CENATAV
Centro de Aplicaciones de
Tecnologías de Avanzada
MINISTERIO DE LA INDUSTRIA BÁSICA

RNPS No. 2143
ISSN 2072-6260
Versión Digital

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Minería de subgrafos conexos frecuentes reduciendo de número de candidatos.

Andrés Gago Alonso, José Medina Pagola
Jesús Ariel Carrasco Ochoa
José Fco. Martínez Trinidad

RT_004 **Octubre 2008**

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
Ciudad de La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada
MINISTERIO DE LA INDUSTRIA BÁSICA

RNPS No. 2143
ISSN 2072-6260

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Minería de subgrafos conexos frecuentes reduciendo de número de candidatos

Andrés Gago Alonso
José Medina Pagola
Jesús Ariel Carrasco Ochoa
José Fco. Martínez Trinidad

RT_004 **Octubre 2008**

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
Ciudad de La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

Minería de subgrafos conexos frecuentes reduciendo el número de candidatos

Andrés Gago Alonso[1,2], José E. Medina Pagola[1], Jesús Ariel Carrasco Ochoa[2], and José Fco. Martínez Trinidad[2]

[1] Departamento de Minería de Datos
Centro de Aplicaciones de Tecnología de Avanzada (CENATAV),
7a ♯ 21812 e/ 218 y 222, Rpto. Siboney, Playa, C.P. 12200, La Habana, Cuba.
{agago,jmedina}@cenatav.co.cu
[2] Coordinación de Ciencias Computacionales,
Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE),
Luis Enrique Erro No. 1, Sta. María Tonantzintla, Puebla, CP: 72840, Mexico.
{ariel,fmartine}@inaoep.mx

RT_004 CENATAV
Oct. 2008

Resumen: En el presente reporte técnico, se presenta un nuevo algoritmo para la minería de subgrafos conexos frecuentes llamado gRed (por sus siglas en inglés, *graph candidate reduction miner*). Este algoritmo utiliza la codificación DFS (una forma de representar grafos introducida por Yan y Han) para representar los grafos durante el proceso de minería. En gRed, el proceso de minería fue optimizado introduciendo nuevas propiedades de la codificación DFS para reducir el espacio de candidatos. El desempeño de gRed es comparado contra cuatro de los algoritmos más eficientes y populares encontrados en la literatura (gSpan, MoFa, FFSM y Gaston). La experimentación sobre colecciones de datos reales muestra que nuestra propuesta mejora a gSpan, MoFa y FFSM. Además, gRed obtiene mejores tiempos de respuesta que Gaston para bajos umbrales de soporte cuando las bases de datos son grandes.
Palabras clave: Minería de datos, minería de grafos, patrones frecuentes, subgrafos frecuentes, grafos etiquetados

Abstract: In this technical report, a new algorithm for mining frequent connected subgraphs called gRed (graph Candidate Reduction Miner) is presented. This algorithm uses DFS code (a kind of graph representation introduced by Yan and Jan) to represent the graphs during the mining process. In gRed, the mining process is optimized introducing new properties of the DFS code to reduce the candidate space. The performance of gRed is compared against four of the most popular and efficient algorithms available in the literature (gSpan, MoFa, FFSM and Gaston). The experimentation on real world datasets shows the performance of our proposal overcoming gSpan, MoFa and FFSM. Moreover, gRed achieves better performance than Gaston for low minimal support when databases are large.
Keywords: Data mining, graph mining, frequent patterns, frequent subgraphs, labeled graphs

Índice general

1. Introducción 1
2. Conceptos preliminares 4
3. Estado del arte 6
   - 3.1. Formas canónicas para grafos etiquetados 6
     - Codificación CAM 7
     - Codificación DFS 8
     - Codificación BFS 10
   - 3.2. Minería de subgrafos conexos frecuentes 12
     - Algoritmos basados en Apriori 12
     - Algoritmos basados en crecimiento de patrones 13
   - 3.3. Estructuras de datos para indexar la colección de grafos 17
   - 3.4. Colecciones de grafos usadas internacionalmente 18
4. Nuevas propiedades de la codificación DFS 20
5. Algoritmo gRed 25
6. Resultados experimentales 28
7. Conclusiones 32

Referencias 33

1. Introducción

Hoy en día, debido a los rápidos avances científicos y tecnológicos, se ha incrementado notablemente las capacidades de creación, almacenamiento y distribución de grandes volúmenes datos. Esta situación ha generado la necesidad de nuevas herramientas para transformar estos datos en información o conocimiento útil para la toma de decisiones. Cuando estos datos son complejos y estructurados, este proceso de transformación requiere de técnicas que usualmente tienen una alta complejidad computacional en tiempo y espacio. Ejemplos de tales técnicas son las relacionadas con la minería de subgrafos frecuentes (SF); es decir, encontrar todos los subgrafos que ocurren frecuentemente en una colección de grafos. La minería de subgrafos frecuentes es una rama dentro de la minería de patrones frecuentes.

En los últimos años se ha observado un significativo avance en la minería de patrones frecuentes [14]. Métodos cada vez más eficientes se han ido desarrollando para la minería de conjuntos frecuentes [1,2,10,52,51,15], patrones secuenciales [3,34] y subárboles frecuentes [4,50,5] en colecciones de datos. Sin embargo, en muchas aplicaciones científicas y comerciales, han estado presentes patrones con una complejidad estructural mayor que la de los conjuntos, las secuencias y los árboles, por ejemplo los grafos. Es por eso que se ha requerido el diseño de métodos especializados en minería de SF sobre colecciones de grafos [14,30]. Estas colecciones también son nombradas en la literatura como bases de grafos o conjuntos de grafos (en inglés, *graph databases* o *graph datasets*).

La minería de subgrafos conexos frecuentes (SCF) sobre colecciones de grafos etiquetados ha sido objeto de estudios recientes dada su amplia variedad de aplicaciones [30]. En problemas de verificación toxicológica predictiva [36], los SCF están relacionados con sustancias tóxicas. Borgelt y Berthold [7] descubrieron estructuras químicas activas en conjuntos de datos sobre el Virus de Inmunodeficiencia Humana (VIH), contrastando la frecuencia de aparición de los SCF entre diferentes clases. En el estudio de redes biológicas también se han buscado subgrafos frecuentes [24]. Los subgrafos conexos repetidos dentro de una colección de grafos proporcionan un mejor entendimiento de las relaciones existentes entre los datos y han sido utilizados como base para crear algoritmos de clasificación [11,53], agrupamiento [38,18] e indexación [49]. Inclusive, existen aplicaciones donde es necesario encontrar sólo aquellos SCF que son suficientemente densos en cuanto al número de aristas [54]. Además de los grafos etiquetados, otros tipos de grafos también han sido utilizados para hacer minería de SCF, por ejemplo: los grafos geométricos [25], los multi-estructurados [46] y otros.

Este reporte técnico se enfoca al estudio de la minería de SCF sobre colecciones de grafos etiquetados. La mayor parte de los algoritmos encontrados en el estado del arte se han diseñado para procesar conjuntos de datos químicos (conjuntos de moléculas) y por tanto trabajan sobre colecciones de grafos simples y no-dirigidos. Este tipo de grafos será el objeto de este trabajo.

Un grafo etiquetado es una forma de representación genérica que es usada en muchos dominios de aplicación. Por ejemplo, la estructura química de una sustancia puede ser modelada mediante un grafo etiquetado, en el cual cada vértice representa un átomo y cada arista corresponde a un enlace [32].

El primer algoritmo de minería de subgrafos frecuentes fue AGM (por sus siglas en inglés, *Apriori based graph mining*) [22], el cual se diseñó para buscar los subgrafos frecuentes, sin tener en cuenta la conexidad. Esto fue una limitante para las aplicaciones donde los SCF son de interés. En respuesta a esta problemática surge el algoritmo FSG (por sus siglas en inglés, *frequent subgraph discovery*) [26] que encuentra todos los SCF dentro de una colección de grafos etiquetados. Este algoritmo pudo acelerar el conteo de frecuencia de los subgrafos candidatos utilizando recursos de memoria para mantener listas de identificadores de transacciones.

Desde entonces se han ido creando nuevos algoritmos cada vez más eficientes para encontrar todos los SCF [14]. En estados del arte escritos sobre esta temática [42,14], los algoritmos se han clasificado, de acuerdo a la estrategia de generación de candidatos, en dos grupos: los basados en Apriori y los basados en crecimiento de patrones.

Los algoritmos basados en Apriori, comparten características similares con el algoritmo Apriori [2] de conjuntos frecuentes. En el contexto de la minería de SCF, los algoritmos basados en Apriori tienen dos grandes dificultades relacionadas con la generación de candidatos: (1) alta complejidad computacional y (2) se generan muchos candidatos (para cada candidato es necesario calcular su frecuencia de aparición).

Para evitar las dificultades mencionadas en los modelos basados en Apriori, se han desarrollado otros algoritmos basados en la metodología de crecimiento de patrones [15]. Estos algoritmos obtienen los subgrafos candidatos creciendo directamente a un solo subgrafo [48,7,31].

En algunas aplicaciones no es práctico considerar todos los SCF debido a que pueden ser demasiados. Existen soluciones para abordar este problema, éstas permiten obtener sólo una parte de los SCF. Una de ellas está relacionada con los patrones frecuentes cerrados [51]. Se dice que un patrón frecuente es cerrado si no existe un patrón que lo contiene con su misma frecuencia. En [47,9] se proponen algoritmos para la minería de subgrafos conexos frecuentes cerrados (SCFC). Otro tipo de patrones frecuentes son los patrones frecuentes maximales los cuales no están contenidos en ningún otro patrón frecuente. La minería de subgrafos conexos frecuentes maximales (SCFM) fue estudiada en [20,37]. Por otra parte, los conceptos de minería de patrones frecuentes con restricciones (en inglés *constraint-based frequent pattern mining*) también han sido adaptados a la minería de grafos [55].

Independientemente de cuanto se ha avanzado en esta temática, los algoritmos existentes sólo pueden ejecutarse en conjuntos de datos estáticos y de tamaño limitado. Es por eso que sigue siendo necesario desarrollar algoritmos capaces de procesar mayores volúmenes de datos, es decir mayor cantidad de grafos y de mayor tamaño.

El este reporte técnico, se presenta un nuevo algoritmo para la minería de SCF llamado gRed (por sus siglas en inglés, *graph candidate reduction miner*) [13]. Pero antes se introducen y demuestran nuevas propiedades de la codificación DFS (una forma de representar grafos introducida por Yan y Han) que nos permiten optimizar el proceso de minería reduciendo el espacio de candidatos.

Este documento está organizado como sigue: en el capítulo 2 se introducen los conceptos básicos para entender el resto del documento. En el capítulo 3 se describen las características esenciales de los trabajos más importantes reportados en la literatura. El capítulo 4 incluye las nuevas propiedades de la codificación DFS que posteriormente se usaron para diseñar el algoritmo gRed. Una descripción de gRed podemos encontrarla en el capítulo 5. Los resultados experimentales se incluyen y se discuten en el capítulo 6. Finalmente, el capítulo 7 expone las conclusiones de este reporte.

2. Conceptos preliminares

En este trabajo nos enfocaremos en los grafos etiquetados simples y no-dirigidos. En lo adelante cuando se hable de grafo se asumen todas estas características y en otro caso se especificará explícitamente. La definición formal de este tipo de grafos es la siguiente.

Definición 2.1 (grafo etiquetado). *Un grafo etiquetado simple y no-dirigido es una tétrada* $\langle V, E, L, l \rangle$*, donde*

- V *es un conjunto cuyos elementos son llamados vértices,*
- $E \subset \{e \,|e \subset V, |e| = 2\}$ *es un conjunto cuyos elementos son llamados aristas (cada arista es un conjunto de vértice con cardinalidad dos),*
- L *es el conjunto de etiquetas y*
- $l : V \cup E \to L$ *es una función que asigna etiquetas a los vértices y aristas del grafo.*

Esta definición puede generalizarse a grafos parcialmente etiquetados si se incluye la etiqueta vacía ϵ.

Definición 2.2 (subgrafo). *Se dice que un grafo* G_1 *es subgrafo de otro grafo* G_2 *si* $V(G_1) \subseteq V(G_2)$*,* $E(G_1) \subseteq E(G_2)$*. En este caso se utiliza la notación* $G_1 \subseteq G_2$*.*

En los algoritmos para el minado de SCF, el conteo de la frecuencia de aparición de un subgrafo candidato se realiza mediante pruebas de sub-isomorfismo.

