CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA, BAJA CALIFORNIA

PROGRAMA DE POSGRADO EN CIENCIAS EN CIENCIAS DE LA COMPUTACIÓN

Algoritmo para encontrar el conjunto independiente fuerte máximo sobre un grafo tipo cactus

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

Alejandro Flores Lamas

Ensenada, Baja California, México

2014

Tesis defendida por

Alejandro Flores Lamas

y aprobada por el siguiente comité

Dr. José Alberto Fernández Zepeda.
Director del Comité

Dr. Carlos Alberto Brizuela Rodríguez.
Miembro del Comité

Dr. Raúl Rangel Rojo.
Miembro del Comité

Dra. Ana Isabel Martínez García
Coordinador del Programa de Posgrado en Ciencias de la Computación

Dr. Jesús Favela Vara
Director de Estudios de Posgrado

Octubre, 2014

Resumen de la tesis que presenta Alejandro Flores Lamas como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Algoritmo para encontrar el conjunto independiente fuerte máximo sobre un grafo tipo cactus

Resumen elaborado por:

Alejandro Flores Lamas

Esta tesis analiza el problema del conjunto independiente fuerte (CIF) máximo sobre un grafo cactus $K = (V_K, E_K)$. El CIF es un problema bien conocido en el área de teoría de grafos y algoritmos, que se define como subconjunto de vértices del grafo tal que para todo par $u, v \in$ CIF, la trayectoria más corta entre ellos es de al menos tres aristas. La solución a esta problemática tiene diversas aplicaciones entre las que se incluyen el estudio de moléculas y átomos, modelado de redes, planificación de rutas, de operaciones y asignación de instalaciones. En el área de teoría de juegos, existen aplicaciones en el campo económico, ingenieril e incluso bélico. Hasta el momento, se desconoce la existencia de algún algoritmo para el grafo cactus o para el grafo general capaz de resolver el CIF máximo en tiempo polinomial y las propuestas que existen únicamente encuentran conjuntos máximos en topologías muy restringidas o conjuntos maximales en grafos más complejos. En este trabajo, se resuelve el CIF máximo sobre grafos tipo cactus mediante un algoritmo (CIF-MAXIMUM-CACTUS) que descompone el grafo en componentes biconectados y que mediante un recorrido sobre el grafo e información acerca de los componentes visitados y vértices previamente marcados, encuentra el CIF en cada componente biconectado. La unión de los vértices marcados produce un CIF máximo sobre el grafo original. La complejidad computacional del CIF-MAXIMUM-CACTUS es de $O(n^2)$ unidades de tiempo, donde $|V_K| = n$. El algoritmo se codificó en lenguaje JAVA versión 1.8.0_05 y se alimentó con dos conjuntos de prueba: el primero con grafos simples cuyo CIF máximo es conocido o fácilmente identificable; el segundo con 120 grafos (árboles, cadenas de ciclos, sistemas de engranes y cactus) generados de forma aleatoria. El análisis formal del algoritmo, así como los resultados obtenidos mediante la implementación y evaluación de los casos de prueba, demuestran la corrección del algoritmo propuesto.

Palabras Clave: **Conjunto independiente fuerte, grafo cactus, conjunto independiente.**

Abstract of the thesis presented by Alejandro Flores Lamas as a partial requirement to obtain the Master of Science degree in Master in Sciences in Computer Science.

Algorithm to find the maximum 2-Packing set in a cactus graph

Abstract by:

Alejandro Flores Lamas

In this thesis we analyze the problem of the maximum 2-packing set in a cactus graph $K = (V_K, E_K)$. The 2-packing set is a well known problem in the fields of graph theory and algorithms. In a graph $G = (V, E)$, a subset $S \subseteq V$ is a 2-packing if for every pair $u, v \in S$ the shortest path between them is at least three edges long. The solution to this problem results in a wide range of applications such as the study of molecules and atoms, network modeling, route planning and operations, and allocation of facilities. Also, there are several applications in game theory, ranging from economics, engineering and warfare. Hitherto, to the best of the author knowledge, there is no algorithm for the cactus nor do exists for the general graph to solve the maximum 2-packing set in polynomial time; other proposals find maximum sets in restricted graph topologies or maximal sets in more complex graphs. In this study, we propose an approach to solve the maximum 2-packing set problem in cactus graphs using an algorithm (CIF-MAXIMUM-CACTUS) which decomposes the graph in biconnected components and finds its 2-packing set following a path through the graph and applying previously gathered information about visited components and marked vertices. The union of marked vertices produces a maximum 2-packing set in the original graph. The running time of the algorithm is $O(n^2)$ time units, where $|V_K| = n$. The CIF-MAXIMUM-CACTUS was implemented in JAVA 1.8.0_05 and two test sets were employed to evaluate its performance: the former includes simple graphs which its maximum 2-packing set is known or easily identified; the other consists of 120 randomly generated graphs (trees, cycle chains, cycle arrangements and cactus). The formal proof of the algorithm, its implementation and results in the test sets show the algorithm's correctness.

Keywords: **2-packing set, cactus graph, independent set.**

Dedicatoria

A mis padres ...

Agradecimientos

A Dios: por darme la vida y por todos estos años, por permitirme a mis padres, por los sueños que me has brindado, por todo aquello que me has dejado ver, ser y hacer.

A mis padres: por apoyarme en los momentos difíciles y celebrar conmigo los momentos de felicidad, porque gracias a su apoyo y consejos, hoy alcanzo otra más de mis grandes metas, el grado de Maestro. Por la formación que me han dado, por su ternura y amor. Gracias por enseñarme a perseverar.

A mi director de tesis: Dr. José Alberto Fernández Zepeda, gracias por guiarme pacientemente durante el desarrollo de este trabajo.

A mis maestros y comité de tesis: mi profundo y sincero respeto por haber compartido desinteresadamente a lo largo de estos años sus conocimientos y por su capacidad de transmitirlos, por sus observaciones y comentarios para mejorar mi investigación.

A mis amigos y compañeros: nada de esto hubiera sido así tan especial sin su apoyo.

Al Centro de Investigación Científica y de Educación Superior de Ensenada: por su apoyo durante mi estancia en esta institución y por la enseñanza académica.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT): por brindarme el apoyo económico para realizar mis estudios de maestría.

A todos ustedes, gracias.

Alejandro Flores Lamas.

Tabla de contenido

Página

Tabla de contenido (continuación)

Lista de figuras

Figura Página

Lista de figuras (continuación)

Lista de tablas

Capítulo 1. Preliminares

El presente trabajo de tesis cae dentro del ámbito de diseño y análisis de algoritmos para grafos. Un *algoritmo* es una herramienta para resolver problemas computacionales bien definidos. Se dice que un algoritmo es un procedimiento ordenado que se puede aplicar a cualquiera de ciertas clases de entradas simbólicas, las cuales eventualmente convergen produciendo salidas simbólicas (Rogers, 1987). Una definición alternativa se encuentra en (Cormen *et al.*, 2009) y describe al algoritmo como un proceso computacional bien definido que recibe como entrada un conjunto de valores y produce un valor o conjunto de valores como salida o respuesta. Asimismo, este trabajo se relaciona fuertemente con el área de *teoría de grafos* que se encarga del estudio de las estructuras matemáticas empleadas para modelar relaciones entre objetos mediante *grafos*. Formalmente, un grafo G es una representación matemática compuesta por el par (V, E), donde V es un conjunto finito de vértices de G, y E es el conjunto de aristas que describen una relación binaria sobre V (Cormen *et al.*, 2009). Los vértices normalmente se representan por pequeños círculos o puntos, mientras que a las aristas por arcos o líneas que unen a los vértices. Existen diversos tipos de grafos, entre ellos se encuentran grafos simples, completos, bipartitos, planos, conexos, ponderados, densos, cíclicos, entre muchos otros más.

En particular, esta tesis presta atención al grafo *cactus*. Un grafo $K = (V_K, E_K)$ es un cactus si cada arista $e \in E_K$ pertenece a lo más a un ciclo en K; equivalentemente, cualquier par de ciclos adyacentes comparten a lo más un vértice en común. La Figura 1 muestra un grafo cactus con 2 componentes conectados. Un grafo $G = (V, E)$ es un grafo *planar* si es posible dibujarlo de tal forma que cualquier par de aristas se crucen únicamente en los vértices. Los cactus son una subclase del grafo *outerplanar*, el cual se define como un grafo $G = (V, E)$ que es planar y cuyos vértices recaen en la periferia del grafo (Trejo-Sánchez y Fernández-Zepeda, 2014). La Figura 2a muestra un grafo outerplanar que no es cactus puesto que la arista $\{a - c\}$ pertenece a dos ciclos. La Figura 2b es un grafo que no es outerplanar ya que el grafo no se puede redibujar de tal forma que los vértices queden en la periferia sin que se crucen en las aristas.

Uno de los objetivos de la teoría de grafos es el desarrollo de algoritmos para resolver

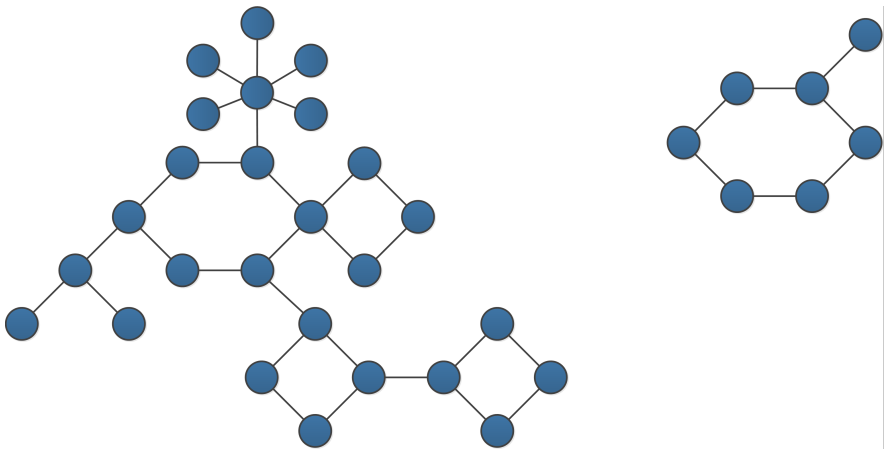

Figura 1: La imagen de la izquierda representa un componente conectado, mientras que el grafo de la derecha es el segundo componentes.

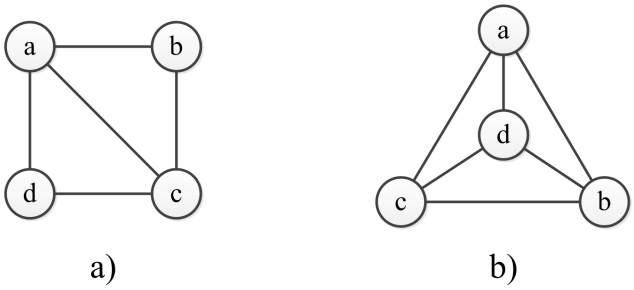

Figura 2: a) Ejemplo de grafo outerplanar. b) Ejemplo de un grafo no outerplanar.

problemas modelados en la forma de grafos; por ejemplo, búsquedas en anchura y profundidad, ordenamiento topológico, arboles de esparcimiento, rutas más cortas entre uno y múltiples puntos, recorridos, caminatas, búsqueda de ciclos, coloreo de grafos, marcado de vértices, entre otros.

Dentro de los problemas de marcado de vértices se encuentra la búsqueda de conjuntos. Dado un grafo $G = (V, E)$ un marcado de vértices $I \subseteq G$ es un *conjunto independiente* si ningún par de vértices arbitrarios u, v que pertenecen a I son vecinos (Haynes *et al.*, 1998; Diestel, 2005). Otra definición de este concepto se encuentra en (Valiente, 2002), el cual define al conjunto independiente como un conjunto de vértices $I \subseteq V$, tal que el subgrafo de G inducido por I no tiene aristas, esto es, $\{u - v\} \notin E$ para todos los vértices distintos $u, v \in I$. La Figura 3 muestra un conjunto independiente sobre G, los vértices marcados u obscuros pertenecen al conjunto.

Frecuentemente, al trabajar con conjuntos es necesario medir su tamaño. De aquí que un conjunto R de vértices sea *maximal* si no existe otro subconjunto R' tal que $R \subset R'$,

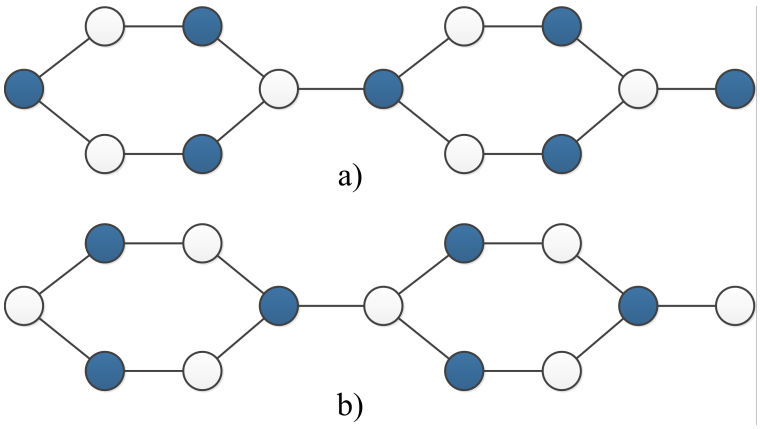

Figura 3: Conjuntos independientes, los vértices marcados pertenecen al conjunto. a) Conjunto máximo. b) Conjunto maximal.

ver la Figura 3a. Un conjunto es *máximo*, si dentro de los conjuntos maximales éste es el de cardinalidad máxima (ver la Figura 3b). Sobre un grafo $G = (V, E)$, el problema del conjunto independiente máximo significa marcar la mayor cantidad de vértices que legalmente puedan existir en el grafo, lo cual es un problema NP-Difícil para topologías de grafo general (Johnson, 1985).

Otro problema de marcado de vértices sobre un grafo es el conjunto *k-packing* que consiste en encontrar aquellos vértices del grafo tal que la trayectoria más corta entre cualquier par de vértices seleccionados sea mayor que k, donde k es un entero positivo y la *distancia* entre un par de vértices se mide como el número mínimo de aristas que los separa (Mjelde, 2004). La Figura 4 muestra un conjunto 3-packing. Al conjunto independiente también se le conoce como 1-packing (Meir y Moon, 1975). Un caso especial del problema k-packing es cuando $k = 2$. Este problema también se conoce como el *conjunto independiente fuerte* (CIF). De esta forma, un CIF es un subconjunto S de vértices, $S \subseteq V$, tal que la longitud del camino más corto entre cualquier par de vértices $u, v \in S$ es mayor que 2 (Gairing *et al.*, 2004).

Los autores en (Castro *et al.*, 2011) emplean la notación $\rho(G)$ para denotar la cardinalidad del CIF máximo sobre el grafo. La Figura 5a presenta un CIF maximal sobre un grafo cactus, mientras que la Figura 5b un CIF máximo. Encontrar un CIF máximo en tiempo polinomial sobre un grafo general es al menos tan complejo como encontrar el conjunto independiente máximo, que es un problema NP-Difícil para grafos en general.

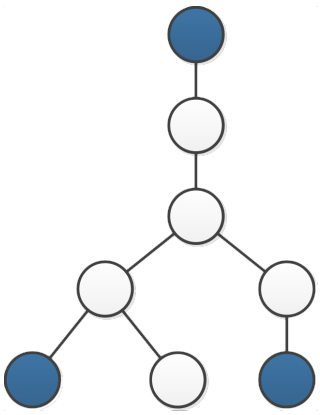

Figura 4: Los vértices marcados sobre el grafo pertenecen al conjunto 3-packing.

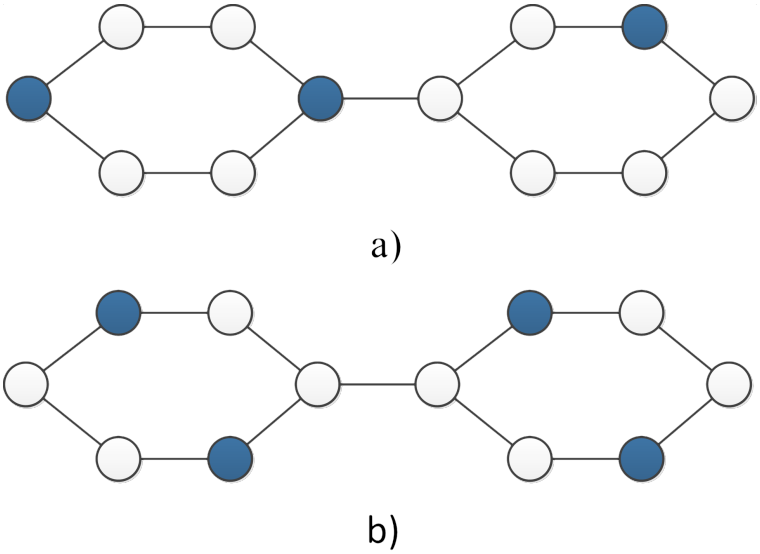

Figura 5: a) CIF maximal. b) CIF máximo.

El *conjunto dominante* es otro problema de marcado de vértices (relacionado al anterior) que se formula de la siguiente manera: dado un grafo $G = (V, E)$, se dice que un conjunto $D \subset V$ es dominante si cada vértice que no está en D es adyacente al menos a un vértice que sí está en D (ver la Figura 6). Al conjunto dominante de cardinalidad mínima se le denota por $\gamma(G)$, mientras que al de cardinalidad máxima como $\Gamma(G)$ (Chellali *et al.*, 2012). Es bien conocido que para cualquier grafo $G = (V, E)$, $\gamma(G) \geq \rho(G)$ (Imrich *et al.*, 2008) (ver la Figura 7). Asimismo, Haynes *et al.* (1998) demostraron que el problema de encontrar el conjunto dominante mínimo es el problema dual del CIF máximo en cuanto a programación lineal se refiere.

Generalmente, para resolver este tipo de problemas sobre un grafo $G = (V, E)$ es necesario conocer el *vecindario* de los vértices. El vecindario de un vértice $v \in V$ son todos

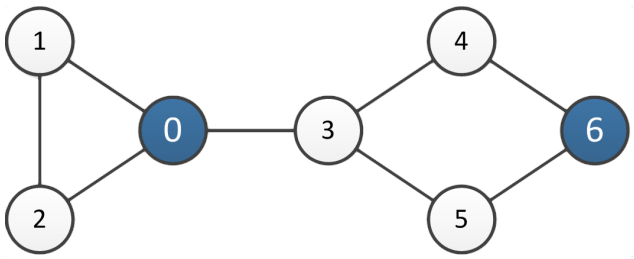

Figura 6: Los vértices 1,2 y 3 son adyacentes al vértice 0, mientras que los vértices 4 y 5 lo son al vértice 6.

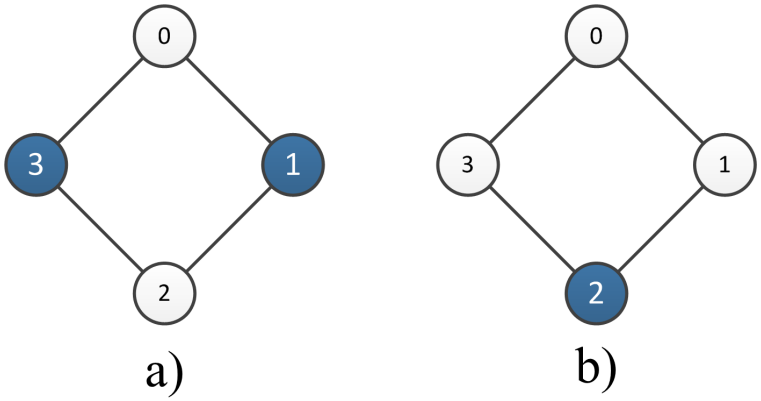

Figura 7: Conjunto dominante mínimo y CIF máximo. a) Conjunto dominante mínimo, b) CIF máximo. En este caso, la cardinalidad del conjunto dominante mínimo es mayor que la cardinalidad del CIF.

los vértices que son adyacentes a v, es decir, aquellos que se encuentran conectados a v por medio de una arista (Chellali *et al.*, 2012). Al vecindario de v se le denota como $N(v)$; por su parte, a la unión de $N(v) \cup v = N[v]$ se le conoce como *vecindario cerrado* de v.

Un vértice de *articulación* o de *corte* en un grafo conectado (ver Figura 8) es aquel vértice que al eliminarse del grafo, junto con todas sus aristas vecinas, parte el grafo en dos o más componentes conectados. Un *componente biconectado* es un grafo maximal que no tiene vértices de articulación. En el caso del grafo cactus, los componentes biconectados pueden tomar la forma de un conjunto de vértices en un ciclo, o de un "clique"de tamaño 2. Un *clique* en un grafo no dirigido $G = (V, E)$ es un conjunto de vértices $C \subseteq V$ tal que todo par $u, v \in C$ se encuentra conectado por una arista (ver Figura 9). Al descomponer el grafo en componentes biconectados es posible generar a partir de éstos, un *árbol de componentes biconectados* (T_B), en el cual los vértices son los componentes biconectados obtenidos de G y donde existe una arista entre aquellos componentes que comparte un vértice de articulación. En (Reingold *et al.*, 1977) se emplea la noción de separar un grafo por componentes biconectados; no obstante, los autores le llaman

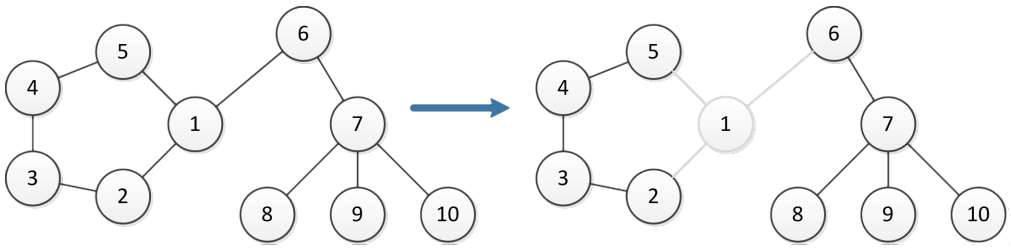

Figura 8: El vértice 1 es un vértice de articulación puesto que descompone el grafo en 2 o más componentes conectados; asimismo lo son los vértices 6 y 7.

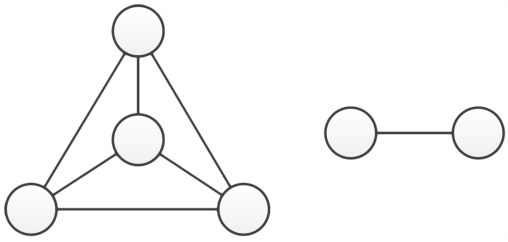

Figura 9: Clique de tamaño 4 (izquierda) y clique de tamaño 2 (derecha).

bloques. Asimismo, Hedetniemi *et al.* (1986) proponen la descomposición del cactus en bloques para encontrar conjuntos dominantes. Posteriormente, Das (2012) utiliza nuevamente este concepto para construir el árbol T_{BC} para calcular el conjunto independiente sobre grafos tipo cactus.

1.1. Conceptos básicos de algoritmos distribuidos

Como se mencionó al inicio de este documento, un algoritmo es un proceso sistemático para resolver problemas específicos mediante una secuencia finita no ambigua de reglas, donde se garantiza que en un número finito de pasos el procedimiento termina (Blass y Gurevich, 2003). En la misma forma en que existen diversos enfoques para atacar un problema, también hay múltiples tipos de algoritmos que se pueden clasificar de acuerdo a cómo ejecutan los pasos involucrados en ellos. Por ejemplo, un algoritmo *secuencial* realiza una operación a la vez, mientras que los algoritmos *paralelos* o *distribuidos* pueden llevar a cabo varias operaciones de forma simultánea (Berman y Paul, 2005).

En este trabajo, se tiene especial interés en el estudio y desarrollo de algoritmos secuenciales para resolver problemas relacionados con el área de teoría de grafos; no obstante, algunos de los mejores algoritmos que se encuentran en la literatura para resolver este tipo de problemas son algoritmos distribuidos. Es por ello que en esta sección se incluyen conceptos adicionales para su comprensión.

De forma general, un algoritmo distribuido se implementa sobre una colección de procesadores, posiblemente heterogéneos, conectados entre sí por medio de algún tipo de red. En este modelo no existe un control central ya que cada procesador o vértice en la red tiene sus propios recursos tanto de hardware como de software, y éstos realizan cálculos de forma independiente a los demás, y donde la sincronización está involucrada en varios niveles (Berman y Paul, 2005; Trejo-Sánchez y Fernández-Zepeda, 2012).

Dentro del cómputo distribuido, un enfoque innovador es la *auto-estabilización*. La auto-estabilización es una propiedad relacionada con la tolerancia a *fallas transitorias* en sistemas distribuidos. Las fallas transitorias son aquellas que generan alteraciones lógicas en un procesador y una vez que se corrigen, el procesador continúa funcionando normalmente. Dijkstra (1974) define a un sistema auto-estabilizante como aquel que, iniciando en un estado arbitrario, garantiza su convergencia en un número finito de pasos. La tolerancia a fallas de los algoritmos auto-estabilizantes es una propiedad muy deseable en sistemas distribuidos donde pueden ocurrir fallas transitorias, de tal manera que le permita al sistema recuperarse y retomar un estado legítimo sin intervención externa (Johnen y Beauquier, 1995).

Shukla *et al.* (1994) consideran tres modelos diferentes para seleccionar cuáles procesadores se ejecutarán en cierto instante, el modelo de calendarizador *central, máximo paralelismo* y *paralelismo restringido*. Con el calendarizador central sólo un procesador puede ejecutarse a la vez; mientras que en el calendarizador de máximo paralelismo, todos los procesadores habilitados pueden ejecutar tareas en paralelo. En el modelo de paralelismo restringido, cuando 2 o más procesadores se encuentran privilegiados, sólo un subconjunto de ellos se selecciona para ejecutar tareas. Además, se dice que un calendarizador es *adversario* cuando éste selecciona los vértices de tal forma que maximiza el tiempo de ejecución del algoritmo que se está ejecutando.

1.2. Visión general del problema del CIF máximo sobre el grafo cactus

En esta tesis se desarrolla un algoritmo secuencial para encontrar un CIF de cardinalidad máxima sobre grafos tipo cactus en tiempo polinomial. En el trabajo de Butenko (2003) se pueden encontrar diversas aplicaciones para conjuntos independientes en áreas co-

mo recuperación de información, teoría de clasificación, economía, planificación, diseño experimental, visión por computadora, ingeniería biomédica, entre otras.

Asimismo, Abello *et al.* (1999) analizan el *grafo de llamadas* que es una representación de un grafo masivo en telecomunicaciones mediante el clique máximo. En esta representación, los vértices son números telefónicos y las aristas unen pares de vértices al realizarse una llamada telefónica entre ellos. En cuanto al modelado de mercados mediante grafos, existe trabajo por parte de Michael y Battiston (2009), donde se estudian las relaciones de agentes económicos en un grafo. De forma particular, Butenko (2003) también presenta una aplicación para el conjunto independiente y clique, donde los mercados se representan por vértices y donde las aristas describen relaciones entre los mercados cuyos precios se encuentran en fluctuación. El mismo autor estudia el problema de construir eficientemente una columna vertebral para una red de sensores *ad-hoc* mediante la solución al problema del conjunto dominante.

Algunos de los problemas modelados por el grafo cactus son: la ubicación de instalaciones (Manne y Mjelde, 2006), designación de servidores en una red, asignación de diferentes frecuencias a estaciones transmisoras de radio para eliminar o minimizar interferencia (Hale, 1980), manejo de materiales en una red al usar vehículos automatizados (Kariv y Hakimi, 1979) y en el modelado de rutas perimetrales de líneas telefónicas para conectar a los usuarios hacia la columna vertebral principal (Koontz, 1980); de forma general, se aplica en situaciones que involucran la distribución de recursos evitando el traslape entre las asignaciones y a la vez cubriendo completamente la red. Además, Paten *et al.* (2010) presentan al grafo cactus como una estructura de datos para comparar conjuntos de genomas relacionados. Siendo el cactus una representación superior a otras para la comparación de genomas, puesto que soporta duplicidad y es capaz de descomponer subestructuras comunes en cadenas jerárquicas y redes.

Como ya se ha mostrado, tanto el modelado y estudio del grafo catus y de los conjuntos independientes tienen diversas aplicaciones para la resolución de problemas. La importancia del CIF recae precisamente sobre el estudio de los conjuntos independientes, puesto que es la continuación del análisis de los mismos. En lo referente al valor teórico, los grafos tipo cactus han atraído la atención de la comunidad científica al descubrirse que

algunos problemas NP-Difíciles para topologías arbitrarias admiten soluciones en tiempo polinomial sobre esta configuración (Ben-Moshe *et al*., 2005; Zmazek y Žerovnik, 2004).

