $T' = \{t_j \in T | i = j \vee A_i \cap A_j = \emptyset\}$ *y* $T'' = T \setminus T'$

   */*En el paper se llama* expansion(s_i) *a esta operación*/*

Algoritmo 4.4 *Algoritmo de verificación de la propiedad HCA en un modelo arco circular (C, A)*

1. *Para cada arco* $A_i \in A$ *hacer*

   Buscar dos arcos A_j, A_k *cuyos extremos satisfagan el ordenamiento* $s_i, t_k, s_j, t_i, s_k, t_j$

2. *Si para algún arco fue posible hallar un ordenamiento como se pide, devolver que el modelo no es HCA. Sino seguir.*

3. *Construir el modelo* $(C, \overline{A})$*.*

4. *Construir el grafo intersección G' de* $(C, \overline{A})$*.*

5. *Verificar si G' es cordal, utilizando el algoritmo de Golumbic. Si lo es, devolver que el modelo es HCA, sino devolver que no lo es.*

Veamos ahora por qué el algoritmo funciona. Primero, los autores demuestran que cualquier modelo CA admite un modelo estable equivalente. Nos interesa encontrar un modelo estable m de G por el Teorema 4.2, porque entonces podemos olvidarnos de todos los posibles modelos de G y concentrarnos únicamente en m. Si m es HCA, G también lo será y viceversa. Faltaría ver por qué 4.4 verifica que el modelo es HCA. Esto se fundamenta en el siguiente:

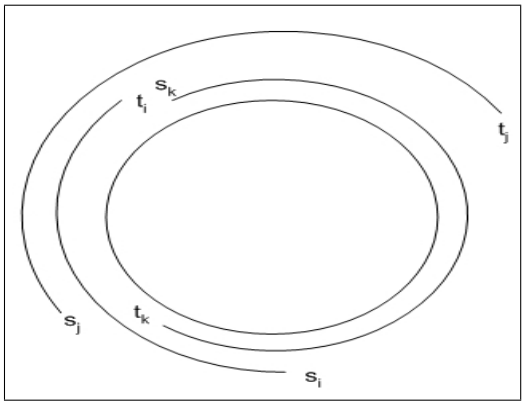

Figura 4.3: Se muestran aquí tres arcos A_i, A_j, A_k con el ordenamiento $s_i, t_k, s_j, t_i, s_k, t_j$. Notar cómo los tres arcos cubren el círculo, pero dos cualquiera no.

Teorema 4.3 *Un modelo arco circular (C, A) de G es HCA sii*

1. *si tres arcos de A cubren a C entonces dos de estos tres arcos tambien lo cubren, y*

2. *el grafo intersección de* $(C, \overline{A})$ *es cordal.*

El procedimiento 4.4 es exactamente la verificación de estas dos condiciones. La razón del paso 1 del algoritmo es la siguiente: Notar que si tenemos tres arcos A_i, A_j, A_k con el ordenamiento $s_i, t_k, s_j, t_i, s_k, t_j$ es precisamente decir que los tres arcos cubren el círculo, pero A_j, A_k no lo cubren (Fig 4.3), con lo cual podemos utilizar esto para verificar si se viola la condición i del teorema anterior.

Describimos ahora el algoritmo de Golumbic de verificación de grafos cordales.
Un vértice v de G es *simplicial* si $N(v)$ es un grafo completo. Un *esquema perfecto de eliminación* es un ordenamiento σ de los vértices de G de manera tal que cada vértice v_i es simplicial con respecto al subgrafo inducido de G formado por los vértices $v_i, \ldots, v_n$. El algoritmo está basado en el siguiente

Teorema 4.4 *Un grafo G es cordal si y sólo si un ordenamiento* σ *que surge de recorrer el grafo en LexBFS es un esquema perfecto de eliminación.*

Algoritmo 4.5 *Algoritmo de Golumbic de reconocimiento de grafos cordales*

1. *Recorrer en LexBFS al grafo input G y generar un arreglo* σ *de los vértices en el orden en que se visitan*

2. *Verificar si sigma es un esquema perfecto de eliminación.*

El procedimiento LexBFS fue introducido en la sección 2.5. Lo resumimos a continuación.

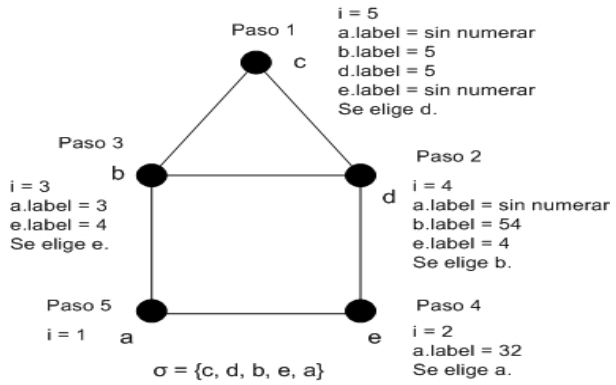

Figura 4.4: Recorrido en LexBFS

Algoritmo 4.6 *LexBFS*

1. *Asignar la etiqueta* $\emptyset$ *a todos los vértices.*
2. *Para* $i = n$ *hasta 1 hacer*

   Elegir un vértice sin numerar con la etiqueta más grande.
   $\sigma(i) \leftarrow v$ */*Quiere decir que v aparece en orden i en el LexBFS*/*
   Para cada vértice sin numerar $w \in N(v)$ *hacer*
   Agregar i a etiqueta(w)