Definición 2.3 (isomorfismo y sub-isomorfismo). *Se dice que* f *es un isomorfismo entre dos grafos etiquetados* G_1 *y* G_2*, si* $f : V(G_1) \to V(G_2)$ *es una función biyectiva y*

- $\forall v \in V(G_1), l_{G_1}(v) = l_{G_2}(f(v)),$
- $\forall (u, v) \in E(G_1), (f(u), f(v)) \in E(G_2)$ *y* $l_{G_1}(u, v) = l_{G_2}(f(u), f(v)).$

Un sub-isomorfismo de G_1 *a* G_2 *es un isomorfismo de* G_1 *a un subgrafo de* G_2*.*

El problema de determinar el isomorfismo entre dos grafos (etiquetados o no) es NP-duro (en inglés *NP-hard*) [35]. Una manera de abordar las pruebas de isomorfismo es utilizando las formas canónicas para representar los grafos. Las formas canónicas más usadas en minería de grafos serán presentadas en el epígrafe 3.1. Los algoritmos Nauty [27] y Ullmann [39], han sido algoritmos muy usados para realizar pruebas de isomorfismo y sub-isomorfismo respectivamente. Tanto estos algoritmos como otros que han sido diseñado posteriormente ha logrado reducir los tiempos de respuesta para algunas familias de grafos; no obstante, siempre quedan casos donde la complejidad temporal es exponencial [12].

Definición 2.4 (camino y ciclo). *Se dice que* $P = (v_1, v_2, \ldots, v_k)$ *es un camino en el grafo* G *si todo vértice de* P *está en* $V(G)$ *y para cada par de vértices* v_i *y* v_{i+1} *(consecutivos en* P*) se cumple que* $\{v_i, v_{i+1}\} \in E(G)$*. En tal caso se dice que* v_1 *y* v_k *están conectados por* P*. Si* $v_1 = v_k$ *se dice que* P *es un ciclo.*

Definición 2.5 (grafo conexo). *Un grafo* G *es conexo si todo par de vértices en* $V(G)$ *están conectados por algún camino.*

Definición 2.6 (árbol libre). *Un árbol libre es un grafo* G *conexo y sin ciclos.*

Cuando en un árbol libre se selecciona un vértice como raíz entonces se tiene un *árbol enraizado*. En adelante se utilizará el término árbol para referirse a los árboles enraizados.

Definición 2.7 (relación padre-hijo en árboles enraizados). *Sea* T *un árbol enraizado con* $v_0 \in V(T)$ *seleccionado como raíz y* $v, u \in V(T)$ *dos vértices cualesquiera de* T*. Se dice que* v *es padre de* u *si* $\{v, u\} \in E(T)$ *y el camino que une a* v *con* v_0 *está completamente contenido en el camino que une a* u *con* v_0*. En ese caso se dice que* u *es un hijo de* v*.*

Definición 2.8 (frecuencia o soporte). *Sea* $D = \{G_0, G_1, \ldots, G_{m-1}\}$ *un conjunto de grafos, un umbral de frecuencia* δ *y un grafo* g*. La frecuencia de aparición (algunos autores utilizan el término soporte) de* g *en* D *que denotaremos* $\sigma(g, D)$*, se define como la cantidad de grafos* $G_i \in D$ *tales que existe un sub-isomorfismo de* g *a* G_i*. Se dice que un grafo* g *es frecuente en* D*, si* $\sigma(g, D) \geq \delta$*.*

La minería de SCF sobre una colección D puede ser definida como el problema de encontrar todos los subgrafos conexos g en D tales que $\sigma(g, D) \geq \delta$.

Definición 2.9 (grafo frecuente cerrado). *Sea* g *un grafo frecuente en* D*. Se dice que* g *es frecuente cerrado en* D*, si no existe* g' *tal que* $g \subset g'$ *y* $\sigma(g, D) = \sigma(g', D)$*.*

El conjunto de los SCF cerrados (SCFC) en una colección D, es por lo general más pequeño que el conjunto de todos los SCF. Es por eso que en algunas aplicaciones resulta importante enfocar la búsqueda hacia los SCFC en lugar de encontrar todos los SCF.

Definición 2.10 (grafo frecuente maximal). *Sea* g *un grafo frecuente en* D*. Se dice que* g *es frecuente maximal en* D*, si no existe* g' *tal que* $g \subset g'$ *y* g' *es frecuente.*

Como se puede apreciar los SCF maximales (SFCM) también son cerrados. Enfocar la minería a encontrar los SFCM reduciría aún más el número de patrones encontrados respecto a los tipos de minería anteriormente mencionados.

3. Estado del arte

En [14], los algoritmos para la minería de SCF se clasificaron, de acuerdo a la estrategia de generación de candidatos, en dos grupos: los basados en Apriori y los basados en crecimiento de patrones. No obstante, existen otros aspectos que pueden incidir en la calidad de un algoritmo para la minería de SCF; por ejemplo, las formas canónicas utilizadas para representar los candidatos y las estructuras utilizadas para el conteo de soporte. En el epígrafe 3.1 se detallan algunas de las formas canónicas más utilizadas en este tipo de minería. Aspectos esenciales sobre las estructuras usadas para el conteo de soporte son presentadas en el epígrafe 3.3.

3.1. Formas canónicas para grafos etiquetados

La idea central es construir una secuencia de símbolos que identifique de manera única a cada grafo. Estos símbolos describen la topología del grafo y el valor de etiquetas.

Para construir estas secuencias se requiere un ordenamiento de los vértices del grafo. Este ordenamiento puede hacerse ya sea mediante la búsqueda de árboles de cobertura (en inglés *spanning trees*) o realizando alguna permutación aleatoria de dichos vértices. Existirán entonces, muchas formas de realizar este ordenamiento y esto hace más complejo el hecho de definir tales secuencias. Mediante formas canónicas es posible construir una descripción única de cada grafo. Esto se realiza seleccionando entre todas las posibles secuencias que representan un grafo, aquella que se distinga por cumplir ciertos cánones o características especiales, según criterios de orden lexicográfico entre los símbolos. En los siguientes epígrafes se presentan ejemplos concretos de cómo lograr estas secuencias.

La propiedad del prefijo plantea que: *cada prefijo válido de una forma canónica es la forma canónica del subgrafo que representa.* Esta propiedad garantiza la completitud de los resultados ya que cada grafo podrá ser generado al menos una vez. Si una forma canónica cumple la propiedad del prefijo, entonces ésta podrá ser utilizada en minería de subgrafos frecuentes por crecimiento de patrones (ver epígrafe 3.2). En los siguiente epígrafes se muestran dos formas canónicas que cumplen esta propiedad: la codificación DFS y la codificación BFS. También se incluye un ejemplo de forma canónica que no cumple con esta propiedad en la manera estricta en que está planteada; sin embargo, pudo ser utilizada en minería de grafos utilizando otros mecanismos para garantizar la completitud.

Las formas canónicas son utilizadas fundamentalmente por los algoritmos de minería de SCF que están basados en el crecimiento de patrones. En la minería de conjuntos frecuentes, por ejemplo, es trivial asegurar que el mismo conjunto es verificado sólo una vez. Sin embargo, en la minería de grafos éste es un problema medular. La razón es que el mismo subgrafo puede ser obtenido por crecimiento en diversas vías, adicionando los mismos vértices y aristas en diferentes órdenes.

La detección de duplicados puede hacerse manteniendo una estructura de datos con los grafos ya analizados y mediante múltiples comprobaciones sobre dicha estructura. Esta variante garantiza validez en los resultados pero puede ser devastador para los tiempos de ejecución y consumo de memoria. Es por eso que se ha hecho tradicional el uso de las formas canónicas en el recorrido del espacio de búsqueda.

Las formas canónicas vistas como secuencias pueden ser ordenadas lexicográficamente de acuerdo con el valor numérico o etiqueta de cada símbolo. El espacio de búsqueda en los algoritmos de minería de SCF es recorrido de acuerdo con este orden lexicográfico. Cuando aparece una secuencia en forma no-canónica durante el recorrido del espacio de búsqueda, es porque la forma canónica del mismo grafo candidato ya fue analizada en un paso anterior. Sería ideal que la heurística que se utilice para recorrer el espacio de búsqueda eliminara toda posibilidad de generar formas no-canónicas, pero no se ha encontrado ninguna variante con estas bondades y por tanto ha sido necesario comprobar en cada candidato si está en forma canónica o no [14].

Comprobar que un grafo está representado por su forma canónica puede ser una operación muy costosa en tiempo, ya que se requiere enumerar exhaustivamente todas las posibles secuencias con el fin de seleccionar de entre ellas aquella que es la secuencia canónica. En algunos casos es posible diseñar heurísticas que permiten reducir el número de comprobaciones [6].

Codificación CAM

La estructura de un grafo etiquetado puede ser expresada usando matrices de adyacencia. A partir de esta representación es posible definir una secuencia única para representar dicho grafo.

Definición 3.1 (matriz de adyacencia). *Dado un grafo etiquetado* $G = \langle V, E, L, l \rangle$ *tal que* $|V| = n$ *y una permutación de los vértices en* $V = \{v_0, v_1, \ldots, v_{n-1}\}$*, la matriz de adyacencia de* G *respecto a dicha permutación es* $X = (x_{i,j})_{i,j=0}^{n-1}$ *donde:*

$$x_{i,j} = \begin{cases} l(v_i) & si \quad i = j \\ l(e) & si \quad e = \{v_i, v_j\} \in E \\ \epsilon & si \quad \{v_i, v_j\} \notin E \end{cases} . \tag{1}$$

El código de G usando X ha sido definido de varias maneras. En adelante, explicaremos dos maneras de construir este código. La primera variante ubica primero las etiquetas de los vértices y luego adiciona las filas de la submatriz estrictamente triangular inferior (la matriz triangular inferior excluyendo la diagonal) (2). Esta forma de codificación fue usada por Inokuchi *et al.* [22,23] y por Hong *et al.* [17] para la minería de subgrafos frecuentes.

$$codeA(X) = x_{0,0}x_{1,1}\ldots x_{n-1,n-1}x_{1,0}x_{2,0}x_{2,1}x_{3,0}x_{3,1}x_{3,2}\ldots x_{n-1,n-2}. \tag{2}$$

Otra variante fue utilizada por Huan *et al.* en el algoritmo FFSM el cual es uno de los algoritmos más prominentes dentro de la minería de SCF [19]. Esta segunda variante concatena las filas de la submatriz triangular inferior (3).

$$codeB(X) = x_{0,0}x_{1,0}x_{1,1}x_{2,0}x_{2,1}x_{2,2}\ldots x_{n-1,n-1}. \tag{3}$$

Un mismo grafo puede tener varias matrices de adyacencia ya que existe una dependencia con el orden de los vértices que se utilizó durante la construcción de la matriz. Caracterizar un grafo etiquetado requiere de una representación que sea única en dicho grafo y en todos aquellos que son isomorfos con éste. Es por eso que se define la matriz de adyacencia canónica (CAM por sus siglas en inglés, *canonical adjacency matriz*) a partir del código de la matriz.

Dos códigos que representen al mismo grafo siempre tendrán el mismo tamaño en cuanto a la cantidad de símbolos que componen la secuencia. Estos códigos podrán ser comparados lexicográficamente teniendo en cuenta el orden de las etiquetas.

Algunos autores han definido la CAM de un grafo G como la matriz de adyacencia que da lugar al código lexicográficamente mayor entre todos los posibles códigos de las matrices de adyacencia asociadas con G [23,17,19]. Aunque, en otro trabajo se utilizó la matriz que logra el código lexicográficamente menor entre tales códigos [22]. Por la forma en que fueron definidas estas secuencias, si dos códigos que representan un mismo grafo son iguales entonces fueron construidos a partir de la misma matriz de adyacencia. Por tanto, cuando ocurre un empate al encontrar el código lexicográficamente mayor (o menor) es porque se está en presencia de la misma matriz CAM.

Codificación DFS

Para representar un grafo etiquetado mediante una única secuencia de aristas, se diseñó una forma canónica llamada codificación DFS (por sus siglas en inglés, *depth first search*) [48]. Esta secuencia es conocida como código DFS mínimo [48].

Árboles DFS. Un árbol DFS T puede ser construido cuando se realiza un recorrido en profundidad en un grafo $G = \langle V; E; L; l\rangle$. El grafo G puede tener varios árboles DFS porque casi siempre existe más de un recorrido en profundidad (ver Fig. 1). Cada árbol define un orden entre todos los vértices del grafo; por tanto, podemos enumerar cada vértice de acuerdo a este orden DFS. La raíz de T será enumerada con índice 0 ya que es el primer vértice en el orden DFS. El último vértice en dicho orden corresponde al *vértice más a la derecha* de T. Por ejemplo en la Fig. 1, los vértices enumerados con 0 y 4 constituyen la raíz y el vértice más a la derecha en cada uno de los árboles $T0$, $T1$, $T2$, ... respectivamente. El camino directo desde la raíz hasta el vértice más a la derecha es llamado la *rama más a la derecha*.

El conjunto de aristas hacia adelante $F(T)$ de un árbol DFS T en un grafo G contiene todas las aristas de G que están en T, y el conjunto de aristas hacia atrás $B(T)$ contiene las aristas de G que no están en T. Por ejemplo en la Fig. 1, las aristas hacia adelante están representadas mediante líneas sólidas y en las aristas hacia atrás con líneas punteadas.

