Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California

Maestría en Ciencias en Ciencias de la Computación

Algoritmos para el problema del conjunto independiente fuerte en grafos dispersos

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de Maestro en Ciencias

Presenta:

Carlos Alberto Oliva Moreno

Ensenada, Baja California, México

2021

Tesis defendida por

Carlos Alberto Oliva Moreno

y aprobada por el siguiente Comité

Dr. José Alberto Fernández Zepeda

Codirector de tesis

Dr. Joel Antonio Trejo Sanchez

Codirector de tesis

Dr. Carlos Alberto Brizuela Rodríguez

Dra. Carmen Guadalupe Paniagua Chávez

Dr. Pedro Gilberto López Mariscal

Coordinador del Posgrado en Ciencias de la Computación

Dr. Pedro Negrete Regagnon

Director de Estudios de Posgrado

Resumen de la tesis que presenta Carlos Alberto Oliva Moreno como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Algoritmos para el problema del conjunto independiente fuerte en grafos dispersos

Resumen aprobado por:

Dr. José Alberto Fernández Zepeda	Dr. Joel Antonio Trejo Sanchez
Codirector de tesis	Codirector de tesis

Sea $G = (V_G, E_G)$ un grafo conectado y no dirigido, un conjunto independiente fuerte (abreviado por las siglas CIF) es un conjunto $S \subseteq V_G$ tal que, para cualquier par de vértices en S, la distancia entre ellos es de al menos tres aristas. El problema de encontrar el CIF de la mayor cardinalidad posible en un grafo arbitrario es un problema NP-difícil. En este trabajo de tesis, se propusieron dos algoritmos para resolver este problema. El primero consiste de una heurística voraz que se puede aplicar sobre grafos arbitrarios pero que solo está garantizado que puede encontrar un CIF maximal. El segundo es un algoritmo secuencial que puede encontrar el CIF máximo en un grafo cactus restringido, denominado cactus lineal, en $O(n)$ pasos. También se hizo una comparación experimental entre la heurística voraz y el Gurobi, un solucionador contemporáneo de programación entera mixta. Esta comparación demostró que la heurística requiere menos tiempo de ejecución que el Gurobi en todos los casos probados, a cambio de una reducción promedio en la cardinalidad de la solución de no más del 10 %. También se demostró que, cuando el grafo es grande, la heurística encuentra soluciones de igual o mayor cardinalidad a la del Gurobi en el caso de los modelos de grafos aleatorios de Gilbert y Watts-Strogatz y en grafos conectados aleatorios.

Palabras clave: algoritmos para grafos,conjunto independiente fuerte,2-packing, cactus lineal,heurística,simulación algorítmica

Abstract of the thesis presented by Carlos Alberto Oliva Moreno as a partial requirement to obtain the Master of Science degree in Computer Science.

Algorithms for solving the 2-packing problem in sparse graphs

Abstract approved by:

Dr. José Alberto Fernández Zepeda
Thesis Co-Director

Dr. Joel Antonio Trejo Sanchez
Thesis Co-Director

Let $G = (V_G, E_G)$ be an undirected and connected graph, a 2-packing set is the set of vertices $S \subseteq V_G$ such that, for any pair of vertices in S, the distance between them is at least three. Finding a 2-packing set of maximum possible cardinality in an arbitrary graph constitutes an NP-hard problem. In this work, we propose two algorithms for tackling this problem. The first one consists of a greedy heuristic that can be applied on arbitrary graphs, but which is only guaranteed to find a maximal 2-packing set. The second one consists of a sequential algorithm that can find the maximum 2-packing set in a restricted kind of cactus graph, called linear cactus, in $O(n)$ time. An experimental comparison was also performed between the greedy heuristic and Gurobi, a contemporary mixed-integer programming solver. This showed that the heuristic requires less running time than Gurobi in all test cases, in exchage of an average reduction in the cardinality of the solution that is no larger than 10 %. It was also shown that, for large graphs, the heuristic finds solutions of equal or greater cardinality than Gurobi in the case of graphs from the Gilbert and Watts-Strogatz random graph models as well as random connected graphs.

Keywords: Graph algorithms,strong independent set,2-packing,lineal cactus, heuristic,algorithm simulation

Dedicatoria

A mis padres, Rosy y Juan Fidel.

Agradecimientos

A mis padres, por incondicionalmente apoyarme y motivarme a alcanzar mis metas académicas y profesionales.

A mis asesores de tesis, el Dr. José Alberto Fernández Zepeda y el Dr. Joel Antonio Trejo Sánchez, por orientarme y transmitirme su experiencia y conocimiento, siempre pacientemente, durante el desarrollo de este trabajo de tesis.