La única diferencia con respecto al LexBFS que vimos antes es que aquí los vértices se numeran *hacia atrás*. Esto es, si el resultado del recorrido LexBFS de antes es un arreglo de vértices $v_1, v_2, \ldots, v_n$ este procedimiento devuelve $\sigma = v_n, v_{n-1}, \ldots, v_1$. La razón de esto es que la caracterización 4.4 presupone este orden para recorrer el grafo. A continuación presentamos un grafo que ilustra el recorrido del algoritmo

Algoritmo 4.7 *Verificar si* σ *es un esquema perfecto de eliminación*

1. *Para* $i = 1$ *hasta* n *- 1 hacer*

   $v \leftarrow \sigma(i)$
   Comprobar si $S = \sigma(j) | \sigma(j) \in N(v), j > i$ *es un grafo completo.*
   Si no lo es, devolver que σ *no es un esquema perfecto de eliminación.*
2. *Devolver que* σ *es un esquema perfecto de eliminación.*

4.4.2. Descripción de la implementación

Describiremos aquí la implementación de cada uno de los procedimientos que conforman el algoritmo, de manera análoga a lo trabajado en el caso de grafos UCA. Trabajaremos en dos niveles de abstracción, con una descripción conceptual primero y luego se explicando la implementación en un lenguaje de programación.

Descripción conceptual

- *Algoritmo 4.3 (Algoritmo de conversión de un modelo (C, A) en un modelo estable)*

Estructuras de datos Suponemos que tenemos una lista L circular de extremos ordenados como aparecen en el modelo, en sentido horario.

```
1. Para cada arco Aj hacer /*expansion(tj)*/
2.    Sea k el indice de A.tj en L
3.    Repetir
4.       i = (k + 1) modulo n
5.       Si L[i] es un extremo inicial si, con i = j o
         adyacentes(Aj, Ai) = false
6.          found = true
7.       Sino
8.           swap(L[i], L[k])
9.    hasta que found = true
10.Fin para
11.Para cada arco Aj hacer /*expansion(si)*/
12.   Sea k el indice de A.tj en L
13.    Repetir
14.       i = (k - 1) modulo n
15.       Si L[i] es un extremo final ti, con i = j o
          adyacentes(Aj, Ai) = false
16.          found = true
17.       Sino
18.          L.swap(i,k)
19.    Hasta que found = true
20.Fin para
```

Los autores demuestran que este algoritmo transforma el modelo en uno estable. La Fig. 4.5 ilustra las operaciones de expansion(t_j) y expansion(s_i). Notar que el modelo resultante es estable.

- *Algoritmo 4.4 (Algoritmo de verificación de la propiedad HCA en un modelo arco circular (C, A))*

Estructuras de datos Utilizaremos para este algoritmo dos listas de extremos, L_1 y L_2. Además, suponemos que tenemos una lista circular E de extremos ordenados como aparecen en el modelo, en sentido horario.

```
/*Condicion i*/
1. Para cada arco Aj hacer
2.    Sea L1 la lista de extremos contenidos entre Aj.sj y Aj.tj en C
      ordenados en sentido horario
3.    Eliminar de L1 los pares de extremos sq, tq del mismo arco.
4.    Sea L2 la lista de extremos formadas por
      los extremos iniciales sk si tk esta en L1 y
```

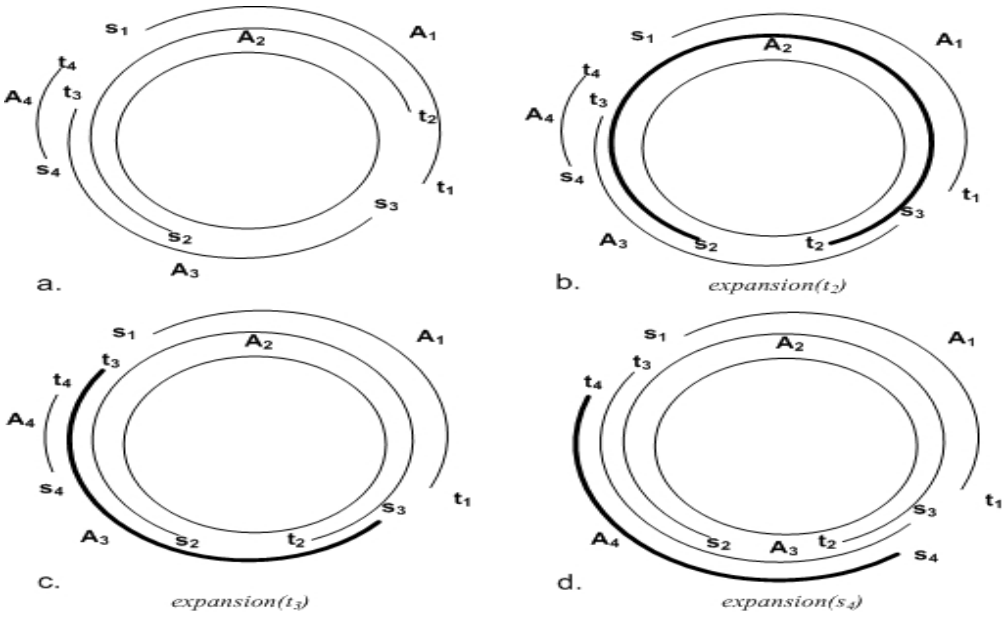

Figura 4.5: Corrida de algoritmo de transformación del modelo en un modelo estable. El primer modelo es el original, y los siguientes ilustran los arcos que se van expandiendo sucesivamente

```
      los extremos finales tl si sl esta en L1,
      ordenados en sentido horario.
5.    Mientras no (L1 es vacio o (L1.primero y L2.ultimo son
      extremos finales)) hacer
6.       Si L1.primero es un extremo inicial sq,
         eliminar sq de L1 y tq de L2
7.       Si L2.ultimo es un extremo inicial sq,
         eliminar sq de L2 y tq de L1
8.    Fin mientras
9.    Si L1 no es vacio, devolver que G no es HCA, sino seguir
9. Fin para
   /*Condicion ii*/
   /*Construir el modelo complemento (C, Acomp)*/
10.Para cada arco Ai hacer
11.   E.swap(Ai.si, Ai.ti)
12.Fin para
13.Guardar este modelo en (C, Acomp)
14.Computar el grafo G' de interseccion de (C, Acomp)
15.Devolver true si G' es cordal y false sino.
```

Graficamos el funcionamiento del algoritmo en la Fig. 4.6

- *Algoritmo 4.6 (LexBFS)*

Para esta implementación no necesitaremos calcular las etiquetas de los vértices, la idea será mantener a los vértices sin numerar en orden lexicográfico.