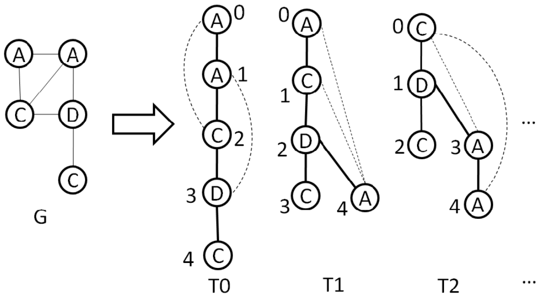


Figura 1. Ejemplos de árboles DFS para el grafo G.

Códigos DFS. Un código DFS es una secuencia de aristas construida a partir de un árbol DFS. Esta secuencia es obtenida ordenando las aristas en $F(T)$ de acuerdo con el orden DFS. Luego, las aristas hacia atrás desde un vértice v son insertadas justo delante de las aristas hacia adelante desde v; si v tiene varias aristas hacia atrás, éstas son incluidas de acuerdo con el orden DFS de sus vértices terminales. La Tabla 1 muestra ejemplos de códigos DFS para los árboles DFS de la Fig. 1. Estas consideraciones para construir la secuencia definen un orden lexicográfico ($\prec_e$) entre dos aristas cualesquiera.

Tabla 1. Ejemplos de códigos DFS para los árboles DFS de la Fig. 1.

Árbol DFS	Código DFS
T0	**(0,1,A,-,A)(1,2,A,-,C)**(2,0,C,-,A)**(2,3,C,-,D)**(3,1,D,-,A)**(3,4,D,-,C)**
T1	**(0,1,A,-,C)(1,2,C,-,D)(2,3,D,-,C)(2,4,D,-,A)**(4,0,A,-,A)(4,1,A,-,C)
T2	**(0,1,C,-,D)(1,2,D,-,C)(1,3,D,-,A)**(3,0,A,-,C)**(3,4,A,-,A)**(4,0,A,-,C)

Cada arista es representada por una 5-tupla, $(i, j, l_i, l_{(i,j)}, l_j)$ donde i y j son los índices de los vértices (origen y destino respectivamente), l_i y l_j son las etiquetas de tales vértices, y $l_{(i,j)}$ es la etiqueta de la arista. Si $i < j$ entonces estamos en presencia de una arista hacia adelante; de otro modo es una arista hacia atrás.

Orden lexicográfico DFS. Si $e_1 = (i_1, j_1, \ldots)$ y $e_2 = (i_2, j_2, \ldots))$, la relación de orden $e_1 \prec_e e_2$ se cumple si y sólo si se cumple una de las siguientes condiciones:

- $e_1, e_2 \in F(T) \wedge (j_1 < j_2 \vee (j_1 = j_2 \wedge i_1 > i_2))$;
- $e_1, e_2 \in B(T) \wedge (i_1 < i_2 \vee (i_1 = i_2 \wedge j_1 < j_2))$;
- $e_1 \in B(T) \wedge e_2 \in F(T) \wedge i_1 < j_2$;
- $e_1 \in F(T) \wedge e_2 \in B(T) \wedge j_1 \leq i_2$;
- $i_1 = i_2 \wedge j_1 = j_2 \wedge e_1 \prec_l e_2$.

El orden lexicográfico considerando las etiquetas ($\prec_l$) compara las aristas e_1 y e_2 respecto a las últimas tres componentes de cada 5-tupla. Para determinar el orden entre dos aristas, la etiqueta l_i tiene prioridad, luego $l_{(i,j)}$, y finalmente l_j.

El orden $\prec_e$ puede ser extendido a un orden lexicográfico ($\prec_s$) entre códigos DFS. Sean $s_1 = (a_1, a_2, \ldots, a_m)$ y $s_2 = (b_1, b_2, \ldots, b_n)$ dos códigos DFS. Decimos que $s_1 \prec_s s_2$ si se cumple una de las siguientes condiciones:

$$\exists t, \forall k < t, a_k = b_k, \text{ y } a_t \prec_e b_t; \tag{4}$$

$$m < n \text{ y } \forall k \leq m, a_k = b_k. \tag{5}$$

Código DFS mínimo. El código DFS mínimo de G es definido como el menor código DFS, respecto al orden $\prec_s$, entre todos los posibles códigos de G; a este código lo denotaremos por minDFS(G).

Por ejemplo entre los códigos DFS de la Tabla 1 se tiene que: Code(T0) $\prec_s$ Code(T1) $\prec_s$ Code(T2). Ahora, para encontrar minDFS(G) es necesario generar todos los posibles códigos DFS para G y seleccionar entre ellos el menor posible. Es por eso que comprobar que un código DFS es mínimo tiene un gran costo computacional. Por ejemplo, para un grafo denso esta comprobación tiene complejidad computacional $O(n!)$.

Este tipo de forma canónica fue introducida por Yan y Han [48] y ha sido utilizada en varios trabajos de minería de SCF [41,43,16,13].

Codificación BFS

Otra codificación basada en recorridos en amplitud, usada para caracterizar un grafo etiquetado mediante una secuencia de símbolos, es llamada código BFS (por sus siglas en inglés, *breadth first search*) mínimo.

Árboles BFS. Un árbol BFS T es construido cuando se realiza un recorrido en amplitud en un grafo $G = \langle V, E, L, l \rangle$. El grafo G puede tener varios árboles BFS porque casi siempre existe más de un recorrido en amplitud. Este árbol T define un orden para los vértices de G, dado por el momento en que fueron visitados durante el recorrido. Este orden es llamado orden BFS. A cada vértice se le puede asignar un índice entre 0 y $|V|-1$ de acuerdo con dicho orden. La raíz de T tendrá asignado el índice 0.

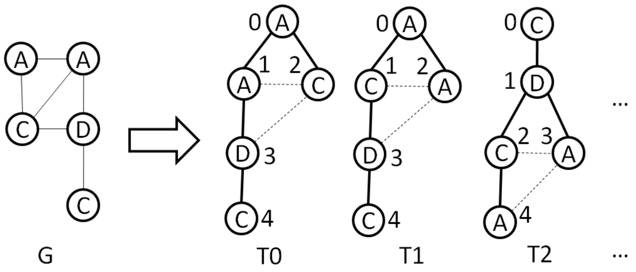


Figura 2. Ejemplos de árboles BFS para el grafo G.

De igual manera que se hizo en la codificación DFS, el conjunto de aristas hacia adelante $F(T)$ contiene todas las aristas de G que están en T, y el conjunto de aristas hacia atrás $B(T)$ contiene las aristas de G que no están en T.

Códigos BFS. El código BFS asociado con el árbol T es una secuencia de símbolos (índices o etiquetas). El primer elemento de esta secuencia es la etiqueta del nodo raíz en T. Este código se va construyendo durante el recorrido en amplitud que originó a T. Las aristas hacia atrás desde cada vértice v son visitadas antes que las aristas hacia adelante que parten de v. Toda vez que una arista es visitada, un fragmento de longitud cuatro es añadido a la secuencia, este fragmento tiene la forma i_0evi_d donde e es la etiqueta de la arista, v la etiqueta del vértice destino, i_0 es el índice del nodo origen e i_d es el índice del nodo destino ($i_0 < i_d$). La Tabla 2 muestra ejemplos de códigos BFS para los árboles BFS de la Fig. 2. De manera resumida, un código BFS puede ser descrito por la siguiente expresión regular:

$$v[i_0evi_d]^m, \tag{6}$$

donde $m = |E(G)|$.

Tabla 2. Ejemplos de códigos BFS para los árboles BFS de la Fig. 2.

Árbol BFS	Código BFS
T0	A(0,-,A,1)(0,-,C,2)(1,-,C,2)(1,-,D,3)(2,-,D,3)(3,-,D,4)
T1	A(0,-,C,1)(0,-,A,2)(1,-,A,2)(1,-,D,3)(2,-,D,3)(3,-,D,4)
T2	C(0,-,D,1)(1,-,C,2)(1,-,A,3)(2,-,A,3)(2,-,A,4)(3,-,A,4)

Código BFS mínimo. Todos los posibles códigos BFS del grafo G pueden ser comparados entre sí lexicográficamente. El menor entre estos códigos es llamado el código BFS mínimo y lo denotaremos $\mathrm{minBFS}(G)$.

Este tipo de forma canónica fue utilizada por el algoritmo MoFa [7] y fue presentada formalmente por Borgelt [6].

3.2. Minería de subgrafos conexos frecuentes

Algoritmos basados en Apriori

Los algoritmos basados en Apriori, comparten características similares con el algoritmo Apriori para minar conjuntos frecuentes [2]. Todos estos algoritmos requieren de una operación `FUSIÓN`(en inglés *join*) para mezclar dos subgrafos y obtener subgrafos candidatos de tamaño mayor. Estos algoritmos se diferencian entre sí casi siempre por los elementos que son usados para medir el tamaño (cantidad de elementos) de cada subgrafo: vértices [22,23], aristas [26], y caminos con aristas disjuntas [40]. En el contexto de la minería de SCF, los algoritmos basados en Apriori tienen dos grandes dificultades relacionadas con la operación `FUSIÓN`: (1) es computacionalmente compleja en tiempo y (2) genera muchos candidatos.

El algoritmo 1 muestra una descripción general de los algoritmos basados en Apriori. En cada llamado recursivo del algoritmo, el tamaño de los nuevos SCF descubiertos es incrementado en uno respecto al llamado anterior. El proceso de generación de candidatos es mostrado en la línea 5 y se realiza mediante la operación `FUSIÓN`. Para invocar este algoritmo por primera vez es necesario calcular S_1 (el conjunto de SCF de tamaño 1) en un procedimiento aparte (dependiendo de las especificaciones de cada algoritmo).

Como ya se mencionó, el primer algoritmo diseñado para la minería de SCF fue FSG [26], aunque hubo un trabajo previo a este que no tenía en cuenta la conexidad [22]. Ambos modelos son basados en Apriori. En FSG el tamaño de los grafos está determinado por el número de aristas. Dos grafos de tamaño k se pueden fusionar cuando comparten un mismo subgrafo de tamaño $k-1$ llamado núcleo (en inglés *core*) y se diferencian en una sola arista. Esta fusión es muy compleja para los candidatos de mayor tamaño, aunque genera menos candidatos que la variante del algoritmo AGM.

Tanto AGM como sus posteriores adaptaciones para la minería de SCF (AcGM [23], Topology [17], Generalized AcGM [21]) miden el tamaño de los grafos contando el número de vértices. Los grafos candidatos son representados mediante su CAM y esta representación es usada para la generación de candidatos. Para que dos grafos de tamaño k se puedan fusionar es necesario que ambas CAM tengan igual submatriz principal de tamaño $k-1$, en otro caso el resultado de la fusión es el conjunto vacío. Esto significa

```
Algoritmo 1: SCF-Apriori($D$, $\delta$, $S_k$, $S$)
Entrada: $D$ - conjunto de grafos, $\delta$ - umbral de frecuencia mínima,
$S_k$ - Conjunto de subgrafos conexos frecuentes de tamaño $k$.
Salida: $S$ - Conjunto de todos los subgrafos conexos frecuentes

$S \leftarrow S \cup S_k$;
$S_{k+1} \leftarrow \emptyset$;
foreach subgrafo frecuente $g_i \in S_k$ do
    foreach subgrafo frecuente $g_j \in S_k$ do
        foreach subgrafo de tamaño $k+1$: $g \in$ FUSIÓN($g_i, g_j$) do
            if $\sigma(g, D) \geq \delta$ then $S_{k+1} \leftarrow S_{k+1} \cup \{g\}$;
        end
    end
end
if $S_{k+1} \neq \emptyset$ then
    SCF-Apriori($D$, $\delta$, $S_{k+1}$, $S$);
end
```

Figura 3. Esquema general del paradigma Apriori.

que ambos grafos deben compartir un subgrafo de tamaño $k - 1$ y diferenciarse en un solo vértice. Los grafos de tamaño $k + 1$ resultantes de la fusión contienen el subgrafo común y los dos vértices que diferencian los grafos. La arista que pudiera unir a estos dos vértices es la que diferencia a los nuevos grafos generados y es el motivo del alto número de candidatos que hay que procesar.

Otra variante muy interesante para la generación de candidatos basándose en caminos disjuntos, es el algoritmo PATH-BASED [40]. En este algoritmo el tamaño de un grafo está dado por la cantidad de caminos disjuntos que tiene. Un grafo candidato de tamaño $k + 1$ es generado a partir dos grafos de tamaño k que sólo se diferencian en uno de los caminos disjuntos.

Algoritmos basados en crecimiento de patrones

Un grafo g puede ser extendido adicionándole una nueva arista e. Hay dos formas de realizar esta extensión en dependencia de que la arista esté compuesta por: (1) dos vértices de g, o (2) un vértice en g y un vértice nuevo. El grafo resultante de esta extensión se denota como $g \diamond e$.