A los miembros de mi comité de evaluación, la Dra. Carmen Guadalupe Paniagua Chávez y el Dr. Carlos Alberto Brizuela Rodríguez, por sus valiosos consejos y retroalimentación.

Al personal académico y administrativo del Centro de Investigación Científica y de Educación Superior de Ensenada, sobre todo el personal del Departamento de Ciencias de la Computación, por su asistencia durante mi estancia como estudiante en esta venerable institución.

Finalmente, al Consejo Nacional de Ciencia y Tecnología por brindarme el apoyo económico para realizar mis estudios de maestría.

Sin la participación de todos ustedes, no hubiera alcanzado esta meta académica. A todos ustedes les entrego mi más sincero agradecimiento.

Tabla de contenido

Página

Tabla de contenido (continuación)

Lista de figuras

Figura Página

Lista de tablas

Capítulo 1. Introducción

Existen muchos objetos y entidades en la vida real que pueden describirse en términos de las relaciones binarias que pueden establecerse entre ellos. Por ejemplo, dos ciudades pueden estar conectadas por medio de una carretera, dos personas pueden conocerse personalmente o dos computadoras pueden comunirse por medio de una conexión alámbrica o inalámbrica. Las relaciones descritas en estos escenarios pueden representarse convenientemente por medio de un diagrama conocido como *grafo*. En un grafo, los objetivos o entidades individuales se representan por medio de puntos distintos (denominados *vértices*) y las relaciones que existen entre ellos se representan por medio de líneas (denominadas *aristas*) que conectan un par de estos puntos.

Los grafos son diagramas muy versátiles; se utilizan para modelar una gran variedad de problemas en diferentes áreas del conocimiento. Por eso es importante desarrollar algoritmos para resolver problemas definidos sobre ellos. En particular, en este trabajo de tesis se pretende desarrollar un par de algoritmos para resolver el *problema del conjunto independiente fuerte* en algunas familias de grafos dispersos.

1.1. Conceptos preliminares

Antes de definir formalmente el problema del conjunto independiente fuerte, es necesario introducir algunos términos. Todos estos términos se tomaron del libro de Diestel (2000).

Sea $G = (V_G, E_G)$ un grafo, donde V_G denota el conjunto de vértices y E_G el conjunto de aristas. El *orden* de G se denota $n \triangleq |V_G|$ y el *tamaño* de G se denota $m \triangleq |E_G|$, para cualquier grafo G. Se supone que todos los vértices de V_G se identifican por medio de un número entero único extraido del conjunto $\{1, 2, \ldots, n_G\}$.

Las aristas de E_G se denotan por medio del par de vértices que conectan. Por ejemplo, la arista que conecta los vértices $i, j \in V_G$ se denota $ij \in E_G$. El par de vértices conectados por una arista son los *extremos* de dicha arista. Una arista es *incidente* a un vértice si dicho vértice es un extremo de dicha arista.

Dos vértices son *adyacentes* o son *vecinos* si están conectados por una arista. Dado un vértice i, el *vecindario abierto* de i, denotado $N(i)$, es el conjunto de todos los vértices adyacentes a i. El *vecindario cerrado* de i se denota $N[i] \triangleq N(i) \cup i$. El *grado* de un vértice i se denota $d(i) \triangleq |N(i)|$.

Un *camino* es un grafo $P = (V_P, E_P)$ tal que $V_P = \{v_1, v_2, \ldots, v_n\}$ y $E_G = \{v_1v_2, v_2v_3, \ldots, v_{n-1}v_n\}$. En este contexto, se dice que m es la *longitud* de P y se tiene que $n = m + 1$. Los vértices v_1 y v_n son los *extremos* de P y se dice que P conecta estos vértices. Se puede ver que $d(v_1) = d(v_n) = 1$ y que $\forall i \in V_P \backslash \{v_1, v_n\}\, d(i) = 2$. Se utiliza P_k, donde $k \in \mathbb{N}$, para denotar un camino de longitud k. Sea $P_k = (V_P, E_P)$ un camino donde $k \geq 3$, un *ciclo* es el grafo $C = (V_P, E_P \cup v_1v_n)$. Se puede ver que $n = m$.

Se dice que un grafo G es un *grafo conectado* si, para todo par de vértices de V_G, existe en G un camino que los conecta. Dados dos vértices, i y j, la *distancia* que los separa en G, denotada $d_G(i, j)$, es la longitud del camino más corto posible que los conecta en G. Por convención, si dicho camino no existe, se tiene que $d_G(i, j) = \infty$. El *vecindario abierto extendido a distancia k* de un vértice i, donde $k \geq 2$, se denota $N_k(i) \triangleq \{j \in V_G : d_G(i, j) \leq k\}$. Así mismo, el *vecindario cerrado extendido a distancia k* de i se denota $N_k[i] \triangleq N_k(i) \cup i$.