En este documento se desarrolla un algoritmo secuencial para encontrar el CIF máximo sobre grafos cactus en tiempo polinomial, bajo la suposición de que todos los grafos a evaluar están formados por un sólo componente conectado; de no ser así, se procede a evaluar cada componente por separado. Para alcanzar este objetivo, se adapta la estrategia de descomposición por componentes biconectados presentada tanto por Hedetniemi *et al.* (1986) como por Das (2012), además de tomar elementos de la metodología de Mjelde (2004).

1.3. Organización de la tesis

Ya se han presentado varios conceptos que forman el sustento de esta tesis y sobre los cuales se construye en capítulos posteriores. En cuanto a la estructura del resto de este trabajo, los capítulos se organizan de la siguiente forma.

- Capítulo II: Se introduce el marco teórico asociado al problema del CIF sobre topologías de grafos como anillo, árbol, cactus y grafo general. Lo anterior, mediante la exposición de los algoritmos que se encuentran en el estado del arte para encontrar conjuntos de cardinalidad maximal y máxima para problemas similares al conjunto independiente. Asimismo, se presenta un algoritmo para calcular el conjunto dominante mínimo sobre un grafo cactus. Tanto las metodologías como los algoritmos que aquí se presentan difieren entre sí, varían desde programación dinámica hasta modelos extendidos del vecindario de un vértice. Adicionalmente, los algoritmos pueden ser secuenciales o auto-estabilizantes.

- Capítulo III: Se describe el algoritmo de tiempo polinomial desarrollado para encontrar el CIF máximo sobre grafos cactus. Primero se presenta la idea general detrás del algoritmo, luego se explica de forma más detallada sus componentes. Posteriormente, se incluyen los pseudocódigos; el capítulo finaliza con la formalización del algoritmo y la incorporación del análisis del tiempo de ejecución.

- Capítulo IV: En este capítulo se presentan detalles adicionales de implementación

del algoritmo descrito en el Capítulo III y de la experimentación realizada. Se explican los tipos de grafos sobre los cuales se evaluó el algoritmo y se comparan los resultados obtenidos, en el caso del grafo tipo árbol, contra aquellos obtenidos por otros algoritmos en la literatura. Además, en la sección final de este capítulo se discuten los desafíos encontrados para la realización del algoritmo y el cómo se solucionaron.

- Capítulo V: Finalmente se presentan las conclusiones y aportaciones de este trabajo de investigación. Asimismo, se discuten brevemente algunos problemas no resueltos que existen y las posibles rutas que puede tomar el trabajo futuro sobre esta línea de investigación.

Capítulo 2. Marco teórico

2.1. Introducción

El objetivo de este capítulo es explicar brevemente los algoritmos existentes para resolver problemas similares al conjunto independiente sobre ciertas tolopogías de grafos como anillo, árbol, cactus, grafo outerplanar y grafo general. Dichos algoritmos y literatura que los acompaña se toman como bases para esta investigación. A pesar de que algunos de estos algoritmos son capaces de encontrar conjuntos de cardinalidad máxima en algunas topologías de grafos, hasta la fecha, el autor de este documento desconoce la existencia de algún algoritmo que pueda calcular el CIF máximo en tiempo polinomial sobre grafos cactus. En la parte final de este capítulo, las tablas 1 y 2 muestran la información condensada de los algoritmos descritos.

2.2. Algoritmos para encontrar conjuntos máximos y mínimos

En esta sección se discuten algunos algoritmos para encontrar conjuntos de cardinalidad máxima para los problemas del conjunto independiente, 2-conjunto independiente, CIF, y conjunto k-packing. También se expone un algoritmo para encontrar el conjunto dominante mínimo. Este último algoritmo es importante porque sienta algunas bases para el algoritmo del CIF desarrollado en esta tesis.

2.2.1. Conjunto independiente y 2-conjunto independiente sobre cactus

El problema del conjunto *2-independiente* máximo busca determinar sobre un grafo G dos conjuntos independientes disjuntos $I_1,\ I_2$ tal que la cardinalidad del conjunto $I_1 \cup I_2$ sea máxima. La generalización de este problema recibe el nombre de *máximo conjunto k-independiente* (MKIS) cuyo objetivo es encontrar k conjuntos independientes disjuntos tal que $\cup_{i=1}^{k} I_i$ sea máxima. Tanto el 2-conjunto independiente (Sarrafzadeh y Lee, 1989) como el MKIS (Yannakakis y Gavril, 1987) son problemas NP-Difícil para topologías de grafos en general.

Para encontrar el 2-conjunto independiente sobre el grafo cactus, Das (2012) propone el siguiente procedimiento: obtener los componentes biconectados y vértices de corte del

cactus. Como siguiente paso, etiquetar como S_O aquellos componentes biconectados con número impar de vértices y como S_E a los que contienen un número par. Para los componentes en S_O se sigue una técnica de borrado para aquellos vértices que no sean de corte. Posteriormente, de forma secuencial se etiquetan los vértices restantes como R y M de tal manera que dos vértices consecutivos no tengan la misma etiqueta. Finalmente, los vértices en R forman el primer conjunto independiente, denotado por S_1 y los vértices en M el segundo conjunto, S_2. Así se obtienen dos conjuntos independientes disjuntos, donde el conjunto 2-independiente máximo es $S_1 \cup S_2$. El tiempo de ejecución de este algoritmo es O(n) unidades de tiempo (u. t.), donde n es el número de vértices. Para encontrar un sólo conjunto independiente máximo, se sigue un procedimiento similar, pero en esta ocasión al etiquetar los vértices alternadamente se selecciona aquel conjunto que sea mayor por componente, no importando si son pares o impares siempre y cuando la selección no afecte a otros vértices.

2.2.2. Conjunto independiente sobre grafo árbol

Un conjunto independiente I en un grafo tipo árbol se puede definir como aquella selección de vértices tal que si un vértice u se marca para pertenecer al conjunto, entonces ningún vértice hijo de u pertenece a I, pero sí los nietos de u. De forma similar, si un vértice w es marcado para formar parte de I, entonces su padre no puede ser marcado para el conjunto (Valiente, 2002). El *número de independencia* de un grafo $G = (V, E)$, denotado por $\beta(G)$, indica la cardinalidad del conjunto independiente máximo que se obtiene de G (Valiente, 2002; Chellali *et al.*, 2012).

Como se mencionó, para encontrar el conjunto independiente en grafos tipo árbol existen dos posibilidades, marcar tanto el vértice u y sus nietos o marcar únicamente a los hijos de u. Esto se puede expresar mediante la ecuación 1, donde $I(u)$ es el tamaño del conjunto independiente máximo del subárbol enraizado en el vértice u.

$$I(u) = \max \left\{ \sum_{v \in hijos(u)} I(v), 1 + \sum_{w \in nietos(u)} I(w) \right\} \tag{1}$$

Para evitar la evaluación exponencial de cada vértice que normalmente aparece en la

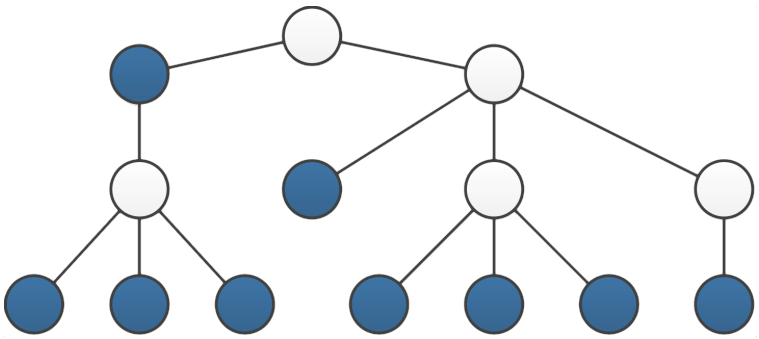

Figura 10: Conjunto independiente máximo en un árbol.

programación dinámica, se puede evaluar el árbol mediante un recorrido postorden. Por lo que el tiempo de ejecución de este algoritmo es $O(n)$, ya que cada arista $u - v \in E$ es examinada a lo más 2 veces durante el recorrido. La Figura 10 muestra una selección posible de vértices para el conjunto independiente máximo sobre el grafo.

2.2.3. Conjunto k-packing sobre grafo árbol

El algoritmo secuencial propuesto por (Mjelde, 2004) se basa en la metodología de programación dinámica y se divide en dos fases. La idea general de la primera fase es comenzar en las hojas del árbol, realizar una serie de cálculos e ir avanzando hacia la raíz. Durante esta fase, se hace distinción entre los vértices del árbol y se les clasifica en: hojas, vértices intermedios y raíz. Al finalizar los cálculos de la primera fase, cada subárbol enraizado en el vértice v conoce el tamaño de la solución óptima, pero no identifica cuáles son los vértices marcados para el conjunto.

La segunda fase comienza en la raíz y se desplaza hacia las hojas. Durante esta fase, se usa la información obtenida en la fase anterior para detallar la solución encontrada y así marcar los vértices del conjunto.

Cada vértice del árbol mantiene una tabla con información de tamaño $k + 1$; para llenar las celdas de cada tabla se necesitan $O(C_v)$ u. t., donde C_v es el número de hijos del vértice v. Por lo tanto, para cada vértice v se utiliza un total de $O(kC_v)$ u. t., puesto que el número total de hijos en el grafo es $n - 1$ (el vértice raíz no es un hijo) el tiempo de ejecución total del algoritmo es $O(kn + n) = O(kn)$ u. t.

Durante la Fase 1, el algoritmo va llenando la tabla que se encuentra asociada a cada

vértice del árbol; las dimensiones de la tabla son k renglones y 4 columnas, donde el índice i de cada renglón indica el número de vértices no marcados que existen debajo del vértice v, incluyéndose a sí mismo. La primera columna de la tabla indica el número de vértices marcados que pueden llegar a existir si se tienen i vértices no marcados debajo de v; el encabezado de la tabla se denota por la letra *M*. La convención que utiliza el autor para indicar cuáles vértices pertenecen al conjunto es como sigue: un vértice está marcado si pertenece al conjunto, apagado en caso contrario. El resto de las columnas de la tabla llevan por encabezado *pd, sd* y *heir* cuyo significado es: distancia de padre, distancia a hermano y heredero, respectivamente. El algoritmo comienza haciendo un recorrido del árbol en postorden para llenar la tabla de cada vértice.

Si el vértice es una hoja, entonces la columna *M* se llena con ceros en todos los renglones, excepto en el índice 0 donde se coloca un 1. Esto significa que la hoja se marca.

Para cualquier otro vértice que no sea hoja, existen tres posibilidades de acuerdo al índice i, éstas son:

- Si el índice $i = k$, entonces para llenar la celda correspondiente en la columna M se utiliza la ecuación 2

$$M_v[k] = \sum_{c \in C_v} M_c[k-1]. \tag{2}$$

- Si el índice i se encuentra en el intervalo: $k - 1 \geq i \geq 1$. En este punto, existe la posibilidad de que un par de vértices hijos de v puedan seleccionarse como vértices marcados pero que se encuentren a una distancia menor que k. Para revisar esta ocurrencia se utiliza la condición 3.

$$\text{if } (i \neq 0 \ \wedge \ 2i - 1 < k). \tag{3}$$

  Si la condición 3 se evalúa verdadera, entonces hay que determinar cuál de los hijos aporta más vértices marcados (a este vértice se le conoce como *heredero*)

al subárbol enraizado en v. Sean u, w los vértices hijos de v, para seleccionar al heredero se hace una revisión de las tablas de los hijos y se guarda en una variable auxiliar $aux1$ el resultado de $M_u[i-1] \ + \ M_w[i]$; posteriormente, en una segunda variable $aux2$ se le asigna $M_u[i] \ + \ M_w[i-1]$. Si $aux1 > aux2$, entonces $heir_v[i] = u$; de lo contrario, el heredero será el vértice v. Este procedimiento se repite para todo $c \in C_v$. Asimismo, el valor de $sd_v[i] = k - (i-1)$. Si la condición 3 es falsa, entonces se puede usar la ecuación 2. La diferencia es que en esta ocasión hay que reemplazar k por el valor adecuado de la posición de la tabla.

- Finalmente, cuando el índice $i = 0$, el valor de $M_v[0]$ es igual a la suma de $M_c[k]$ $\forall c \in C_v$ y a este resultado se le suma una unidad.

A pesar de que ya se ha explicado el llenado de la tabla de cada vértice, hay una opción que no se ha considerado, que una solución obtenida en algún índice $i-1$ no sea tan buena como la solución calculada para un índice i (recuerde que el llenado de la tabla comienza por el índice $i = k$, y decrece hasta llegar a $i = 0$). Por ello hay que realizar una operación adicional que el autor llama *verificación de antecedentes*, en la cual, cada resultado obtenido en $M_v[i]$ se compara contra $M_v[i+1]$. De aquí se obtiene la ecuación 4 que asigna a $M_v[i]$ la mejor solución encontrada.

$$M_v[i] = \text{max}\{M_v[i], \ M_v[i+1]\}. \tag{4}$$

El último campo por llenar es la distancia de padre $pd_v[i]$, cuyo valor por defecto es $pd_v[i] = i$. Este campo indica la posición de la tabla $M_v[\,]$ en donde se encontró la mejor solución para $M_v[i]$. Es decir, cada vez que la ecuación 4 encuentra una mejor solución para $M_v[i]$ el valor de $pd_v[i]$ cambia. Al finalizar el llenado de la tabla del vértice raíz se tiene toda la información necesaria para iniciar la Fase 2.

El propósito de la Fase 2 es marcar los vértices que formarán parte del conjunto; esto se hace con base en la información obtenida en la Fase 1. Para marcar los vértices, es necesario introducir un nuevo término: la distancia final $final_v$. La distancia final indica la posición en $M_v[\,]$ donde se encuentra el vértice que se marcará. Si $final_v = 0$, entonces

el vértice se marca; por el contrario, si el vértice no debe de marcarse, entonces $final_v$ muestra el número mínimo de vértices no marcados que se encuentran por abajo de v (incluyendo a v) antes del próximo vértice marcado.

La distancia final de cada vértice v puede caer en cualquiera de los siguientes casos:

1. El vértice v es la raíz.

$$final_r = pd_r[0]. \tag{5}$$

2. El vértice v es el heredero o su padre p no tiene heredero.

$$final_v = pd_v[(final_p - 1) mod (k+1)]. \tag{6}$$

3. El vértice v no es el heredero, pero su padre p sí tiene heredero.

$$final_v = pd_v[(sd_p[final_p] - 1) mod (k+1)]. \tag{7}$$

De esta forma finaliza la Fase 2 y también el algoritmo. Es importante notar que mientras el recorrido del árbol en la Fase 1 se realiza en postorden, la Fase 2 emplea un recorrido preorden. El tiempo de ejecución total del algoritmo es $O(kn)$ u.t. Asimismo, existe la versión distribuida de este algoritmo, cuya complejidad es $O(n^3)$ u. t. La Figura 11 muestra un conjunto 3-packing encontrado mediante este algoritmo.

2.2.4. Conjunto dominante mínimo en grafos cactus

Para encontrar el conjunto dominante de cardinalidad mínima en un grafo cactus Hedetniemi *et al.* (1986) proponen el algoritmo *MCACTUS*, el cual se compone de tres rutinas: *MCYCLE, CREATEPATH, y DOMSET*. MCACTUS identifica los componentes biconectados del cactus y los vértices de corte. Posteriormente, hace uso del algoritmo MCYCLE para identificar los vértices que forman parte del conjunto dominante.

La rutina MCYCLE trabaja bajo la suposición que los vértices del grafo se dividen en tres conjuntos *free, bound, required* (F, B, R). Los vértices que se encuentran en R son aquellos marcados para el conjunto dominante, mientras que los vértices de B se

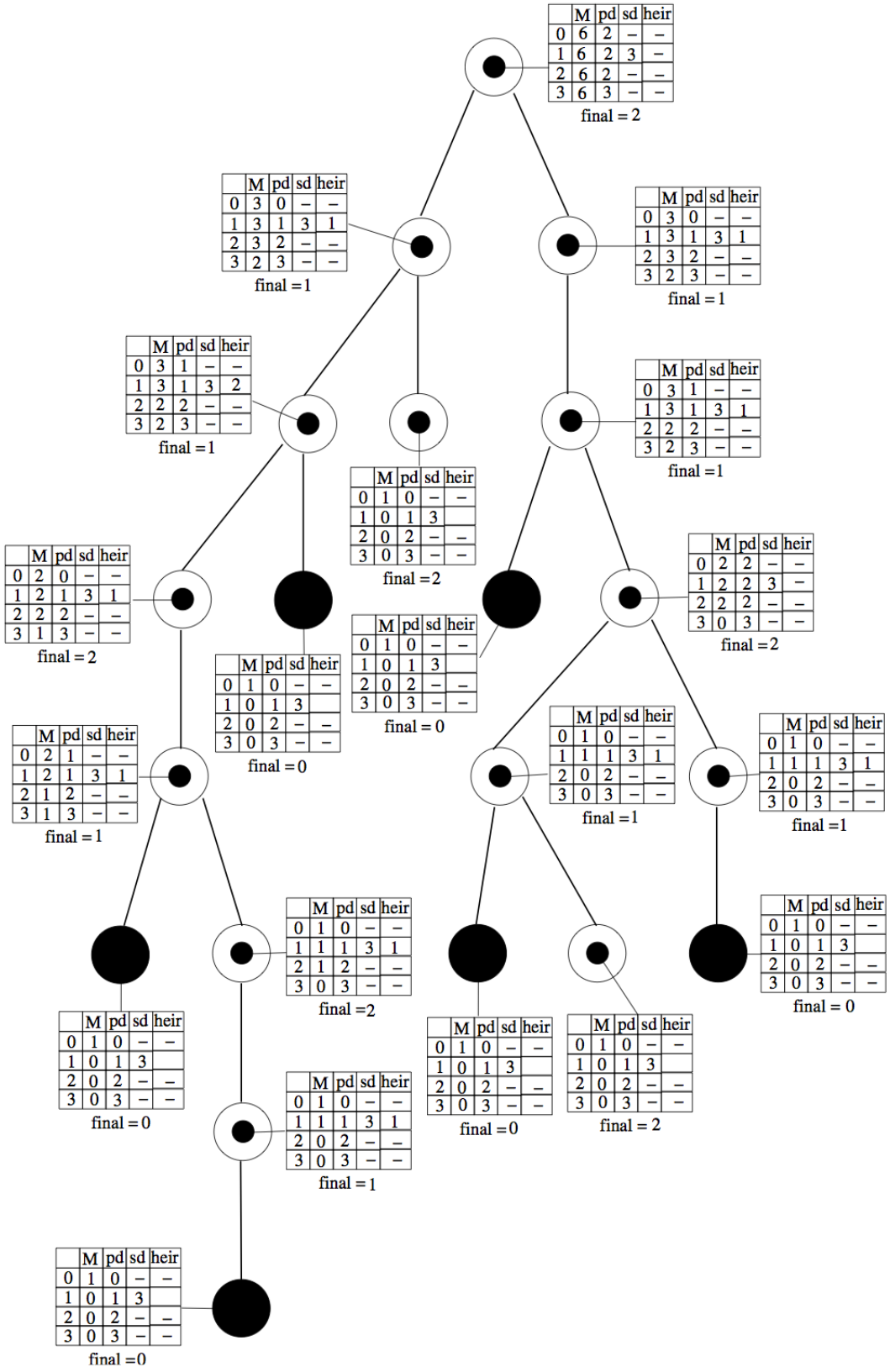

Figura 11: Conjunto 3-Packing sobre un grafo tipo árbol. Imagen tomada del trabajo de Mjelde (2004).

encuentran dominados por los vértices de R. Los vértices libres no se encuentran dominados, pero pueden incluirse en R para dominar a los vértices de B. MCYCLE encuentra el conjunto dominante mixto, es decir encuentra los vértices que se encuentran tanto en R como en B, de un componente biconectado. Lo anterior se lleva a cabo mediante las rutinas DOMSET y CREATEPATH. La primera rutina encuentra el conjunto dominante en un componente no cíclico, mientras que la rutina CREATEPATH recibe un vértice u etiquetado como R y un ciclo C, el vecindario de u se re-etiqueta como F y u se elimina del ciclo. Finalmente, CREATEPATH regresa el camino generado al borrar al vértice u del ciclo.

Para encontrar los componentes biconectados, el autor hace uso del algoritmo 5.3 de (Aho y Hopcroft, 1974) cuya complejidad es de $O(|E|)$, es decir, para el grafo cactus es $O(n)$ u. t. El tiempo de ejecución tanto de CREATEPATH como MCYCLE es lineal. Asimismo, MCACTUS manda llamar a MCYCLE a lo más 3 veces por componente. De aquí que la complejidad total del algoritmo MCACTUS es $O(n)$ unidades de tiempo.

2.3. Algoritmos que encuentran conjuntos maximales

Diversos procedimientos se emplean para encontrar conjuntos de vértices sobre un grafo; estos procedimientos suelen emplear tanto secuencias de pasos bien definidos, como selección voraz, e incluso aleatoriedad. No obstante, el resultado que se obtiene no siempre es el óptimo (debido a la complejidad del problema) y en caso de encontrar el conjunto de mayor tamaño el tiempo de ejecución puede llegar a ser muy elevado. Esta sección presenta algunos de los algoritmos que es posible encontrar en la literatura para calcular conjuntos maximales sobre el grafo.

2.3.1. Algoritmo autoestabilizante para un CIF maximal en cactus

El algoritmo propuesto por Trejo-Sánchez y Fernández-Zepeda (2012) se divide en 2 fases; los autores denominan la primera como elección de líder. Esta fase primero inicializa las variables locales de cada vértice. El propósito de estas variables es construir de forma gradual un árbol de esparcimiento enraizado en cada vértice que contenga la etiqueta identificadora con valor más pequeño. La segunda fase o fase de marcado, asigna un color diferente del conjunto $\{rojo, azul\}$ a cada vértices $v \in V_K$. El coloreo

se realiza partiendo de la raíz del árbol de esparcimiento y avanza asignando el mismo color a los vértices que se encuentren en el mismo nivel. La idea de este coloreo es que los vértices rojos sean el conjunto de vértices S que forman el CIF maximal. Dentro del conjunto de vértices de color azul existen dos tonalidades $azul = \{azul_1, azul_2\}$, que se asignan a cada vértice dependiendo de su cercanía a un vértice rojo. La complejidad del algoritmo es $O(D_K)$ u. t., donde D_K es el diámetro del cactus. Si el grafo es un anillo, el algoritmo encuentra el CIF máximo.

2.3.2. Algoritmo autoestabilizante para encontrar el CIF en un grafo arbitrario

Gairing *et al.* (2004) describe un algoritmo auto-estabilizante para encontrar un CIF maximal sobre un grafo arbitrario; no obstante, el número de movimientos requeridos para alcanzar el objetivo es exponencial. La estrategia que sigue el autor es utilizar punteros para conocer la información del vecindario de un vértice a distancia 2. De esta forma se van construyendo árboles de esparcimiento donde la raíz de cada subárbol son los vértices marcados para el CIF. Un vértice se agrega al conjunto sólo si todos sus vecinos apuntan hacia él. Un vértice $v \in$ CIF se puede remover del conjunto si existe otro vértice u vecino de v tal que u también está en el CIF y tiene menor identificador que v. El calendarizador que emplea este algoritmo es el adversario. Posteriormente, Shi (2012) mejora notablemente este enfoque, reduciendo el sobre costo a $O(mn)$ movimientos bajo cualquier calendarizador y cuya estabilización se alcanza en $O(n^2)$ rondas. El algoritmo está compuesto de 4 reglas : *c-Decrease, Leave, Join, c-Orphan*. La regla *c-Decrease* se encarga de actualizar la distancia de un nodo hacia otros si es que existe un nodo en el vecindario con menor identificador. La regla *Leave* asegura que la distancia entre cada par de nodos en el conjunto sea por lo menos 3; esta regla se acompaña de la regla *c-Orphan* que se encarga de que la raíz de cada subárbol enraizado en un nodo que pertenece al conjunto tenga el identificador mayor del vecindario. Finalmente, la regla *Join* agrega un nodo u al conjunto si el vecindario de u se encuentra por lo menos a distancia 2, lo que indica que u se encuentra a distancia de por lo menos 3 de cualquier otro nodo en el conjunto.

2.3.3. Metodologías de información a distancia k

Los autores en (Goddard *et al.*, 2008) proponen una metodología (modelo extendido) para expandir la información que un vértice puede tener acerca de los vértices en su vecindario, extendiendo así el conocimiento de este vértice hasta aquellos vértices que se encuentren a una distancia k (ver la Figura 12). En esta metodología, cada vértice tiene una variable binaria f, tal que $f(i) = 1$ si el i-ésimo vértice pertenece al conjunto; $f(i) = 0$, en caso contrario. También se emplea la variable σ que se encarga de almacenar una copia de la información a distancia k del vértice. Finalmente, se emplea un apuntador $\longrightarrow$ hacia algún miembro del vecindario cerrado del vértice que se encuentre a lo más a distancia k. El algoritmo se compone de 4 reglas *UPDATE, ASK, RESET* y *CHANGE*. La regla *UPDATE* actualiza y verifica que la información guardada por σ sea correcta, mientras que la regla *ASK* comprueba si un vértice se encuentra privilegiado para realizar cambios o corregir sus valores en σ. Por su parte *RESET* monitorea los punteros de los vértices para que cada uno de ellos apunte hacia el vértice con menor identificador dentro de su vecindario cerrado a distancia k. Finalmente, *CHANGE* le permite que un vértice se agregue al conjunto si todos sus vecinos a distancia k apuntan hacia él y si los valores de σ son correctos. Este algoritmo tiene una complejidad de $O(n^2m)$ movimientos. El Algoritmo 1 encuentra un CIF maximal bajo el modelo extendido.

Algoritmo 1: Algoritmo: 2-Packing suponiendo el modelo extendido

```
1 ENTER: if i ∉ S ∧ S ∩ N²(i) = ∅ then
2 |  enter S;
3 end
4 LEAVE: if i ∈ S ∧ S ∩ N²(i) ≠ ∅ then
5 |  leave S;
6 end
```

Posteriormente, Turau (2012) retoma la transformación anterior (renombrándola como modelo de expresión) y la mejora a tan sólo $O(m)$ movimientos. El autor desarrolla dos algoritmos auto-estabilizantes, uno para el calendarizador central y otro para el calendarizador distribuido. Para el calendarizador central, el algoritmo se llama *Transformador C* y consiste de 5 reglas: *Update, Request, Grant, Execute,* y *Reset*. *Update* garantiza que las variables de las expresiones tengan valores correctos. Un vértice hace uso de la regla *Request* cuando éste quiere realizar algún movimiento; el permiso se otorga mediante la

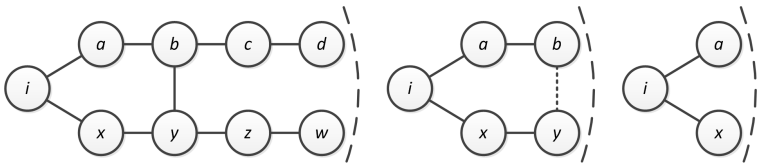

Figura 12: De izquierda a derecha, información a distancia 4, 2, y 1 desde el vértice i. Bajo este modelo, la arista punteada $\langle b, e\rangle \in E$ no es visible a distancia dos desde el vértice i. Imagen tomada de (Goddard *et al.,* 2008).

regla *Grant*, pero únicamente para ejecutar las reglas *Execute* o *Reset*. La regla *Execute* ejecuta cualquier algoritmo que se encuentre dentro se esta regla; conversamente, la regla *Reset* cancela los permisos otorgados al vértice.

Bajo un planificador distribuido, los vértices vecinos no pueden llevar a cabo las reglas *Execute* y *Reset* de forma concurrente puesto que estos nodos se podrían aprobar de forma simultánea; por ello, para este calendarizador, se agregan un par de reglas adicionales *LockR y LockE*, que simplemente verifican si un vértice quiere pedir permiso y si quiere ejecutar alguna acción. Ambas reglas le avisan al resto del vecindario que no ejecuten ninguna acción hasta que el vértice al que se le concedió permiso realice sus tareas. Posteriormente, las reglas *UnlockR y UnlockE* cancelan las peticiones del vértice. El Algoritmo 2 encuentra un conjunto k-dominante minimal bajo el modelo de expresión.

Algoritmo 2: Algoritmo A_1: Conjunto k-dominante (modelo de expresión)

$\varepsilon_A = \{INcount :: |\{w \in N(v) : w.state = IN\}|\}$
R1: **if** $v.state = OUT \wedge v.Incount < k$ **then**
  $v.state \leftarrow IN$;
end
R2: **if** $v.state = IN \wedge v.INcount \geq k \wedge (\forall w \in N(v) : w.state = IN \vee w.INcount > k)$ **then**
  $v.state \leftarrow OUT$;
end

Tanto Goddard *et al.* (2008) como Turau (2012) presentan aplicaciones de la metodología en problemas como k-packing, conjuntos irredundantes y otros conjuntos maximales o minimales. Asimismo, Manne y Mjelde (2006) toman la metodología desarrollada por Goddard *et al.* (2008) para encontrar un CIF maximal donde la complejidad temporal del algoritmo se mantiene, pero la complejidad espacial se reduce.