El esquema de los algoritmos basados en crecimiento de patrones se describe de forma general en el algoritmo 2.

La generación de candidatos en este paradigma está basada en la operación de extensión. Se han diseñado varias estrategias utilizando diferentes formas canónicas de grafos

```
Algoritmo 2: SCF-Crecimiento($g$, $D$, $\delta$, $S$)
Entrada: $g$ - un grafo frecuente, $D$ - conjunto de grafos, $\delta$ - umbral de frecuencia mínima.
Salida: $S$ - Conjunto de todos los subgrafos conexos frecuentes

if $g \in S$ then return;
$S \leftarrow S \cup \{g\}$;
$E_g \leftarrow$ Conjunto de aristas $e$ tales que exista en $D$ la extensión $g \diamond e$;
foreach $e \in E_g$ do
    if $\sigma(g \diamond e, D) \geq \delta$ then SCF-Crecimiento($g \diamond e$, $D$, $\delta$, $S$);
end
```

Figura 4. Esquema general del paradigma crecimiento de patrones.

(ver epígrafe 3.1), pero en todas es necesario tener en cuenta la posibilidad de la existencia de duplicados. En muchos casos la eficiencia de un algoritmo está fuertemente ligada a disminuir el número de duplicados.

Algoritmo gSpan. El primer algoritmo de minería de SCF basado en el crecimiento de patrones fue gSpan (por sus siglas en inglés, *graph-based substructure pattern mining*) [48]. Este trabajo introduce la codificación DFS como una prometedora forma canónica para representar grafos etiquetados en la minería de grafos [33] y se propone una heurística no tan exhaustiva para la detección de duplicados. La definición 3.2 nos permite entender el espacio de búsqueda en gSpan.

Definición 3.2 (extensión más a la derecha). *Sea s un código DFS mínimo y e una arista que no pertenece a s. Se dice que e es una extensión más a la derecha de s si e conecta al vértice más a la derecha con otro vértice en la rama más a la derecha de s (extensión hacia atrás); o e introduce un nuevo vértice conectado a un vértice de la rama más a la derecha de s (extensión hacia adelante). En tales casos, el código $s' = s \diamond e$ denotará al código obtenido luego de extender a s con e; se dice que s' es un hijo de s o que s es el padre de s'.*

El espacio de búsqueda en gSpan es definido como un árbol que en sus nodos contiene códigos DFS y el concepto de padre-hijo viene dado por la definición 3.2. Como la codificación DFS cumple la propiedad del prefijo (ver epígrafe 3.1) entonces cada código DFS mínimo (de longitud m, $m > 1$) es un hijo de algún otro código DFS mínimo (de longitud $m - 1$); es decir, recorriendo el espacio de búsqueda para cada arista frecuente (código de longitud 1) en D se logra completitud en la búsqueda. El árbol del espacio de búsqueda tiene que ser podado a partir de cualquier nodo que contenga un código DFS no-mínimo. Cada código no-mínimo representa un candidato duplicado ya que el código DFS mínimo del grafo asociado debe haberse generado anteriormente, pues los códigos DFS se generan ascendentemente en el orden lexicográfico.

Todos los algoritmos basados en crecimiento de patrones generan duplicados durante el recorrido del espacio de búsqueda. En gSpan, los candidatos duplicados son aquellos códigos DFS que son no-mínimos. Como se ha dicho anteriormente, para comprobar la minimalidad o no-minimalidad de un código DFS se requieren pruebas exhaustivas con un alto costo computacional. Para comprobar si un código DFS s es mínimo o no, gSpan propone su propia heurística la cual es computacionalmente más eficiente que la prueba exhaustiva. Se generan aquellos prefijos de códigos DFS que son menores o iguales que el correspondiente prefijo en s. Si algún prefijo de estos es menor estricto que s, entonces s es un código no-mínimo y la prueba concluye. Como se puede ver esta prueba de minimalidad sigue siendo exhaustiva y computacionalmente costosa; por ejemplo, en los casos cuando este prefijo es encontrado tardíamente o cuando el código s es realmente mínimo.

El esquema del algoritmo gSpan ha sido utilizado como punto de partida para diseñar algoritmos más eficientes o mejor adaptados a condiciones específicas de aplicación. El algoritmo CloseGraph fue presentado como una adaptación de gSpan para la minería de SCFC [47]. En otro trabajo, una estructura llamada ADI (por sus siglas en inglés, *adjacency index*) es utilizada para incrementar la capacidad de procesamiento de gSpan [41]. Con el uso de esta estructura fue posible procesar conjuntos de datos de gran tamaño los cuales no podían ser cargados en memoria por los algoritmos conocidos hasta entonces. El algoritmo Edgar (por sus siglas en inglés, embeding-based graph miner) introduce mejoras en gSpan mediante técnicas de optimización de códigos y una estrategia de poda en el espacio de búsqueda basada en estructuras de correspondencias [43]. Estos dos últimos trabajos no modificaron el esquema general de gSpan para la generación del espacio de búsqueda.

Recientemente, se han descubierto nuevas propiedades de la codificación DFS y esto ha permitido modificar en mayor medida el esquema general del algoritmo gSpan para obtener mejores resultados. En el algoritmo FSP (por sus siglas en inglés, *frequent substructure pattern mining*) se introdujo el concepto de código DFS balanceado y se estudió la estructura del espacio de búsqueda basándose en propiedades de las extensiones más a la derecha [16].

Otros ejemplos de algoritmos desarrollados bajo el paradigma de crecimiento de patrones son MoFa [7], FFSM [19] y Gaston [31]. Estos, junto a gSpan, pueden ser considerados como la base de todos los trabajos desarrollados en la minería de SCF bajo este paradigma.

Estos tres algoritmos (MoFa, FFSM y Gaston), mantienen en memoria estructuras de correspondencias (EC), que permiten realizar un conteo eficiente de la frecuencia de los candidatos. La implementación original de gSpan utiliza listas de identificadores (LI) y por eso necesita de algoritmos para las pruebas de sub-isomorfismo que permiten localizar a un candidato dentro de los grafos de D incluidos en LI. Los algoritmos que usan EC no necesitan realizar las pruebas de sub-isomorfismo, pero ocupan más espacio en memoria que las LI (sobre todo en los candidatos de tamaño menor). En el epígrafe 3.3 abordaremos un poco más sobre las estructuras de datos y estrategias que se han utilizado para realizar el conteo de soporte.

Algoritmo MoFa. El algoritmo MoFa (por sus siglas en inglés, *mining molecular fragments*) fue inspirado en el algoritmo Eclat [52] para minar conjuntos frecuentes. Este algoritmo utiliza la codificación BFS para representar los candidatos. MoFa brinda muchas funcionalidades para su aplicación en conjuntos de datos moleculares, sin embargo ha mostrado demoras en los tiempos de respuesta en estudios comparativos realizados previamente [44]. Es por eso que se propuso un modelo híbrido entre MoFa y FSG que permite reducir el costo de mantener las EC para los candidatos de menor tamaño [28]. El esquema de MoFa también fue utilizado para la minería de SCFC introduciendo novedosas estrategias para podar el espacio de búsqueda [9].

Recientemente, se realizaron experimentos con MoFa para comparar dos formas de evitar los duplicados [8]. Como se ha visto anteriormente, comprobar que un candidato está representado en su forma canónica es una forma de garantizar que éste no es un duplicado. Otra forma de hacer esta comprobación es mantener una estructura de datos (tabla Hash) con todos los subgrafos que ya han sido procesados y consultarla toda vez que un nuevo candidato es enumerado. En dicho trabajo, Borgelt y Fiedler mostraron que el hecho de ajustar la función Hash en la segunda variante puede implicar ligeras mejoras en el tiempo para minar SCF con respecto al uso de formas canónicas.

Algoritmo FFSM. El algoritmo FFSM (por sus siglas en inglés, *fast frequent subgraph mining*) es un método que mezcla ideas de ambos paradigmas (Apriori y crecimiento de patrones). La codificación CAM es utilizada por este algoritmo para representar los grafos etiquetados. Este algoritmo restringe notablemente el número de posibles extensiones $g \diamond e$, de tal modo que no se garantiza la completitud de los resultados. Esto no es un problema porque el resto de los candidatos se genera mediante una operación `FUSIÓN`, análoga a la del paradigma Apriori. A partir de FFSM se diseñó el algoritmo SPIN (por sus siglas en inglés, *spanning tree based graph mining*) [20] el cual se enfoca a la minería de los SCFM.

Un algoritmo llamado FSMA, muy parecido a FFSM, fue publicado recientemente por Wu y Chen [45]. Este algoritmo propuso utilizar matrices de incidencias en lugar de las matrices de adyacencias usadas por FFSM.

Algoritmo Gaston. El algoritmo Gaston (por sus siglas en inglés, *graph/sequence/tree extraction*) es considerado por muchos autores el más eficiente entre los cuatro algoritmos de minería de SCF más renombrados (gSpan, MoFa, FFSM y Gaston). No obstante estudios comparativos posteriores han mostrado que esta supremacía no es absoluta [44,33,13]. En Gaston se utilizan ideas anteriormente desarrolladas para la búsqueda de subestructuras más simples (caminos y árboles) antes de buscar los subgrafos frecuentes con ciclos. Primero se detectan los subcaminos frecuentes, éstos se usan para formar subárboles (grafos acíclicos) y finalmente, se generan los subgrafos cíclicos.

3.3. Estructuras de datos para indexar la colección de grafos

En los algoritmos de minería de patrones frecuentes, el conteo de frecuencia es uno de los procesos fundamentales. Las pruebas de sub-isomorfismo en la minería de SCF, hacen que este proceso sea crítico. Para este tipo de minería es prácticamente imprescindible el uso de estructuras de datos que indexen la colección de grafos; de este modo, es posible optimizar algunos procesos dentro de la búsqueda como el conteo de soporte. Otra manera de clasificar a los algoritmos existentes para minar SCF es teniendo en cuenta las estructuras de datos que utiliza para indexar la colección de grafos. Han existido cuatro tendencias: (1) búsqueda exhaustiva, (2) usando lista de identificadores, (3) usando estructuras de correspondencias, e (4) híbridos entre (2) y (3).

(1) Búsqueda exhaustiva: consiste en que dado un candidato g, es necesario recorrer D para encontrar cuántos grafos G_i contienen al menos un sub-isomorfismo de g a D. Por el costo computacional asociado a las pruebas de sub-isomorfismo esta variante sólo es funcional para candidatos de tamaño muy pequeño. La ventaja que tiene este método es que no necesita memoria adicional para realizar el conteo durante el proceso de generación de candidatos. La búsqueda exhaustiva sólo ha sido utilizada en variantes simplificadas de los algoritmos existentes para bases de datos pequeñas, casi siempre utilizando el algoritmo de Ullmann [39] para las pruebas de sub-isomorfismo. En esta variante no se utilizan estructuras de datos para indexar la colección.

(2) Listas de identificadores: (LI) consiste en mantener para cada grafo generado g, una lista con los identificadores i de los grafos G_i que contienen al menos un sub-isomorfismo de g a G_i. Esta variante no elimina las pruebas de sub-isomorfismo pero reduce el número de éstas notablemente. El proceso de mantener las listas en estado consistente no requiere de gran poder de cómputo; la lista de un nuevo candidato puede obtenerse fácilmente a partir de las listas de los grafos que le dieron origen durante el proceso de generación. Se requiere memoria adicional para almacenar estas listas pero no tanta como las estructuras de correspondencias. Los algoritmos FSG, Topology y gSpan, usan este tipo de estructuras.

(3) Estructuras de correspondencias: (EC) consiste en mantener para cada grafo generado g, estructuras que indexen todas las ocurrencias de g en D. Nótese que g puede ocurrir más de una vez en algunos $G_i \in D$. Las EC requieren mucho más espacio en memoria que las LI, pero eliminan por completo las pruebas exhaustivas de sub-isomorfismo. Este tipo de estructuras se ha usado con éxito en los algoritmos AcGM, FFSM, MoFa, Gaston y en algunas modificaciones de gSpan como Edgar y ADI.

(4) Híbridos: consiste en utilizar LI durante el procesamiento de los candidatos de menor tamaño y a partir de cierto tamaño empezar a usar las EC. Las EC requieren mucha más memoria en los candidatos pequeños, ya que éstos pueden ocurrir en muchas partes dentro de los grafos de D. Es conocido que las pruebas de sub-isomorfismo son menos costosas en los candidatos de menor tamaño. Esto justifica el uso de la variante (2) para empezar la búsqueda y luego continuar con la variante (3). Para un mejor diseño de estas variantes, surge la siguiente pregunta: ¿Cómo seleccionar el tamaño de los candidatos entre las dos variantes?; que es equivalente a la pregunta: ¿A partir de qué momento las pruebas de sub-isomorfismo se hacen lo suficientemente costosas para que sea preferible mantener las EC en memoria?. En [28] se propuso usar una variante de este tipo que permitió reducir notablemente los tiempos de ejecución del algoritmo MoFa.