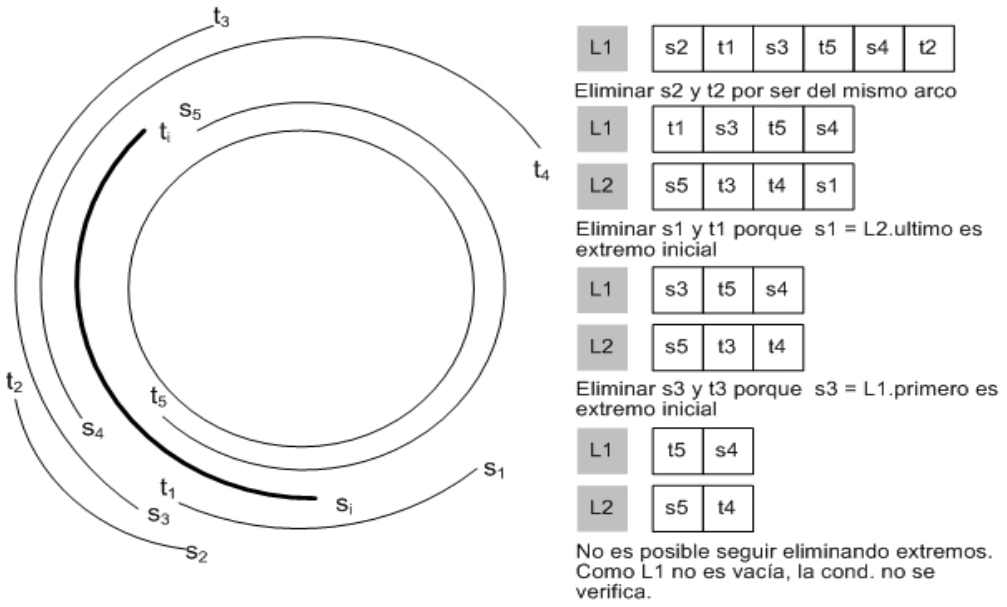

Figura 4.6: Verificación de la condición i. El arco A_i sobre el que se está iterando se muestra en negrita.

Estructuras de datos Utilizaremos una lista doblemente linkeada Q de conjuntos $S_i = \{v \in N(v) | etiqueta(v) = i \wedge \sigma(v) = indefinido\}$, los i ordenados lexicográficamente de mayor a menor. $\sigma(v) = indefinido$ quiere decir que los elementos todavía no fueron recorridos. Cada S_i puede ser implementado con una lista doblemente linkeada. Aumentamos cada S_i además con un FLAG inicialmente en 0. Contamos también con un arreglo SET tal que el elemento SET(w) apunta al conjunto $S_{etiqueta(i)}$.

Procedimiento1 LexBFS

```
1. Crear un conjunto S con todos los vertices y colocarlo en Q
2. Para i = n hasta 1 hacer
3.    Elegir del ultimo conjunto S1 de Q algun elemento v
4.    Hacer sigma[i] = v
5.    Borrar a v de S1
6.    Llamar al procedimiento Update
7. Fin para
```

Procedimiento2 Update

```
 1.Para cada vertice v sin numerar en N(v) hacer
 2.   Si FLAG(SET(w)) = 0
 3.      Crear un nuevo conjunto S' e insertarlo
 4.      en Q inmediatamente detras de SET(w)
 5.      FLAG(SET(w)) = 1
 6.      FLAG(S') = 0
 7.      Poner un puntero a SET(w) en la lista FIXLIST
```

```
 8.   Fin Si
 9.   Sea S' el conjunto inmediatamente detras de SET(w) en Q.
10.   Borrar a w de SET(w)
11.   Agregar a w a S'
12.   SET(w) = S'
13.Fin para
14.Para cada conjunto S en FIXLIST hacer
15.   FLAG(S) = 0
16.   Si S esta vacio
17.      Borrar a S de Q
10.Fin para
```

Dijimos que la lista Q mantiene el invariante que los S_i estan en orden lexicogrfico de los i, de menor a mayor. Entonces en cada paso del procedimiento principal tomamos un elemento cualquiera del último conjunto de Q, que sabemos que tiene etiqueta lexicográficamente más grande. El procedimiento Update mantiene la estructura para que conserve este invariante. Cuando elegimos el vértice v en el paso 3 del procedimiento LexBFS, y lo borramos del conjunto correspondiente en el paso 5, porque los S_i contienen solamente vértices sin numerar. Ahora creamos un nuevo conjunto S_{l-i} para cada conjunto S_l que contenga un vértice sin numerar $w \in N(v)$ (Usamos la variable FLAG para los casos en que dos vértices $w_1, w_2 \in N(v)$ pertenezcan al mismo conjunto S_l, para no crear dos veces el nuevo conjunto). De estos S_l borramos todos estos vértices w y los colocamos en el nuevo conjunto S_{l-i}. Este conjunto S_{l-i} se agrega a la lista Q inmediatamente después de S_l. Es claro que este método mantiene el orden lexicográfico correcto sin necesidad de calcular las etiquetas. La lista FIXLIST se utiliza para almacenar aquellos S_i cuyo FLAG debe resetearse para la próxima iteración y para eliminar aquellos que puedan haber quedado vacíos en el ciclo anterior.