Asimismo, Turau (2007) presenta un algoritmo auto-estabilizante para encontrar con-

Tabla 1: Algoritmos para conjuntos máximos y mínimos.

Problema	Grafo	*T*(*n*)
Conjunto independiente y 2-conjunto independiente (Das, 2012)	cactus	$O(n)$
Conjunto independiente	árbol	$O(n)$
Conjunto k-packing (Mjelde, 2004)	árbol	$O(kn)$
Conjunto dominante (Hedetniemi *et al.*, 1986)	cactus	$O(n)$
Conjunto k-packing autoestabilizante (Mjelde, 2004)	árbol	$O(n^3)$

juntos independientes maximales y conjuntos dominantes minimales. La estrategia requiere únicamente de un número lineal de movimientos bajo un planificador central. El algoritmo se compone de tres estados *IN, OUT* y *WAIT*. El estado *IN* indica que un vértice se marca para el conjunto. Por su parte, el estado *OUT* etiqueta al vértice que no pertenece al conjunto, y *WAIT* indica que un vértice u podría cambiar hacia el estado *IN* si es que el vecindario de u no tiene otro vértice que también este en *WAIT* y cuyo identificador sea menor.

Tabla 2: Algoritmos para conjuntos maximales y minimales.

Problema	Grafo	*T*(*n*)
CIF auto-estabilizante (Trejo-Sánchez y Fernández-Zepeda, 2012)	cactus	$O(D_k)$
CIF auto-estabilizante (Gairing *et al.*, 2004)	general	exponencial
CIF auto-estabilizante (Shi, 2012)	general	$O(n^2)$
CIF auto-estabilizante, modelo extendido (Goddard *et al.*, 2008)	general	$O(n^2m)$
CIF auto-estabilizante, modelo de expresión (Turau, 2012)	general	$O(m)$
CIF distribuido (Trejo-Sánchez y Fernández-Zepeda, 2014)	outerplanar	$O(n)$
Conjunto k-packing auto-estabilizante (Manne y Mjelde, 2006)	general	$O(n^2m)$
Conjunto independiente auto-estabilizante (Turau, 2007)	general	$O(n)$
Conjunto dominante auto-estabilizante (Turau, 2007)	general	$O(n)$

Capítulo 3. Algoritmo para encontrar el CIF máximo en grafos cactus

3.1. Introducción

En este capítulo se presenta el algoritmo CIF-MAXIMUM-CACTUS, que es el algoritmo de tiempo polinomial desarrollado en este trabajo para encontrar el CIF máximo en un grafo tipo cactus. El capítulo inicia con la descripción de alto nivel de cómo funciona el algoritmo. Posteriormente, se explican más detalles acerca de sus componentes; asimismo, se presentan los pseudocódigos y formalización del mismo. Finalmente, se analiza su tiempo de ejecución.

3.2. Descripción de alto nivel

El algoritmo para encontrar el CIF sobre un cactus $K = (V_K, E_K)$, se descompone en las siguientes partes:

1. Si la cantidad de vértices del grafo es exactamente igual a 1, entonces marque el vértice y termine el proceso. Si la condición anterior no se cumple, continue con los siguientes incisos.
2. Divida el grafo de entrada K en componentes biconectados.
3. Genere un árbol de componentes biconectados T_B, y seleccione cualquier componente como raíz para el árbol. De los vértices que integran el componente raíz, seleccione un vértice raíz.
4. Siga el recorrido postorden sobre T_B y calcule el CIF en cada componente de la siguiente manera:

    a) Si el componente biconectado es una hoja en el árbol, marque la mayor cantidad de vértices posibles de tal forma que no se violen las propiedades del CIF, y donde la distancia más corta del vértice de articulación hacia el vértice marcado más cercano sea lo más grande posible, llame a esta distancia *restricción* y pásela al bloque padre.

b) Si el componente es un vértice intermedio o raíz del árbol T_B, primero evalúe las restricciones que recibe de sus hijos y, posiblemente, de su padre, manteniendo aquella cuya distancia sea menor (restricción de mayor peso).

c) Dentro del proceso de marcado de vértices, evite marcar el vértice de articulación hacia el componente padre; el marcado de dicho vértice depende del componente padre (excepto en la raíz.)

5. Reconstruya la respuesta en K a partir del marcado realizado en T_B.

3.3. Notación empleada

Así como se menciona en la Sección 1, un grafo cactus se denota por $K = (V_K, E_K)$ donde V_K y E_K son el conjunto finito de vértices y aristas de K, respectivamente. El árbol de componentes biconectados se representa por $T_B = (B, E_B)$, los vértices de este árbol son los componentes biconectados del grafo K. Al conjunto de vértices de T_B se les denota por $B = \{B_1, B_2, ...B_i, ..., B_N\}$, donde el i-ésimo elemento denota el orden de aparición del componente i en el recorrido postorden sobre T_B, de aquí que el componente B_N sea el vértice raíz del árbol de componentes biconectados.

Los vértices que integran al componente B_i, cuando éste es un ciclo, se denotan siguiendo una numeración en el sentido de las manecillas del reloj de la siguiente manera: para el vértice $v_{i,j}$, el subíndice i indica que pertenece al componente B_i, mientras que el subíndice j, para $j \geq 0$, indica que es el j-ésimo vértice que se encuentra en el componente B_i. Considere también que cada B_i tiene un único vértice de articulación $v_{i,0}$ hacia su componente padre, y que $v'_{i,j}$ es el vértice marcado más cercano hacia $v_{i,0}$. Además, si B_i es un componente intermedio, entonces $\bar{v}_{i,j}$ denota el vértice de articulación hacia uno de los componentes hijos. Si B_i es un clique de tamaño 2, entonces $v_{i,0}$ es el vértice que tiene una arista hacia su componente padre y $v_{i,1}$ es el vértice restante en el componente. El resto de la notación que se emplea en los componentes cíclicos se conserva para los componentes cliques.

La distancia más corta entre entre un vértice $v_{i,j}$ y el vértice marcado más cercano se denota mediante *dist*($v_{i,j}$), mientras que la distancia más corta entre un par arbitrario

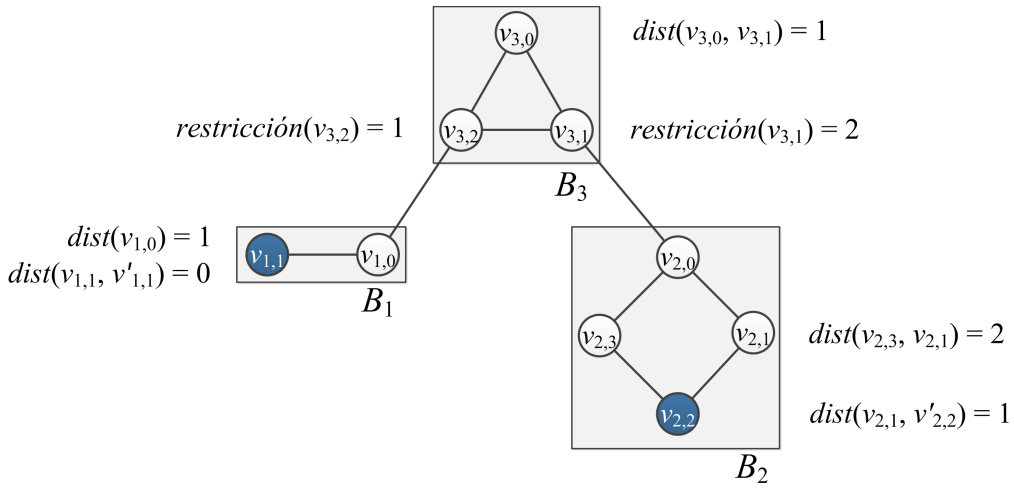

Figura 13: En el árbol T_B que muestra la figura, el conjunto de vértices es: $B = \{B_1, B_2, B_3\}$, mientras que el conjunto de aristas es: $E_B = \{B_1 - B_3, B_2 - B_3\}$. El componente B_3 tiene dos vértices de articulación hacia componentes hijos, $\overline{v}_{3,2}$ que es el vértice $v_{3,2}$ y de forma similar $\overline{v}_{3,1} = v_{3,1}$. Asimismo, para el componente B_3, el conjunto de componentes hijos es $C_{B_3} = \{B_1, B_2\}$.

de vértices $v_{i,j}$ y $v_{i,k}$ se escribe mediante la notación *dist*$(v_{i,j},\ v_{i,k})$ y la restricción que recibe un vértice se denota con: *restricción*$(v_{i,0})$. Finalmente, al conjunto de componentes biconectados hijos de un vértice $B_i \in T_B$ se denota por C_{B_i} (ver la Figura 13).

3.4. Descripción del algoritmo CIF-MAXIMUM-CACTUS

Esta sección describe las partes que integran el algoritmo CIF-MAXIMUM-CACTUS. Cada subsección, iniciando con la descomposición de componentes biconectados hasta la reconstrucción de la respuesta en K, detalla en prosa las operaciones que se llevan a cabo; asimismo, en la subsección de marcado de vértices, se explica el criterio de desempate entre las diversas configuraciones de vértices marcados.

3.4.1. Cálculo de los componentes biconectados de K

La búsqueda de componentes biconectados para el T_B se lleva a cabo mediante 2 procedimientos preliminares: la *búsqueda de ciclos* y *búsqueda de cliques de tamaño 2*.

1. *Búsqueda de ciclos*: se lleva a cabo mediante una búsqueda en profundidad (*DFS*) sobre el grafo K, y haciendo uso de las aristas traseras se encuentran los ciclos en el grafo. Cada ciclo se considera un componente biconectado. Los vértices así como aristas que forman parte del componente se marcan como visitadas en K, esto es

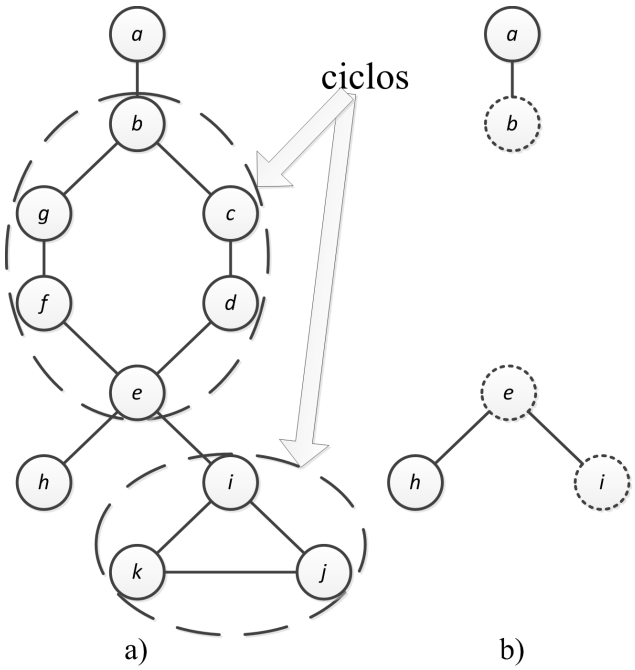

Figura 14: Búsqueda de ciclos. a) Grafo K de entrada. b) grafo K sin ciclos. Los vértices punteados indican que también se encuentran en algún ciclo.

para que un componente biconectado no aparezca en más de una ocasión (ver la Figura 14).

2. *Búsqueda de cliques de tamaño 2*: Al finalizar la búsqueda de ciclos, el grafo K se compone un bosque. Cada arista con sus vértices extremos (clique de tamaño 2) es un componente biconectado. La Figura 15 muestra los cinco componentes biconectados que se desprenden del grafo K de la Figura 14a.

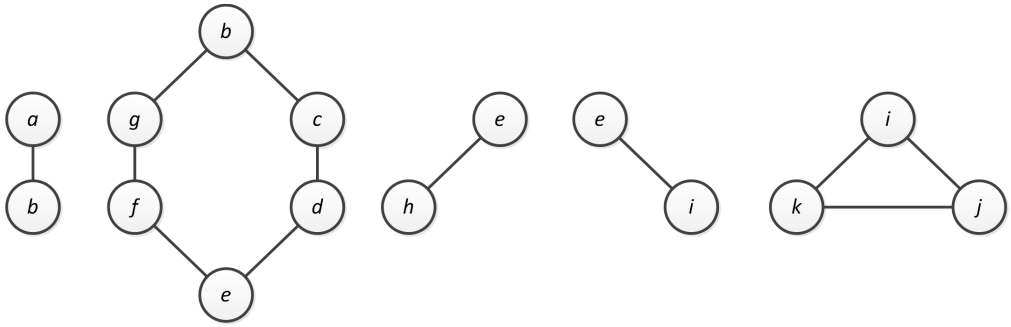

Figura 15: Componentes biconectados del grafo K.

3.4.2. Creación del árbol de componentes biconectados

Un paso previo a la construcción del árbol T_B es la construcción de un grafo intermedio de bloques $G_B = (B, E')$, donde B es el conjunto de componentes biconectados de K

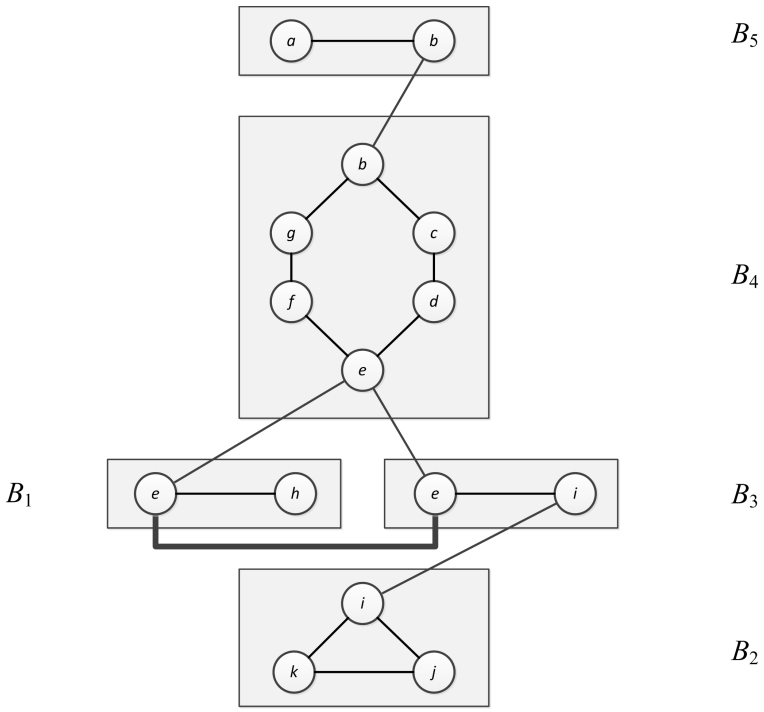

Figura 16: Grafo G_B con un ciclo.

encontrados por la búsqueda (ciclos y de cliques de tamaño 2) y donde $E' \subseteq E_K$. Para formar el grafo intermedio se elige aleatoriamente un componente como raíz para G_B. Posteriormente, para todo par $B_i,\ B_k$ de componentes de B se agrega una arista entre los componentes B_i, B_k, si existe un vértice $v_{i,j}$ y un vértice $v_{k,l}$ tal que $v_{i,j}$ y $v_{k,l}$ tienen el mismo identificador. La Figura 16 muestra el grafo de componentes obtenido; los vértices $v_{5,1}$ y $v_{4,0}$ tienen el mismo identificador, la letra b y los componentes $\{B_1, B_3, B_4\}$ poseen aristas entre sí puesto que tienen un vértice en común, el vértice e.

A partir de G_B se forma el árbol de bloques $T_B = (B, E_B)$, donde $E_B \subseteq E'$, eliminando las aristas entre los bloques que se encuentran en el mismo nivel, esto se puede realizar mediante una búsqueda en anchura (*BFS*). Por ejemplo, en la Figura 16, la arista entre B_1 y B_3 (pintada en negritas) se elimina generando así el árbol que se muestra en la Figura 17.

3.4.3. Cálculo del CIF por componente

El calculo del CIF sobre el árbol T_B sigue un recorrido en *postorden* (visita el vértice raíz después de visitar los vértices en sus subárboles (Cormen *et al.*, 2009)), a cada componente se le aplican los procedimientos descritos en las Secciones 3.4.3.1 a 3.4.4.

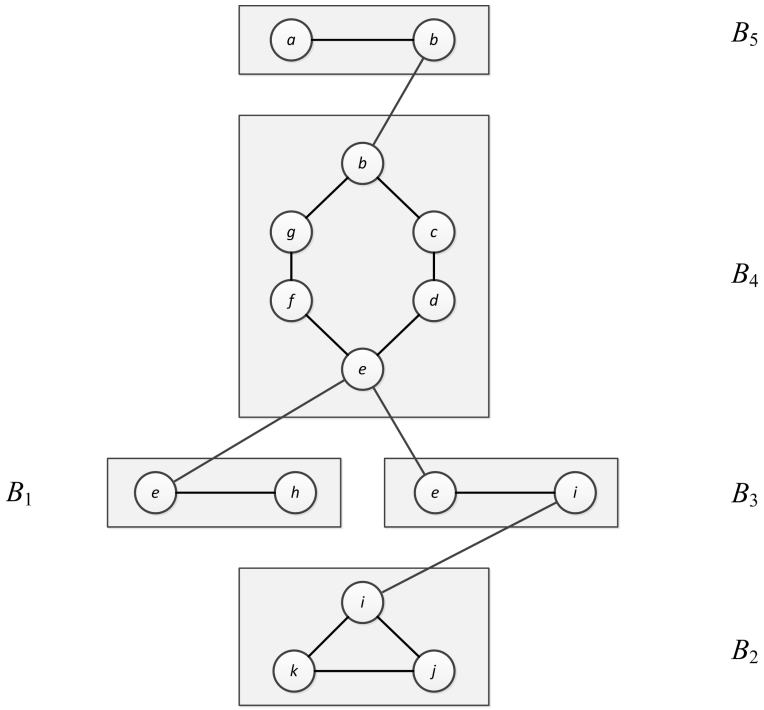

Figura 17: Árbol T_B obtenido al remover las aristas entre componentes hermanos.

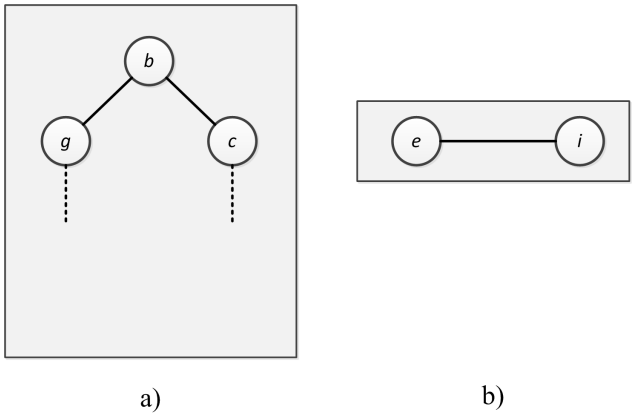

Figura 18: Vecindario izquierdo y derecho de un vértice. a) El vecino izquierdo de b es el vértice g y el vecino derecho es c. b) Para el vértice e, el vecino izquierdo y derecho es el vértice i; similarmente, para i, su vecino izquierdo y derecho es e.

3.4.3.1. Cálculo del vecino izquierdo y derecho

Cada componente biconectado tiene al menos dos vértices $v_{i,0}$ y $v_{i,1}$. Es necesario que éstos conozcan su vecindario inmediato, en otras palabras, aquellos que se encuentran a distancia 1. Además, cada vértice debe conocer la posición de sus vecinos, es decir, cuál de ellos se encuentra a la izquierda y a la derecha. En caso de que el componente sea un clique, se considera que el vecino izquierdo de $v_{i,0}$ es el vértice $v_{i,1}$; asimismo, el vecino derecho de $v_{i,0}$ también es $v_{i,1}$ (ver la Figura 18).

3.4.3.2. Empaquetamiento

La selección de vértices para el CIF sigue cuatro procesos:

1. Marcar legalmente la mayor cantidad de vértices que se pueda, evitando marcar el vértice de articulación.
2. Si dos o más configuraciones válidas (entiéndase por configuración al marcado de vértices) aportan la misma cantidad de vértices para al CIF, entonces elija aquella donde la distancia más corta del vértice marcado hacia el punto de articulación sea la mayor posible, es decir: dist($v_{i,0}$ $v'_{i,j}$) sea máxima.
3. Si existe la posibilidad de marcar el vértice de articulación para que la cardinalidad del conjunto sea mayor, entonces no hacerlo. Note que el vértice $v_{i,0}$ por ser vértice de articulación también existe en el componente padre B_p como $\bar{v}_{p,j}$; por lo tanto, si se marca o no dicho vértice es decisión del componente padre.
4. Cada vez que se marca un vértice para formar parte del conjunto, la distancia más corta de cada vértice hacia el vértice marcado más cercano debe actualizarse.

3.4.3.3. Cálculo del CIF cuando el componente es una hoja

Considere el componente B_2 (Figura 17), el cual contiene los vértices $\{i, j, k\}$ y donde el vértice de articulación es el vértice i. Cualquiera de estos vértices se puede marcar para formar parte del CIF; no obstante, la configuración debe ser aquella cuya distancia

hacia al vértice de articulación sea máxima, esta distancia se pasa hacia el componente padre como restricción. En este ejemplo, podría marcarse el vértice i para formar parte del CIF; sin embargo, dicho marcado implica que la distancia de i hacia el vértice de articulación (que es el mismo) sea cero. Otro posible marcado es tomar como parte del conjunto ya sea al vértice j o el vértice k, lo cual aporta (al igual que el marcado anterior) un sólo vértice al conjunto; sin embargo, al seleccionar j, la distancia hacia i es de una arista (algo similar ocurre al seleccionar k), por lo que ésta es una mejor configuración. En caso de que se genere un empate en cuanto a qué vértice se marca, elija aquel con el menor identificador; particularmente, para este ejemplo el vértice j se marcará para formar parte del CIF.

3.4.3.4. Cálculo del CIF para componentes intermedios o raíz

Si el componente es intermedio significa que sus componentes hijos ya se han evaluado (recorrido postorden) y por lo tanto, puede existir alguna restricción. Un componente puede tener tantas restricciones como número de hijos. Asimismo, cada $\bar{v}_{i,j}$ puede tener varias aristas hacia sus componentes hijos (una por hoja). Es responsabilidad de cada componente intermedio, en particular de cada $\bar{v}_{i,j}$, ponderar todas las restricciones que recibe y mantener la de mayor peso, es decir, la que entre todas las distancias sea menor. Tome como ejemplo el componente B_4 de la Figura 17, el cual tiene como hijos a B_1 y B_3. Se supone que sus hijos así como los subárboles enraizados en cada componente hijo ya han sido evaluados, entonces se tiene que el componente B_1 pasa una restricción de distancia uno al componente B_4 (esto es porque el vértice h se ha seleccionado para formar parte del CIF). De forma similar, el componente B_3 pasaría una restricción de distancia dos. Dadas las anteriores restricciones, el componente B_4 toma la restricción del componente B_1 (por ser aquella de distancia menor) y la asigna al vértice e como distancia hacia su vértice marcado más cercano.

Una vez que el componente conoce todas las restricciones, procede a buscar el marcado de vértices en el componente actual, respetando los cuatro principios generales mencionados en la sección de *Empaquetamiento*. Si el componente a evaluar es el vértice raíz B_N, entonces la evaluación procede igual que si se tratara de un nodo intermedio, la diferencia es que ya no existe ningún componente padre al cual pasar restricciones;

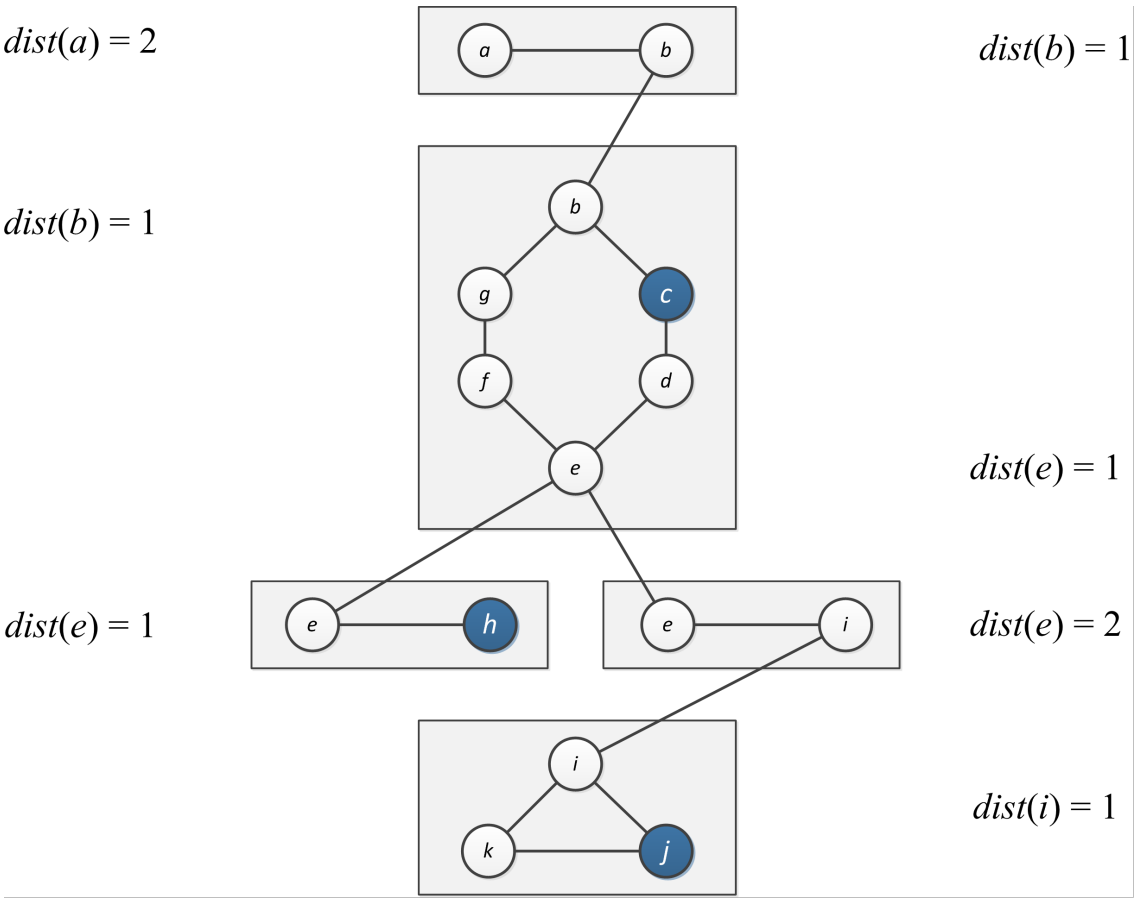

Figura 19: Árbol T_B una vez que sus componentes han sido evaluados. Los vértices marcados u obscuros pertenecen al conjunto S.

además, de ser necesario se permite el marcado del vértice raíz para aumentar la cardinalidad del conjunto.

Dentro del algoritmo existe un procedimiento llamado *incrementa*ρ cuya tarea es simple: contar el número de vértices que se han marcado dentro de cada componente de T_B. De forma similar, existe un método llamado *calcula*ρ que se encarga de contar el total de los vértices marcados en todo el árbol T_B.

3.4.4. Mejor lista y mejor distancia

Durante la ejecución del algoritmo se emplean diversas estructuras de datos, entre ellas se puede nombrar una lista llamada *mejorLista* que se encarga de mantener el marcado de vértices que realiza el algoritmo sobre un componente B_i que hasta cierto momento de la ejecución del algoritmo es la mejor, es decir, aquellos vértices que pertenecen al CIF.

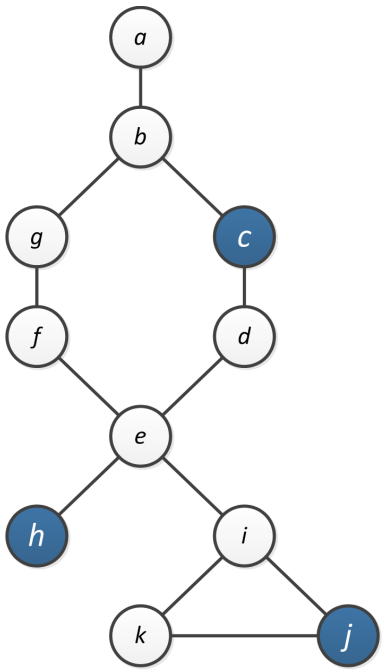

Figura 20: Grafo $K = (V_K,\ E_K)$ con vértices marcados u obscuros que pertenecen al CIF.

Existe un parámetro adicional llamado *mejorDistancia* que mantiene la distancia encontrada del vértice $v'_{i,j}$ al vértice de articulación, en caso necesario, este parámetro sirve para desempatar distintas configuraciones de vértices almacenadas en *mejorLista* tal que su cardinalidad sea igual entre ellas y a la vez máxima en B_i.