3.4. Colecciones de grafos usadas internacionalmente

Las bases de datos bioquímicas, en especial los conjuntos de moléculas, constituyen uno de los principales campos de aplicación de la minería de SFC sobre grafos etiquetados. Estas colecciones de grafos han sido usadas para evaluar el desempeño de los algoritmos de minería de SFC. En la Tabla 3, se describen algunas de las colecciones más citadas dentro de esta temática.

La colección PTE [36] es la base de datos más pequeña que usaremos en este trabajo. Esta contiene solamente 337 grafos que representan moléculas utilizadas en competencias internacionales de verificación toxicológica predictiva [36]. Esta colección se caracteriza por tener una enorme cantidad de SCF a pesar de su pequeño tamaño; por ejemplo, posee 136981 patrones frecuentes (SCF) tomando como umbral de frecuencia el 2 % del tamaño de la colección.

Entre las bases de datos de mediado tamaño encontramos CAN2DA99 y HIV. La colección CAN2DA99 (http://dtp.nci.nih.gov/docs/cancer/cancer_data.html) contiene la representación en forma de grafo de 32557 moléculas presentes en tumores cancerígenos mientras que en HIV (http://dtp.nci.nih.gov/docs/aids/aids_data.html) se incluye la información referente a 42689 estructuras moleculares del Virus de Inmunodeficiencia Humana.

Para el estudio del desempeño de los algoritmos de minería de grafos se han utilizado partes de la colección HIV que contienen las moléculas moderadamente activas (HIV-CM) y las moléculas activas (HIV-CA). Como puede apreciarse en la Tabla 3, estas dos

Tabla 3. Colecciones de grafos.

Colección	# de grafos	promedio de aristas por grafo	tamaño del mayor grafo	# de etiquetas en los vértices
PTE	337	26	213	66
HIV-CA	423	42	186	21
HIV-CM	1 083	34	234	27
CAN2DA99	32 557	28	236	69
HIV	42 689	27	234	58
NCI	237 771	22	276	78

colecciones son pequeñas en cuanto a la cantidad de grafos pero, al igual que PTE, poseen una gran cantidad de SCF.

La colección NCI (http://cactus.nci.nih.gov/ncidb2/download.html) es la más grande entre las bases de datos que tradicionalmente se utilizan en comparaciones de algoritmos de minería de grafos. Los algoritmos de minería de SCF que usan estructuras de correspondencias (ver sección 3.3) necesitan demasiada memoria RAM para poder procesar dicha colección.

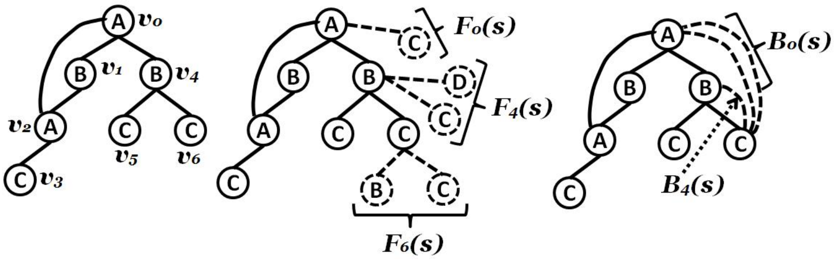
Además de estos conjuntos de grafos reales, se han estado usando colecciones artificiales producidas por generadores automáticos. Uno de los generadores automáticos de grafos sintéticos más usado es el propuesto por Kuramochi y Karypis [26]. Utilizar este tipo de colecciones artificiales permite estudiar el comportamiento de los algoritmos manteniendo o variando algunas propiedades de la colección de grafos.

4. Nuevas propiedades de la codificación DFS

Sea $s = e_0, e_1, \ldots, e_m$ un código DFS mínimo que representa un grafo etiquetado $G = \langle V, E, L, l \rangle$ donde $|V| = n$, y $|E| = m$. Denotemos mediante $RE(s)$ a el conjunto de todas las extensiones más a la derecha desde s que surgen cuando el algoritmo gSpan recorre la colección de grafos D (ver línea 3 del Algoritmo 2). Este conjunto puede ser particionado en varios subconjuntos:

$$RE(s) = B_0(s) \cup \ldots \cup B_n(s) \cup F_0(s) \cup \ldots \cup F_n(s),$$

donde $B_i(s)$ contiene las extensiones hacia atrás que tienen vértice destino con índice i, y $F_i(s)$ es el conjunto de las extensiones hacia adelante desde el vértice con índice i. Para cada vértice v_i en la rama más a la derecha, supongamos que tenga índice i, $i \neq n - 1$, la arista f_i representa la arista hacia adelante que nace en v_i y que forma parte de la rama más a la derecha. La notación e^{-1} será utilizada para referirnos a la arista e pero en el sentido inverso al que se le impuso en el código s.



$$RE(s) = B_0(s) \cup B_4(s) \cup F_0(s) \cup F_4(s) \cup F_6(s)$$

Figura 5. Ejemplo de un árbol DFS y las particiones en la rama más a la derecha.

Los siguientes dos teoremas constituyen **condiciones suficientes de no-minimalidad** para los hijos de un código DFS mínimo.

Teorema 4.1. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i y distinto del vértice más a la derecha. Si $e \in F_i(s)$ y $e \prec_l f_i$, entonces $s' = s \diamond e$ es un código DFS no-mínimo.*

Demostración. Sea h el número entero tal que $e_h = f_i$. Las aristas e_h y e parten del vértice v_i y ambas son aristas hacia adelante en s'. Por tanto, es posible alterar el recorrido DFS que da origen a s visitando e antes que e_h. Si e es visitada inmediatamente antes que e_h, el código DFS resultante tiene el siguiente formato $s'_1 = e_0, \ldots, e_{h-1}, e, e'_h, \ldots, e'_m$, donde e'_j es la misma arista e_j pero con los nuevos índices para cada $j \geq h$. Los códigos s' y s'_1 tienen el mismo prefijo $e_0, \ldots, e_{h-1}$ y estamos considerando que $e \prec_l e_h$; entonces, la condición (4) del orden lexicográfico (ver epígrafe 3.1) asegura que $s'_1 \prec_s s'$. Así, concluimos que s' es un código no-mínimo. $\square$

El teorema 4.1 permite asegurar la no-minimalidad de algunos de los hijos de un código DFS mínimo que fueron obtenidos por extensiones hacia adelante. Esta no-minimalidad es garantizada sin necesidad de realizar una prueba exhaustiva de minimalidad. Resultados similares para las extensiones hacia atrás son mostrados en el teorema 4.2.

Teorema 4.2. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Si $e \in B_i(s)$ y $e^{-1} \prec_l f_i$, entonces $s' = s \diamond e$ es un código DFS no-mínimo.*

Demostración. De manera similar a como se hizo en la demostración del teorema 4.1, sea h el número entero tal que $e_h = f_i$. Es posible alterar el recorrido DFS que da origen a s visitando e antes que e_h. Si e^{-1} es visitada inmediatamente antes que e_h, el código DFS resultante tiene el siguiente formato $s'_1 = e_0, \ldots, e_{h-1}, e^{-1}, e'_h, \ldots, e'_m$. Los códigos s' y s'_1 tienen el mismo prefijo $e_0, \ldots, e_{h-1}$ y estamos considerando que $e^{-1} \prec_l e_h$; entonces, $s'_1 \prec_s s'$. Así, concluimos que s' es un código no-mínimo. $\square$

Sea $\overline{RE}(s)$ el conjunto de todas las extensiones más a la derecha de s que se obtiene directamente desde $RE(s)$ eliminando aquellas que, según los teoremas 4.1 y 4.2, dan lugar a códigos no-mínimos ($\overline{RE}(s) \subseteq RE(s)$).

A continuación se presentan algunos resultados que permiten asegurar la minimalidad o no-minimalidad para algunos códigos DFS resultantes de aplicar las extensiones que quedaron en $\overline{RE}(s)$ sobre s. En dichos casos se puede asegurar el resultado sin necesidad de realizar pruebas exhaustivas de minimalidad. A los siguientes dos teoremas los llamaremos **condiciones de ruptura** para los hijos de s.

Teorema 4.3. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Si $e, e' \in F_i(s)$; entonces, se cumple que:*

1. *si $s \diamond e$ es un código DFS mínimo y $e \preceq_l e'$, entonces $s \diamond e'$ también es código DFS mínimo;*
2. *si $s \diamond e$ es un código DFS no-mínimo y $e' \preceq_l e$, entonces $s \diamond e'$ también es un código DFS no-mínimo.*

Demostración. Se demostrará cada caso por separado.

En el primer caso, tenemos que $s \diamond e$ es un código DFS mínimo y $e \preceq_s e'$. Supongamos que $s \diamond e'$ es un código DFS no-mínimo, entonces existe al menos un código

$s_1 = a_0, a_1, \ldots, a_{m+1}$ tal que $s_1 \prec_s s \diamond e'$. Usando la definición de $\prec_s$ (ver epígrafe 3.1), se asegura que existe un entero t, $0 \leq t \leq m$ tal que $a_k = e_k$ para todo $k < t$, y $a_t \prec_e e_t$. Es importante hacer notar que $t < m + 1$ porque s es un código DFS mínimo. Luego, de la ecuación (4) se deduce que $s_1 \prec_s s$.

Como e y e' parten desde el mismo vértice, se puede reemplazar la arista que representa a e' en s_1 por la arista e. Sea s_1' el código DFS obtenido a partir de s_1 luego del reemplazo; este código DFS es válido para el grafo codificado por $s \diamond e$ y se tiene que $s_1' \prec_s s_1 \prec_s s$. Por la condición de la equación (5), tenemos que $s_1' \prec_s s \prec_s s \diamond e$. Entonces, $s \diamond e$ es un código DFS no-mínimo, lo cual es una contradicción. Por lo tanto lo supuesto ($s \diamond e'$ es un código DFS no-mínimo) tiene que ser falso. De este modo se concluye la demostración del primer caso.

En el segundo caso, se tiene que $s \diamond e$ es un código DFS no-mínimo y $e' \preceq_s e$. Entonces, existe al menos un código $s_1 = a_0, a_1, \ldots, a_{m+1}$ tal que $s_1 \prec_s s \diamond e'$. Sea t el entero tal que $0 \leq t \leq m$, $a_k = e_k$ para todo $k < t$, y $a_t \prec_e e_t$. Luego, de la ecuación (4), se obtiene que $s_1 \prec_s s$. Como e y e' parten del mismo vértice, se puede reemplazar la arista que representa a e en s_1 por e'. El código DFS resultante (asumamos que es s_1') es válido para el grafo codificado por $s \diamond e'$ y se tiene que $s_1' \prec_s s_1 \prec_s s \prec_s s \diamond e'$. Por tanto, $s \diamond e'$ es un código no-mínimo lo que demuestra el segundo caso. □

Un resultado similar puede ser planteado para las extensiones hacia atrás en el siguiente teorema.

Teorema 4.4. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Si $e, e' \in B_i(s)$; entonces, se cumple que:*

1. *si $s \diamond e$ es un código DFS mínimo y $e \preceq_l e'$, entonces $s \diamond e'$ también es código DFS mínimo;*
2. *si $s \diamond e$ es un código DFS no-mínimo y $e' \preceq_l e$, entonces $s \diamond e'$ también es un código DFS no-mínimo.*

Demostración. La demostración es similar a la que fue dada para el teorema 4.3. □

Los teoremas 4.3 y 4.4, plantean que se puede reutilizar el resultado de una prueba de minimalidad que ya se hizo en un hijo de s para predecir la minimalidad o no-minimalidad en algunos de los restantes hijos de s. Los elementos de $\overline{RE}(s)$ pueden ser ordenados utilizando $\prec_e$, de este modo los subconjuntos que definen la partición antes descrita quedarían dispuestos en el siguiente orden:

$$\bar{B}_0(s), \ldots, \bar{B}_n(s), \bar{F}_0(s), \ldots, \bar{F}_n(s),$$

donde cada $\bar{B}_i(s)$ (o $\bar{F}_i(s)$) se obtuvo a partir de su correspondiente $B_i(s)$ (o $F_i(s)$) luego de realizar el filtrado utilizando las condiciones de no-minimalidad (teoremas 4.1 y 4.2). Dentro de cada conjunto $\bar{B}_i(s)$ (o $\bar{F}_i(s)$) los elementos quedarán ordenados según el orden de las etiquetas $\prec_l$ ya que todos comparten el mismo origen y el mismo destino. En lo adelante se asumirá que estos conjuntos están ordenados de esta manera.

Un resultado que se sigue del teorema 4.3 se muestra a continuación. A este corolario lo llamaremos **principio de corte** para las extensiones hacia adelante, ya que éste establece la manera en que los códigos DFS mínimos y no-mínimos están distribuidos dentro de cada $\bar{F}_i$.