1.2. Definición del problema

Sea $G = (V_G, E_G)$ un grafo conectado y no dirigido y sea $k \in \mathbb{N}$. Se dice que un conjunto $S \subseteq V_G$ es un conjunto independiente a distancia k (conocido en inglés como *k-packing*) si, para cualquier par de vértices distintos pertenecientes a S, la distancia que los separa es al menos $k + 1$ (Meir y Moon, 1975). Cuando $k = 2$, se dice que S es un *conjunto independiente fuerte* (Hochbaum y Shmoys, 1985), nombre que se abrevia a partir de ahora con las siglas CIF.

Un CIF, S, es *maximal* si no existe otro CIF, $S' \subseteq V_G$, tal que $S \subset S'$ (Gairing *et al.*, 2004a). Un CIF es *máximo* si es maximal y si tiene la mayor cardinalidad de todos los CIFs maximales posibles de G (Mjelde, 2004). Cabe señalar que un CIF máximo no necesariamente es único. La Figura 1 muestra ejemplos de diferentes CIFs que pueden formarse en un mismo grafo. Un conjunto determinado, S, es un CIF maximal si se cumple que $|N[i] \cap S| \leq 1$ y que $|N_2(j) \cap S| \geq 1$ para toda $i \in V_G$ y para toda $j \in V_G \backslash S$.

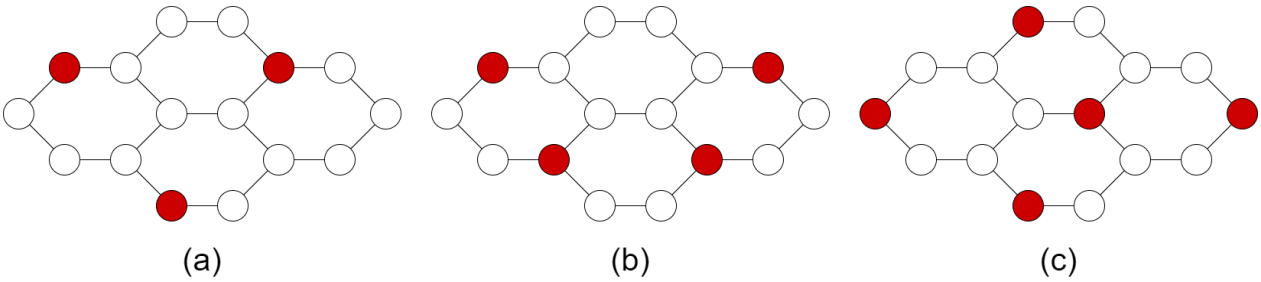


Figura 1. Tres CIFs de un mismo grafo de ejemplo. Los vértices rojos son aquellos que constituyen un CIF. (a) Un CIF que no es maximal; aún es posible marcar de rojo un vértice blanco para obtener un CIF de mayor cardinalidad. (b) Un CIF maximal; no es posible marcar ningún vértice blanco sin que éste se encuentre a una distancia menor a tres de algún otro vértice rojo. (c) Un CIF máximo; es maximal y tiene la mayor cardinalidad posible para este grafo particular.

El problema del CIF máximo se define formalmente de la siguiente manera:

- Entrada: un grafo conectado y no dirigido, G.
- Salida: un CIF máximo de G.

El problema del CIF máximo en grafos arbitrarios es un problema NP-difícil(Hochbaum y Shmoys, 1985). También se puede definir el problema del CIF maximal de la manera similar al problema anterior:

- Entrada: un grafo conectado y no dirigido, G.
- Salida: un CIF maximal de G.

El problema del CIF maximal se puede resolver en tiempo polinomial utilizando una heurística voraz aleatoria ingenua.

Un concepto que aparece frecuentemente cuando se habla de algoritmos para resolver los problemas del CIF es el de *marcado de vértices*. Sea $L \subseteq \mathbb{N}_0$ un conjunto de “etiquetas”, un marcado de vértices es una función $\varphi : V_G \to L$. En el contexto de los problemas del CIF, el marcado de vértices se utiliza para distinguir cuáles vértices pertencen al CIF y cuáles no.