3.4.5. Construcción de la respuesta en K

La traducción de la respuesta encontrada en el T_B hacia el grafo K es una tarea trivial que se construye siguiendo un recorrido en postorden sobre los bloques del T_B y copiando el marcado de vértices hacia el grafo K.

3.5. Pseudocódigos

En esta sección se presenta el pseudocódigo del algoritmo propuesto. La sintaxis empleada se describe a continuación:

1. Los objetos tales como grafos, nodos y variables se escriben en cursiva: G.
2. La asignación de valores se indican mediante el signo "$\leftarrow$".
3. El símbolo "="se utiliza en instrucciones condicionales para verificar igualdad.

4. Los comentarios se denotan de la siguiente manera: `/*éste es un comentario*/`.

5. Los procedimientos se escriben en minúsculas si requieren una palabra, por ejemplo, “calcular(<valor>)”, en caso de necesitar más de una palabra para su descripción entonces se utilizan minúsculas y mayúsculas: “calcularRecorrido(<valor>)”. Si el procedimiento es bien conocido, como la búsqueda en profundidad, entonces se utiliza la abreviación del mismo (BFS).

6. Si un procedimiento se explica en algún otro algoritmo de esta sección, se denota de la siguiente manera RUTINA(<valor>).

7. Las variables se denotan con letras cursivas: $variable$.

8. Los atributos de un objeto se acceden mediante el operador “.” y se nombran siguiendo la misma convención empleada en los procedimientos, i.e, K.nodo(<identificador>).

9. Para acceder específicamente a un objeto o atributo dentro del objeto se hace mediante el uso de paréntesis: $objeto$.atributo(<atributo especifico>); en las estructuras tipo arreglo, el acceso se realiza mediante índices, arreglo[<índice>].

Algoritmo 3: CIF-MAXIMUM-CACTUS

Entrada : Un grafo cactus $K = (V_K,\ E_K)$, donde $\forall u \in V_K$, u.marcado = falso.

Salida : Un conjunto independiente fuerte $S \subseteq V_K$, tal que los vértices en S se marcan sobre K.

```
1  begin
2      if |K| = 1 then
3          K.nodo.marcado ← verdadero
4      else
5          Sea T_B un árbol de componentes biconectados
6          Obtenga los componentes biconectados de K, y agréguelos a T_B
7          Agregue una arista entre cada par de componentes B_i, B_k si existen los
           vértices v_{i,j}, v_{k,l} tal que sus etiquetas sean iguales
8          Elija de los componentes de T_B un componente raíz, B_N
9          agregarRaíz(T_B, B_N, falso) /*agregue una raíz pero no enraíce el
           grafo*/
10         BFS(T_B, B_N)
11         Elimine las aristas entre los componentes de T_B que se encuentren a la
           misma altura
12         agregarRaíz(T_B, B_N, verdadero) /*enraíce el grafo en B_N*/
13         OBTENERCIF(T_B)
14         /*Marcado de vértices en el grafo original*/
15         for 1 ≤ i ≤ |B| do
16             for 0 ≤ j < |B_i| do
17                 if v_{i,j}.marcado = verdadero then
18                     K.nodo(v_{i,j}).marcado ⟵ verdadero
19                 end
20             end
21         end
22     end
23     return K
24 end
```

Algoritmo 4: OBTENERCIF(T_B).

Entrada : Un árbol de componentes biconectados T_B.
Salida : El árbol de entrada con los vértices marcados que pertenecen al CIF.

begin
  Sea $B = \{B_1, B_2, ..., B_i, ...B_N\}$ el conjunto de componentes biconectados de T_B donde el i-ésimo elemento denota el orden de aparición del componente B_i en el recorrido postorden sobre T_B.
  Identifique los vértices de articulación en cada componente B_i.
  Sea B_p el componente padre de B_i.
  for $1 \leq i \leq |B|$ **do**
    restricción $\leftarrow$ restricción($\bar{v}_{p,j}$)
    restricción($v_{i,0}$) $\leftarrow$ *restricción*
    EMPAQUETAMIENTO(B_i)
    if dist($v_{i,0}$) = 0 **then**
      $distancia \leftarrow \infty$
    else
      $distancia \leftarrow$ dist($v_{i,0}$)
    end
    if *restricción* $< distancia$ **then**
      nuevaRestricción $\leftarrow$ *restricción*
    else
      nuevaRestricción $\leftarrow distancia$
    end
    restricción(B_p.nodo($v_{i,0}$)) $\leftarrow$ *nuevaRestricción*
    actualizarDistancias(B_i, $v_{i,0}$)
  end
  calculaρ(T_B)
  return T_B
end

Algoritmo 5: EMPAQUETAMIENTO(B_i)

Entrada : Un componente biconectado $B_i \in T_B$.
Salida : El componente de entrada con los vértices marcados para el CIF.

begin
  calculaVecinoIzquierdoDerecho(B_i)
  if $|B_i| > 2$ **then**
    PROCEDIMIENTO1(B_i)
  else
    PROCEDIMIENTO2(B_i)
  end
  reiniciarDistancias(B_i)
  distancia($v_{i,0}$, $v'_{i,j}$) $\leftarrow mejorDistancia$
  actualizarDistancias(B_i, $v_{i,0}$)
end

Algoritmo 6: PROCEDIMIENTO1(B_i).

Entrada : Un componente biconectado $B_i \in T_B$ con más de dos vértices sobre los cuales se realizará el recorrido.
Salida : El componente biconectado de entrada con los vértices marcados para el CIF.

```
1  begin
2      if tieneRestricciones(B_i) = verdadero then
3          u ← v_{i,0}
4          reiniciarDistancias(B_i)
5          actualizarDistancias(B_i, u)
6          RECORRIDO(B_i, u, "derecha")
7          u ← v_{i,0}
8          reiniciarDistancias(B_i)
9          actualizarDistancias(B_i, u)
10         RECORRIDO(B_i, u, "izquierda")
11     else
12         restricción(v_{i,0}) ← 3
13         actualizarDistancias(B_i, v_{i,0})
14         EMPAQUETAMIENTO(B_i)
15         return B_i
16     end
17 end
```

Algoritmo 7: RECORRIDO($B_i, u, sentido$)

Entrada : El sentido del recorrido {"derecho", "izquierdo"}.
Salida : Un marcado de vértices sobre B_i iniciando por el vértice u y cuyo recorrido
sigue la dirección indicada por la variable $sentido$.

```
1  begin
2      Sea sentido la dirección por la que se realizará el recorrido
3      for 0 ≤ l < |B_i| do
4          BUSCARMARCADO(B_i, u)
5          reiniciarDistancias(B_i)
6          actualizarDistancias(B_i, u)
7          if sentido = "derecho" then
8              u ← B_i.nodo(vecinoDerecho(u))
9          else
10             u ← B_i.nodo(vecinoIzquierdo(u))
11         end
12     end
13 end
```

Algoritmo 8: PROCEDIMIENTO2(B_i)

Entrada : Un componente biconectado B_i con más de dos vértices sobre los cuales se realizará el recorrido.

Salida : El componente biconectado de entrada con los vértices marcados para el CIF.

```
1  begin
2      Sea val una variable que mantiene un valor entero
3      if restricción(v_{i,0}) < (restricción(vecinoDerecho(v_{i,0})) + 1) then
4          val ← restricción(v_{i,0})
5      else
6          val ← restricción(vecinoDerecho(v_{i,0})) + 1
7          if val < ∞ then
8              restricción(v_{i,0}) ← val
9          else
10             restricción(v_{i,0}) ← 2
11         end
12         reiniciarDistancias(B_i)
13         actualizarDistancias(B_i, v_{i,0})
14         u ←vecinoDerecho(v_{i,0})
15         if dist(B_i.nodo(u)) ≥ 3 then
16             mejorLista ∪ u
17             dist(B_i.nodo(u), v'_{i,j}) ← 0
18             actualizarDistancias(B_i, u)
19         end
20         if dist(v_{i,0}, v'_{i,j}) ≥ 3 then
21             mejorLista ∪ v_{i,0}
22             dist(v_{i,0}, v'_{i,j}) ← 0)
23             actualizarDistancias(B_i, v_{i,0})
24         end
25         mejorDistancia ← dist(v_{i,0}, v'_{i,j})
26     end
27     for 0 ≤ l ≤ |mejorLista| do
28         if B_i.nodo(mejorLista[l]) = v_{i,0} then
29             B_i.nodo(mejorLista[l]).marcado ←"rojo"
30         else
31             B_i.nodo(mejorLista[l].marcado) ← verdadero
32             incrementarρ(B_i)
33             restricción(B_i.nodo(mejorLista[l])) ← 0
34         end
35     end
36     return B_i
37 end
```

Algoritmo 9: BUSCARMARCADO(B_i, u)

Entrada : El grafo que representa el componente B_i y un vértice u para empezar a buscar el marcado.
Salida : Una lista con los vértices de B_i que pertenecen a s_i^*.

```
1  begin
2      inicial ← u
3      for 0 ≤ l < |B_i| do
4          if dist(u, v_{i,0}) ≥ 3 then
5              lista ∪ u
6              dist(B_i.nodo(u), v_{i,0}) ← 0
7              actualizarDistancias(B_i, u)
8          end
9          u ← vecinoDerecho(B_i.nodo(u))
10     end
11     asignarMejorLista()
12     u ← inicial
13     reiniciarDistancias(B_i)
14     actualizarDistancias(B_i, u)
15     for 0 ≤ l < |B_i| do
16         if dist(B_i.nodo(u) ≤ 3 then
17             lista ∪ u
18             dist(B_i.nodo(u), v'_{i,j}) ← 0
19             actualizarDistancias(B_i, u);
20         end
21         u ← vecinoIzquierdo(B_i.nodo(u))
22     end
23     asignarMejorLista()
24 end
```

3.6. Estrategia de formalización del algoritmo CIF-MAXIMUM-CACTUS

Esta sección presenta la corrección del algoritmo CIF-MAXIMUM-CACTUS. La demostración se compone por las ecuaciones 8 y 9 que son expresiones matemáticas del problema del CIF sobre el árbol de componentes biconectados T_B. Las ecuaciones 10 y 11 muestran el proceso de decisión del algoritmo CIF-MAXIMUM-CACTUS para resolver el problema anterior. Asimismo, la demostración está integrada por los lemas 3.1 a 3.4 que contemplan casos específicos básicos y sin restricciones del marcado de vértices sobre el árbol T_B, mientras que los lemas 3.5 y 3.6 son versiones más elaboradas de los lemas anteriores y que consideran restricciones en el proceso de marcado de vértices. A continuación se muestra de forma general la descripción de estos lemas.

- Lema 3.1: presenta la cantidad de vértices que un grafo con $|V| \leq 2$ aporta al CIF.
- Lema 3.2 y Lema 3.3: indican el número de vértices marcados que un componente biconectado cíclico aporta al CIF; asimismo, señala la distancia que existe desde el vértice marcado más cercano ($v'_{i,j}$) al vértice de articulación ($v_{i,0}$).
- Lema 3.4: muestra el marcado de vértices sobre un grafo tipo árbol enraizado en r y cuyos hijos están únicamente a distancia 1 de r.
- Lema 3.5: demuestra que el algoritmo CIF-MAXIMUM-CACTUS encuentra el CIF máximo sobre un componente hoja del T_B, incluye tres casos que combinan uno o más de los lemas anteriores e incorpora restricciones para el marcado de vértices.
- Lema 3.6: demuestra que el algoritmo *CIF-Maximum-Cactus* encuentra el CIF máximo sobre un componente intermedio (sin considerar el vértice raíz) del T_B, considera dos casos para el marcado de vértices con y sin restricciones.

La demostración concluye con el Teorema 3.1 que mediante inducción matemática, englobando las ecuaciones 8 y 9 y los lemas anteriores demuestran que el marcado de vértices que realiza el algoritmo CIF-MAXIMUM-CACTUS sobre el árbol T_B es máximo y correcto. La Figura 21 muestra visualmente el flujo que sigue la demostración formal del algoritmo CIF-MAXIMUM-CACTUS.

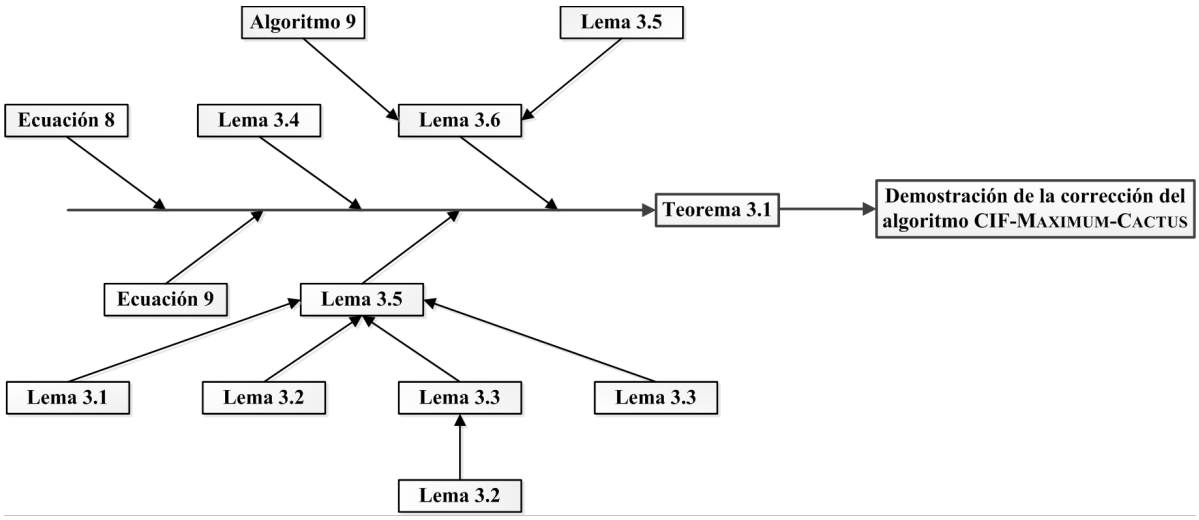

Figura 21: Diagrama con el flujo que sigue la demostración formal del algoritmo CIF-MAXIMUM-CACTUS.

3.6.1. Demostración formal de que el algoritmo CIF-MAXIMUM-CACTUS es correcto

Sea $S \subseteq V$ el conjunto de vértices seleccionados para formar el CIF que describe las siguientes propiedades:

$$S_i = S_{C_{B_i}} \cup \{ \text{ máximo número de vértices } \{u \in \{B_i - v_{i,0}\},\ v \in B_i \mid$$

$$(dist(u,v) \geq 3 \wedge dist(v,x) \geq 3\ \forall x \in S_{C_{B_i}}) \wedge dist(v'_{i,j}, v_{i,0}) \text{ es máxima}\}\} \text{ para } 1 \leq i \leq N. \tag{8}$$

$$S = \max \begin{cases} S_N. \\ S_N \cup r, \quad \text{si } \ (dist(r, v'_{N,j}) \geq 3\ \wedge\ dist(u,x) \geq 3\ \forall x \in S_{C_{S_N}}). \end{cases} \tag{9}$$

Donde S_i denota el CIF que se obtiene al considerar $B_i \cup C_{B_i}$, mientras que $S_{C_{B_i}}$ es el CIF de todos los subárboles enraizados en B_i.

Por su parte, s_i es el máximo número de vértices $\{u, v \in \{B_i - v_{i,0}\} | dist(u,v) \geq 3\}$ y sea $C = \{c_1, c_2, ...c_k, ...c_{|C|}\}$ el conjunto universo de todos los s_i posibles tal que $\forall u \in s_i \wedge \forall w \in S_{C_{B_i}}$, la $dist(u,w) \geq 3$. Asimismo, $C' = \{c'_1, c'_2, ...c'_l, ...c'_{|C'|}\}$ es el conjunto de mejores soluciones de C (aquellas cuya cardinalidad es la mayor), tal que:

$$C' = \begin{cases} C' \cup_{l=1}^{|C|} c_l \mid dist(v'_{i,j}, v_{i,0}) \text{ es máxima} & \text{si } |C| \geq 2. \\ c_1 & \text{si } |C| = 1. \\ \emptyset & \text{si } |C| = 0. \end{cases} \tag{10}$$

Además, sea s_i^* el CIF óptimo de $B_i - v_{i,0}$ donde:

$$s_i^* = \begin{cases} c'_l \in C' \mid \forall j \in \text{a los vértices de } c'_l, j \text{ es el menor identificador} & \text{si} |C| \geq 2. \\ c'_1 & \text{si } |C| = 1. \\ \emptyset & \text{si } |C| = 0. \end{cases} \quad (11)$$

Finalmente, $S_i = s_i^* \cup S_{C_{B_i}}$.

Para demostrar que el algoritmo es correcto, se proponen los siguientes lemas.

Lema 3.1 *Un grafo con $|V| \leq 2$ contribuye a lo más con un vértice a S.*

Demostración. Se consideran dos casos.

1. Si $|V| = 1$, el vértice se marca.
2. Si $|V| = 2$, entonces G tiene la forma de un clique de tamaño dos. Sean u, v los vértices de G. Tanto el vértice u como v se pueden marcar para pertenecer a S, pero no ambos a la vez. Suponga que u es el vértice raíz. De acuerdo a la Sección 3.4.3.3, el algoritmo marca el vértice v ya que la $dist(v'_{i,j} = v, v_{i,0} = u)$ es máxima, entonces $|S| = 1$.

Se completa la demostración. ■

Lema 3.2 *Sea G un ciclo que contiene $3k$, $3k+1$ o $3k+2$ vértices, para algún $k \in \mathbb{Z}^+$, entonces G aporta exactamente k vértices al CIF, si no hay restricciones.*

Demostración. Se demuestra por contradicción en el número de vértices del ciclo. Primero se demuestra que no pueden existir más de k vértices marcados para el CIF en el ciclo, posteriormente que no hay menos de k vértices.

Suponga que hay al menos $k+1$ vértices que pertenecen al conjunto S en un ciclo con z vértices donde $3k \leq z \leq 3k+2$. Puesto que los vértices seleccionados forman parte

de un CIF, entonces éstos deben de estar separados entre sí por lo menos por 3 aristas o 2 vértices que no pertenecen al CIF. De aquí que si existen $k+1$ vértices en el CIF debe haber al menos otros $2(k+1)$ vértices que los separan. Por lo que el número total de vértices en el ciclo debe ser $3k+3 > z$. Lo cual es una contradicción al número supuesto de vértices.

Ahora suponga que el ciclo aporta a lo más $k-1$ vértices al CIF y que el primer vértice que pertenece al CIF es el $v_{i,1}$, después los 2 vértices $v_{i,2}$ y $v_{i,3}$ no pertenecen al conjunto, mientras que el vértice $v_{i,4}$ sí pertenece y así sucesivamente. Siguiendo este procedimiento, el i-ésimo vértice marcado está dado por la fórmula $3(i-1)+1$, de aquí que el $k-1$ vértice que pertenece al CIF debe ser $v_{i,3k-5}$. Dado que existen 5 vértices disponibles no marcados, se puede marcar al vértice $v_{i,3k-2}$. Lo anterior contradice la suposición inicial de que existen a lo más $k-1$ vértices en el CIF. Por lo tanto, se demuestra que el CIF tiene exactamente k vértices. ■

Lema 3.3 *Sea B_i un ciclo libre de restricciones donde $3k \leq |B_i| \leq 3k+2$ vértices, para algún $k \in \mathbb{Z}^+$; entonces, la distancia del vértice de corte $v_{i,0}$ hacia el vértice marcado más cercano $v'_{i,j}$ está dado por la ecuación 12.*

$$dist(v_{i,0}, v'_{i,j}) = \begin{cases} 1, & \textit{si } |B_i| \textit{ es múltiplo de 3.} \\ 2, & \textit{en caso contrario.} \end{cases} \tag{12}$$

Demostración. La demostración se realiza sobre el número de vértices que separan a los vértices marcados, se consideran 3 casos.

1. Si $|B_i| = 3k$, entonces el máximo número de vértices que se pueden marcar bajo esta configuración es k (por el Lema 3.2). Bajo esta selección de vértices marcados, la distancia entre cada par de vértices $u, v \in s_i$ es exactamente 3; además, como el vértice $v_{i,0}$ no se puede marcar para s_i, entonces $v_{i,0}$ debe quedar entre 2 vértices marcados que se encuentran separados por el par $x, y \notin s_i$. De aquí que el vértice $v_{i,0}$ debe ser el vértice x o y y la distancia más cercana desde $v'_{i,j}$ es 1.

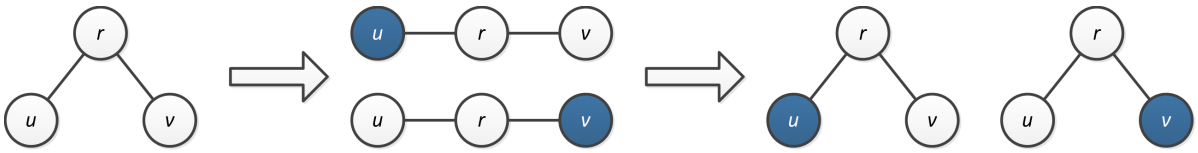

Figura 22: La distancia del vértice u a v es dos ya sea que la selección sea u o v, sólo un vértice se aporta al conjunto S.

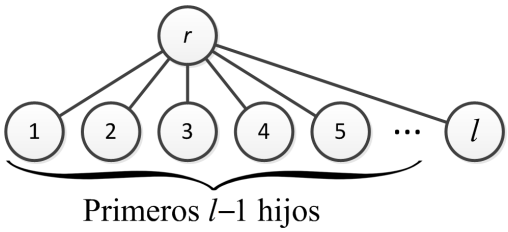

Figura 23: No importa el número de hijos que tenga el subárbol enraizado en el vértice r, si sus hijos están a distancia 1 de la raíz y estos ya no tienen descendientes, entonces cualquier par $u, v \in C_r$ se encuentra a distancia 2; debido a esto, sólo uno de ellos puede ser seleccionado para pertenecer al conjunto.

2. Si $|B_i| = 3k+1$, entonces el máximo número de vértices que se pueden marcar es k (por el Lema 3.2). Al cumplirse esta condición, cada par de vértices que pertenecen a s_i se encuentran separados exactamente por 2 vértices, excepto para un par $u, v \in s_i$ cuya separación es de 3 vértices entre ellos. De aquí que el $v_{i,0}$ debe ser el vértice intermedio para maximizar la distancia al vértice u y v. Por lo tanto, la $dist(v'_{i,j}, v_{i,0}) = 2$.

3. Si $|B_i| = 3k+2$, éste es similar al caso 2, excepto que la separación entre u y v deben ser 4 vértices; nuevamente, el vértice $v_{i,0}$ debe ser algún vértice intermedio para maximizar la distancia, por lo que la $dist(v'_{i,j}, v_{i,0}) = 2$.

Se completa la demostración. ■

Lema 3.4 *Sea G un árbol de altura 1 enraizado en r y donde C_r es el conjunto de vértices descendientes. Bajo esta configuración, a lo más un vértice de C_r puede pertenecer al CIF (Ver las figuras 22 y 23).*

Demostración. El diámetro del subgrafo de expansión generado por el conjunto $r \cup C_r$ es dos; por lo tanto, se puede seleccionar a lo más un vértice para el CIF. ■

Lema 3.5 *El algoritmo* CIF-MAXIMUM-CACTUS *encuentra el CIF máximo sobre un componente hoja B_i del árbol T_B.*

Demostración. Se consideran 3 casos.

1. El componente hoja B_i se encuentra libre de restricciones o no tiene hermanos. Se consideran 2 posibilidades:
   - Si $|B_i| = 2$, el caso se maneja como indica el Lema 3.1.
   - Si $|B_i| \geq 3$, se maneja como lo indican los Lemas 3.2 y 3.3.
2. El componente hoja B_i tiene restricciones y el vértice de articulación $v_{i,0}$ se comparte entre bloques hermanos. Suponga, sin perdida de generalidad, un componente padre B_p que tiene al menos 2 componentes hijos enraizados en $\bar{v}_{p,j}$. Sean B_{i-1} y B_i dichos componentes. Además, suponga que B_{i-1} es el primer componente evaluado, y que S_{i-1} es máximo. Nótese que $\bar{v}_{p,j} = v_{i-1,0} = v_{i,0}$. Se consideran 2 subcasos.
   - La restricción que B_{i-1} impone al componente B_i es 1 (ver la Figura 24). El algoritmo CIF-MAXIMUM-CACTUS, busca restricciones antes de evaluar al componente hijo. Posteriormente, el algoritmo realiza las operaciones pertinentes para encontrar el CIF en B_i. Sin embargo, la selección a distancia 1 de su bloque hermano B_{i-1} evita que algunos vértices de B_i se puedan marcar para pertenecer a S_i; no obstante, dicha restricción no disminuye el tamaño del conjunto (ver el Lema 3.4).
   - La restricción que B_{i-1} impone es 2 (ver la Figura 25). En esta configuración, el vértice de articulación $v_{i,0}$ se encuentra a distancia 2 del vértice más cercano marcado en B_{i-1}. Bajo esta configuración, cualquier otro vértice en $B_i - v_{i,0}$ debe estar al menos a distancia 3, por lo cual, la selección de vértices que realice el componente B_i no entra en conflicto con aquella realizada por B_{i-1}.
3. El componente B_i tiene restricciones pero los componentes hermanos no comparten el vértice de articulación con B_i. Suponga sin pérdida de generalidad que un

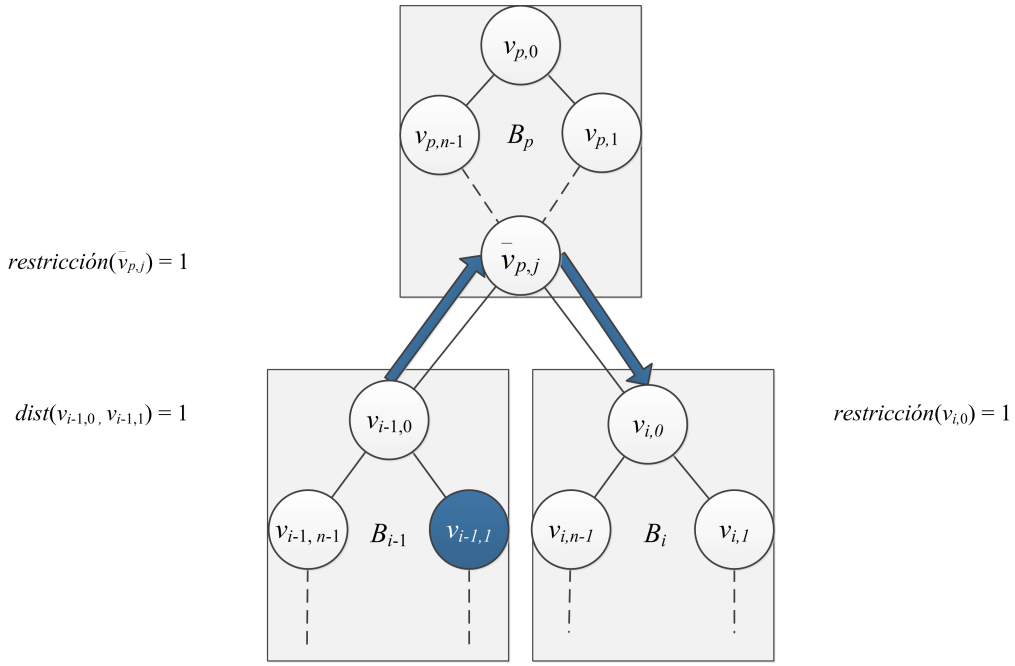

Figura 24: El componente hoja a evaluar es B_i, el cual recibe restricciones por el vértice $\bar{v}_{p,j}$.

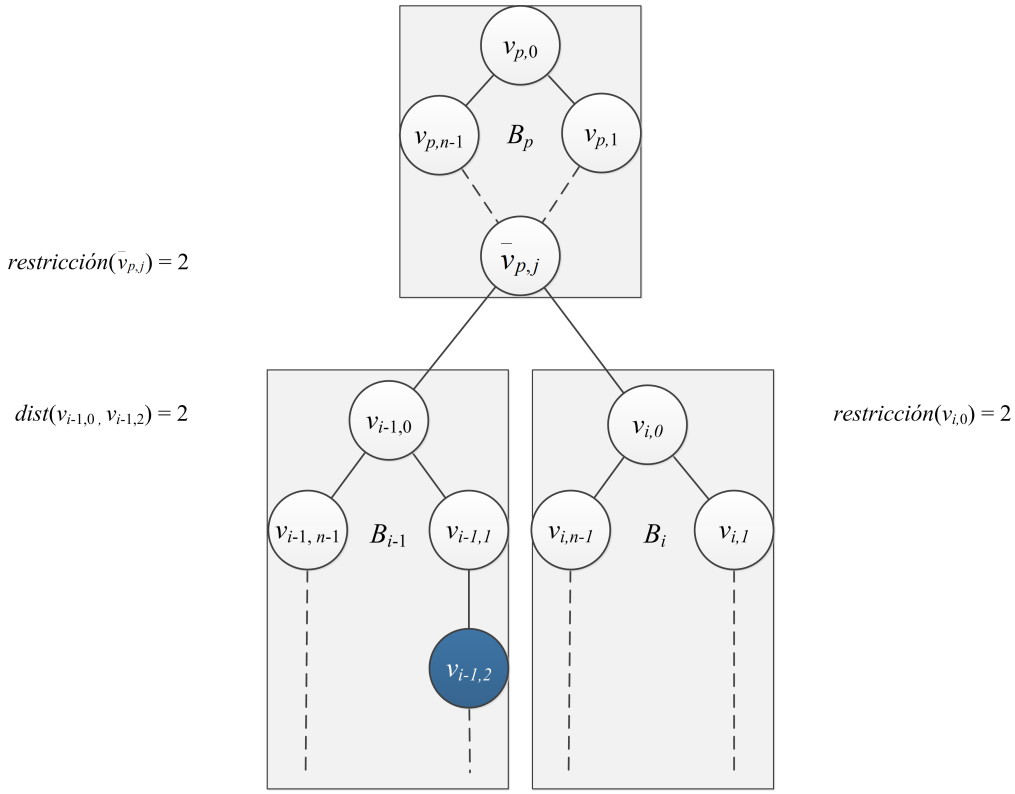

Figura 25: Todo vértice en $B_i - v_{i,0}$ se encuentra a distancia 3.