Corolario 4.1. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Si existen extensiones en $F_i(s)$ que producen códigos DFS no-mínimos entonces tales extensiones se encuentran al principio de $F_i(s)$. Además, si existen extensiones que producen códigos DFS mínimos, entonces tales extensiones se encuentran al final de $F_i(s)$.*

Demostración. Si todas las extensiones en $F_i(s)$ producen códigos DFS no-mínimos o todas producen códigos DFS mínimos entonces está clara la veracidad del corolario. Sólo queda el caso en que existan ambos tipos de extensiones en $F_i(s)$. Sean $\hat{e}, \check{e} \in F_i(s)$ dos extensiones tales que $\hat{e}$ produce un código DFS no-mínimo y $\check{e}$ produce un código DFS mínimo. Del teorema 4.3 se deduce directamente que $\hat{e} \prec_e \check{e}$, porque en caso contrario $s \diamond \check{e}$ debería ser no-mínimo, y esto contradice las hipótesis. Por tanto, las extensiones que producen códigos DFS no-mínimos van primero que aquellas que producen códigos DFS mínimos. □

De este corolario se deduce que cada conjunto $F_i(s)$, puede ser particionado en dos subconjuntos $F_i(s) = \hat{F}_i(s) \cup \check{F}_i(s)$ tal que todas las extensiones en $\hat{F}_i(s)$ producen códigos DFS no-mínimos, todas las extensiones en $\check{F}_i(s)$ producen códigos DFS mínimos y las extensiones de $\hat{F}_i(s)$ son anteriores a las de $\check{F}_i(s)$ según la relación de orden $\prec_l$.

Para las extensiones hacia atrás se obtiene un corolario del teorema 4.4 similar al anterior. A este resultado lo llamaremos **principio de corte** para las extensiones hacia atrás.

Corolario 4.2. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Si existen extensiones en $B_i(s)$ que producen códigos DFS no-mínimos entonces tales extensiones se encuentran al principio de $B_i(s)$. Además, si existen extensiones que producen códigos DFS mínimos, entonces tales extensiones se encuentran al final de $B_i(s)$.*

Demostración. La demostración es similar a la que fue dada para el corolario 4.1.□

De este modo, cada conjunto $B_i(s)$ también puede ser particionado en $B_i(s) = \hat{B}_i(s) \cup \check{B}_i(s)$ donde las extensiones en $\hat{B}_i(s)$ producen códigos DFS no-mínimos, las de $\check{B}_i(s)$ producen códigos DFS mínimos y los elementos de $\hat{B}_i(s)$ son anteriores a los de $\check{B}_i(s)$ según el orden lexicográfico.

Así el subconjunto $ME(s) \subseteq \overline{RE}(s)$ que contiene sólo aquellas extensiones que producen códigos DFS mínimos viene dado por la ecuación (7).

$$ME(s) = \check{B}_0(s) \cup \ldots \cup \check{B}_n(s) \cup \check{F}_0(s) \cup \ldots \cup \check{F}_n(s). \tag{7}$$

En la minería de SCF las extensiones que producen códigos DFS no-mínimos no son de interés, en todos los casos son podadas del espacio de búsqueda. Con los resultado expuestos en esta capítulo es posible localizar las extensiones que producen códigos DFS no-mínimos de manera eficiente. En el próximo capítulo se ilustrará el modo en que pueden ser utilizados dichos resultados en la minería de SCF.

5. Algoritmo gRed

Los resultados presentados en el capítulo 4 pueden ser usados para mejorar el desempeño del algoritmo gSpan. Por ejemplo, supongamos que $|RE(s)| = N$, entonces para obtener el conjunto $ME(s)$ siguiendo el procedimiento de gSpan se requiere realizar N pruebas exhaustivas de minimalidad. En lo que sigue mostraremos cómo el algoritmo gRed [13] reduce este número de pruebas.

Teniendo en cuenta que las condiciones suficientes de minimalidad pueden comprobarse con costo computacional $O(1)$, éstas pueden ser ejecutadas mientras se recorre D (ver línea 3 del Algoritmo 2). Así el conjunto de posibles extensiones en el algoritmo gRed luego de recorrer D es reducido a $\overline{RE}(s)$ ($|\overline{RE}(s)| = \bar{N} \leq N$).

Supongamos que $|\bar{B}_0(s)| = \bar{b}_0, \ldots, |\bar{B}_n(s)| = \bar{b}_n, |\bar{F}_0(s)| = \bar{f}_0, \ldots, |\bar{F}_n(s)| = \bar{f}_n$, entonces

$$\sum_i \bar{b}_i + \sum_i \bar{f}_i = \bar{N}, \tag{8}$$

donde las sumatorias $\sum_i$ se realiza sobre los vértices de la rama más a la derecha del árbol DFS que di origen a s.

Tanto el algoritmo gSpan como gRed, ordenan el conjunto de posibles extensiones según el orden lexicográfico $\prec_e$ mientras se recorre D. En el caso del algoritmo gSpan las bondades de dicho ordenamiento no son explotadas al máximo. Los siguientes teoremas ilustran los recursos que utilizó gRed para explotar dichas bondades.

Primero, demostraremos el siguiente lema que más adelante será de utilidad.

Lema 5.1. *Si* $N(1) = 1$ *y* $N(f) \leq N(\lfloor f/2 \rfloor) + 1$ *para cualquier* $f > 1$, *entonces* $N(f) \leq log_2(f) + 1$.

Demostración. Demostremos este lema por inducción matemática. El *caso base* de la inducción está garantizado con $N(1) = 1$.

Hipótesis de inducción: Supongamos que para $h \leq f - 1$, $N(h) \leq log_2(h) + 1$.

Como $\lfloor f/2 \rfloor \leq f - 1$ para $f \geq 2$, se utilizará la hipótesis de inducción para asegurar que

$$N(f) \leq N(\lfloor f/2 \rfloor) + 1 \leq (log_2(\lfloor f/2 \rfloor) + 1) + 1 = log_2(\lfloor f/2 \rfloor) + 2.$$

Utilizando que $\lfloor f/2 \rfloor \leq f/2$ obtenemos que

$$N(f) \leq log_2(f/2) + 2.$$

Luego, se emplean las propiedades de los logaritmos $log_2(f/2) = log_2(f) - log_2(2)$ y $log_2(2) = 1$, para concluir que

$$N(f) \leq log_2(f) + 1.$$

Así queda probado el *paso de inducción* y se concluye la demostración del lema.□

Teorema 5.1. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Supongamos que $|\bar{F}_i(s)| = \bar{f}_i > 0$. Entonces, el número de pruebas exhaustivas de minimalidad que se requiere para separar cada conjunto $\bar{F}_i(s)$ en $\hat{F}_i(s) \cup \check{F}_i(s)$ es a lo sumo $log_2(\bar{f}_i) + 1$.*

Demostración. Si $\bar{f}_i = 1$ se requiere una sola prueba exhaustiva de minimalidad, esto demuestra que el teorema es válido para este caso.

Para simplificar las notaciones hagamos que $F = \bar{F}_i(s)$ y $f = \bar{f}_i$. Denotemos mediante $N(f)$ el número de pruebas exhaustivas de minimalidad que se requiere para procesar un conjunto con f elementos (extensiones).

Sea e la mediana del conjunto F según el orden $\prec_e$ y denotemos $F_< = \{e' \in F | e' \prec_l e\}$ y $F_> = \{e' \in F | e' \succ_l e\}$. Del teorema 4.3 se puede predecir la minimalidad o no-minimalidad de los códigos DFS que se obtienen a partir de las extensiones de uno de los conjuntos $F_<$ ó $F_>$ respectivamente. En efecto si se comprueba que $s \diamond e$ es mínimo, entonces las extensiones de $F_>$ generan códigos mínimos. En el otro caso, las extensiones de $F_<$ generan códigos no-mínimos. En cualquiera de los dos casos, de uno de los dos conjuntos $F_<$ ó $F_>$ no se puede decir nada y por tanto se requerirán más pruebas exhaustivas de minimalidad. El tamaño de dichos conjuntos no supera $\lfloor f/2 \rfloor$.

Como se puede apreciar, hemos reducido el problema de tamaño f a un subproblema de tamaño a lo sumo $\lfloor f/2 \rfloor$ y sólo se ha realizado una prueba exhaustiva de minimalidad para la mediana del conjunto F. Por tanto se puede plantear la siguiente desigualdad $N(f) \leq N(\lfloor f/2 \rfloor) + 1$. Como $N(1) = 1$ se puede aplicar el lema 5.1; concluyendo así la demostración del teorema. □

Un resultado análogo también se tiene para las extensiones hacia atrás.

Teorema 5.2. *Sea s un código DFS mínimo y sea v_i un vértice en la rama más a la derecha de s con índice i. Supongamos que $|\bar{B}_i(s)| = \bar{b}_i > 0$. Entonces, el número de pruebas exhaustivas de minimalidad que se requiere para separar cada conjunto $\bar{B}_i(s)$ en $\hat{B}_i(s) \cup \check{B}_i(s)$ es a lo sumo $log_2(\bar{b}_i) + 1$.*

Demostración. La demostración es similar a la del teorema 5.1.□

Usando los teoremas 5.1 y 5.2, el número de pruebas exhaustivas de minimalidad (K), que se requiere para obtener el conjunto $ME(s)$ a partir de $\overline{RE}(s)$, cumple la siguiente desigualdad

$$K \leq \sum_i (log_2(\bar{f}_i) + 1) + \sum_i (log_2(\bar{b}_i) + 1), \tag{9}$$

donde las sumatorias $\sum_i$ se realizan sobre los vértices de la rama más a la derecha. Denotemos $r(s)$ el número de vértices en la rama más a la derecha de s. Por otra parte, como la función $log_2(x) : [1, +\infty) \to [0, +\infty)$ es convexa, la desigualdad (9) se transforma en

$$K \leq 2r(s)log_2\left(\frac{\sum_i \bar{f}_i + \sum_i \bar{b}_i}{2r(s)}\right) + 2r(s).$$

El argumento del $log_2(\cdot)$ puede ser simplificado mediante (8) y quedaría

$$K \leq 2r(s)log_2\left(\frac{\bar{N}}{2r(s)}\right) + 2r(s) \leq 2r(s)log_2\left(\frac{N}{2r(s)}\right) + 2r(s). \tag{10}$$

Algoritmo 3: gRed (s, D, δ, S)

```
Entrada: $s$ - un código DFS mínimo (representa a un grafo frecuente), $D$ - conjunto de grafos, $\delta$ -
         umbral de frecuencia mínima.
Salida: $S$ - Conjunto de todos los subgrafos conexos frecuentes
$S \leftarrow S \cup \{s\}$;
$\overline{RE}(s) \leftarrow$ Conjunto de aristas $e$ tales que exista en $D$ la extensión $s \diamond e$ y $e$ no es filtrada luego de
comprobar las condiciones suficientes de no-minimalidad;
Eliminar de $\overline{RE}(s)$ las extensiones infrecuentes;
Obtener el conjunto $ME(s)$ (ver ecuación (7)) según el principio de corte y mediante búsquedas
binarias;
foreach extensión $e \in ME(s)$ do
    gRed ($s \diamond e$, $D$, $\delta$, $S$);
end
```

Figura 6. Esquema general del algoritmo gRed.

La gráfica de la Fig. 7 muestra una comparación entre la cantidad de pruebas exhaustivas de minimalidad que requieren los algoritmos gSpan y gRed luego de extender un código s. En esta gráfica, se presenta el caso peor de gRed asumiendo igualdad en la desigualdad (10), y por tanto es de esperar que en los resultados reales de gRed el número de estas costosas pruebas sea menor que lo que se muestra.

En la demostración del teorema 5.1, se puede apreciar el procedimiento para reducir las pruebas exhaustivas de minimalidad. Para cada conjunto $\bar{F}_i(s)$ (o $\bar{B}_i(s)$) se realiza una búsqueda binaria que permite separar las extensiones que producen códigos no-mínimos de las que producen códigos mínimos. Esta búsqueda binaria se justifica teóricamente por las condiciones de ruptura y los principios de corte. El algoritmo gRed propone utilizar este procedimiento para optimizar el proceso de minería de SCF. La Fig. 6 muestra el esquema general de gRed.

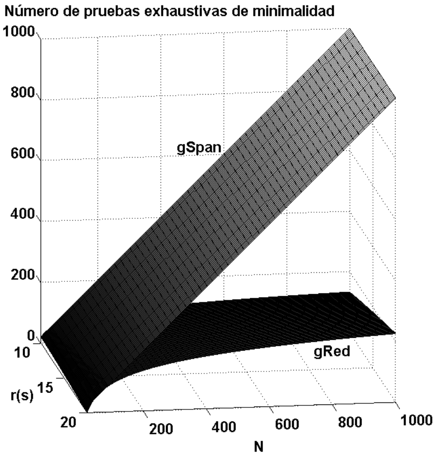


Figura 7. Número de pruebas exhaustivas de minimalidad en dependencia del tamaño de la rama más a la derecha $r(s)$ y el número de extensiones posibles N.