Diferentes algoritmos en la literatura utilizan diferentes procedimientos para construir gradualmente el CIF de un grafo, incluyendo un procedimiento de marcado de vértices. A continuación se propone un vocabulario para describir, en un marco uniforme, las diferentes acciones que realizan estos algoritmos. Sea S un CIF, se dice que

un vértice i está *empaquetado* si $i \in S$. En este sentido, se utiliza el verbo *empaquetar* para referirse a la acción de agregar algún vértice particular a S. Cuando se tiene que $i \notin S$ y que $\forall j \in S\, i \notin N_2(j)$, se dice que i está *disponible*. En cambio, si $i \notin S$ pero $\exists j \in S\, i \in N_2(j)$, se dice que i está *restringido*. Se utiliza el vértice *restringir* para referirse a que un vértice disponible pasó a estar restringido.

1.3. Antecedentes

En la literatura se han propuesto varios algoritmos para resolver los problemas del CIF utilizando diferentes modelos computacionales. Flores-Lamas *et al.* (2020) presentaron un listado de los algoritmos más representativos para resolver este problema, dicho listado se muestra en la Tabla 1. Se puede observar que la mayoría de los algoritmos presentados son *distribuidos* o *autoestabilizantes*. También se puede observar que la mitad de los algoritmos trabajan sobre grafos arbitrarios, mientras que la otra mitad lo hace sobre familias particulares de grafos. Lo más importante a señalar es que los únicos algoritmos que resuelven el problema del CIF máximo son los propuestos por Flores-Lamas *et al.* (2018) y por Mjelde (2004). Estos dos algoritmos trabajan sobre una familia restringida de grafos. El resto de esta sección se dedica a describir brevemente la estrategia general que se utiliza en cada uno de estos algoritmos.

1.3.1. Algoritmos distribuidos

Antes de discutir los algoritmos distribuidos que pueden resolver los problemas del CIF, primero se deben introducir algunos conceptos sobre sistemas distribuidos. Estos conceptos se obtuvieron del libro de Lynch (1996). Un *algoritmo distribuido* es un procedimiento bien definido para resolver un problema computacional en un *sistema distribuido*. Un sistema distribuido es una colección de unidades de cómputo que se comunican por algún medio y que cooperan para realizar una tarea. Algunos ejemplos de sistemas distribuidos son los procesadores multinúcleos con memoria compartida y las computadoras conectadas en red. Un sistema distribuido puede modelarse como un grafo, donde los vértices representan las unidades de cómputo y un par de estos se conectan por una arista si existe algún medio de comunicación entre ellos.

Tabla 1. Algunos algoritmos para resolver los problemas del CIF. La columna “Grafo” indica la familia de grafos que el algoritmo admite como entrada. La columna “CIF” indica si el algoritmo encuentra un CIF maximal (m) o máximo (M). La columna “Tipo” indica si el algoritmo es centralizado (C), paralelo (P), distribuido (D) o una metaheurística (H). La columna “AE” indica si el algoritmo es autoestabilizante o no. La columna “$T(n)$” expresa el comportamiento asintótico del tiempo de ejecución del algoritmo. Se utiliza D_k para denotar el diámetro de un grafo cactus. Esta tabla fue adaptada de Flores-Lamas *et al.* (2020).

Autores	**Año**	**Problema**	**Grafo**	**CIF**	**Tipo**	**AE**	$T(n)$
Gairing et al.	2004	CIF	Arbitrario	m	D	✓	Exponencial
Gairing et al.	2004	CIF	Arbitrario	m	D	✓	$O(mn)$
Mjelde	2004	k-packing	Árbol	M	D	✓	$O(n^3)$
Mjelde	2004	k-packing	Árbol	M	C		$O(kn)$
Goddard et al.	2006	k-packing	Arbitrario	m	D	✓	$\Omega(n^3)$
Manne et al.	2006	k-packing	Arbitrario	m	D	✓	$O(n^2k)$
Turau	2012	CIF	Arbitario	m	D	✓	$O(mn)$
Trejo-Sánchez et al.	2012	CIF	Cactus	m	D	✓	$O(D_k)$ rondas
Shi	2012	CIF	Arbitrario	m	D	✓	$O(mn)$
Christou et al.	2013	CIF	Arbitrario	m	P/H		-
Trejo-Sánchez et al.	2014	CIF	1-outerplanar	m	D		$O(n)$
Ding et al.	2014	CIF	Arbitrario	m	D	✓	$O(n)$
Flores-Lamas et al.	2018	CIF	Cactus	M	C		$O(n^2)$
Flores-Lamas et al.	2020	CIF	Halin	m	D		$O(n)$

En un sistema distribuido, cada nodo ejecuta un proceso de forma independiente a los, por lo que surge la necesidad de utilizar un modelo que represente el cómo los diferentes nodos ejecutan una instrucción en cada unidad de tiempo. En el *modelo síncrono*, se supone que cada nodo ejecuta una instrucción al mismo tiempo que los demás. En contraste, en el *modelo asíncrono*, se supone que los nodos ejecutan diferentes instrucciones que requieren una cantidad diferente de tiempo. A menos que se especifique lo contrario, todos los algoritmos distribuidos y autoestabilizantes descritos a partir de ahora utilizan un modelo síncrono de cómputo distribuido.