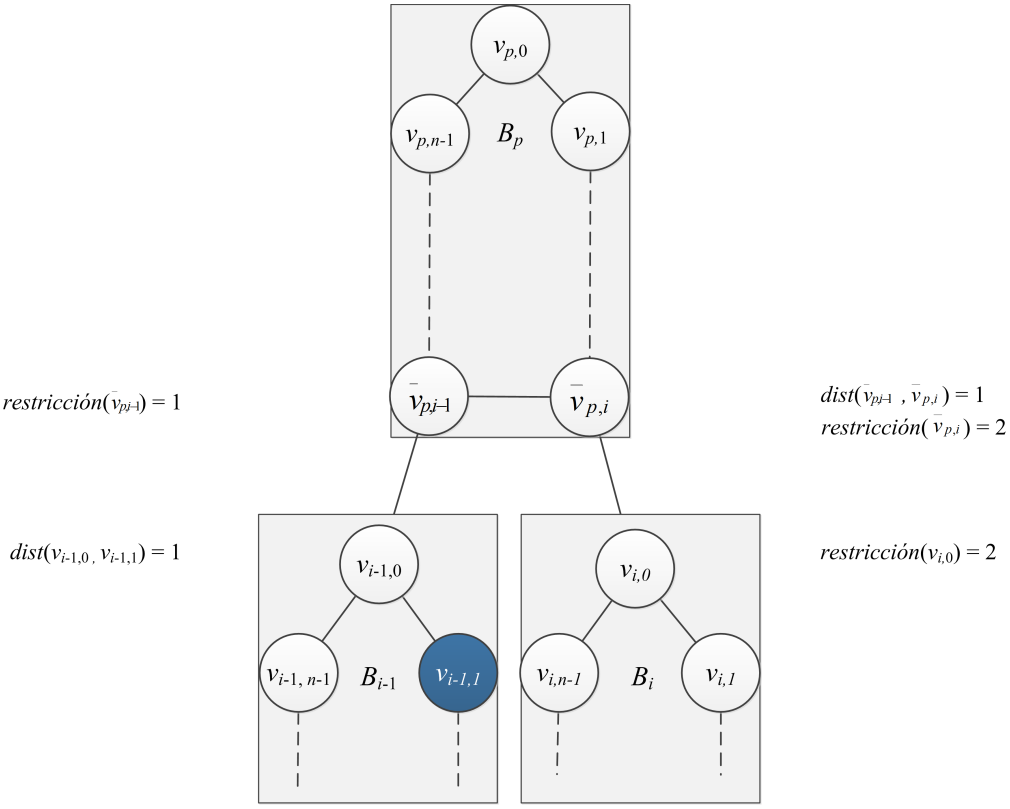

Figura 26: La restricción que recibe B_i es 2.

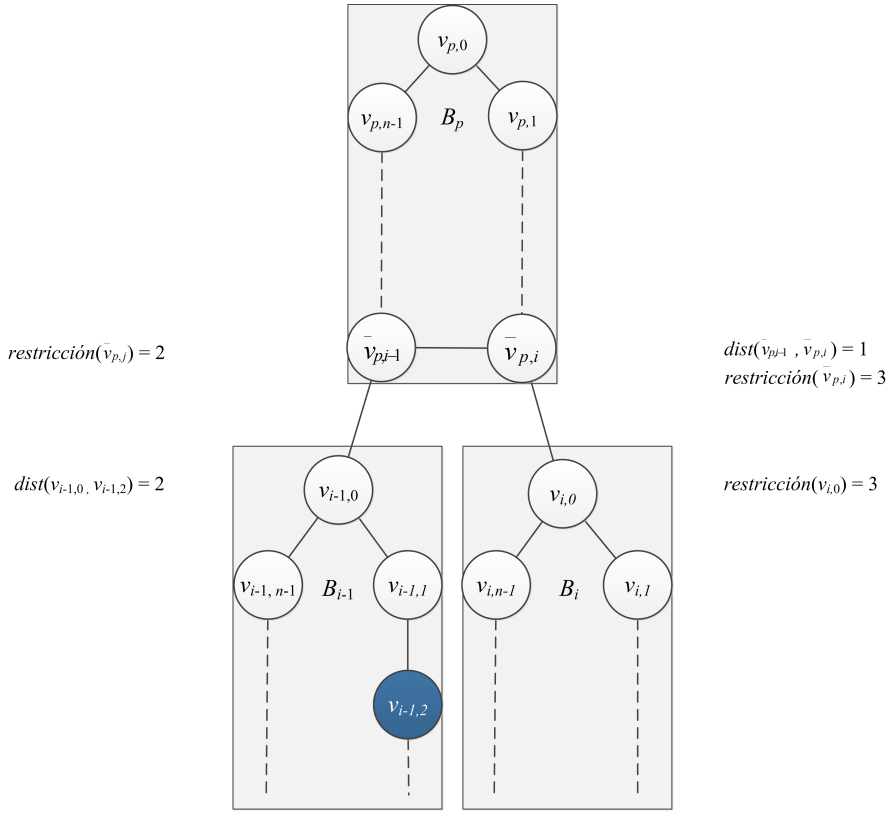

Figura 27: La restricción que recibe B_i es 3; por lo tanto, B_i se evalúa como si no tuviese restricciones.

componente padre B_p tiene al menos 2 componentes hijos B_{i-1} y B_i. Además, suponga que B_{i-1} se encuentra enraizado en $\bar{v}_{p,i-1}$ mientras que B_i en $\bar{v}_{p,i}$. Adicionalmente, suponga que la $dist(\bar{v}_{p,i-1}, \bar{v}_{p,i}) = 1$ arista y que el componente B_{i-1} se evalúa antes que B_i. Se consideran 2 opciones.

- La restricción que imponen los componentes hermanos sobre B_i es 2 (ver la Figura 26). Suponga que la restricción que impone B_{i-1} sobre $v_{i-1,0} = 1$. De aquí que la restricción recibida por $\bar{v}_{p,i-1}$ sea también 1. Además, como la distancia entre $dist(\bar{v}_{p,i-1}, \bar{v}_{p,i}) = 1$, la restricción sobre $\bar{v}_{p,i}$ debe ser 2. Bajo esta configuración, cualquier otro vértice en $B_i - v_{i,0}$ debe estar al menos a distancia 3, por lo cual, la selección de vértices que realice el componente B_i no entra en conflicto con aquella realizada por B_{i-1}.
- La restricción que imponen los componentes hermanos sobre B_i es igual o mayor que 3 (ver la Figura 27). Suponga que la restricción que impone B_{i-1} sobre $\bar{v}_{p,i-1}$ es igual o mayor que 2, y como la $dist(\bar{v}_{p,i-1}, \bar{v}_{p,i}) = 1$, entonces la restricción que recibe $\bar{v}_{p,i}$ debe ser al menos de 3; por lo tanto, el componente hoja B_i se puede evaluar como si no tuviese restricciones.

Se completa la demostración. ■

Lema 3.6 *El algoritmo* CIF-MAXIMUM-CACTUS *calcula el CIF máximo (sin considerar el vértice raíz)* S_i *sobre los bloques intermedios del* T_B*.*

Demostración. Sea B_i un componente biconectado que es ancestro de al menos un componente B_{i-1}. Se consideran 2 casos.

1. B_i tiene restricciones. El componente B_i recibe restricciones por medio de los vértices de articulación $\bar{v}_{i,j}$ de cada uno de sus hijos; asimismo, B_i llama a una rutina para actualizar restricciones, esto para que todo $v \in B_i$ conozca la distancia a la que se encuentra del vértice marcado más cercano. Posteriormente, el Algoritmo 9 itera sobre los vértices de B_i, en sentido izquierdo y derecho para encontrar el CIF.

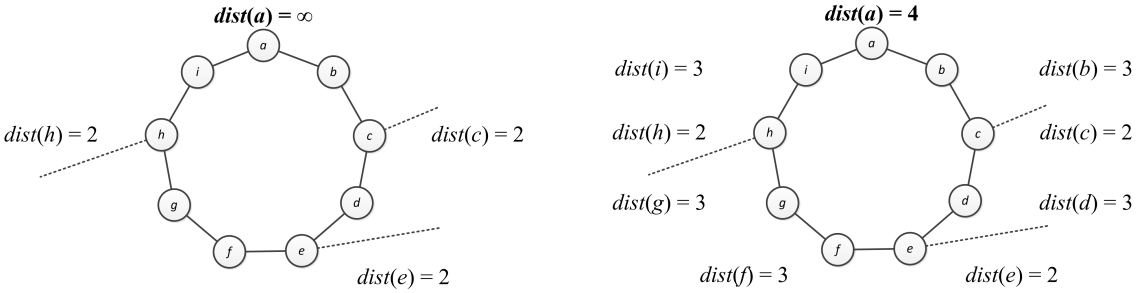

Figura 28: Lado izquierdo: componente a evaluar indicando las restricciones que recibe de los bloques hijos. Lado derecho: componente después de que las restricciones fueran propagadas hacia el resto de los vértices.

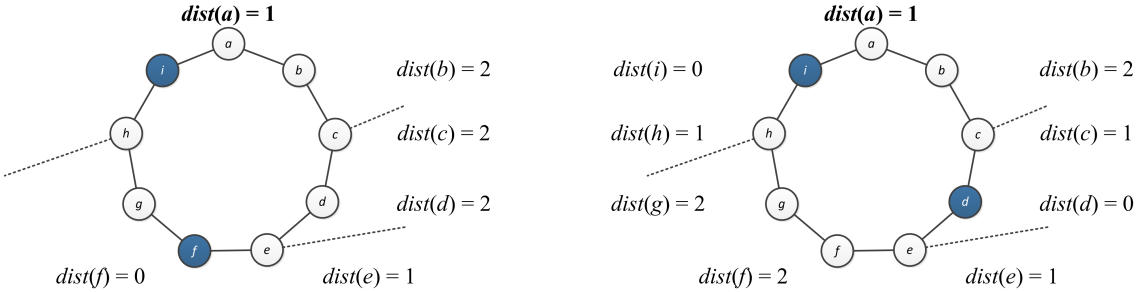

Figura 29: Posibles marcado de vértices cuando el recorrido se hace por la izquierda.

Las respuestas (selección de vértices y distancias) se almacenan en las variables *lista* y *distancia*. El criterio de desempate entre las configuraciones (en orden de importancia) es el siguiente:

a) Conservar la configuración de mayor cardinalidad.

b) Seleccionar la configuración tal que la $dist(v'_{i,j}, v_{i,0})$ es máxima.

c) Seleccionar la respuesta cuyos vértices tengan el menor identificador.

2. El componente B_i no recibe restricciones. En este caso, B_i se comporta como si fuera un componente hoja de T_B, por lo que la demostración es como en el Lema 3.5.

Las figuras 28, 29 y 30 muestran posibles configuraciones de salida. Se completa la demostración. ■

Teorema 3.1 *Sea $T_B = (B, E'')$ un árbol de bloques con $|B| = N$ y sea $S \subseteq V$ el Conjunto Independiente Fuerte con las propiedades de las ecuaciones 8 y 9, entonces S es un CIF y es máximo.*

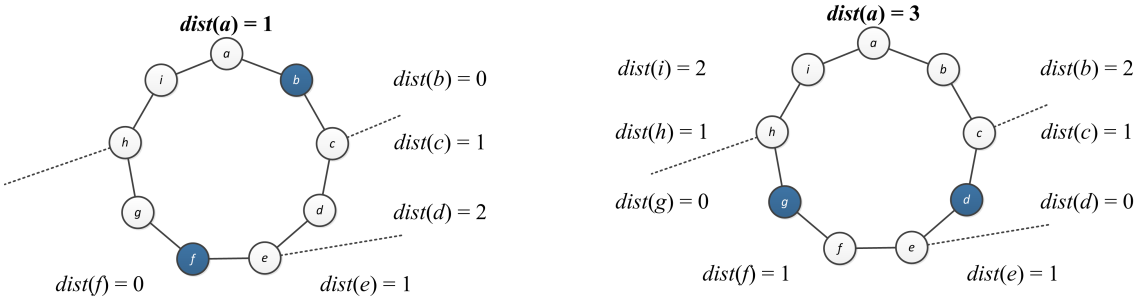

Figura 30: Posible marcado cuando el recorrido se hace por la derecha. La mejor configuración encontrada es la del lado derecho.

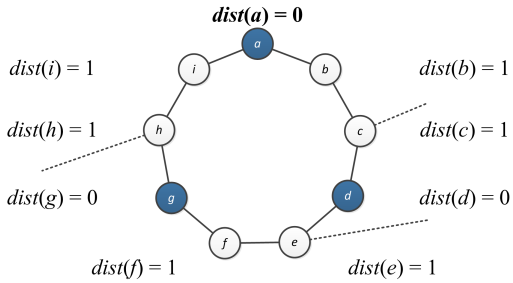

Figura 31: Mejor configuración, después de aplicar la ecuación 9.

Demostración. La demostración de que S es máximo se realiza mediante inducción sobre la altura del árbol T_B.

Como caso base considere que el componente B_i es una hoja del T_B, la corrección de este caso se discute en el Lema 3.5. Observe que el conjunto S_1 contiene el máximo número de vértices (excluyendo a $v_{i,0}$) que el T_B puede aportar y que la distancia desde $v_{i,0}$ hacia el vértice marcado más cercano es máxima. Además, de acuerdo al Lema 3.4, el marcado de vértices no interfiere con la cardinalidad del conjunto S_i, por lo cual se cumple la ecuación 8.

Para el caso inductivo, suponga que el teorema se mantiene para todo subárbol del T_B enraizado en el componente B_{i-1} para $1 < i \leq N$. Se consideran 2 casos.

1. El componente B_i se encuentra libre de restricciones. Por el Lema 3.6, si para el $(i-1)$-ésimo componente el S_{i-1} es correcto, entonces el S_i correspondiente a $B_i - v_{i,0}$ también se calculará correctamente.

2. El componente B_i recibe restricciones. Este caso es la combinación del Lema 3.6 y los casos 2 y 3 del Lema 3.5.

Finalmente, si $B_i = B_N$, entonces la ecuación 9 verifica si es posible agregar r al conjunto S. La Figura 31 muestra la salida del algoritmo cuando la ecuación 9 se aplica al componente B_N, considere que el vértice a es la raíz del grafo. La cardinalidad del conjunto es máxima. Se completa la demostración. ■

3.7. Análisis del tiempo de ejecución del algoritmo

Esta sección se analiza el tiempo de ejecución del algoritmo CIF-MAXIMUM-CACTUS. Cada una de los pasos que aquí se mencionan se describen en la Sección 3.2.

El primer paso requiere de $O(1)$ u. t., puesto que se trata de una simple comparación del número de vértices en el grafo.

El segundo paso, se encarga de dividir el grafo en componentes biconectados; esto a la vez se subdivide en la búsqueda de ciclos y búsqueda de cliques de tamaño 2. Para la búsqueda de ciclos, se hace uso de la búsqueda en profundidad (DFS) la cual requiere de $O(n+m)$ u. t. Por otro lado, la búsqueda de cliques de tamaño 2 simplemente compara el conjunto de aristas inicial contra aquellas retiradas por la búsqueda de ciclos, lo cual se resuelve en $O(m)$ u. t. Puesto que el grafo tipo cactus es un grafo disperso, es decir $|E_K| \approx O(V_K)$, el tiempo necesario para dividir el grafo en componentes biconectados es $O(n)$ u. t.

El tercer paso, generar un árbol de componentes biconectados T_B, requiere de $O(n^2)$ u. t. puesto que por componente se revisa si existe adyacencia con el resto de los componentes.

Para el cuarto paso, cálculo del CIF en cada componente, el marcado se realiza mediante el recorrido por la izquierda y por la derecha, el cual toma O(n') u. t. donde n' es el número de vértices que integran el componente que se está evaluando. Dichos recorridos se realizan tomando como vértice inicial cada uno de los vértices en el componente. En el peor de los casos $n' \approx O(n)$, por lo que el tiempo de ejecución es $O(n^2)$ u. t.

Finalmente, la reconstrucción de la respuesta en K (paso 5) toma $O(n)$ u. t., ya que simplemente consta del recorrido en postorden sobre los componentes en T_B y de copiar el marcado de los vértices en cada B_i hacia el grafo K.

Algoritmo 3: CIF-MAXIMUM-CACTUS

Entrada : Un grafo cactus $K = (V_K,\ E_K)$, donde $\forall u \in V_K$, u.marcado = falso.
Salida : Un conjunto independiente fuerte $S \subseteq V_K$, tal que los vértices en S se marcan sobre K.

```
1  begin
2      if |K| = 1 then
3          K.nodo.marcado ← verdadero
4      else
5          Sea T_B un árbol de componentes biconectados
6          Obtenga los componentes biconectados de K, y agreguelos a T_B
7          Agregue una arista entre cada par de componentes B_i, B_k si existen los vértices
           v_{i,j}, v_{k,l} tal que sus etiquetas sean iguales
8          Elija de los componentes de T_B un componente raíz, B_N
9          agregarRaíz(T_B, B_N, falso) /*agregue una raíz pero no enraíce el
           grafo*/
10         BFS(T_B, B_N)
11         Elimine las aristas entre los componentes de T_B que se encuentren a la misma
           altura
12         agregarRaíz(T_B, B_N, verdadero) /*enraíce el grafo en B_N*/
13         OBTENERCIF(T_B)
14         /*Marcado de vértices en el grafo original*/
15         for 1 ≤ i ≤ |B| do
16             for 0 ≤ j < |B_i| do
17                 if v_{i,j}.marcado = verdadero then
18                     K.nodo(v_{i,j}).marcado ⟵ verdadero
19                 end
20             end
21         end
22     end
23     return K
24 end
```

paso 1, $O(1)$ u. t.

paso 2, $O(n)$ u. t.

paso 3, $O(n^2)$ u. t.

paso 4, $O(n^2)$ u. t.

paso 5, $O(n)$ u. t.

Figura 32: Análisis de la complejidad del algoritmo CIF-MAXIMUM-CACTUS.

Por lo tanto, la complejidad total del algoritmo es $O(n^2)$ u.t. en el peor de los casos (ver la Figura 32).

3.8. Ejemplo final

Ahora que los componentes del algoritmo se han descrito, y después de revisar los pseudocódigos y analizar su complejidad, se presenta un ejemplo más desarrollado de la ejecución del algoritmo CIF-MAXIMUM-CACTUS. Considere que el cactus $K = (V_K, E_K)$ que se muestra en la Figura 33 es el grafo a evaluar.

De acuerdo con la descripción de alto nivel mostrada en la Sección 3.2 y al Algoritmo 3, el primer paso es contar el número de vértices del grafo K, puesto que $|V_K| > 1$, entonces se deben obtener los componentes biconectados del grafo (segundo paso). El tercer paso es generar el árbol de componentes biconectados T_B, el cual se muestra en la Figura 34.

El cuarto paso del algoritmo encuentra el CIF sobre T_B siguiendo el recorrido postorden sobre sus vértices. La evaluación del componente B_1 es trivial puesto que es una

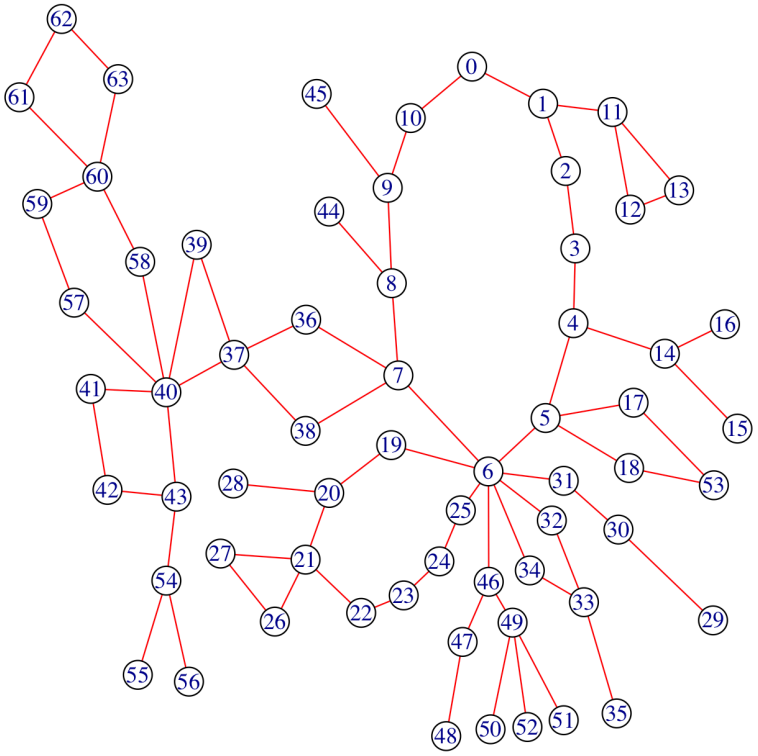

Figura 33: Grafo cactus con 64 vértices.

hoja libre de restricciones, de aquí que el vértice $\{v_{1,1} = 45\}$ se marca. Debido al proceso anterior, el vértice $v_{1,0}$ se encuentra ahora a distancia 1 del vértice marcado, es decir, $dist(v_{1,0}, v'_{1,1}) = 1$. La distancia anterior se pasa como restricción al componente padre B_{32}. Es responsabilidad de B_{32} actualizar las distancias de todos sus vértices de acuerdo a la restricción recibida por el vértice de articulación hacia su hijo, por lo que, la distancia de $v_{32,8}$ ahora es dos hacia el vértice marcado más cercano, de aquí que el vértice $v_{2,0}$ reciba una restricción de valor 2 (ver la Figura 35). Considerando la restricción anterior, el vértice $v_{2,1}$ se encuentra a distancia 3, por lo que se marca.

De forma similar, se evalúa el subárbol enraizado en B_{10} (ver la Figura 36). Observe que el vértice $v_{10,0}$ se encuentra a distancia 3 del vértice marcado más cercano, y podría marcarse; no obstante, el algoritmo indica que en este caso la decisión la toma el componente padre, de aquí que el vértice $v_{32,7}$ no reciba restricciones. El resultado de evaluar los componentes B_{11} a B_{25} se muestra en la Figura 37. El vértice de articulación de los componentes B_{13}, B_{16}, B_{18} y B_{25} se encuentra a distancia 3 de su vértice marcado más cercano. La decisión de marcar el vértice depende del componente B_{32}.

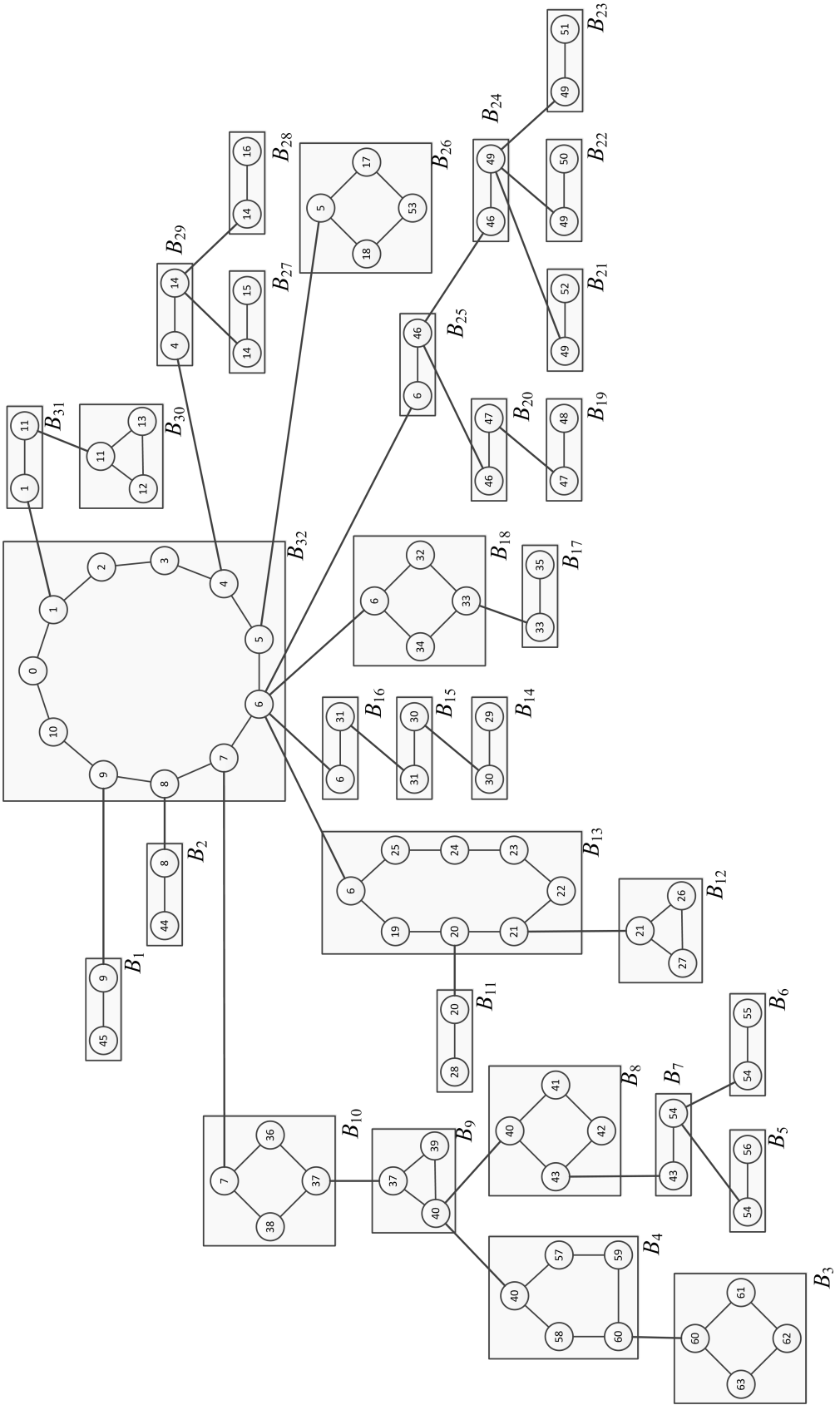

Figura 34: Árbol de componentes biconectados obtenido del grafo de la Figura 33.

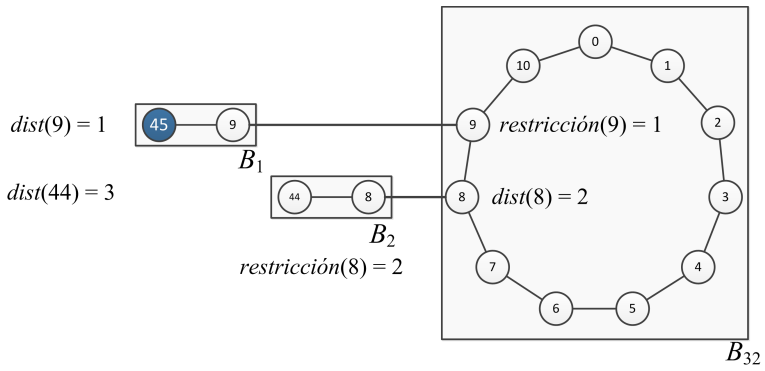

Figura 35: Subgrafo después de evaluar el componente B_1.