6. Resultados experimentales

El algoritmo gRed es comparado con los cuatro algoritmos de minería de SCF que más se han citado recientemente: gSpan, MoFa, FFSM y Gaston. Estos últimos, además de ser los algoritmos más referenciados, describen las ideas que más éxito han tenido en los más recientes estudios comparativos [44,33]. Los algoritmos cuya estrategia de búsqueda es en amplitud, entre ellos los basados en Apriori, han mostrado ser menos eficientes que aquellos diseñados con búsqueda en profundidad [33,14]. Los algoritmos especializados en minería de SCFC, SCFM o alguna otra variante tampoco son incluidos en la comparación con gRed, ya que este último busca todos los SCF. Otros algoritmos concebidos como optimizaciones de gSpan (por ejemplo Edgar, ADI y FSP), no son incluidos en la comparación porque las mismas ideas introducidas por gRed pueden insertarse fácilmente en dichos algoritmos y mejorarían aún más sus desempeños.

Los algoritmos gSpan, MoFa, FFSM y Gaston fueron implementados en el paquete ParMol [29] con el fin de realizar un estudio comparativo [44]. Uno de los aspecto más criticados de esta comparación fue el hecho de que el paquete ParMol fue implementado en el lenguaje de programación Java y por tanto numerosas optimizaciones, alcanzables en los lenguajes de más bajo nivel como ANCI C, no pudieron ser utilizadas. No obstante, fue la primera vez que estos algoritmos fueron comparados en igualdad de condiciones; por ejemplo, todos los algoritmos usan las mismas bibliotecas para el manejo de los grafos. El paquete ParMol es distribuido gratuitamente bajo licencia GNU.

El algoritmo gRed fue implementado también en Java, utilizando las mismas bibliotecas para el manejo de grafos del paquete ParMol. Con el fin de complementar el estudio comparativo realizado por Wörlein *et al.* [44] se repitieron los experimentos sobre bases de datos reales incluyendo los resultados de gRed. Para realizar las corridas de los algoritmos se utilizó la máquina virtual de Java 1.5.0 de la SUN. Todos los experimentos fueron realizados en una PC Intel Core 2 Duo a 2.2 GHz con 2 GB de memoria RAM.

El principal aporte de gRed es la optimización que introduce en la detección y poda de los elementos duplicados durante el recorrido del espacio de búsqueda. Para tener una idea más clara sobre la mejora que introduce gRed, se estimará el porcentaje del tiempo de ejecución que representa cada una de las etapas en la minería de SCF. Para calcular los tiempos de ejecución de cada etapa se contabilizó la duración de cada llamado a las funciones correspondientes. De este modo se estimó el porcentaje del tiempo de ejecución que representó cada etapa en las corridas. Los resultados utilizando las colecciones HIV-CM y PTE con el umbral de frecuencia 5 % se muestran en la Tabla 4. Los resultados con las colecciones HIV-CA y HIV-CM fueron muy similares en cuanto a los porcentajes en tiempo de ejecución por eso no se incluyó HIV-CA.

Tabla 4. Porcentaje en tiempo de ejecución que representó cada una de las etapas de la minería de SCF en los algoritmos utilizando las colecciones HIV-CM y PTE.

Etapa	HIV-CM					PTE				
	gRed	gSpan	Gaston	FFSM	MoFa	gRed	gSpan	Gaston	FFSM	MoFa
Detección y poda de duplicados	0.9 %	1.4 %	1.0 %	1.2 %	12.3 %	4.9 %	10.5 %	6.0 %	5.7 %	15.5 %
Construcción de estructuras	2.7 %	2.9 %	62 %	11.4 %	45.3 %	3.3 %	2.7 %	59 %	10.4 %	40.2 %
Enumeración de candidatos	85.2 %	84 %	33 %	62.3 %	19.2 %	79 %	75 %	25 %	61.2 %	22.3 %
Tiempo total de la ejecución	33.7s	34.3s	17.2s	38.1s	52.6s	5.6s	7.5s	3.5s	9.4s	43.6s

En la Tabla 4 puede observarse que la etapa de la detección y poda de los duplicados no representa un gran porcentaje del tiempo de ejecución de los algoritmos. En dependencia de la topología de los grafos en la colección, puede ser mayor o menor el tiempo que se requiera para esta etapa. El algoritmo MoFa es el que más tiempo dedica ya que éste incluye pruebas de isomorfismo y podas por el cumplimiento de ciertas reglas estructurales. El segundo algoritmo que más tiempo relativo dedica a dicha etapa es gSpan, en contraposición con gRed que es el algoritmo que menos tiempo relativo emplea. Esto último muestra el efecto positivo de las optimizaciones introducidas por gRed utilizando las nuevas propiedades de la codificación DFS.

En cuanto al tiempo total mostrado en la Tabla 4, el algoritmo gRed fue aventajado por Gaston. Esto último tiene una explicación clara, gRed emplea la mayor parte de su tiempo en la enumeración de los candidatos mientras Gaston emplea la mayor parte del tiempo en la construcción de las estructuras de correspondencias. La enumeración de los candidatos consume mucho tiempo cuando hay que realizar las costosas pruebas de sub--isomorfismo para localizar los posibles candidatos y este es el caso de gRed y gSpan. Las estructuras de correspondencias utilizadas en Gaston eliminan totalmente las costosas pruebas de sub-isomorfismo. El costo de mantener estas estructuras en Gaston fue bajo en parte porque las colecciones de grafos usadas en este experimento son pequeñas. En lo que sigue se mostrarán experimentos para colecciones de mayor tamaño.

Con gRed se obtienen mejores tiempos de ejecución que con los algoritmos Gaston, FFSM y MoFa (ver Tabla 5) en las colecciones de mediano y mayor tamaño como CAN2DA99, HIV y NCI. Los mejores resultados con NCI fueron obtenidos por gRed y gSpan ya que los otros tres algoritmos no pudieron procesar dicha colección; es por eso que la Tabla 5 sólo contiene los experimentos con CAN2DA99 y HIV. Los algoritmos FFSM y MoFa mostraron los peores tiempos de ejecución en todos los experimentos que se muestran. El algoritmo gSpan no se incluyó en este experimento porque más adelante se presentará un estudio comparativo para resaltar mejor las diferencias entre gSpan y gRed.

Tabla 5. Comparación de los tiempos de ejecución de gRed respecto a los algoritmos Gaston, FFSM y MoFa en las colecciones HIV y CAN2DA99.

HIV					CAN2DA99				
%	**gRed**	Gaston	FFSM	MoFa	%	**gRed**	Gaston	FFSM	MoFa
3	**849.188s**		899.172s	2328.328s	3	**651.437s**		812.922s	4188.3s
4	**583.985s**		629.438s	1140.750s	4	**370.922s**		461.282s	1789.8s
5	**459.172s**	603.120s	511.297s	809.157s	5	**258.563s**	279.27s	318.938s	1026.3s
6	**375.500s**	461.360s	442.500s	613.110s	6	**193.812s**	198.516s	246.359s	710.25s
7	315.156s	**311.402s**	389.406s	488.484s	7	153.5s	**130.344s**	195.047s	504.56s

En este experimento, gRed muestra su supremacía en valores bajos del umbral de soporte. Los umbrales bajos son los más usados en las aplicaciones de la minería de SCF, ya que en la medida que baja este umbral, los candidatos a procesar poseen una complejidad estructural mayor (aumenta la cantidad de aristas y es mayor la cantidad de ciclos) desde el punto de vista de su representación como grafos. Esto último también es la causa de que los tiempos de ejecución en Gaston se incrementen para umbrales de soporte pequeños; ya que es en la parte de generar los grafos cíclicos donde este algoritmo muestra peor desempeño. En la Tabla 5 se muestra el crecimiento en los tiempos de ejecución que experimenta Gaston para umbrales de soporte menores o iguales que 6 %. Inclusive, los registros de Gaston para los umbrales de soporte menores o iguales que 4 % no pudieron ser mostrados porque las estructuras de correspondencias, usadas por este algoritmo para el indexado de la colección de grafos, no cupieron en memoria. En cambio, Gaston supera los registros de gRed para umbrales mayores o iguales que 7 %. Esto último está en concordancia con el hecho de que los grafos frecuentes para umbrales altos casi siempre son caminos o árboles, precisamente donde Gaston tiene sus mejores heurísticas.

El último experimento, que se muestra en la Tabla 6, tiene como objetivo mostrar el aporte de las ideas de gRed en comparación con gSpan. Se seleccionaron dos colecciones de mediano tamaño (CAN2DA99 y HIV), una colección pequeña (PTE) y la colección más grande (NCI) entre las reportadas en la literatura. Los umbrales de soporte se seleccionaron entre 2 % y 5 %, ya que para valores más altos de dicho umbral se procesan menos candidatos y por tanto las diferencias entre los desempeños son menores. Además del tiempo de ejecución (T.E.), se incluyeron los registros de la cantidad de candidatos duplicados (N.C.D.), la cantidad de pruebas exhaustivas realizadas para verificar si un candidato está en forma canónica (N.P.E.), y la cantidad de SCF obtenidos en cada prueba.

Tabla 6. Comparación entre gRed y gSpan en las colecciones de datos reales respecto a sus tiempos de ejecución y variando los umbrales de soporte.

PTE							
%	|SCF|	T.E. (seg.)		N.C.D.		N.P.E.	
		gSpan	gRed	gSpan	gRed	gSpan	gRed
2	136981	224.891	131.266	189610	37842	326559	161202
3	18146	34.328	22.828	28324	6303	46445	21947
4	5955	14.297	9.484	9774	2073	15709	7123
5	3627	7.547	5.625	5568	1172	9176	4227

CAN2DA99							
%	|SCF|	T.E. (seg.)		N.C.D.		N.P.E.	
		gSpan	gRed	gSpan	gRed	gSpan	gRed
2	17091	2090.796	1987.562	19726	14788	36808	29765
3	6277	683.500	651.437	6817	5043	13085	10442
4	3507	383.219	370.922	3716	2699	7216	5699
5	2336	263.828	258.563	2390	1726	4720	3704

HIV							
%	|SCF|	T.E. (seg.)		N.C.D.		N.P.E.	
		gSpan	gRed	gSpan	gRed	gSpan	gRed
2	25127	1660.046	1578.453	22116	12681	47235	33974
3	11491	884.750	849.188	10012	5602	21495	15253
4	6643	607.672	583.985	5852	3233	12488	8770
5	4422	474.578	459.172	3958	2209	8374	5919

NCI							
%	|SCF|	T.E. (seg.)		N.C.D.		N.P.E.	
		gSpan	gRed	gSpan	gRed	gSpan	gRed
2	6636	5981.063	4929.953	7395	5482	14023	11395
3	2593	1451.859	1385.484	2645	1825	5230	4034
4	1546	920.531	878.031	1438	965	2977	2288
5	1054	677.484	640.343	892	576	1941	1478

Leyenda	
%	Umbral de soporte utilizado
|SCF|	Número de patrones frecuentes encontrados
T.E.	Tiempo de ejecución en segundos
N.C.D.	Número de candidatos duplicados
N.P.E.	Número de pruebas exhaustivas

La primera conclusión que se obtiene al analizar la Tabla 6 es que gRed obtiene siempre mejores resultados que gSpan. El N.C.D. en gRed estuvo casi siempre entre del 50 % y el 70 % del N.C.D. obtenido por gSpan. En el caso de la colección PTE para el umbral de soporte igual a 2 %, se obtuvo que el N.C.D. de gRed es menor que la quinta parte del N.C.D. de gSpan. Algo similar ocurre con los valores obtenidos para el N.P.E., en muchos casos gRed realizó alrededor de la mitad de las pruebas exhaustivas de gSpan. Otro aspecto importante es que mientras más grande es la cantidad de SCF más grande es la mejora mostrada por gRed en comparación con gSpan para todos los aspectos considerados.

7. Conclusiones

Encontrar algoritmos eficientes para la minería de SCF sigue siendo una línea de investigación abierta. Es cierto que en los últimos ocho años esta temática ha tenido un notable avance; no obstante, quedan problemas teóricos que no han sido estudiados con suficiente profundidad. Los algoritmos diseñados bajo el paradigma llamado *crecimiento de patrones* han mostrado su supremacía respecto a aquellas soluciones propuestas bajo el paradigma *Apriori*, sobre todo cuando es necesario procesar colecciones de datos de gran tamaño y para bajos umbrales de frecuencia. Recorrer el espacio de búsqueda en profundidad, mediante el crecimiento patrones, permite obtener buenos tiempos de respuestas y ahorrar notablemente los recursos de memoria.

En este reporte técnico, fue introducido un nuevo algoritmo para la minería de subgrafos conexos frecuentes llamado gRed. Nuevas propiedades de la codificación DFS, fueron utilizadas para optimizar el proceso de minería reduciendo el espacio de candidatos. En esta investigación, se reafirma que la codificación DFS es una prometedora forma de representación de grafos para la minería de grafos y que aún no ha sido suficientemente estudiada.