De los algoritmos descritos en la Tabla 1, dos son distribuidos y se utilizan para encontrar el CIF maximal en grafos 1-outerplanar y grafos Halin.

El algoritmo que trabaja sobre grafos 1-outerplanar, propuesto por Trejo-Sánchez y Fernández-Zepeda (2014), utiliza un modelo asíncrono de cómputo distribuido y consiste de tres pasos. Primero, se elige un vértice del grafo como el líder. Para ello, se hace uso del algoritmo propuesto por Awerbuch (1987). Después, partiendo del líder, se envían “tokens” a los demás vértices con el propósito de identificar el camino de todas las aristas que tocan la cara externa del plano. Por último, se realiza un marcado

de vértices, donde se identifica con color rojo aquellos que pertenecen al CIF. Cabe señalar que este marcado se lleva a cabo por medio de una descomposición de orejas sobre el camino encontrado en el segundo paso.

El algoritmo propuesto por Flores-Lamas *et al.* (2020) trabaja sobre grafos Halin y consiste de dos pasos. Primero, se identifica el esqueleto y el ciclo externo que conforman el grafo Halin. Después, se construye un CIF en ambos subgrafos. En el primer paso, se utilizan las reglas de reducción propuestas por Eppstein (2016), las cuales disminuyen gradualmente el número de vértices en el grafo hasta llegar a un grafo completo de cuatro vértices. Además, cada vez que un vértice se remueve del grafo, su información se almacena en sus vecinos. Así, en el segundo paso, el grafo puede restaurarse a su forma original a la vez que se realiza un marcado de vértices para identificar el ciclo externo del grafo. Finalmente, se aplica un segundo marcado de vértices, aprovechando los colores que fueron asignados en el paso anterior, para decidir cuáles vértices pertenecen al CIF. Al terminar la ejecución del algoritmo, los vértices de color rojo conforman el CIF maximal del grafo.

1.3.2. Algoritmos autoestabilizantes

Los *algoritmos autoestabilizantes* son algoritmos distribuidos que tienen la propiedad adicional de que converge en un estado válido en una cantidad finita de tiempo y sin importar el estado inicial del sistema. El concepto de autoestabilización fue introducido por Dijkstra (1974).

Los algoritmos autoestabilizantes consisten de un conjunto de reglas que se aplican sobre cada nodo individual. Cada regla realiza una acción especifica si el estado del nodo correspondiente cumple con cierta condición preestablecida. Cuando un nodo satisface dicha condición, se dice esa regla está *habilitada* y que dicho nodo está *privilegiado*. Cuando al menos dos nodos están privilegiados, se requiere de un *calendarizador* para decidir cuál nodo debe ejecutar las acciones de su regla habilitada. Existen varios tipos diferentes de calendarizadores. El *calendarizador central* (también denominado *calendarizador serial*), en todo momento solamente un nodo puede ejecutar la acción de su regla habilitada. En un *calendarizador síncrono*, todos los nodos privilegiados pueden ejecutar las acciones de sus reglas habilitadas al mismo tiempo.

De los algoritmos presentados en la Tabla 1, nueve son autoestabilizantes. A menos que se especifique lo contrario, todos ellos utilizan un calendarizador síncrono.

Trejo-Sánchez y Fernández-Zepeda (2012) propusieron un algoritmo para encontrar el CIF maximal de un grafo cactus. El algoritmo consiste de dos pasos. Primero, se elige un vértice como líder y se construye un árbol de expansión enraizado en dicho vértice. Después, se recorre el árbol construido y se realiza un marcado de vértices. Al concluir la ejecución del algoritmo, los vértices de color rojo conforman el CIF maximal. Una característica particular de este algoritmo es que su rendimiento se mide en rondas. Los autores de este algoritmo definen una ronda como el mínimo período de tiempo en el que cada vértice privilegiado ejecuta al menos una instrucción cada uno. El número de rondas ejecutadas por este algoritmo particular crece de forma proporcional al diámetro del grafo de entrada.

El algoritmo anterior es el único de los algoritmos autoestabilizantes cuya entrada está restringida a una familia particular de grafos. Los algoritmos autoestabilizantes que se mencionan a continuación admiten como entrada grafos arbitrarios.