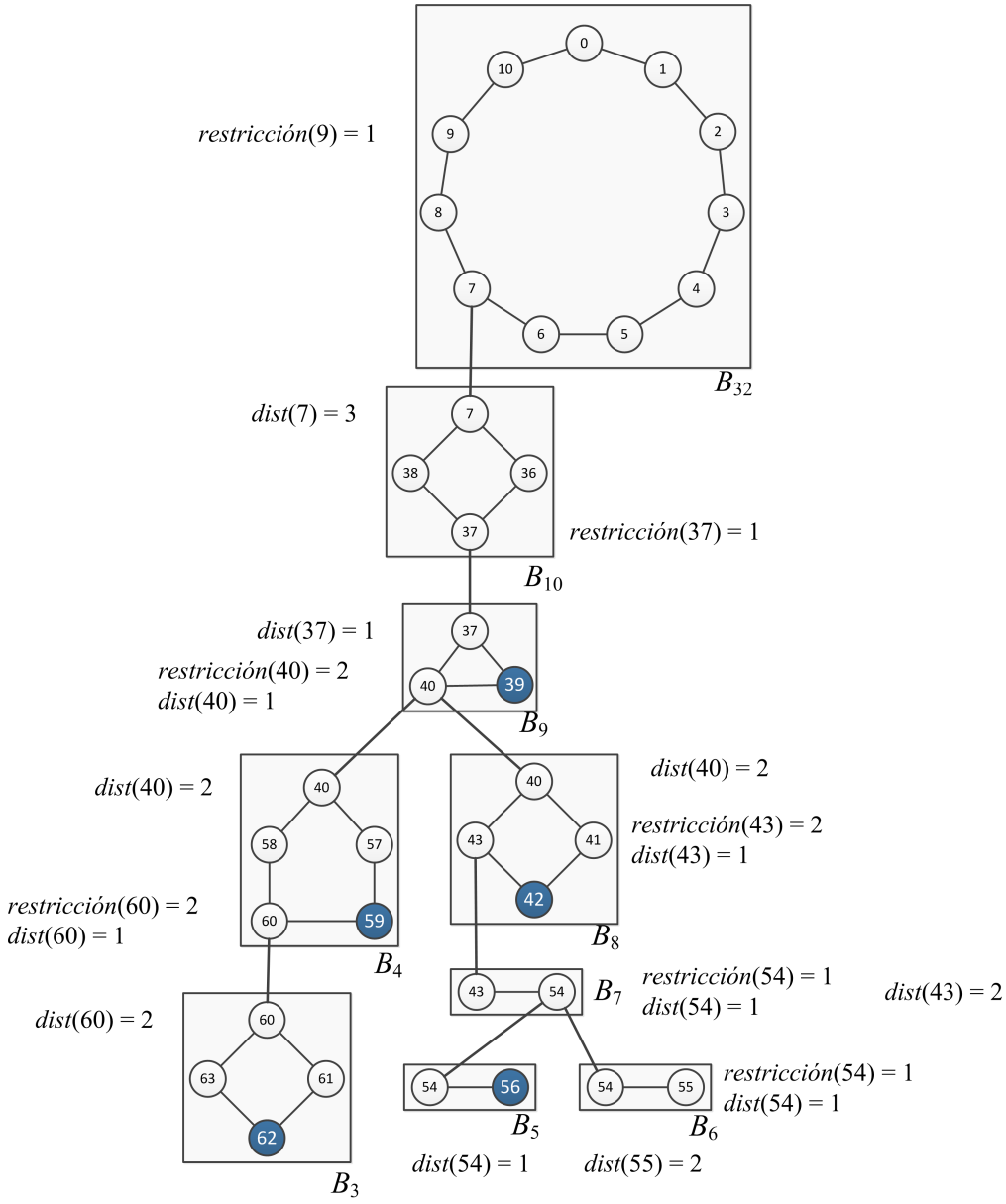

Figura 36: El vértice $v_{10,0}$ no se marca.

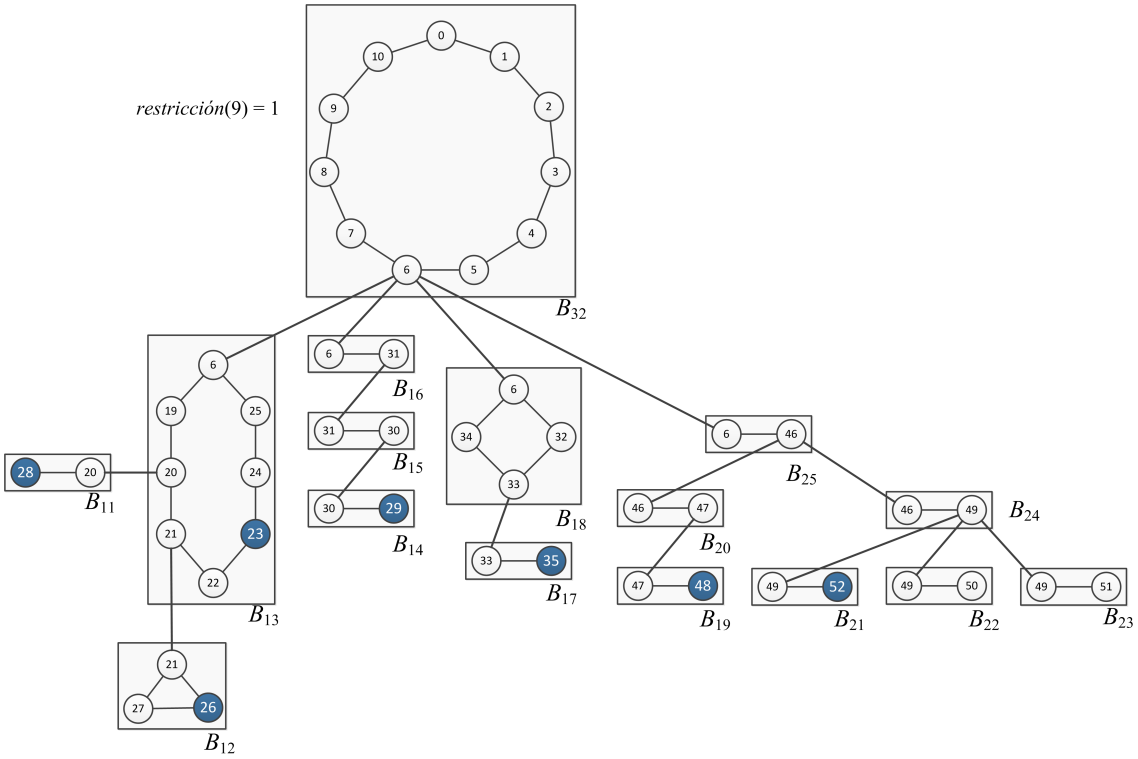

Figura 37: El vértice 6 de cada uno de los componentes de $C_{B_{32}}$ está a distancia 3 de su vértice marcado más cercano; a B_{32} le corresponde decidir si dicho vértice se marca.

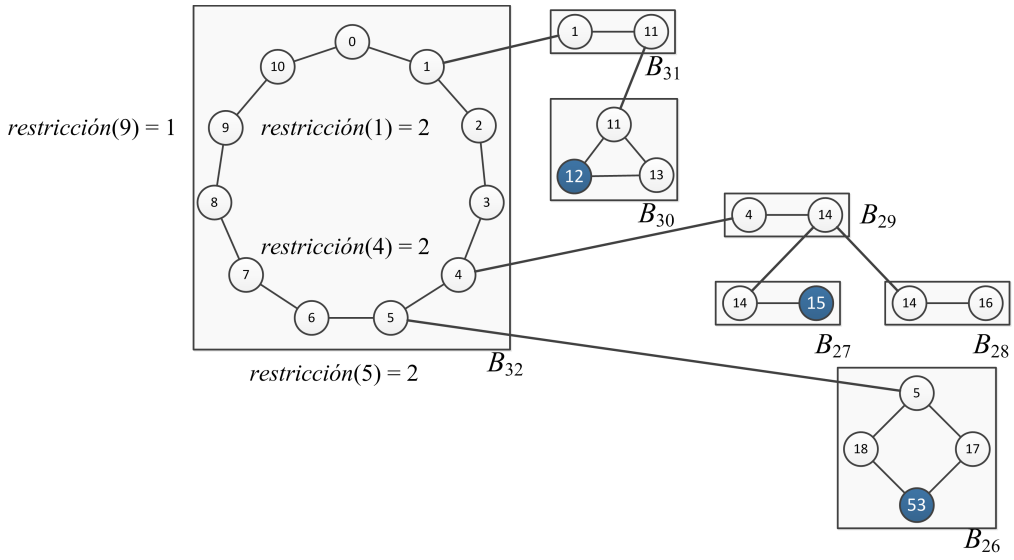

Figura 38: El componente B_{32} ya tiene todas las restricciones de $C_{B_{32}}$, por lo que ya está listo para evaluarse.

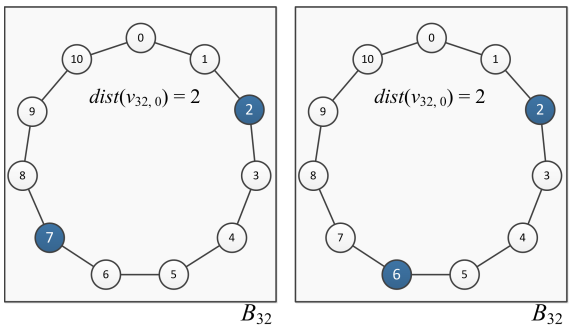

Figura 39: Dos posibles configuraciones que muestran el marcado de vértices para B_{32}. No obstante, el algoritmo no conserva ninguna de éstas, puesto que la distancia de $v'_{32,j}$ hacia $v_{32,0}$ no es máxima.

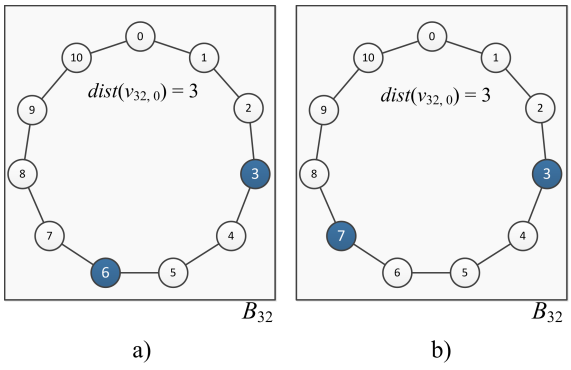

Figura 40: Dos configuraciones válidas cuya distancia es máxima de $v'_{32,j}$ hacia $v_{32,0}$. El algoritmo conserva la solución mostrada en el inciso a.

La evaluación de los componentes B_{26} a B_{31} se muestra en la Figura 38. Finalmente, se evalúa el componente B_{32}. Las figuras 39 y 40 muestran todas las posibles configuraciones de vértices válidas dadas las restricciones impuestas por $C_{B_{32}}$. Observe que si selecciona cualquier configuración mostrada en la Figura 39, no se podría marcar el vértice $v_{32,0} = r = 0$, así que estas configuraciones quedan descartadas.

Cualquiera de las configuraciones de vértices que se muestran en la Figura 40 permiten que el vértice raíz pueda marcarse y con esto, aumenta la cardinalidad del conjunto. Por el proceso de desempate, el algoritmo conserva la configuración de vértices con el menor identificador (ver la Figura 40a). Observe también que el vértice $\{6\}$ que anteriormente no se marcó en $C_{B_{32}}$, ahora ya pertenece al CIF.

En la Figura 41 se muestra el árbol T_B completo con los vértices marcados para el CIF. Finalmente, la Figura 42 presenta el marcado de vértices para el CIF sobre el cactus.

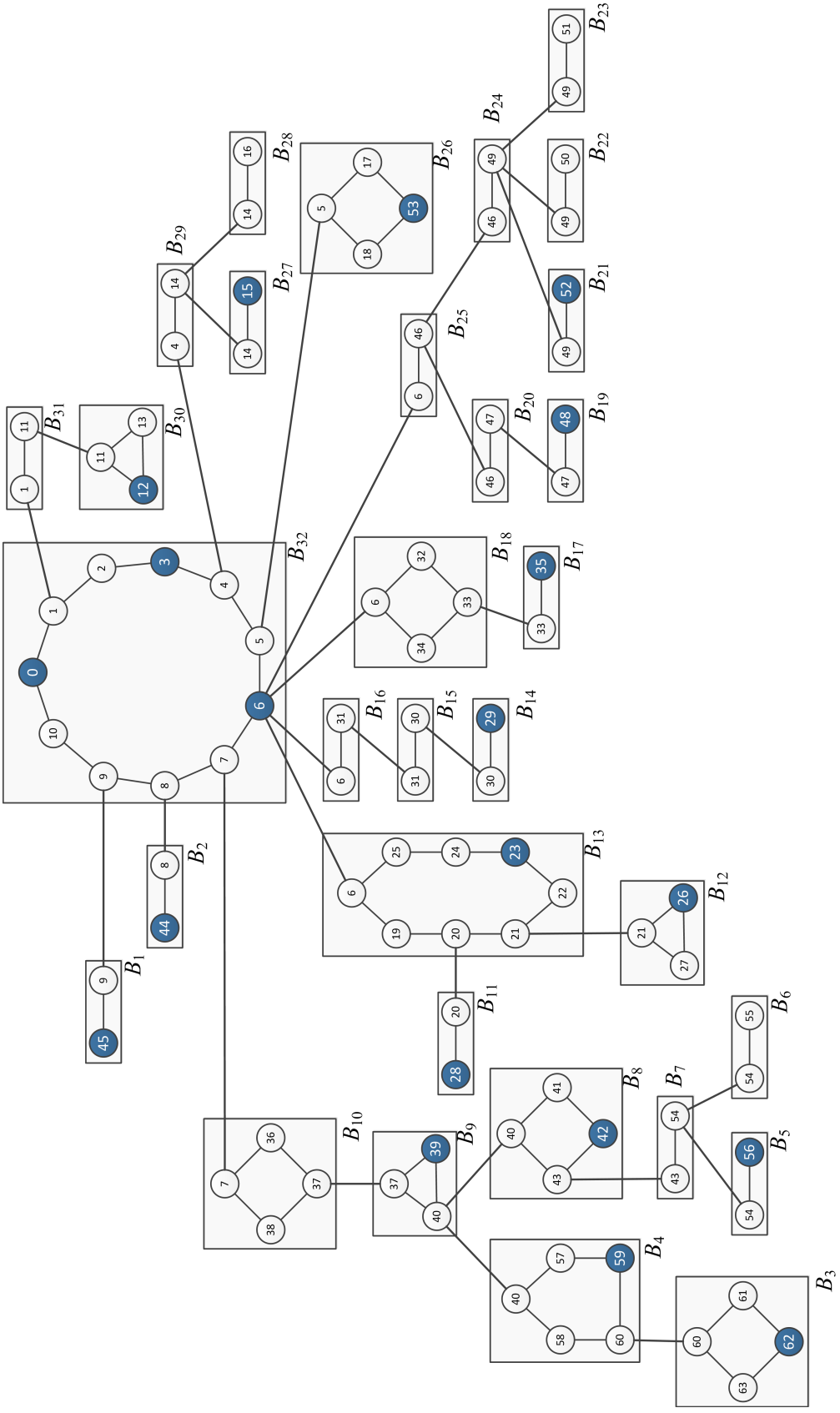

Figura 41: Árbol T_B completo con todos los vértices marcados para el CIF.

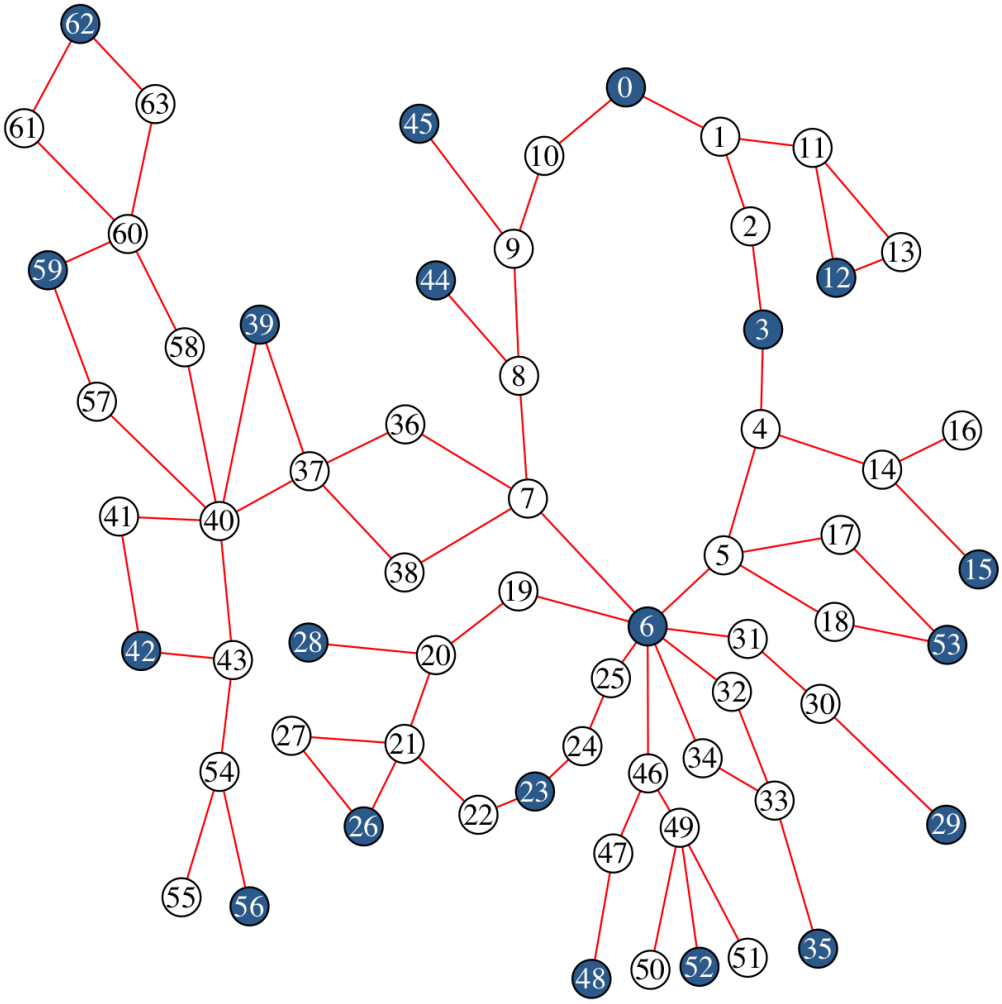

Figura 42: Marcado de vértices sobre el cactus.

Capítulo 4. Resultados

4.1. Introducción

Esta sección presenta las herramientas computacionales empleadas tanto para la creación de los grafos como para la codificación y prueba del algoritmo propuesto en esta tesis, así como otros encontrados en la literatura. Para corroborar el funcionamiento del algoritmo CIF-CACTUS-MAXIMUM se construyeron dos conjuntos de pruebas. El primero contiene grafos tipo cactus que se encuentran en la literatura: (Mjelde, 2004; Hedetniemi *et al.*, 1986; Trejo-Sánchez y Fernández-Zepeda, 2012); el marcado de vértices para el CIF máximo de estos cactus ya se conoce y se usó para comparar los resultados de estos contra los generados por nuestro algoritmo. El segundo conjunto contiene 120 grafos cactus generados aleatoriamente mediante la codificación y uso de rutinas en lenguaje C, scripts en R y Java. Cada grafo se encuentra compuesto aproximadamente por 4096 vértices y sirvieron como entrada para el algoritmo de conjunto k-packing de Mjelde (2004) para el algoritmo del conjunto dominante mínimo de Hedetniemi *et al.* (1986) y para el CIF-CACTUS-MAXIMUM; nuevamente, se comparan los resultados.

El capítulo finaliza discutiendo cuestiones relacionadas a la solución del CIF máximo sobre el grafo cactus.

4.2. Software de apoyo y simuladores

Dentro de las herramientas computacionales empleadas se encuentra el software *R project for Statistical Computing* (*R*) el cual se encuentra en (R Core Team, 2013). Asimismo, se empleó la biblioteca igraph (Csardi y Nepusz, 2006). R es un lenguaje de programación y ambiente de desarrollo para realizar cómputo estadístico y gráfico. Por otra parte, igraph es una colección de paquetes de software y biblotecas que permite trabajar con diversos elementos de teoría de grafos. De acuerdo a la página principal de igraph, las bibliotecas base que lo componen están codificadas tanto en lenguaje C como C++.

Las herramientas comerciales anteriores, así como las que se describen en las Secciones 4.2.1 y 4.2.2, se emplearon para codificar, visualizar y guardar el archivo fuente de los diferentes grafos con los que se trabaja.

4.2.1. Simulador BAP

El *Simulador BAP* fue creado por el grupo de algoritmos del Departamento de Ciencias de la Computación que pertenece a la División de Física Aplicada en CICESE. El simulador se diseñó para trabajar sobre un problema de teoría de juegos conocido como el problema de "asignación de guardaespaldas", la descripción de este trabajo se encuentra en (Zatarain Aceves, 2011). Los autores del Simulador BAP son: *Brizuela C.A., Brubeck D., Fajardo D., Fernandez J.A., Zatarain H.* La codificación se realizó en lenguaje JAVA y se encuentra integrado por los paquetes *Algorithms, Data, Enums, Experiments, Graph, Utils,* y *bap_simulator*, donde de forma respectiva, se encuentran los algoritmos para resolver las diferentes configuraciones del problema de asignación de guardaespaldas, la forma de cómo guardar los resultados obtenidos en hojas de Microsoft Excel, valores enumerados que toman las estructuras, el archivo de configuración que indica los experimentos realizar, las clases para representar al grafo y finalmente, código auxiliar para lectura y escritura de los archivos fuente del grafo.

Específicamente, dentro del paquete Graph se encuentran las clases de java Node.java y Graph.java. La primera contiene atributos y métodos necesarios para representar e identificar un vértice en el grafo mientras que la segunda clase incluye la estructura del grafo, así como procedimientos para almacenar, borrar, y acceder a los vértices.

Para realizar la codificación del algoritmo CIF-MAXIMUM-CACTUS fue necesario cambiar algunos elementos del Simulador BAP. La clase Graph.Graph.java ahora acepta herencia de clases tipo $<$T$>$ genérico, de esta forma soporta diversos tipos de vértices con propiedades distintas. Asimismo, se agregaron métodos adicionales para enraizar el grafo, realizar recorridos (preorden, postorden, anchura y profundidad), encontrar componentes biconectados, ciclos de vértices, borrado de aristas, y otros métodos auxiliares. La clase Graph.Node.java también se modificó para aceptar herencia de clases genéricas. De esta forma, se elimina la necesidad de adaptar los tipos de datos que se emplean. Las tablas 3 y 4 muestran los atributos de la clase Graph.Graph.java y Graph.Node.java.

Tabla 3: Atributos de la clase Node.class del Simulador BAP.

Atributo	Tipo de dato	Descripción
col	$Z = \{ 0, 1, 2 \}$	Utilizado para realizar la búsquedas en anchura y profundidad.
d	Z^+	Indica el momento en que se descubre el vértice durante DFS.
pi	Z^+	Identificador del vértice padre, empleado únicamente para BFS y DFS.
id	Z^+	Identificador único del vértice.
$distance$	Z^+	Distancia del vértice $v_{i,j}$ a $v'_{i,k}$.
$color$	Enum	Mantiene la lista de colores que puede tomar el vértice.
$parent$	Node<E>	Apuntador hacia el vértice padre.
$neighbors$	Lista	Contiene el vecindario abierto del vértice.
$label$	String	Información adicional del vértice.

Tabla 4: Atributos de la clase Graph.class del Simulador BAP.

Atributo	Tipo de dato	Descripción
ρ	Z^+	Cardinalidad del CIF
$listBFS$	Lista	Lista los vértices del recorrido en anchura sobre el grafo.
$ciclo$	Lista	Mantiene los ciclos encontrado del grafo.
$Block$	Lista	Contiene los componentes biconectados del grafo.
restricción	Booleano	Valor lógico que se activa si se reciben restricciones.
$listaPo$	Lista	Orden los vértices en el recorrido postorden.
tE	Lista	Orden de los vértices en el tour de Euler.

Asimismo, se agregaron nuevas clases al paquete Graph con propiedades y comportamientos para representar nuevos vértices y grafos dependiendo de las necesidades de los diferentes algoritmos. Dichas clases se describen a continuación:

1. *NodoComun*. Representación básica de un vértice en el grafo, está integrada por las características y comportamiento de la clase Node.java.

2. *NodeB*. Es la especialización de la clase Node.java. Contiene atributos y métodos para trabajar con el grafo tipo árbol cuyos vértices son componentes biconectados.

3. *NodeM*. Hereda los atributos y comportamiento de la clase Node.java, además,

tiene los elementos y comportamientos requeridos para el algoritmo del conjunto k-packings (Mjelde, 2004).

4. *NodoH*. También hereda los atributos y comportamiento de la clase Node.java, además de tener las propiedades necesarias para trabajar con el algoritmo de conjunto dominante mínimo sobre un árbol y un cactus que se puede encontrar en (Hedetniemi *et al*., 1986).

5. *GraphB*. Esta clase provee los metodos para encontrar los vecinos izquierdo y derecho de un vértice. Además, hereda (atributos y comportamiento) de la clase Graph.java y forma parte de la clase Block.java. Adicionalmente, el método distancia de Graph.java se sobrescribió para calcular la trayectoria más corta desde cualquier vértice hacia el vértice marcado más cercano, es decir, para obtener la dist($v_{i,j},\ v'_{i,j}$).

6. *GraphH*. Al igual que GraphB, hereda las propiedades de Graph y contiene la estructura para trabajar con el grafo requerido por Hedetniemi *et al.* (1986).

7. *Block*. Esta clase contiene los métodos necesarios para acceder y manipular la clase GraphB.

8. *BlockH*. Implementa los métodos para acceder a la clase GraphH. Adicionalmente, contiene el método MCyle, que es parte del algoritmo de Hedetniemi *et al.* (1986).

Además de codificar la estructura del grafo, se codificaron las siguientes clases:

1. *RLauncher*. Ejecuta la consola de R mediante instrucciones propias del sistema operativo y muestra en la pantalla el grafo que con el que se está trabajando.

2. *DomSet*. Codificación del algoritmo de Hedetniemi *et al.* (1986) para encontrar el conjunto dominante de cardinalidad mínima sobre un grafo tipo árbol.

3. *MCactus*. Implementación del algoritmo de Hedetniemi *et al.* (1986) para encontrar el conjunto dominante mínimo sobre un grafo cactus.

4. *Mjelde*. Contiene el algoritmo propuesto por Mjelde (2004) para encontrar el k-packing máximo en grafos tipo árbol. Por el momento, el valor de k está fijo en 2 para encontrar el CIF.

5. *CIF_Total*. Implementación del Algoritmo 4 para encontrar el CIF máximo sobre un grafo cactus. Esta clase se apoya tanto de la codificación del Simulador_BAP como la codificación de CIF_Subgrafo.

6. *CIF_Subgrafo*. Encuentra el CIF máximo de un componente $B_i \in T_B$. Es la implementación del Algoritmo 5.

4.2.2. Generadores de los casos de prueba

Los 120 grafos de prueba se crearon mediante las rutinas que se describen en esta sección. Tal como se mencionó en la Sección 4.1, cada grafo contiene aproximadamente 4096 vértices. La razón detrás de esta aproximación se debe a que el número de vértices por componente (ciclos de vértices, número mínimo y máximo de hijos) se genera aleatoriamente. Ahora se presenta de forma muy breve las rutinas empleadas, así como su pseudocódigo.

1. *RandomTree.c* Es una adaptación del código encontrado en (Nepusz, 2010). El programa está codificado en lenguaje C e incluye la librería *igraph.h* para manejar el grafo generado como objeto de *igraph*. La firma del programa recibe 3 parámetros de entrada: número deseado de vértices intermedios, el número mínimo de hijos por vértice y el número máximo de hijos por vértice. El pseudocódigo se presenta en el Algoritmo 10. La descripción de las funciones IGRAPH_CHECK y igraph_create se encuentra en (Csardi y Nepusz, 2006).

2. *generaCadenas.java.* Este programa hace uso de los recursos del *Simulador BAP*. Como parámetros de entrada recibe el número mínimo y máximo de vértices que la cadena de anillos debe tener. Posteriormente, el programa genera aleatoriamente los vértices en el grafo y agrega las aristas pertinentes para generar un ciclo de vértices. Si no se excede el número de vértices requeridos, entonces, aleatoriamente se escoge un vértice del último ciclo generado (de tal forma que dicho vértice no se comparta con ningún otro ciclo) y sobre éste se construye el siguiente ciclo del grafo. Si se excede el número máximo de vértices que se solicitó, entonces se destruye el último ciclo y se genera otro aleatoriamente (ver el Algoritmo 11).

Algoritmo 10: RandomTree para generar árboles aleatorios.

Entrada : $numVerticesIntermedios$, $minChilden$, $maxChilden$.
Salida : Un grafo árbol con los parámetros especificados.

begin
  Sean:
  T un árbol vacío
  $nP = 0$ el identificador del vértice padre
  $nC = 1$ el identificador de un vértice hijo
  $aristas$ un vector con la lista de aristas
  while $nP < numVerticesIntermedios$ **do**
    $numChilden \leftarrow random()\,\%(maxChilden - minChilden + 1) - minChilden$
    if $nP \geq nC$ **then**
      ERROR
    else
      for $0 \leq i \leq nC$ **do**
        IGRAPH_CHECK(igraph_vector_push_back($aristas$, nP))
        IGRAPH_CHECK(igraph_vector_push_back($aristas$, nC))
        $nC++$
      end
    end
    $nP++$
  end
  igraph_create(T, $aristas$, 0, 0)
  return T
end

Algoritmo 11: generaCadenas.