El desempeño de gRed es comparado con cuatro algoritmos reportados en la literatura (gSpan, MoFa, FFSM y Gaston).

La experimentación muestra que gRed mejora a gSpan, MoFa y FFSM en todos las pruebas. Además, gRed obtiene mejores tiempos de ejecución que Gaston para bajos umbrales de soporte cuando las bases de datos son grandes.

Esta versión de gRed fue implementada utilizando *listas de identificadores*(LI) para indexar la colección de grafos, durante el recorrido del espacio de búsqueda y el conteo de soporte. En los resultados experimentales se muestra que un gran porcentaje del tiempo de ejecución de gRed se dedica a las costosas pruebas de sub-isomorfismo. Por tanto, si se le adapta a gRed una estructura de datos que indexe mejor la colección de grafos que las LI, se logrará un mejor desempeño.

Referencias

1. R. Agrawal, T. Imielinski and A.N. Swami. Mining Association Rules Between Sets of Items in Large Databases. *In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, pp. 207-216, 1993.
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *In Proceedings of the 1994 International Conference on Very Large Data Bases (VLDB'94)*, Santiago, Chile, pp. 487-499, 1994.
3. R. Agrawal and R. Srikant. Mining Sequential Patterns. *In Proceedings of the International Conference on Data Engineering (ICDE'95)*, Taipei, Taiwan, pp. 3-14, 1995.
4. T. Assai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto and S. Arikawa, Efficient Substructure Discovery from Large Semi-structured Data. *In Proceedings of the 2002 SIAM International Conference on Data Mining (SDM'02)*, Arlington, VA, 2002
5. J.L. Balcázar, A. Bifet and A. Lozano. Mining Frequent Closed Unordered Trees Through Natural Representations. *In Proceedings of the 15th International Conference on Conceptual Structures (ICCS 2007)*, pp. 347-359, 2007.
6. C. Borgelt. Canonical Forms for Frequent Graph Mining. *In Proceedings of the 30th Annual Conference of the Gesellschaft für Klassifikation e.V.*, Freie Universität, Berlin, pp. 8-10, 2006.
7. C. Borgelt and M.R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. *In Proceedings of the 2002 International Conference on Data Mining (ICDM'02)*, Maebashi, Japan, pp. 211-218, 2002.
8. C. Borgelt and M. Fiedler. Graph mining: Repository vs. Canonical Form. *In Data Analysis, Machine Learning and Applications: Proceedings of the 31st Annual Conference of the Gesellschaft für Klassifikation*, pp. 229-236, 2007.
9. C. Borgelt, T. Meinl and M.R. Berthold. Advanced Pruning Strategies to Speed Up Mining Closed Molecular Fragments. *In Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics*, Vol. 5, pp. 4535-4570, 2004.
10. D. Burdick, M. Calimlim and J. Gehrke. Mafia: A Maximal Frequent Itemset Algorithm for Transactional Databases. *In Proceedings of the International Conference on Data Engineering (ICDE'01)*, Heidelberg, Germany, pp. 443-452, 2001.
11. Deshpande, M., Kuramochi, M., N. Wale and G. Karypis. Frequent Substructure-Based Approaches for Classifying Chemical Compounds. *IEEE Transaction on Knowledge and Data Engineering*, Vol. 17, Number 8, pp. 1036-1050, 2005.
12. P. Foggia, C. Sansone and M. Vento. A Performance Comparison of Five Algorithms for Graph Isomorphism. *In Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pp. 188-199, 2001.
13. A. Gago-Alonso, J.E. Medina-Pagola, J.A. Carrasco-Ochoa and J.F. Martínez-Trinidad. Mining Frequent Connected Subgraphs Reducing the Number of Candidates. *In Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD'08)*, pp. 365–376, 2008.
14. J. Han, H. Cheng, D. Xin and X. Yan. Frequent Pattern Mining: Current Status and Future Directions. *Data Mining and Knowledge Discovery*, 10th Anniversary Issue, Vol. 15, Number 1, pp. 55-86, 2007.
15. J. Han, J. Pei and Y. Yin. Mining Frequent Patterns without Candidate Generation. *In Proceedings of the 2000 ACM-SIGMOD International Conference on Management of Data (SIGMOD'2000)*, Dallas, TX, pp. 1-12, 2000.
16. S. Han, W.K. Ng and Y. Yu. FSP: Frequent Substructure Pattern Mining. *In Proceedings of the 6th International Conference on Information, Communications & Signal Processing*, pp. 1-5, 2007.

17. M. Hong, H. Zhou, W. Wang and B. Shi. An Efficient Algorithm of Frequent Connected Subgraph Extraction. *In Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'03)*, LNAI 2347, pp. 40-51, 2003.
18. M.S. Hossain and R.A. Angryk. GDClust: A Graph-Based Document Clustering Technique. *In Proceedings of the 7th IEEE International Conference on Data Mining Workshops*, pp. 417-422, 2007.
19. J. Huan, W. Wang and J. Prins. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In Proceedings of the 2003 International Conference on Data Mining (ICDM'03)*, Melbourne, FL, pp. 549-552, 2003.
20. J. Huan, W. Wang, J. Prins and J. Yang. SPIN: Mining Maximal Frequent Subgraphs from Graph Databases. *In Proceedings of the 2004 ACM SIGKDD international conference on knowledge discovery in databases (KDD'04)*, Seattle, WA, pp. 581-586, 2004.
21. A. Inokuchi. Mining Generalized Substructures from a Set of Labeled Graphs. *In Proceedings of the 4th IEEE International Conference on Data Mining (ICDM'04)*, Los Alamitos, CA, pp. 415-418, 2004.
22. A. Inokuchi, T. Washio and H. Motoda. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data, *In Proceedings of the 2000 European symposium on the principle of data mining and knowledge discovery (PKDD'00)*, Lyon, France, pp. 13-23, 2000.
23. A. Inokuchi, T. Washio, K. Nishimura and H. Motoda. A Fast Algorithm for Mining Frequent Connected Subgraphs, *Technical Report RT0448, In IBM Research, Tokyo Research Laboratory*, pp. 10, 2002.
24. M. Koyutürk, A. Grama and W. Szpankowski. An Efficient Algorithm for Detecting Frequent Subgraphs in Biological Networks, *Bioinformatics*, Vol. 20, Suppl. 1, pp. i200-i207, 2004.
25. M. Kuramochi and G. Karypis. Discovering Frequent Geometric Subgraphs. *Information Systems*, Vol. 32, pp. 1101–1120, 2007.
26. M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. *In Proceedings of the 2001 International Conference on Data Mining (ICDM'01)*, San Jose, CA, pp. 313-320, 2001.
27. B.D. Mckay. Practical Graph Isomorphism. *Congressus Numerantium*, Vol. 30 pp. 45-87, 1981.
28. T. Meinl and M.R. Berthold. Hybrid Fragment Mining with MoFa and FSG. *In Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics*, Vol. 5, pp. 4559-4564, 2004.
29. T. Meinl, M. Wörlein, O. Urzova, I. Fischer, and M. Philippsen. The ParMol Package for Frequent Subgraph Mining. *In Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs'06)*, pp. 94-105, 2006.
30. H. Motoda. Pattern Discovery from Graph-Structured Data: A Data Mining Perspective. *In Proceedings of the 20th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pp. 12-22, 2007.
31. S. Nijssen and J. Kok. A Quickstart in Frequent Structure Mining can Make a Difference. *In Proceedings of the 2004 ACM SIGKDD International Conference on Kowledge Discovery in Databases (KDD'04)*, Seattle, WA, pp. 647-352, 2004.
32. S. Nijssen and J. Kok. Frequent Graph Mining and its Application to Molecular Databases. *In Proceedings of the 2004 IEEE Conference on Systems, Man and Cybernetics (ICSMC2004)*, Vol. 5, pp. 4571-4577, 2004.
33. S. Nijssen and J. Kok. Frequent Subgraph Miners: Runtimes Don't Say Everything. *In Proceedings Mining and Learning with Graphs (MLG 2006), workshop held with ECML-PKDD 2006*, Berlin, Germany, pp. 173-180 2006.
34. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-projected Pattern Growth. *In Proceedings of the International Conference on Data Engineering (ICDE'01)*, Heidelberg, Germany, pp. 215-224, 2001.

35. R.C. Read and D.G. Corneil. The Graph Isomorphism Disease. *Journal of Graph Theory*, Vol. 1, pp. 339-334, 1977.
36. A. Srinivasan, R.D. King, S.H. Muggleton and M. Sternberg. The Predictive Toxicology Evaluation Challenge. *In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'1997)*, Nagoya, Japan, pp. 1-6, 1997.
37. L. Thomas, S. Valluri and K. Karlapalem. MARGIN: Maximal Frequent Subgraph Mining. *In Proceedings of the 6th International Conference on Data Mining (ICDM'06)*, Hong Kong, pp. 1097-1101, 2006.
38. K. Tsuda and T. Kudo. Clustering Graphs by Weighted Substructure Mining. *In Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, Pennsylvania, pp. 953-917, 2006.
39. J.R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, Vol. 23, Number 1, pp. 31-42, 1976.
40. N. Vanetik, E. Gudes and S.E. Shimony. Computing Frequent Graph Patterns from Semistructured Data. *In Proceedings of the 2002 International Conference on Data Mining (ICDM'02)*, Maebashi, Japan, pp. 458-435, 2002.
41. C. Wang, W. Wang, J. Pei, Y. Zhu and B. Chi. Scalable Mining of Large Disk-based Graph Databases. *In Proceedings of the 2004 ACM SIGKDD international conference on knowledge discovery in databases (KDD'04)*, Seattle, WA, pp. 316-325, 2004.
42. T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl*, New York, NY, Vol. 5, Number 1, pp. 59-68, 2003.
43. M. Wörlein, A. Dreweke, T. Meinl, I. Fischer and M. Philippsen. Edgar: the Embedding-baseD GrAph MineR. *In Proceedings of the International Workshop on Mining and Learning with Graphs*, Berlin, Germany, pp. 221-228, 2006.
44. M. Wörlein, T. Meinl, I. Fischer and M. Philippsen. A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. *In Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'05)*, Porto, Portugal, pp. 392-403, 2005.
45. J. Wu and L. Chen. Mining Frequent Subgraph by Incidence Matrix Normalization. *Journal of Computers*, Vol. 3, Number 10, pp. 109-115, October 2008.
46. T. Yamamoto, T. Ozaki and T. Ohkawa. Discovery of Frequent Graph Patterns that Consist of the Vertices with the Complex Structures. *In Post Proceedings of the Third International Workshop on Mining Complex Data (MCD'07)*, Warsaw, Poland, Revised Selected Papers, LNCS 4944 pp.143-156, 2008.
47. X. Yan and J. Han. CloseGraph: Mining Closed Frequent Graph Patterns. *In Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, pp. 286-295, 2003.
48. X. Yan and J. Han. gSpan: Graph-based Substructure Pattern Mining. *In Proceedings of the 2002 International Conference on Data Mining (ICDM'02)*, Maebashi, Japan, pp. 721-724, 2002.
49. X. Yan, P.S. Yu and J. Han. Graph indexing: A Frequent Structure-based Approach. *In Proceedings of the 2004 ACM-SIGMOD International Conference on Management of Data (SIGMOD'2004)*, Chicago, IL, pp. 324-333, 2004.
50. M. Zaky. Efficiently mining frequent trees in a forest: Algorithms and Applications. *In IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, Number 8, pp. 1021-1035, 2005.
51. M. Zaky and C.J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. *In Proceedings of the 2003 SIAM International Conference on Data Mining*, Arlington, VA, pp. 457-473, 2002.

52. M. Zaky, S. Parthasarathy, M. Ogihara and W. Li. New Algorithm for fast Discovery of Association Rules. *In Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD'97), AAAI Press, Menlo*, CA, USA, pp. 283-296, 1997.
53. Z. Zeng, J. Wang and L. Zhou. Efficient Mining of Minimal Distinguishing Subgraph Patterns from Graph Databases. *In Proceedings of the 2008 Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'08)*, pp: 1062-1068, 2008.
54. Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-Core coherent closed quasi-clique mining from large dense graph databases. *ACM transactions on database systems*, Vol. 32, Number 2, Article 13, 40 pages, 2007.
55. F. Zhu, X. Yan, J. Han and P.S. Yu. gPrune: A Constraint Pushing Framework for Graph Pattern Mining. *In Proceedings of the 2007 Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'07)*, pp. 388-400, 2007.

RT_004, Octubre 2008
Aprobado por el Consejo Científico CENATAV

Editor: Lic. Margarita Ilisástigui Avilés
Diseño de Portada: DCG Matilde Galindo Sánchez
RNPS No. 2143
ISSN 2072-6260
Indicaciones para los Autores:
Seguir la plantilla que aparece en www.cenatav.co.cu
CENATAV
7ma. No. 21812 e/218 y 222, Rpto. Siboney, Playa;
Ciudad de La Habana. Cuba. C.P. 12200
Impreso en Cuba