Gairing *et al.* (2004b) propusieron un algoritmo simple donde cada vértice $v \in V_G$ contiene dos valores relevantes: una bandera booleana que indica si v pertenece al CIF y un apuntador que señala a algún vértice adyacente a v. El algoritmo consiste de seis reglas: tres para modificar la bandera booleana y tres para modificar el apuntador. Estas reglas se aplican continuamente hasta que los apuntadores de todos los vértices adyacentes a algún vértice empaquetado apuntan a dicho vértice. La desventaja de este algoritmo es que su tiempo de ejecución crece de forma exponencial.

Estos mismos autores propusieron en (2004a) un algoritmo que tiene por objetivo otorgar, a cada vértice del grafo, acceso directo a la información de aquellos vértices a distancia dos del mismo. De esta forma, el algoritmo para encontrar el CIF consiste únicamente de dos reglas. Para lograr este acceso a distancia dos, cada vértice debe almacenar una copia del estado de todos sus vecinos. El problema con este método es que la información copiada puede diferir del estado actual del vértice correspondiente, por lo que se proponen cinco reglas que permiten mantener sincronizada esta información. Así, el algoritmo consiste de empaquetar vértices iterativamente hasta agotarse los vértices disponibles.

Goddard *et al.* (2008) extienden el algoritmo anterior para que cada vértice pueda acceder a información de aquellos vértices que se encuentren a una distancia *k* del mismo. Ésto se logra aplicando recursivamente la misma técnica de almacenar en cada vértice una copia del estado de sus vecinos. El problema es que el procedimiento de recabar y almacenar la información de los vecinos a distancia *k* es muy tardado y, como consecuencia, el tiempo de ejecución del algoritmo crece de forma cúbica (en el mejor de los casos) con respecto al número de vértices. Cabe mencionar que este algoritmo utiliza un calendarizador serial.

Manne y Mjelde (2006) propusieron un algoritmo donde cada vértice almacena la distancia y el identificador único de los dos vértices empaquetados más cercanos a él. Así, el algoritmo consiste únicamente de una función y una regla. La función se utiliza para determinar cuáles distancias se deben almacenar en cada vértice a partir del estado actual del mismo. Utilizando la información proporcionada por esta función, el algoritmo decide entonces si el vértice correspondiente puede empaquetarse o no. En cuanto a la única regla del algoritmo, ésta se utiliza para determinar si los valores obtenidos por la función corresponden al estado actual de los vértices correspondientes. De no ser así, el vértice se señala para corregir estos valores en la siguiente iteración.

Otro autor que propone una mejora al algoritmo de Gairing *et al.* (2004a) es Turau (2012). En su trabajo, este autor define lo que se conoce como un “modelo de expresión”. En dicho modelo, cada vértice contiene un conjunto de pares donde cada par consiste de un nombre y una expresión. La expresión se utiliza para asignar un valor (de cualquier tipo de dato) al vértice correspondiente, según el estado de dicho vértice y de sus vecinos. La única restricción es que las expresiones de un vértice particular no pueden contener las expresiones de sus vecinos, pero sí tiene acceso a ellas. De esta forma, un vértice puede acceder a la información contenida en otro vértice a distancia dos por medio de las expresiones de un vecino mutuo. Esta técnica es más eficiente en el uso de memoria que la propuesta por Gairing *et al.* Turau también propone un método para transformar un algoritmo definido sobre el modelo de expresión (que utiliza un calendarizador distribuido) al modelo de cómputo distribuido (que utiliza un calendarizador serial). Este método de transformación permite generar un algoritmo eficiente para encontrar el CIF maximal de un grafo.

Mjelde (2004) propone dos algoritmos diferentes para encontrar el CIF máximo de un árbol, uno centralizado y otro autoestabilizante. El algoritmo centralizado consiste de dos pasos y utiliza una estrategia de programación dinámica. Primero, se recorren los vértices, comenzando en las hojas. En cada vértice visitado, se calcula el tamaño de la solución óptima para el sub-árbol enraizado en él. Así, al llegar a la raíz del árbol, se obtiene la cardinalidad de la solución óptima para el árbol completo. En el segundo paso, se recorren los vértices una segunda vez, esta vez comenzando en la raíz, y se utiliza la información recabada en el paso anterior para decidir cuáles vértices deben empaquetarse y así construir gradualmente el CIF máximo.

Mjelde después convirtió su algoritmo centralizado en uno autoestabilizante. Esto lo logró convirtiendo ambos pasos del algoritmo anterior en algoritmos autoestabilizantes individuales. Para el algoritmo correspondiente al segundo paso, sin embargo, se requiere que cada vértice elija uno de sus vecinos (o a sí mismo) como la raíz de un sub-árbol. Para ello, el autor hace uso del algoritmo autoestabilizante de selección de raíz propuesto por Blair y Manne (2003). Por otro lado, en el algoritmo correspondiente al primer paso, cada vértice registra no solo la cardinalidad del CIF máximo del sub-árbol enraizado en él, sino también la cardinalida del CIF máximo de los sub-árboles enraizados en cada uno de sus vecinos. Dado que estos algoritmos son independientes, pueden ejecutarse en paralelo.