Entrada : $minVertices$, $maxVertices$.
Salida : Una cadena de vértices con los parámetros especificados.

begin
  Sea n un número aleatorio con $minVertices \leq n \leq maxVertices$
  Cree un ciclo con n vértices y agréguelo a G
  Sean:
  u un vértice cualquiera de G
  v el vértice de G con el mayor identificador
  while $minVertices \leq |G| \leq maxVertices$ **do**
    $n = random()$
    Iniciando en el vértice u agregue un ciclo de n vértices
    Elija un nuevo vértice u de forma aleatoria tal que $N(u) = 2$
  end
  return G
end

3. *generaAnillos.java*. Se basa en la codificación de generaCadenas.java en cuanto a los parámetros de entrada, código en común y el uso del *Simulador BAP*. La diferencia es que esta clase genera un ciclo de vértices de forma recursiva, el cual alimenta otro método auxiliar llamado *generador* que construye anillos perimetrales al anillo base. Asimismo, cada ciclo adicional se pasa al método *generador* para continuar con el proceso anterior. La recursividad se detiene cuando se alcanza el intervalo de vértices deseado en el grafo.

4. *RandomCactus.R* es cortesía de Trejo-Sánchez y Fernández-Zepeda (2012). Dicha rutina se basa en el modelo Erdös-Rényi para generar grafos aleatorios; posteriormente se utiliza búsqueda en profundidad (DFS) y mediante probabilidad se eliminan aristas para generar los cactus.

Cada una de estas rutinas genera como salida un archivo con extensión *.gml* el cual consiste del conjunto de vértices con sus parámetros de identificador, etiqueta y color (el color asignado es blanco, por omisión), también contiene el conjunto de aristas.

4.3. Detalles de implementación

Ahora que se han presentado las herramientas computacionales auxiliares en este trabajo y se ha descrito el algoritmo CIF-MAXIMUM-CACTUS, se proveen detalles más prácticos de su implementación. Como lo indica la Sección 4.2.1, el algoritmo propuesto se implementó en lenguaje JAVA, específicamente con la versión 1.8.0_05. En esta sección se describen las estructuras y tipos de datos empleados en la codificación.

La estructura de datos que alberga a los vértices es un mapa hash cuya llave es un entero y contiene elementos de la clase node de tipo genérico $<$T$>$. Asimismo, otro mapa hash almacena las aristas del grafo, cuya llave es un String de la forma $u - v$, donde u es el vértice extremo de menor identificador y v el otro extremo de la arista. Para facilitar la identificación de la etiqueta menor se utiliza el método Utils.Utils.getEdgeKey(int u, int v), cuyo parámetro de retorno es precisamente la llave identificadora de la arista. El motivo por el cual se ha utilizado el mapa hash, es debido a la habilidad de soportar las operaciones de diccionario de insertar, buscar y borrar; además de poder hacer uso

de prestaciones en cuanto a simplicidad de manejo para acceder directamente a alguna posición en el mapa en tiempo constante. De acuerdo a (Cormen *et al.*, 2009), el buscar un elemento en el mapa hash puede requerir $O(n)$ u. t. en el peor de los casos pero por lo general el desempeño es $O(1)$ u. t.

Para mantener el orden de aparición de los vértices durante los diversos recorridos, se emplean estructuras dinámicas tipo *ArrayList* con la finalidad de tener los elementos ordenados en forma lineal y mantener las propiedades de un vector de datos, tal como acceder a un elemento en particular por medio de su índice. Otra estructura empleada es la lista ligada, esto para poder hacer uso del comportamiento de pila y cola necesarios para realizar las búsquedas BFS y DFS.

En cuanto a la asignación de restricciones y distancias, los algoritmos 4 y 8 mencionan el valor infinito; este valor se simula mediante el método estático Integer.MAX_VALUE que pertenece a la clase java.lang.Integer cuyo valor es $2^{31} - 1$.

Finalmente, el resultado generado por el algoritmo se guarda en una hoja de Microsoft Excel que contiene el nombre del archivo del grafo de entrada, tipo de grafo, el número de vértices y componentes biconectados, así como la cardinalidad del CIF. Lo anterior se logra mediante el uso del open source *Java Excel Api* (http://jexcelapi.sourceforge.net), que es un conjunto de bibliotecas codificadas en JAVA para escribir y modificar hojas de datos de Excel.

4.4. Casos de prueba

Como se menciona en la introducción de este capítulo, se cuenta con 2 conjuntos de grafos de prueba: aquellos grafos encontrados en la literatura y que su CIF máximo es conocido y para el segundo conjunto, se generaron 30 grafos tanto de tipo árbol, cadena de ciclos, sistemas de engranes y cactus (estas configuraciones se muestran en la Figura 43) cada uno de ellos creado con las herramientas mencionadas en la Sección 4.2.2. El algoritmo CIF-CACTUS-MAXIMUM se alimentó con cada uno de estos grafos de la siguiente manera: se dividió el grafo en componentes biconectados y se utilizó cada uno de estos componentes como raíz del T_B dando un total de 237,087 árboles T_B distintos, sin contar que en cada componente biconectado, cada uno de los vértices se empleó como raíz.

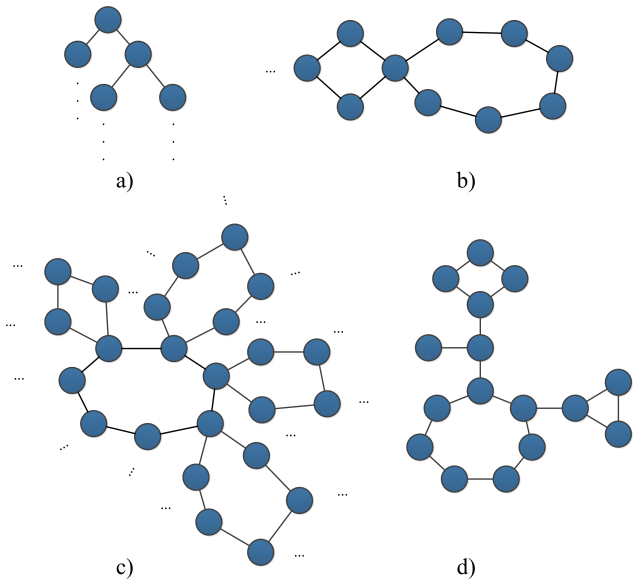

Figura 43: Grafos cactus. a) Árbol. b) Cadenas de ciclos. c) Sistemas de engranes. d) Cactus (combinación de diferentes configuraciones).

El motivo por el cual cada grafo se enraizó en cada componente biconectado y en cada vértice, es para asegurar que no importando la forma del cactus, el algoritmo es capaz de encontrar el CIF máximo sobre éste, y que la cardinalidad del mismo es igual en todos los casos. La notación empleada para nombrar los grafos es la siguiente:

1. Para el grafo árbol, el nombre es: t_a_b_c_d.gml, donde a es el número mínimo de hijos por vértice intermedio, b indica la cantidad deseada de vértices intermedios, c es el máximo número de vértices hijos por vértices intermedio, y d denota el número total de vértices en el grafo.

2. Las cadenas de vértices se identifican como XX_cadenas_$YYYY$.gml, donde XX es un número entre $\{01 - 30\}$ y $YYYY$ es $|V(G)|$.

3. Los sistemas de engranes se identifican como XX_engranes_$YYYY$.gml, la notación XX y $YYYY$ tiene el mismo significado que la empleada en las cadenas.

4. Los cactus se identifican mediante cac_$YYYY$_XX, y conservan el significado de las cadenas.

Las tablas 5 a 8 presentan información relacionada a los casos de prueba; mientras

que las figuras 44, 45 y 46 son una representación visual de los grafos evaluados. La disposición de vértices en dichas figuras utilizan los siguientes algoritmos de dibujo:

1. *Kamada-Kawai*, es un algoritmo dirigido por fuerzas que coloca los vértices del grafo de tal manera que las aristas se crucen lo menos posible (Kamada y Kawai, 1989).
2. *Reingold-Tilford* emplea la estrategia de divide y vencerás para encontrar una disposición de vértices en el grafo tal que el vértice padre esté centrado sobre sus hijos y que el grafo ocupe el menos espacio posible (Reingold y Tilford, 1981).

4.4.1. Resultados

El primer conjunto consta de grafos pequeños como el que se muestra en la Figura 47. Estos son grafos cuya cardinalidad de vértices es pequeña, por lo que incluso mediante una búsqueda exhaustiva es posible encontrar el CIF máximo. Al ejecutar el algoritmo CIF-MAXIMUM-CACTUS, el tamaño del conjunto encontrado contra el CIF reportado en este tipo de grafos es el mismo; no obstante, la selección de vértices difiere.

Para el segundo conjunto de prueba, los grafos tipo árbol se evaluaron tanto con algoritmos de conjunto dominante mínimo, conjunto k-packing y CIF-MAXIMUM-CACTUS. La cardinalidad encontrada por cada algoritmo en cada uno de estos grafos es la misma.

El resto de los grafos se evaluaron con el algoritmo de Hedetniemi *et al.* (1986) y el propuesto en esta tesis. Los resultados muestran que se mantiene la propiedad descrita en (Imrich *et al.*, 2008), la cual indica que para cualquier grafo $G = (V, E)$, la cardinalidad del conjunto dominante mínimo es mayor o igual que la cardinalidad del CIF máximo, es decir $\gamma(G) \geq \rho(G)$. Satisfactoriamente, no importando la forma del árbol de componentes biconectados T_B, ni el componente raíz seleccionado, la cardinalidad del CIF es siempre igual, aunque la selección de vértices varía. Las figuras 48 y 49 presentan una gráfica comparativa que relaciona los grafos evaluados y la cardidalidad de los conjuntos encontrados; *CDM* denota al conjunto dominante mínimo. La relación observada en cuanto a vértices marcados contra el número de vértices del grafo se muestra en las tablas 9 y 10.

Tabla 5: Casos de prueba del grafo tipo árbol.

Grafo	$\|V(G)\|$	**Componentes**	$\|S\|$	$\|V(G)\|$ / $\|S\|$
t_0_16_512_4056.gml	4057	4056	447	0.110179936
t_11_31_199_4284.gml	4285	4284	191	0.044574096
t_1_10_777_4292.gml	4293	4292	637	0.148381085
t_1_11_677_4007.gml	4008	4007	565	0.140968064
t_1_11_709_4324.gml	4325	4324	598	0.138265896
t_1_11_857_5126.gml	5127	5126	707	0.137897406
t_1_2_3000_4510.gml	4511	4510	1440	0.319219685
t_1_32_256_4354.gml	4355	4354	240	0.05510907
t_1_3_2000_4008.gml	4009	4008	1133	0.282614118
t_1_3_2048_4126.gml	4127	4126	1175	0.284710443
t_1_7_1024_4277.gml	4278	4277	777	0.181626928
t_2_13_587_4469.gml	4470	4469	518	0.115883669
t_2_32_256_4633.gml	4634	4633	240	0.051791109
t_2_64_128_4239.gml	4240	4239	125	0.029481132
t_2_7_1000_4484.gml	4485	4484	784	0.174804905
t_2_7_999_4489.gml	4490	4489	788	0.175501114
t_2_8_1000_4986.gml	4987	4986	804	0.16121917
t_3_12_640_4836.gml	4837	4836	555	0.114740542
t_3_5_997_3975.gml	3976	3975	763	0.191901408
t_3_5_999_4008.gml	4009	4008	760	0.18957346
t_3_7_701_3486.gml	3487	3486	572	0.164037855
t_3_7_811_4059.gml	4060	4059	656	0.161576355
t_3_7_888_4417.gml	4418	4417	718	0.162516976
t_3_7_997_5063.gml	5064	5063	806	0.159162717
t_3_8_768_4128.gml	4129	4128	625	0.15136837
t_3_9_666_4090.gml	4091	4090	564	0.137863603
t_4_128_64_4520.gml	4521	4520	64	0.01415616
t_4_21_320_4118.gml	4119	4118	295	0.071619325
t_59_97_53_4095.gml	4096	4095	53	0.012939453
t_7_11_460_4156.gml	4157	4156	410	0.098628819

Tabla 6: Casos de prueba del grafo tipo cadenas.

Grafo	$\|V(G)\|$	**Componentes**	$\|S\|$	$\|V(G)\| \ / \ \|S\|$
01_cadenas_4050.gml	4050	577	1236	0.305185185
02_cadenas_4051.gml	4051	596	1236	0.305109849
03_cadenas_4050.gml	4050	570	1229	0.30345679
04_cadenas_4054.gml	4054	576	1240	0.305870745
05_cadenas_4054.gml	4054	599	1236	0.304884065
06_cadenas_4053.gml	4053	574	1235	0.304712559
07_cadenas_4051.gml	4051	588	1234	0.304616144
08_cadenas_4054.gml	4054	556	1231	0.303650715
09_cadenas_4053.gml	4053	575	1232	0.303972366
10_cadenas_4060.gml	4060	551	1237	0.304679803
11_cadenas_4052.gml	4052	594	1228	0.303060217
12_cadenas_4055.gml	4055	578	1230	0.303329223
13_cadenas_4057.gml	4057	578	1235	0.304412127
14_cadenas_4051.gml	4051	578	1235	0.304862997
15_cadenas_4053.gml	4053	584	1225	0.30224525
16_cadenas_4056.gml	4056	595	1223	0.3015286
17_cadenas_4050.gml	4050	576	1244	0.307160494
18_cadenas_4059.gml	4059	578	1238	0.305001232
19_cadenas_4050.gml	4050	590	1227	0.302962963
20_cadenas_4054.gml	4054	586	1228	0.302910705
21_cadenas_4050.gml	4050	574	1229	0.30345679
22_cadenas_4053.gml	4053	584	1228	0.302985443
23_cadenas_4051.gml	4051	574	1230	0.303628734
24_cadenas_4053.gml	4053	573	1237	0.30520602
25_cadenas_4051.gml	4051	580	1231	0.303875586
26_cadenas_4053.gml	4053	569	1236	0.304959289
27_cadenas_4054.gml	4054	589	1230	0.303404045
28_cadenas_4052.gml	4052	591	1229	0.303307009
29_cadenas_4053.gml	4053	583	1231	0.303725635
30_cadenas_4055.gml	4055	581	1240	0.305795314

Tabla 7: Casos de prueba del grafo tipo engranes.

Grafo	$\|V(G)\|$	**Componentes**	$\|S\|$	$\|V(G)\| \ / \ \|S\|$
01_engranes_4098.gml	4098	901	1136	0.277208394
02_engranes_4053.gml	4053	910	1119	0.276091784
03_engranes_4061.gml	4061	912	1121	0.276040384
04_engranes_4114.gml	4114	910	1144	0.278074866
05_engranes_4129.gml	4129	938	1129	0.273431824
06_engranes_4113.gml	4113	919	1131	0.274981765
07_engranes_4068.gml	4068	886	1126	0.276794494
08_engranes_4093.gml	4093	918	1128	0.275592475
09_engranes_4119.gml	4119	907	1144	0.277737315
10_engranes_4106.gml	4106	918	1139	0.277398928
11_engranes_4148.gml	4148	929	1151	0.277483124
12_engranes_4104.gml	4104	919	1140	0.277777778
13_engranes_4108.gml	4108	905	1144	0.278481013
14_engranes_4072.gml	4072	914	1123	0.275785855
15_engranes_4051.gml	4051	895	1120	0.276474944
16_engranes_4109.gml	4109	912	1132	0.275492821
17_engranes_4142.gml	4142	945	1145	0.276436504
18_engranes_4052.gml	4052	921	1113	0.274679171
19_engranes_4085.gml	4085	896	1134	0.277600979
20_engranes_4130.gml	4130	919	1148	0.277966102
21_engranes_4064.gml	4064	905	1128	0.277559055
22_engranes_4094.gml	4094	926	1131	0.276257938
23_engranes_4111.gml	4111	896	1141	0.277548042
24_engranes_4078.gml	4078	917	1139	0.27930358
25_engranes_4107.gml	4107	899	1130	0.275140005
26_engranes_4110.gml	4110	920	1144	0.278345499
27_engranes_4148.gml	4148	911	1152	0.277724204
28_engranes_4101.gml	4101	918	1135	0.276761765
29_engranes_4081.gml	4081	896	1132	0.277382994
30_engranes_4117.gml	4117	926	1143	0.277629342

Tabla 8: Casos de prueba del grafo tipo cactus.

Grafo	$\|V(G)\|$	**Componentes**	$\|S\|$	$\|V(G)\| \ / \ \|S\|$
cac_4096_19.gml	4096	2096	1276	0.311523438
cac_4096_20.gml	4096	2078	1276	0.311523438
cac_4096_29.gml	4096	2130	1277	0.311767578
cac_4096_6.gml	4096	2053	1283	0.313232422
cac_4098_4.gml	4098	2105	1284	0.313323572
cac_4098_7.gml	4098	2106	1281	0.312591508
cac_4099_10.gml	4099	2065	1285	0.313491095
cac_4099_21.gml	4099	2084	1277	0.3115394
cac_4099_25.gml	4099	2059	1290	0.314710905
cac_4099_3.gml	4099	2076	1266	0.308855818
cac_4101_2.gml	4101	2132	1291	0.314801268
cac_4102_14.gml	4102	2120	1288	0.313993174
cac_4102_23.gml	4102	2113	1282	0.312530473
cac_4102_30.gml	4102	2048	1281	0.312286689
cac_4102_5.gml	4102	2043	1271	0.309848854
cac_4103_17.gml	4103	2034	1281	0.312210578
cac_4103_26.gml	4103	2078	1284	0.31294175
cac_4104_13.gml	4104	2072	1283	0.312621832
cac_4104_15.gml	4104	2035	1279	0.311647173
cac_4104_22.gml	4104	2119	1273	0.310185185
cac_4105_16.gml	4105	2108	1288	0.313763703
cac_4105_24.gml	4105	2162	1287	0.313520097
cac_4105_28.gml	4105	2084	1273	0.310109622
cac_4105_8.gml	4105	2119	1277	0.311084044
cac_4106_12.gml	4106	2067	1286	0.313200195
cac_4106_9.gml	4106	2083	1271	0.309547004
cac_4109_1.gml	4109	2074	1287	0.313214894
cac_4109_11.gml	4109	2087	1294	0.314918472
cac_4111_27.gml	4111	2142	1287	0.313062515
cac_4112_18.gml	4112	2115	1283	0.312013619

Tabla 9: Relación tamaño del CIF contra tamaño del grafo.

Grafo:	Árbol	Cadenas	Engranes
Máximo	0.319219685	0.307160494	0.27930358
Mínimo	0.012939453	0.3015286	0.273431824
Promedio	0.139410429	0.304131863	0.276839431

Tabla 10: Relación tamaño del CIF contra tamaño del grafo, continuación.

Grafo:	Cactus	**Concentrado**
Máximo	0.314918472	0.319219685
Mínimo	0.308855818	0.012939453
Promedio	0.312335344	0.258179267

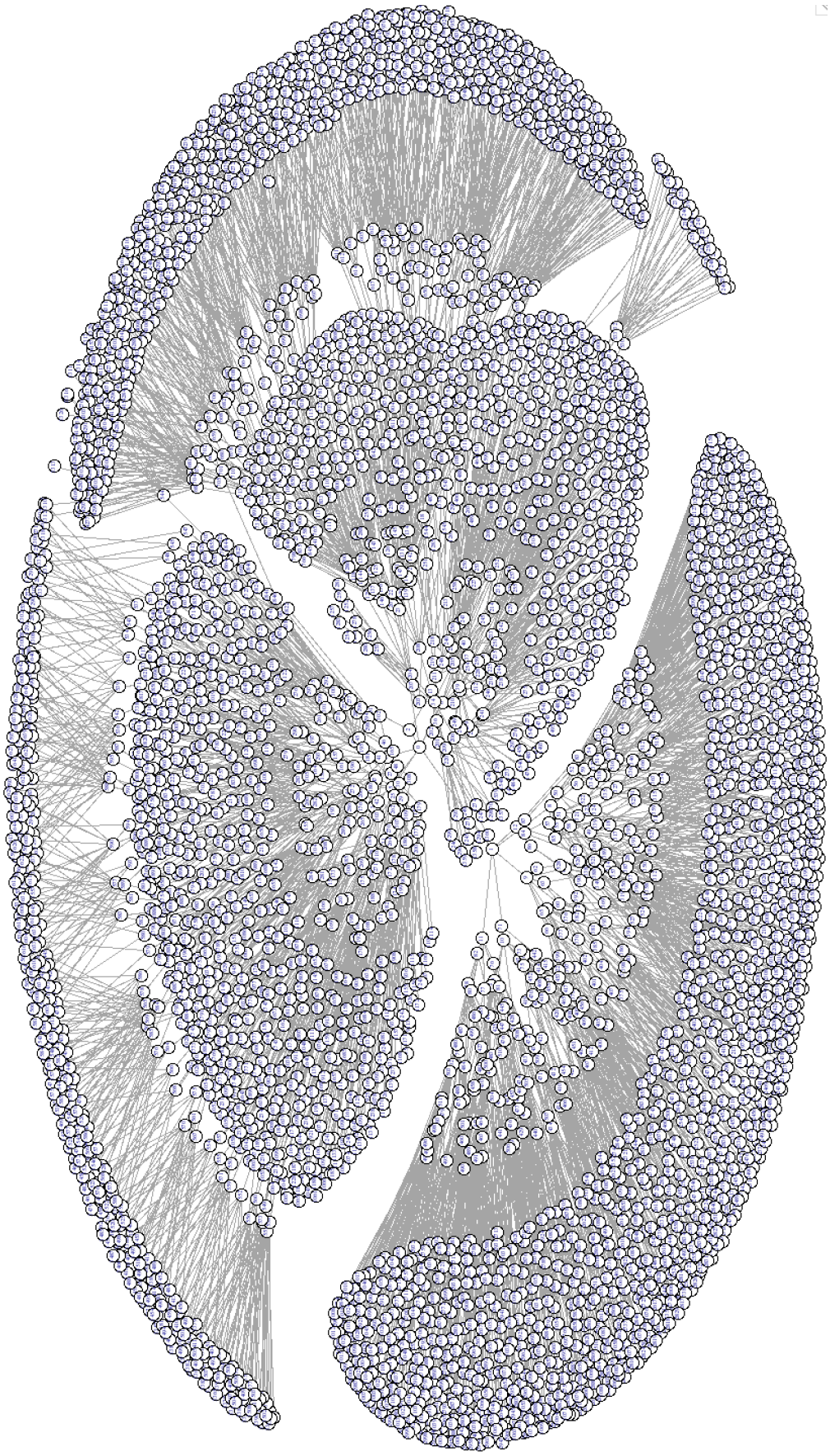

Figura 44: Grafo t_1_11_709_4324.gml, la disposición de los vértices que se emplea es Kamada-Kawai.

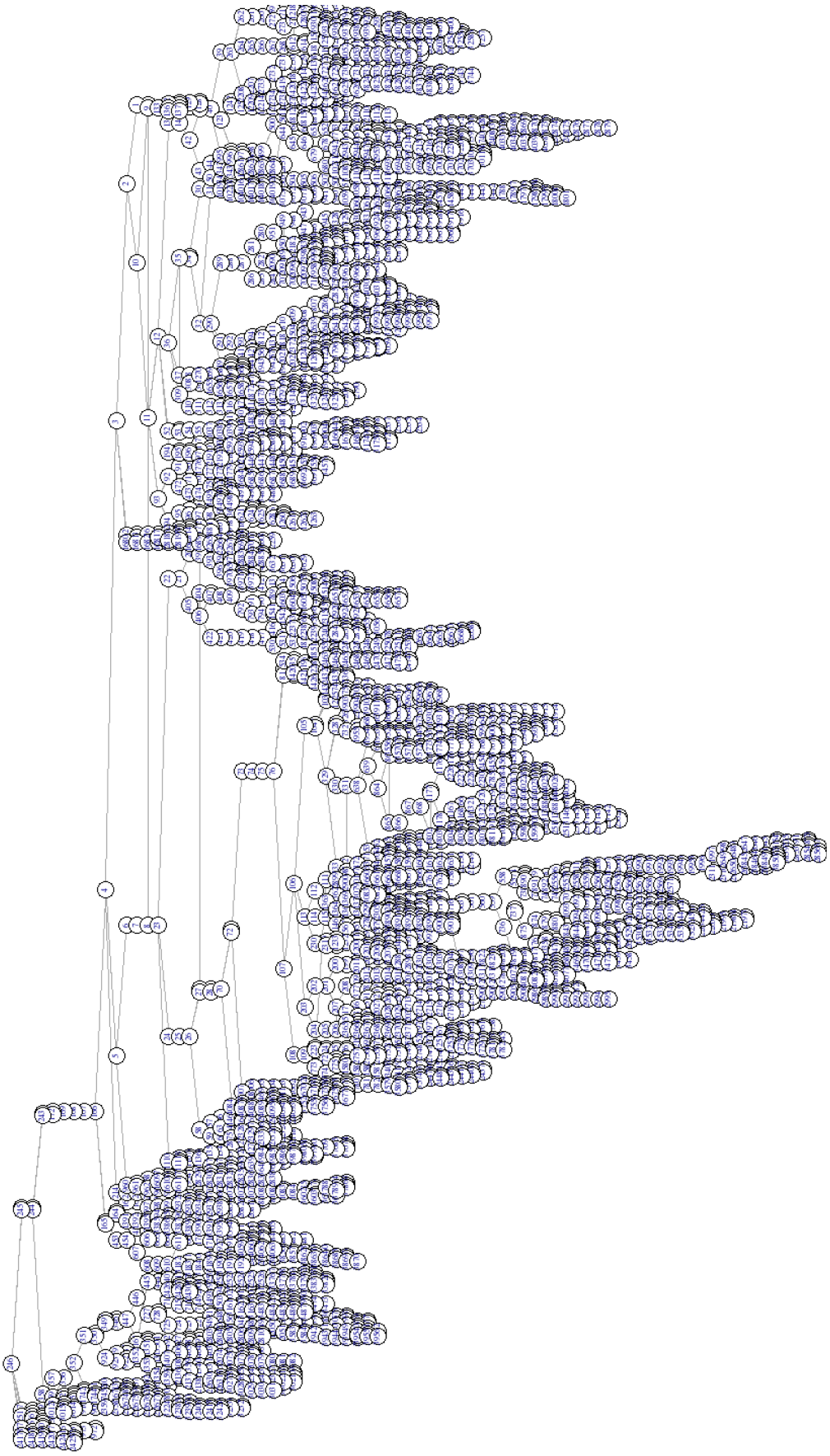

Figura 45: Grafo cactus de 4096 vértices, la disposición de los vértices empleada es Reingold-Tilford.

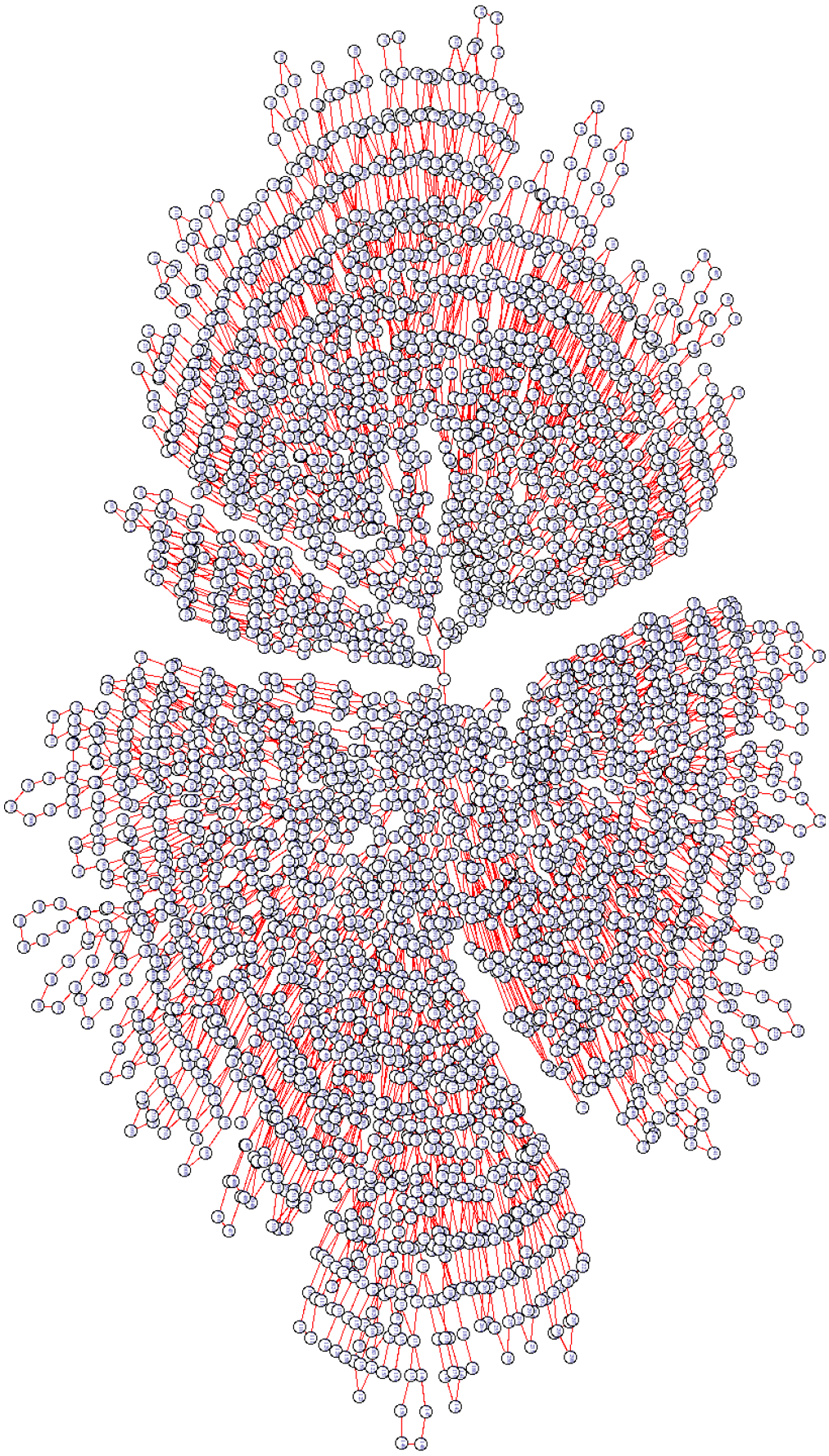

Figura 46: Grafo de sistemas de engranes con 4148 vértices en disposición Kamada-Kawai.