- *Algoritmo 4.4 (Verificar si σ es un esquema perfecto de eliminación)*

Estructuras de datos Utilizaremos para representar a σ y σ^{-1} dos arreglos. A[u] es un conjunto con repeticiones (multiconjunto), donde vamos guardando todos los vértices que buscamos testear si son adyacentes a u.

```
1. Para cada vertice v hacer
2.    A[v]= vacio
3. Fin para
4. Para i = 1 hasta n-1 hacer
5.    v = sigma[i]
6.    X = el conjunto de los vecinos x de v tales
      que sigmaInverse(v) < sigmaInverse(x)
7.    Si X no es vacio
8.       j = Computar el minimo sigmaInverse(x), con x en X.
9.       u = sigmaInverse(j)
10.   Agregar X - {u} a A[u]
11.   Si A[v]-N(v) no es vacio
```

```
12.      Devolver false
13.Fin Para
14.Devolver true
```

Como no se ve trivialmente que el algoritmo verifica esquemas perfectos de eliminación, explicaremos la demostración de correctitud de Golumbic:
Demostración de Correctitud:

El algoritmo contesta FALSE en el paso $\sigma^{-1}(u)$-esimo si y sólo si existen vértices $u, v, w, \sigma^{-1}(v) < \sigma^{-1}(u) < \sigma^{-1}(w)$, con u definido en el paso 9 del algoritmo en el paso $\sigma^{-1}(v)$-esimo y u, w adyacentes a v, (con lo cual w se almacena en A[u]), pero u, w no son adyacentes, entonces el paso 11 no es vacío. Pero entonces σ no es un esquema perfecto de eliminación, porque u y w aparecen después de v en el ordenamiento, son adyacentes a v pero no son adyacentes entre s (v no es simplicial).

Supongamos ahora que el algoritmo contesta TRUE y σ no es un esquema perfecto de eliminación. Sea v un vértice con $\sigma^{-1}(v)$ lo más grande posible tal que $X = \{w | w \in N(v) \wedge \sigma^{-1}(v) < \sigma^{-1}(w)\}$ no es completo. Este v tiene que existir, si los X fueran todos completos, el esquema sería perfecto. Sea ahora u el vértice de X definido en el paso 6 en el paso $\sigma^{-1}(v)$-esimo, luego de lo cual $X - u$ se agrega a $A[u]$. Como cuando le llega el turno al vértice u (en la iteración $\sigma^{-1}(u)$-esima) no se ejecuta el paso 12 (porque devuelve TRUE), se cumple que cada $x\ in X - \{u\}$ es adyacente a u. Además, cada par $x, y\ in X - \{u\}$ es adyacente, por la maximalidad de $\sigma^{-1}(v)$. Entonces X es completo, un absurdo. $\triangle$

Complejidad y alcance del algoritmo

La complejidad del algoritmo publicado por Lin y Szwarcfiter es lineal. En esta implementación, algunas partes del algoritmo son lineales y otras son $O(n^2)$, a saber:

- En el algoritmo 4.3, en el segundo ciclo de los pasos 11 a 20, como se examinan para cada extremo inicial si la secuencia completa de extremos finales, esto lleva un costo de $O(n^2)$. Los autores sugieren una implementación alternativa más complicada para mejorar este orden.
- Los pasos 4 y 5 del algoritmo 4.4 pueden resultar en un grafo G' con un número cuadrático de aristas. Para optimizar esto, Lin y Szwarcfiter muestran que el complemento $\overline{G'}$ tiene a lo sumo la misma cantidad de aristas que el grafo input G. Entonces se podría verificar si $\overline{G'}$ es *co-cordal* (grafos cuyo complemento es cordal). Para ello el autor cita algunos algoritmos publicados que verifican esta propiedad.

Descripción de la implementación en máquina

Describiremos a continuación algunos puntos de interés en la implementación, por no ser una aplicación inmediata de los procedimientos que se explican arriba.

- El paso 10. del procedimiento 4.4 puede implementarse en orden $O(|N(v)|+|A[v]|)$ con ayuda de un arreglo TEST de n posiciones, inicialmente en cero, como sigue:

```
1. Para cada w en N(v) hacer
2.    TEST(w) = 1
3. Fin para
4. Para cada w en N(v) hacer
5.    Si TEST(w) = 0 entonces
6.       Devolver NO VACIO
7. Fin para
8. Para cada w en N(v) hacer
9.    TEST(w) = 1
10.Fin para
11.Devolver VACIO
```

- El algoritmo de Golumbic de reconocimiento de grafos cordales trabaja sobre grafos conexos. Fue necesario entonces calcular las componentes conexas de G en orden lineal. Esto se puede hacer fácilmente recorriendo el grafo en BFS mientras se pueda. Cada vez que el recorrido BFS se detiene se encontró una componente conexa, se comienza de nuevo mientras existan vértices sin recorrer.

- En esta implementación también se presentó el problema que citamos antes, que fue necesario prescindir de las listas que provee Java (tipo LinkedList), por razones de eficiencia. (Ver sección 3.3.2). Específicamente, para este algoritmo la dificultad se presentó en el procedimiento LexBFS, en los conjuntos S_i y en la lista Q, donde se necesitan inserciones y borrados en $O(1)$, que las listas de Java no implementan como se busca.