Shi (2012) propuso un algoritmo que trabaja únicamente sobre los identificadores de los vértice del grafo. El objetivo de este algoritmo es dividir el grafo en sub-árboles de tal forma que cada vértice pertenece exactamente a un sub-árbol y donde las raíces de dichos sub-árboles son todos vértices empaquetados. Para lograr esto, cada vértice registra la distancia entre dicho vértice y algún vértice empaquetado y la distancia entre este último y otros vértices disponibles. El algoritmo puede construye el CIF maximal empaquetando y desempaquetando vértices iterativamente según las distancias registradas en cada vértice. Este algoritmo funciona con cualquier tipo de calendarizador.

Por último, Ding *et al.* (2014) propusieron un algoritmo que se caracteriza por tener una “convergencia segura”. Los autores de este trabajo explican que un problema con los algoritmos autoestabilizantes es que, al pasar de un estado arbitrario a uno válido, el estado resultante no necesariamente satisface los requerimientos de salida del pro-

blema que se está tratando. Para corregir esto, los autores introdujeron el concepto de convergencia segura. Así, se dice que un algoritmo autoestabilizante tiene una convergencia segura si y sólo si converge en un “estado seguro” en una cantidad constante de pasos y después converge en un estado válido sin salir del estado seguro. Con respecto al problema del CIF maximal, el algoritmo propuesto por estos autores consiste de dos pasos. Primero, se desempaquetan vértices hasta llegar a un CIF. Después, se empaquetan vértices únicamente si éstos están disponibles, por lo que se permanece en un estado seguro. Este procedimiento continúa hasta que se agotan los vértices disponibles.

1.3.3. Metaheurísticas

Christou y Vassilaras (2013) propusieron un algoritmo paralelo que puede encontrar un CIF maximal cuya cardinalidad es muy cercana a la del CIF máximo. Dicho algoritmo consiste de la aplicación de un esquema denominado GRASP (*Greedy Randomized Adaptive Search Procedure* en inglés), el cual fue propuesto por primera vez por Feo y Resende (1995) para resolver problemas de optimización combinatoria.

En el esquema GRASP, cada iteración consiste de dos pasos principales. Primero, se construye una solución factible de forma iterativa, donde, de los elementos que pueden agregarse a la solución, se toman aquellos que más conviene agregar (según alguna función objetivo) y se elige uno de ellos de forma aleatoria. Una vez construida la solución factible, se aplica una búsqueda local, centrada en dicha solución, hasta encontrar un óptimo local. En cada iteración del esquema GRASP, se registra la mejor solución encontrada hasta ese momento y el procedimiento continúa hasta que se encuentra la solución óptima o se alcanza alguna otra condición de paro. El esquema GRASP no garantiza encontrar la solución óptima para todos los casos, pero sí puede encontrar soluciones muy cercanas a la óptima.

En el caso particular del algoritmo de Christou y Vassilaras, la construcción de las soluciones factibles se lleva a cabo por medio de un algoritmo voraz aleatorio, donde los vértices que más conviene agregar a la solución son aquellos que tienen la mayor cantidad de vértices restringidos en su vecindario abierto a distancia dos. Una vez que se eligieron los vértices que conviene empaquetar, se aplica una búsqueda tipo

branch-and-bound para explorar todas las posibles soluciones que se forman al empaquetar estos vértices de forma maximal. Entonces, el algoritmo se aplica de nuevo sobre cada una de estas posibles soluciones hasta que ya no quedan vértices disponibles en el grafo.

La búsqueda branch-and-bound fue propuesta por primera vez por Land y Doig (1960) como un método genérico para resolver problemas de programación lineal entera mixta. En términos generales, esta búsqueda construye un árbol de nodos conforme se exploran todas las posibles configuraciones en las que se puede construir una solución factible para el problema en cuestión. En el caso del algoritmo de Christou y Vassilaras, cada nodo de la búsqueda branch-and-bound contiene un CIF (no necesariamente maximal). Dado un nodo particular, se puede calcular una cota superior para la cardinalidad del CIF más grande que se puede construir a partir de los vértices disponibles del grafo contenido en dicho nodo. Esta cota se puede entonces utilizar para decidir si la búsqueda branch-and-bound debe explorar el sub-árbol enraizado en dicho nodo. Si la cota es mayor a la cardinalidad de la mejor solución encontrada hasta ese momento, entonces se continúa explorando ese sub-árbol. En caso contrario, el sub-árbol se “poda”.