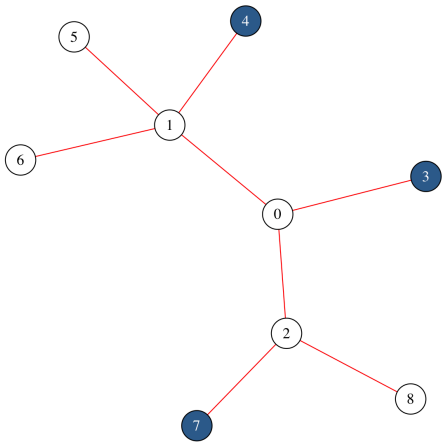

Figura 47: Árbol con $|V| = 9$ y $r = 3$. Los vértices que pertenecen al CIF máximo se encuentran marcados.

4.5. Discusión

Se ha presentado un algoritmo para encontrar el conjunto independiente máximo sobre un grafo cactus, así como algunos detalles de su implementación. Ahora, en esta sección se discuten las interrogantes que guiaron el transcurso del presente trabajo de investigación.

La característica que describe a un CIF, es que la distancia mínima entre cualquier par de vértices marcados es al menos de 3 aristas. No obstante, el marcado de vértices a distancia 3 no debe ser arbitrario, puesto que se cae en el riesgo de crear un conjunto maximal en lugar de uno máximo. Debido a lo anterior, se debe seguir un procedimiento ordenado e informado para marcar los vértices del grafo. Como se mencionó en la Sección 2.2, existen algoritmos que dividen el grafo por secciones como lo son (Hedetniemi *et al.*, 1986; Das, 2012). También existen algoritmos que, mediante recorridos sobre grafos acíclicos y de forma ordenada, calculan el conjunto máximo hasta cierto punto o vértice en el grafo, tal es el caso del algoritmo descrito en la Sección 2.2.2 y el trabajo de (Mjelde, 2004). Una de las preguntas que participaron en el proceso para generar el algoritmo es si existe la necesidad o no de romper los ciclos presentes en el cactus y cómo llevarlo a cabo. Una forma de lograr lo anterior es eliminando aristas del ciclo, para así obtener una estructura similar a la de un árbol. No obstante, esto genera el inconveniente de que los vértices extremos a los que se les elimina la arista, desconocen que en realidad son vecinos. Por lo tanto, al borrar la arista entre ellos, existe la posibilidad

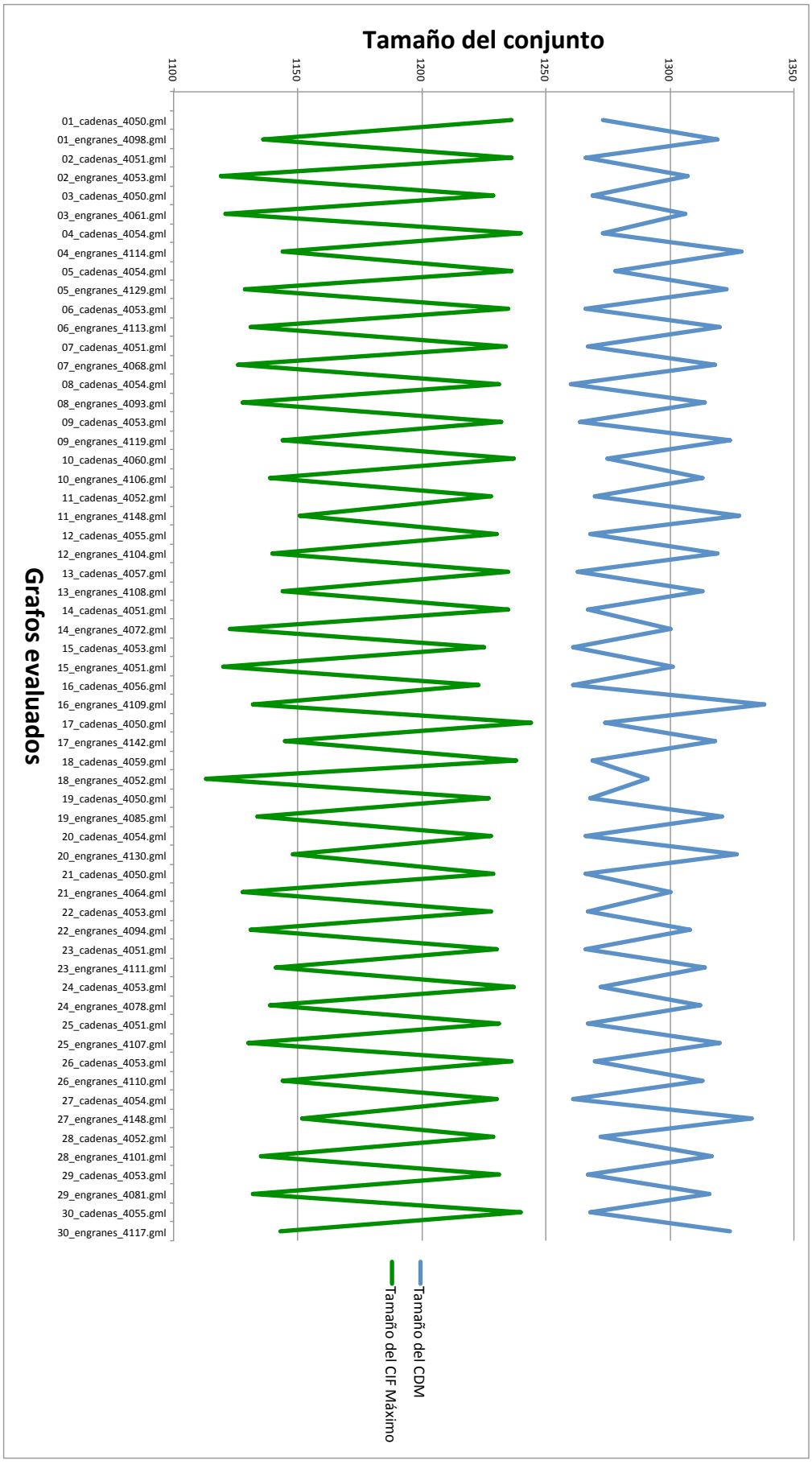

Figura 48: CDM contra el CIF máximo.

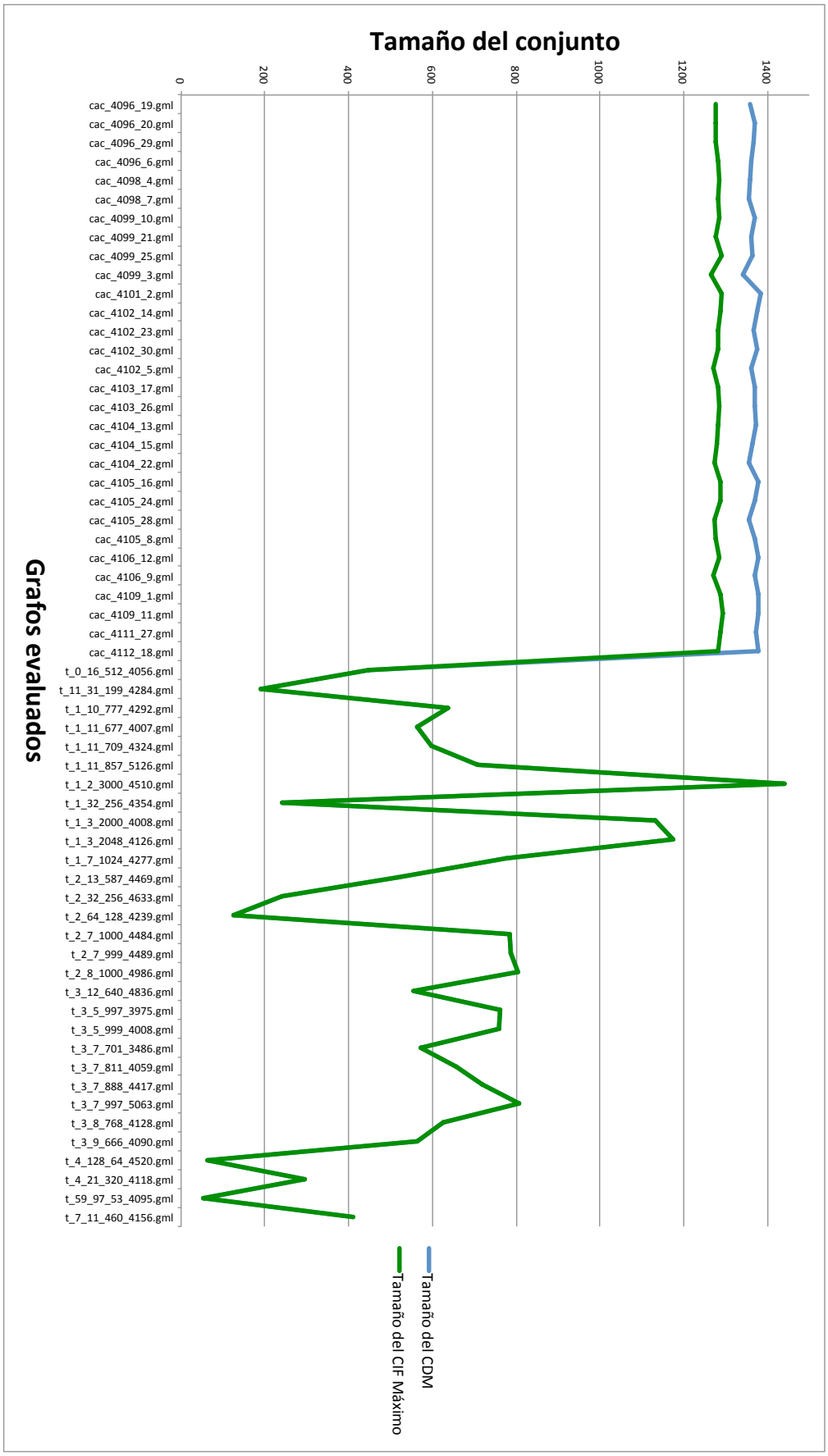

Figura 49: CDM contra el CIF máximo. En los grafos tipo árbol, las cardinalidades de los conjuntos son iguales.

que ambos sean marcados para el conjunto, lo cual viola las propiedades del CIF, puesto que por ser vecinos la distancia entre ambos es 1.

Otra forma para romper ciclos que fue considerada es la de eliminar tanto un vértice del mismo como todas sus aristas adyacentes, este procedimiento se sigue en el trabajo de Hedetniemi *et al.* (1986). Por ejemplo, sean $\{...u, v, w, x, y...\}$ vértices que pertenecen a un ciclo arbitrario del grafo, tal que v tiene aristas hacia u y hacia w; el vértice w tiene también arista hacia x, y así sucesivamente. Si se elimina el vértice w, el ciclo se descompone y la $dist(v, x) = 2$, puesto que se eliminan 2 aristas. Nuevamente, existe el riesgo de que los vértices v, x puedan ser marcados y de esta forma se viola la propiedad de distancia entre vértices del CIF.

En la literatura es posible encontrar otros métodos para romper los ciclos presentes en el grafo, tal como manejar aristas dirigidas en lugar de no dirigidas, evadir ciertos vértices e incluso aristas para así generar trayectorias acíclicas. Sin importar cual fuera la metodología, se obtendría un segundo grafo G' tal que $V(G') \subseteq V(G)$ y $E(G') \subseteq E(G)$. Entonces, surge la inquietud de por cuál arista o vértice iniciar, qué conjunto de vértices o aristas borrar, conservar o qué combinaciones realizar para obtener el CIF máximo. Además, se observó que G' tendría la forma de un árbol. Por ello, se tomó la metodología de descomponer el grafo en componentes biconectados, tal y como se hace en (Hedetniemi *et al.*, 1986). De esta forma, se conservan todos los vértices del grafo original G y los componentes biconectados también conservan sus aristas originales.

Retomando la relación de la cardinalidad del CIF contra el número de vértices del grafo, la tercera columna Tabla 10 indica que en los experimentos realizados, sólo el $26\,\%$ (aproximadamente) de los vértices es marcado para el CIF. No obstante, existen configuraciones en las cuales dicho porcentaje es cercano a $33\,\%$; por ejemplo, en los ciclos de vértices. Incluso, es posible elevar la tasa al $50\,\%$ en cliques de tamaño 2 o en grafos que exhiben una configuración de vértices como muestra la Figura 50. Si se tiene un grafo trivial, i.e. de un sólo vértice, la relación $|S|\ /\ V(G) = 100\,\%$. Entonces, la tasa promedio obtenida por los experimentos realizados se debe a que en los grafos árbol, se tienen árboles cuyos vértices tienen grafo alto, es decir, con muchos vértices hijos. Por lo cual, para un vértice intermedio v cualquiera y del conjunto C_v, es posible marcar un sólo vérti-

ce para el CIF; de aquí que, el porcentaje de vértices marcados es considerablemente menor.

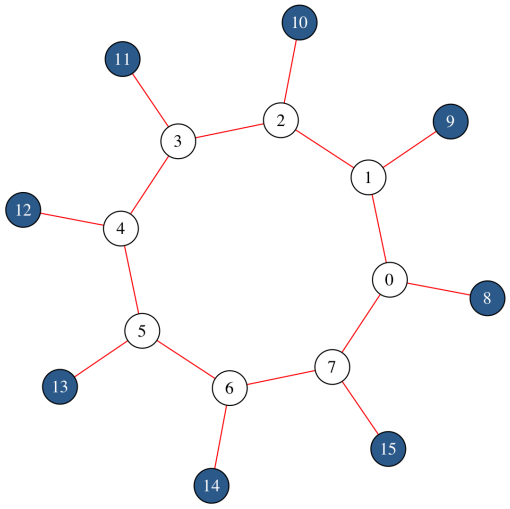

Figura 50: Grafo con 16 vértices, de los cuales 8 pertenecen al CIF.

Capítulo 5. Conclusiones

En esta tesis se presentó la literatura concerniente al estado del arte para resolver problemas similares al conjunto independiente; particularmente se propuso el algoritmo CIF-MAXIMUM-CACTUS para resolver el problema del CIF máximo sobre grafos cactus y se realizó la formalización y experimentación para demostrar su corrección. La estrategia empleada para resolver el CIF une metodologías y algoritmos de diversos autores. Siguiendo sus ideas, se hicieron recorridos sobre el grafo, búsquedas informadas y descomposición en componentes biconectados, entre otras técnicas. Todo esto forma parte de nuestro algoritmo.

Es importante resaltar que el algoritmo desarrollado resuelve en tiempo polinomial el problema del CIF máximo sobre grafos cactus para el cual, como se menciona en el Capítulo 2, hasta el momento sólo existían algoritmos eficientes para obtener conjuntos maximales y en el caso del conjunto máximo, los algoritmos se reducen a 2 opciones: considerar topologías más sencillas que el cactus o incrementar exponencialmente el tiempo de ejecución.

5.1. Aportaciones y discusión

La aportación principal de esta tesis es el algoritmo CIF-MAXIMUM-CACTUS (ver el Capítulo 3), el cual es capaz de identificar los componentes biconectados del grafo, sus vértices de articulación y generar un grafo secundario (el cual recibió el nombre de árbol de componentes biconectados), en donde mediante un recorrido informado se resuelve el problema del conjunto independiente fuerte. Posteriormente, el algoritmo reconstruye a partir de la respuesta encontrada en el grafo secundario el marcado de vértices sobre el cactus denotando así el CIF máximo. Adicionalmente, se tienen los siguientes productos secundarios:

- Actualización del Simulador BAP, el cual ahora contiene la codificación del algoritmo propuesto (CIF-MAXIMUM-CACTUS), así como los algoritmos secuenciales encontrados en la literatura para solucionar el problema del CIF máximo sobre grafos tipo árbol y el conjunto dominante mínimo para grafos tipo árbol y cactus. Asimismo, el

simulador BAP contiene otras mejoras en cuanto a la codificación y ejecución automática de la consola de R para visualizar los resultados obtenidos tras la ejecución de algún algoritmo (ver la Sección 4.2.1).

- También se cuenta con scripts en R, C/C++ y JAVA para generar de forma aleatoria tanto grafos tipo árbol como cactus y guardar su representación en formato .gml (ver la Sección 4.2.2).

- Un conjunto de grafos de prueba descritos en la Sección 4.4 donde $|V(G)| \approx 2^{12}$ vértices por grafo, así como los libros de Microsoft Excel que contienen los resultados de evaluar cada uno de dichos grafos mediante la implementación del CIF-MAXIMUM-CACTUS.

5.2. Trabajo futuro y problemas no resueltos

La estrategia que se propone en esta tesis para resolver el CIF máximo en grafos cactus es tan sólo el inicio de un gran abanico de oportunidades y problemas que se pueden trabajar. Comenzando con el algoritmo que se describe en el Capítulo 3, se considera viable la exploración de la optimización de recursos empleados, es decir, disminuir la complejidad espacial de las estructuras de datos y variables. Adicionalmente, estudiar la posibilidad de eliminar la construcción y uso del árbol de componentes biconectados y en su lugar, emplear alguna estrategia similar a la unión de vértices como la encontrada en (Hedetniemi *et al.*, 1986).

También existe la posibilidad de reducir el tiempo de ejecución del algoritmo. Actualmente, la complejidad total del CIF-MAXIMUM-CACTUS es de $O(n^2)$ u. t. No obstante, con base en la experimentación realizada y análisis de algoritmos similares, se conjetura que es posible disminuir el tiempo de ejecución a $O(n)$ u. t. Sin embargo, sería muy aventurado el tratar de reducir aún más dicha cota, puesto que en algún momento, se tiene que recorrer al menos en una ocasión cada vértice del cactus.

Otra área de oportunidad concerniente al CIF, es realizar la versión ponderada del mismo, es decir, resolver el problema del CIF máximo considerando que ahora los vértices o aristas tienen pesos asociados, lo cual es utilizado en problemas como el de asignación

de instalaciones y el modelado de rutas que manejan algún tipo de tráfico. Asimismo, se puede estudiar el problema del conjunto k-packing máximo sobre el grafo cactus, dado que es la generalización del conjunto independiente y del CIF. Lo anterior debido a que hasta el momento y al mejor de nuestro entendimiento, para el cactus únicamente existen algoritmos para $k = 1$ (conjunto independiente en (Das, 2012)) y $k = 2$ que es el algoritmo que aquí se propone. Aunado a esto, se considera de interés el extender nuestra metodología a este problema, seguir trabajando con el cactus y evaluar los resultados que se deriven de ella.

Asimismo, se puede evaluar la pertinencia de utilizar esta metodología o alguna adaptación, como la descomposición de orejas para marcar vértices en el grafo que correspondan a conjuntos tanto de cardinalidad máxima como mínima. Entre los conjuntos que se pueden afrontar se encuentran el problema del conjunto de cobertura, cubrimiento de aristas, cubrimiento de vértices e incluso el conjunto dominante extendido.

Además del trabajo futuro descrito, existen otro par de nichos de investigación muy grandes, el primero es realizar el rediseño del algoritmo CIF-MAXIMUM-CACTUS hacia su versión distribuida e incluso autoestabilizante. El segundo, la modificación de este algoritmo o diseño de otros para encontrar un CIF máximo sobre topologías de grafos más generales que el cactus, por ejemplo, el grafo outerplanar y el grafo cordal.

Lista de referencias

Abello, J., Pardalos, P. M., y Resende, M. G. C. (1999). On maximum clique problems in very large graphs. En: *In External Memory Algorithms*. American Mathematical Society, pp. 119–130.

Aho, A. V. y Hopcroft, J. E. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., primera edición. Boston, MA, USA, p. 470.

Ben-Moshe, B., Bhattacharya, B., y Shi, Q. (2005). Efficient algorithms for the weighted 2-center problem in a cactus graph. En: X. Deng y D.-Z. Du (eds.), *Algorithms and Computation*, Vol. 3827 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 693–703.

Berman, K. y Paul, J. (2005). *Algorithms: Sequential, Parallel, And Distributed*. Thomson. p. 996.

Blass, A. y Gurevich, Y. (2003). Algorithms: A quest for absolute definitions. *Bulletin of the European Association for Theoretical Computer Science*, p. 30.

Butenko, S. (2003). *Maximum Independent Set and Related Problems, with Applications*. Tesis de doctorado, University of Florida, Gainesville, FL, USA.

Castro, A., Klavar, S., Mollard, M., y Rho, Y. (2011). On the domination number and the 2-packing number of fibonacci cubes and lucas cubes. *Comput. Math. Appl.*, **61**(9): 2655–2660.

Chellali, M., Favaron, O., Hansberg, A., y Volkmann, L. (2012). k-domination and k-independence in graphs: A survey. *Graphs and Combinatorics*, **28**(1): 1–55.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Clifford, S. (2009). *Introduction to Algorithms*. The MIT Press. p. 1312.

Csardi, G. y Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, **Complex Systems**(5): 1695.

Das, K. (2012). Some algorithms on cactus graphs. *Annals of Pure and Applied Mathematics*, **2**(2): 114–128.

Daskin, M. S. (1995). *Network and Discrete Location: Models, Algorithms, and Applications*. Wiley-Interscience, primera edición. p. 520.

Diestel, R. (2005). Graph theory. 2005. *Grad. Texts in Math*, p. 431.

Dijkstra, E. W. (1974). Self-stabilizing systems in spite of distributed control. *Commun. ACM*, **17**(11): 643–644.

Gairing, M., Geist, R. M., Hedetniemi, S. T., y Kristiansen, P. (2004). A self-stabilizing algorithm for maximal 2-packing. *Nordic Journal of Computing*, **11**(1): 1–11.

Gamst, A. (1986). Some lower bounds for a class of frequency assignment problems. *Vehicular Technology, IEEE Transactions on*, **35**(1): 8–14.

Goddard, W., Hedetniemi, S. T., Jacobs, D. P., y Trevisan, V. (2008). Distance- k knowledge in self-stabilizing algorithms. *Theoretical Computer Science*, **399**(1-2): 118–127.

Hale, W. K. (1980). Frequency assignment: Theory and applications. En: *IEEE journals and magazines*. Proceedings of the IEEE, Vol. 68, pp. 1497–1514.

Haynes, T., Hedetniemi, S., y Slater, P. (1998). *Domination in Graphs: Advanced Topics*. Chapman and Hall/CRC Pure and Applied Mathematics Series. Marcel Dekker, Incorporated. p. 520.

Hedetniemi, S., Laskar, R., y Pfaff, J. (1986). A linear algorithm for finding a minimum dominating set in a cactus. *Discrete Applied Mathematics*, **13**(2a3): 287 – 292.

Hedetniemi, S., Hedetniemi, S. T., Jacobs, D., y Srimani, P. (2003). Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers & Mathematics with Applications*, **46**: 805–811.

Hochbaum, D. S. y Shmoys, D. B. (1985). A best possible heuristic for the *k*-center problem. *Mathematics of Operations Research*, **10**: 180–184.

Imrich, W., Klavzar, S., y Rall, D. F. (2008). *Topics in Graph Theory: Graphs and Their Cartesian Product*. AK Peters Ltd. p. 219.

Johnen, C. y Beauquier, J. (1995). Space-efficient, distributed and self-stabilizing depth-first token circulation. En: *In Proceedings of the Second Workshop on Self-Stabilizing Systems*. pp. 4–1.

Johnson, D. S. (1985). The NP-completeness column: An ongoing guide. *Journal of Algorithms*, **6**(1): 145–159.

Kamada, T. y Kawai, S. (1989). An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, **31**(1): 7–15.

Kariv, O. y Hakimi, S. L. (1979). An algorithmic approach to network location problems. I: The *p*-centers. *SIAM Journal on Applied Mathematics*, **37**(3): pp. 513–538.

Koontz, W. (1980). Economic evaluation of loop feeder relief alternatives. *Bell System Technical Journal*, **59**(3): 277–293.

Manne, F. y Mjelde, M. (2006). A memory efficient self-stabilizing algorithm for maximal k-packing. En: A. Datta y M. Gradinariu (eds.), *Stabilization, Safety, and Security of Distributed Systems*, Vol. 4280 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 428–439.

Meir, A. y Moon, J. W. (1975). Relations between packing and covering numbers of a tree. *Pacific Journal of Mathematics*, **61**(1): 225–233.

Michael, D. y Battiston, S. (2009). From graph theory to models of economic networks. a tutorial. En: A. Naimzada, S. Stefani, y A. Torriero (eds.), *Networks, Topology and Dynamics*, Vol. 613 de *Lecture Notes in Economics and Mathematical Systems*. Springer Berlin Heidelberg, pp. 23–63.

Mjelde, M. (2004). *K-packings and K-domination on tree graphs*. Tesis de maestría, Department of Informatics, University of Bergen, Norway.

Nepusz, T. (2010). Simple random tree generation. Recuperado el 27-12-2013 de: http://web.archiveorange.com/archive/v/yj6N4R5LUhb6EaHyCs87.

Paten, B., Diekhans, M., Earl, D., St. John, J., Ma, J., Suh, B., y Haussler, D. (2010). Cactus graphs for genome comparisons. En: B. Berger (ed.), *Research in Computational Molecular Biology*, Vol. 6044 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 410–425.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Reingold, E. y Tilford, J. (1981). Tidier drawings of trees. *IEEE Transactions on Software Engineering*, **7**(2): 223–228.

Reingold, E., Nievergelt, J., y Deo, N. (1977). *Combinatorial Algorithms: Theory and Practice*. Pearson Education Canada. p. 930.

Rogers, H. (1987). *Theory of recursive functions and effective computability*. McGraw-Hill series in higher mathematics. McGraw-Hill. p. 504.

Sarrafzadeh, M. y Lee, D. T. (1989). A new approach to topological via minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, **8**(8): 890–900.

Shi, Z. (2012). A self-stabilizing algorithm to maximal 2-packing with improved complexity. *Information Processing Letters*, **112**(13): 525 – 531.

Shukla, S., Rosenkrantz, D., y Ravi, S. (1994). Developing self-stabilizing coloring algorithms via systematic randomization.

Trejo-Sánchez, J. A. y Fernández-Zepeda, J. A. (2012). A self-stabilizing algorithm for the maximal 2-packing in a cactus graph. En: *IPDPS Workshops*. IEEE Computer Society, pp. 863–871.

Trejo-Sánchez, J. A. y Fernández-Zepeda, J. A. (2014). Distributed algorithm for the maximal 2-packing in geometric outerplanar graphs. *Journal of Parallel and Distributed Computing*, **74**(3): 2193 – 2202.

Turau, V. (2007). Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, **103**(3): 88 – 93.

Turau, V. (2012). Efficient transformation of distance-2 self-stabilizing algorithms. *Journal of Parallel and Distributed Computing*, **72**(4): 603 – 612.

Valiente, G. (2002). *Algorithms on Trees and Graphs*. Springer. p. 489.

Yannakakis, M. y Gavril, F. (1987). The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, **24**(2): 133 – 137.

Zatarain Aceves, H. (2011). *Resolviendo el problema de asignación de guardaespaldas utilizando algoritmos genéticos*. Tesis de maestría, Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California, México.

Zmazek, B. y Žerovnik, J. (2004). The obnoxious center problem on weighted cactus graphs. *Discrete Applied Mathematics*, **136**(2-3): 377–386.