- Explicaremos la construcción de las listas L1 y L2 en el algoritmo de verificación de la propiedad HCA en un modelo arco circular (C, A). Recordar que tanto la lista L1 como la lista L2 deben estar ordenadas en sentido horario de C. La idea será construir las listas de todos los arcos antes de comenzar el algoritmo. Procedemos así,

```
1. Para cada arco Ai hacer
2.    Guardar todos los vertices entre Ai.si y Ai.ti en una lista L1aux.
      /* Esto es inmediato porque contamos con una lista de extremos E ordenada
      en el sentido horario de C (L1aux es una sublista de E). A medida que
      recorremos la lista, marcamos que arcos tienen ambos extremos en L1aux.*/
3.    Construir L1 con todos los elementos de L1aux, salvo los
      extremos del mismo arco, que marcamos antes.
4.    /*Guardar en un arreglo de listas L2aux, de 2n elementos, en las
      posiciones correspondientes a los extremos opuestos de L1 el arco que se
      esta procesando:*/
      Para cada elemento ej de L1
5.       L2aux[ej] = Ai
```

```
6.    Fin para
7. Fin para
8. /**Ahora recorremos en sentido horario los extremos, y vamos agregando
   en las listas L2 que correspondan los extremos a medida que van
   apareciendo. Esto garantiza que las listas L2 esten luego ordenadas
   en sentido horario*/
   Para cada extremo ek de E
      Para cada arco Ai en L2aux[ek]
         Agregar ek al final de L2[Ai]
      Fin para
   Fin para
```

4.4.3. Resultados computacionales

Antes de presentar los resultados del comportamiento del algoritmo en máquina, nos detendremos brevemente sobre el problema del testing, de por sí interesante. En este caso la estrategia seguida es distinta de la implementación UCA, que generaba modelos para los que se sabía previamente si correspondían a grafos UCA o no. Lo que se hizo fue constatar mediante otros procedimientos mucho más fáciles de programar y con peor orden de complejidad cada una de las partes del algoritmo, a saber:

- *Convertir m en un modelo estable m1.* Se verificó aquí que el modelo m y el modelo m1 sean equivalentes, es decir, que correspondan al mismo grafo intersección G.

- *Condición i del algoritmo: Si tres arcos de A cubren a C en m1 entonces dos de estos tres arcos tambien lo cubren.* Verificamos en $O(n^3)$ si cuando el algoritmo contesta que esta condición se cumple efectivamente todo conjunto de tres arcos de A la cumple. Adicionalmente, si el algoritmo presenta un certificado de que encontró tres arcos donde esto no verifica, se testea si el certificado es correcto.

- *Condición ii del algoritmo: El grafo intersección de* $(C, \overline{A})$ *es cordal.* Aquí utilizamos un algoritmo de fuerza bruta para eliminar de a uno los vértices simpliciales del grafo, para comprobar si es cordal(si lo es, debería poder hacer esta operación $n-1$ veces). Encontramos un vértice simplicial mirando cada vértice v y verificando si $N(v)$ es completo, hasta que encontramos un vértice que cumpla la condición.

Veamos ahora los resultados. Se contaron todas la operaciones. No se consideró necesario discriminarlas por categorías, ya que son todas operaciones aritméticas, comparaciones y asignaciones; no existen multiplicaciones ni divisiones que son operaciones más costosas. La Fig. 4.7 muestra la cantidad de operaciones en función de los arcos y las adyacencias (se considera $max(n, m)$) del modelo input. La curva no es claramente lineal porque como vimos arriba existen partes del algoritmo que tienen un orden cuadrático. De ahí las fluctuaciones en los puntos. La línea de tendencia es calculada por el método numérico de mínimos cuadrados, aplicado al caso lineal.

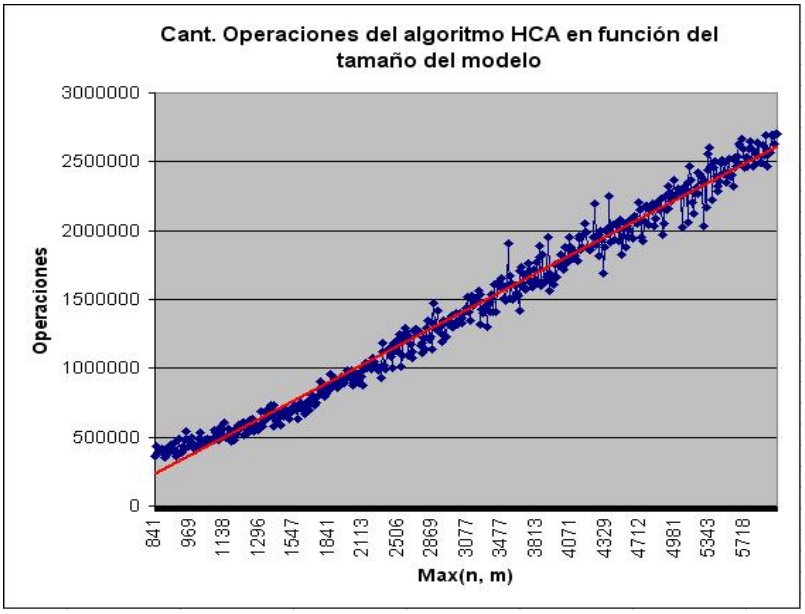

Figura 4.7: Eficiencia general del algoritmo de reconocimiento HCA

Nos interesa ahora comprobar específicamente algunas partes del algoritmo, para ver si efectivamente tienen el orden buscado. La primera de ellas es el reconocimiento de grafos cordales de Golumbic. Notar que debemos considerar como input el grafo intersección del complemento del modelo inicial (Ver algoritmo). La Fig. 4.8 muestra el comportamiento del procedimiento en función de la cantidad de $max(n, m)$ de este modelo complemento. Podemos ver que es aproximadamente lineal.

El cómputo de este modelo complemento y su grafo intersección puede costar $O(n^2)$. Efectivamente, la Fig. 4.9 nos muestra que se comporta así en la corrida en máquina.

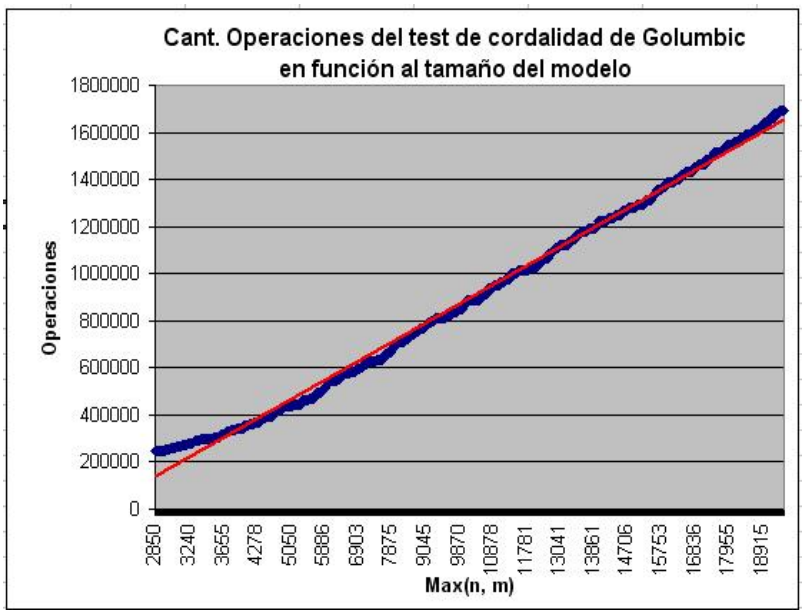

Figura 4.8: Eficiencia del test de reconocimiento de grafos cordales

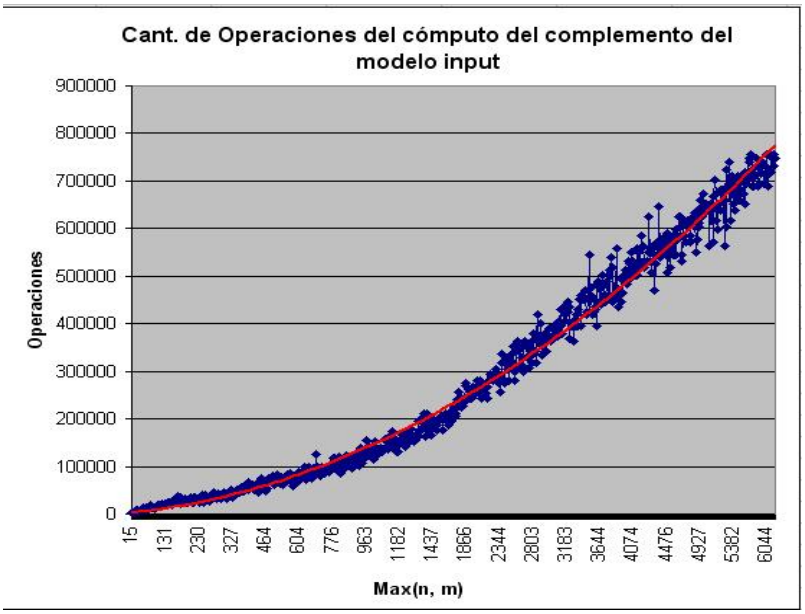

Figura 4.9: Eficiencia del cómputo del modelo complemento.

Capítulo 5

Conclusiones

Hemos presentado en este trabajo los principales algoritmos de reconocimientos de los grafos arco circulares y de las subclases arco circular propio, unitario y Helly. Para el caso de cubrimiento por dos cliques del algoritmo PCA de Tucker, completamos el esquema sugerido por Spinrad en [24]. Spinrad reduce el problema al reconocimiento de grafos bipartitos de permutación. Nosotros aplicamos los resultados de Hell y Huang para reconocimiento de estos grafos de [13] y [12], para obtener un algoritmo completo.

En el capítulo 1, seguimos la sugerencia de Spinrad en [24] y describimos la forma de reducir el orden de complejidad cuadrático del algoritmo de Tucker a orden lineal, aplicando algoritmos modernos al problema de unos consecutivos y al caso del cubrimiento por dos cliques de G. Esto es interesante porque el algoritmo de Tucker con estas modificaciones es más fácil de implementar que el algoritmo lineal de Deng et al.

Es notable la fuerte relación de los algoritmos presentados en esta tesis con el problema de reconocer grafos de comparabilidad, y encontrar una orientación transitiva en caso afirmativo. El caso I del algoritmo UCA de Tucker (cubrimiento del grafo G con dos cliques) se reduce al reconocimiento de grafos de permutación (G y $\overline{G}$ son de comparabilidad), igual que el caso de dos cliques del algoritmo PCA de Tucker. En el algoritmo PCA de Deng et al. también se utiliza. Si bien existen algoritmos lineales para reconocer grafos de comparabilidad ([23]), todavía son complejos para servir de base a otros algoritmos.

La implementacíon de los algoritmos de las subclases arco circular unitario y arco circular Helly arrojaron resultados que están dentro del orden teórico calculado en los papers. Observamos en estas implementaciones que la eficiencia puede obligar a prescindir de una implementación con tipos de datos completamente encapsulados. Por ejemplo, dado un tipo de datos T, en algunos casos es demasiado costoso utilizar un tipo lista, que esté en una capa inferior a T y se debió incluir en los elementos de T punteros anterior/siguiente.

La implementación del recorrido LexBFS para el algoritmo HCA ofrece posibilidades interesantes de ampliación, como una posible continuación de este trabajo. Con pequeñas modificaciones, el LexBFS puede utilizarse para reconocer, además de los grafos bipartitos de permutación y los grafos cordales, los grafos de intervalos y los grafos propios de intervalos. El interés radica además

en que el recorrido es relativamente sencillo de programar y se realiza en orden lineal.

Sería interesante desarrollar una implementación del algoritmo de reconocimiento UCA de Lin y Szwarcfiter, ya que este algoritmo genera un modelo arco circular unitario, a diferencia del algoritmo UCA que implementamos en este trabajo. Además, el orden de complejidad se reduce a $O(n + m)$.

Hemos recibido información acerca de la reciente publicación de dos nuevos algoritmos de reconocimiento PCA y UCA, por parte de Kaplan y Nussbaum, de orden $O(n)$. Se trata de un resultado importante, en la medida que provee certificados en el caso de que el grafo input G no sea PCA o UCA. Hasta ahora no se contaba con algoritmos de certificación. Esto haría también relevante una implementación de estos algoritmos.

Apéndice

Explicaremos aquí cómo utilizar el software que implementa los algoritmos. Para ello, la PC donde se correrá el programa debe contar con el entorno de ejecución de Java. El mismo se encuentra en el CD de la presente tesis, puede instalarse desde all. Luego, desde el directorio dentro del entorno de ejecución de Java, donde se encuentra el archivo java.exe (en general está dentro del directorio bin), debe tipearse lo siguiente: java -jar c:\Tesis\tesis.jar, donde 'c:\Tesis\' se reemplaza por la ruta donde se encuentre el archivo. La Fig. 1 muestra la interfaz que se visualiza cuando se ejecuta el programa.

Ahora, debe informarse al programa un archivo con el modelo arco circular que es input para el algoritmo UCA y para el algoritmo HCA. Notar que el modelo debe ser PCA en el caso del algoritmo UCA. El archivo se informa a través de la opción 'Archivo / Abrir' de la barra de menú de la ventana descripta antes. La estructura del archivo es como sigue.

```
# comentarios
etiqueta_arco1, extremo_inicial1, extremo_final1
etiqueta_arco2, extremo_inicial2, extremo_final2
...
```

Las líneas deben estar separadas por retorno de carro y avance de línea. Las líneas que comienzan con `#` son comentarios. En cuanto a las demás líneas, `etiqueta_arco` es un número entero positivo que asigna un número a un arco, que lo identifica. `extremo_inicial` es un número positivo con decimales que hace referencia al extremo s_i del arco A_i, donde i es la etiqueta. Análogamente, `extremo_final` es un número positivo con decimales que hace referencia al extremo t_i del arco A_i, donde i es la etiqueta. Los números sirven a los fines de establecer un orden de cómo aparecen los extremos alrededor de C, comenzando por el 0. Por ejemplo, la Fig. 2 muestra un modelo y el archivo que lo describe.

Debe tenerse en cuenta que no existan extremos con el mismo número.

Finalmente, presionando los botones 'Reconocer UCA' o 'Reconocer HCA' de la ventana de la aplicación, se corren los algoritmos correspondientes y se muestra la respuesta de los mismos. Para informar otro archivo, se deberá pulsar nuevamente la opción 'Archivo / Abrir' de la barra de men'ɣ

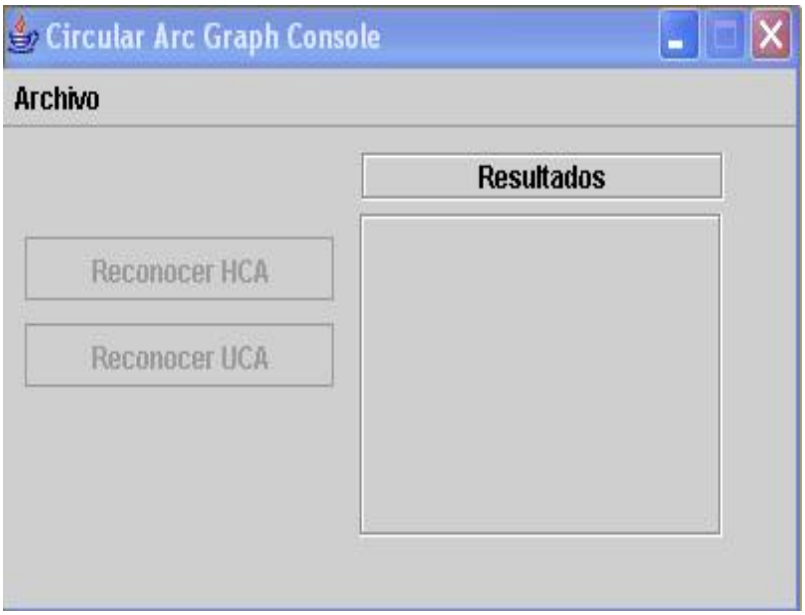

Figura 1: Ventana de la aplicación que implementa los algoritmos.

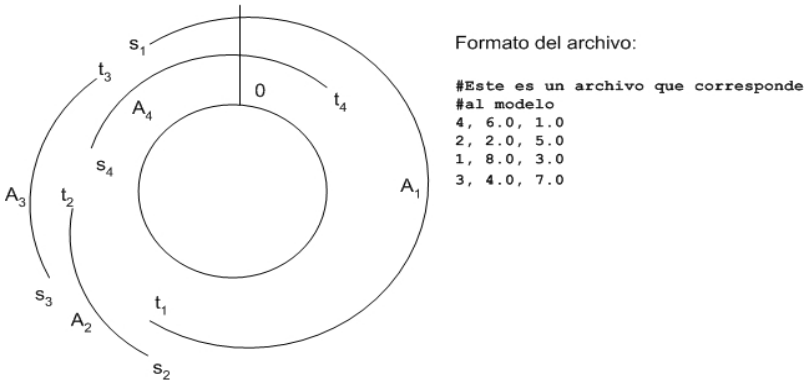

Figura 2: Ilustración del formato del archivo.

Bibliografía

[1] K. S. Booth, G. S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, J. Comput. Syst. Sci., 13(3) (1976), pp. 335 - 379

[2] G. Confessore, P. Dell'Olmo and S. Giordani, An approximation result for a periodic allocation problem, Discrete Applied Mathematics 112 (2001), pp. 53 – 72

[3] V. Conitzer, J. Derryberry, and T. Sandholm. Combinatorial auctions with structured item graphs. In Proceedings of the Nineteenth National Conference on Artificial Intelligence (2004), pp. 212 - 218

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, Introduction to Algorithms, McGraw Hill Boston (2001)

[5] X. Deng, P. Hell and J. Huang, Linear time representation algorithms for proper circular-arc graphs and proper interval graphs, SIAM Journal of Computing, 25 (1996), pp. 390 – 403

[6] G. A. Durán, Sobre grafos intersección de arcos y cuerdas en un círculo, Tesis Doctoral (2000)

[7] G. A. Durán, A. Gravano, R. M. McConnell, J. Spinrad, A. Tucker, Polynomial time recognition of unit circular-arc graphs, Journal of Algorithms, Volume 58 (2006), pp. 67 – 78

[8] D. R. Fulkerson, O. A. Gross, Incidence matrices and interval graphs, Pacific J. Math 15 (1965), pp. 835 – 855

[9] F. Gavril, Algorithms on circular-arc graphs, Networks 4 (1974), pp. 357 – 369

[10] M. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press New York (1980), pp. 81 – 90

[11] H. Hadwiger, H. Debrunner, V. Klee, Combinatorial Geometry in the Plane, Holt Rinehardt and Winston, New York, (1964), pp. 54

[12] P. Hell and J. Huang, Interval bigraphs and circular arc graphs, J. Graph Theory 46 (2004), pp. 313 - 327.

[13] P. Hell and J. Huang, Certifying LexBFS Recognition Algorithms for Proper Interval Graphs and Proper Interval Bigraphs, Manuscript (2003)

[14] W. L. Hsu, $O(mn)$ algorithms for the recognition and isomorphism problems on circular-arc graphs, SIAM J. Comput. 24 (1995), pp. 411 – 439

[15] W.L. Hsu, A simple test for the consecutive ones property, J. Algorithms 42 (2002), pp. 1 - 16.

[16] W. L. Hsu, R. M. McConnell, PC-trees and circular-ones arrangements, Theor. Comput. Sci.,296(1) (2003), pp. 99 - 116.

[17] L. Hubert, Some applications of graph theory and related non-metric techniques to problems of approximate seriation: the case of symmetry proximity measures, British J. Math. Statist Psychology 27 (1974), pp. 133 – 153.

[18] H. Kaplan, Y. Nussbaum, A Simpler Linear-Time Recognition of Circular-Arc Graphs, The Tenth Scandinavian Workshop on Algorithm Theory (2006)

[19] M. C. Lin, J. L. Szwarcfiter, Characterizations and Linear Time Recognition of Helly Circular-Arc Graphs, The Twelfth Annual International Computing and Combinatorics Conference(2006)

[20] M. C. Lin, J. L. Szwarcfiter, Efficient Construction of Unit Circular-Arc Models, Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (2006), pp. 309 – 315.

[21] M. C. Lin, Comunicación personal.

[22] R. M. McConnell, Linear-time recognition of circular-arc graphs, Algorithmica 37(2) (2003), pp. 93 – 147

[23] R. M. McConnell, and J. P. Spinrad, Modular decomposition and transitive orientation, Discrete Mathematics, 201(1-3) (1999), pp. 189 - 241.

[24] J. Spinrad, Efficient Graph Representations, American Mathematical Society, Providence RI (2003)

[25] J. Spinrad, Circular-arc graphs with clique cover number two, Journal of Combinatorial Theory Series B 44(3) (1988), pp. 300 - 306

[26] E. Eschen, J. Spinrad, An $O(n^2)$ Algorithm for Circular-Arc Graph Recognition, Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (1993), pp. 128 – 137

[27] F. Stahl, Circular genetic maps, J. Cell Physiol. 70 (Suppl. 1) (1967), pp. 1 – 12.

[28] S. Stefanakos, T. Erlebach, Routing in All-Optical Ring Networks Revisited, Proceedings of the 9th IEEE Symposium on Computers and Communications (ISCC) (2004), pp. 288 – 290

[29] K. Stouffers, Scheduling of traffic lights - A new approach, Transportation Res. 2 (1968), pp. 199 – 234.

[30] A. Tucker, Characterizing circular-arc graphs, Bull. Amer. Math. Soc. 76 (1970), pp. 1257 – 1260

[31] A. Tucker, Matrix characterizations of circular-arc graphs, Pacific J. Math. 38 (1971), pp. 535 – 545

[32] A. Tucker, Structure theorems for some circular-arc graphs, Discrete Mathematics 7 (1974), pp. 167 – 195.

[33] A.Tucker, Coloring a family of circular-arc graphs, SIAM J. Appl. Math. 29 (1975), pp. 493 – 502.

[34] A. Tucker, An efficient test for circular-arc graphs, SIAM J. Comput. 9 (1980), pp. 1 – 24