A diferencia de los presentados anteriormente, Christou y Vassilaras no presentaron un análisis del orden de crecimiento de su algoritmo. En lugar de eso, realizaron una comparación experimental del desempeño de su algoritmo con el de otros, ejecutándolos sobre un conjunto grafos de prueba de diferentes tamaños. En particular, el algoritmo de estos autores se comparó con dos solucionadores de programación entera mixta, denominados SCIP y Gurobi, el algoritmo autoestabilizante de Manne y Mjelde (2006), y con una heurística aleatoria ingenua propuesta por Balakrishnan *et al.* (2004). El resultado de esta comparación experimental demostró que el algoritmo de Christou y Vassilaras se desempeñó mejor que todos los demás algoritmos, tanto en el tiempo de ejecución como en la cardinalidad de la solución encontrada.

1.3.4. Algoritmos centralizados

Flores-Lamas *et al.* (2018) propusieron un algoritmo centralizado para resolver el problema del CIF máximo en un grafo cactus. Informalmente, un grafo cactus es un

grafo conectado donde cualquier par de ciclos adyacentes comparten a lo más un vértice. Flores-Lamas *et al.* partieron del trabajo presentado por Mjelde (2004) y diseñaron un algoritmo que encuentra el CIF máximo en un grafo uniciclo (esto es, un grafo que posee exactamente un ciclo). Después extienderon dicho algoritmo para que trabaje con grafos cactus. El algoritmo consiste de reorganizar los vértices del grafo en una estructura de árbol denominada *árbol de bloques* y aplicar un algoritmo de programación dinámica sobre dicho árbol para encontrar el CIF máximo del gráfo original.

Para construir el árbol de bloques de un grafo uniciclo o un grafo cactus, los vértices se agrupan en bloques, de los cuales se distinguen dos tipos. Un *bloque clique* es un bloque cuyos vértices conforman un clique de orden dos (esto es, dos vértices conectados por una arista) en el grafo original. Un *bloque cíclico* es un bloque cuyos vértices conforman un ciclo en el grafo original. Una vez que se identifican los bloques del grafo, se construye el árbol de bloques al conectar dos bloques por medio de una arista si éstos comparten un mismo vértice, cuidando que no se generen ciclos. Una vez hecho ésto, se elige un bloque cíclico como la raíz del árbol.

Ya que se tiene el árbol de bloques, se asigna una tabla a cada bloque, donde se registra el tamaño del CIF máximo de diferentes grafos inducidos por los vértices contenidos en el sub-árbol enraizado en el bloque correspondiente. Los autores de este algoritmo también utilizan un marcado de vértices para decidir cuáles vértices se deben empaquetar en cada bloque, tomando en cuenta los vértices empaquetados en los bloques del sub-árbol enraizado en dicho bloque. De esta forma, se recorre el árbol de bloques, comenzando por las hojas, y se aplican las reglas antes mencionadas sobre cada bloque.

En el caso del grafo uniciclo, el algoritmo se detiene cuando se llega al bloque cíclico. En ese momento, para calcular el CIF máximo del ciclo se generan todos los posibles árboles de expansión que se forman al remover una arista del ciclo. Después, se calcula el CIF máximo de cada árbol resultante y se elije aquél de mayor cardinalidad. El problema con este procedimiento es que no se consideran las restricciones introducidas por los vértices empaquetados en los demás bloques del grafo. Entonces, al reintroducir al ciclo la arista que se había removido para crear el árbol de expansión, se puede provocar que dos vértices empaquetados se encuentren a una distancia menor a tres. Para remediar este problema, los autores diseñaron un algorit-

mo que identifica todos los diferentes tipos de conflictos que pueden ocurrir y aplica un conjunto de reglas para resolver cada uno conforme se van presentando. Finalmente, el CIF máximo del grafo uniciclo se obtiene al unir el empaquetado del ciclo con el de las demás ramas.

En cuanto al algoritmo para encontrar el CIF máximo de un grafo cactus, este consiste de recorrer el árbol de bloques del grafo cactus, comenzando por las hojas, para aplicar las mismas rutinas que se utilizan en el algoritmo para el grafo uniciclo para procesar los bloques cíclicos y bloques clique. Al llegar al bloque raíz, se obtiene el CIF máximo del grafo cactus original.

1.4. Programación lineal

Otra forma de resolver el problema del CIF máximo es formulándolo como un problema de *programación lineal*. Los conceptos que se presentan a continuación sobre programación lineal se obtuvieron de Cormen *et al.* (2009). La programación lineal es un método matemático para resolver problemas de optimización que pueden expresarse como un conjunto de ecuaciones o inecuaciones lineales. Un problema expresado en programación lineal se denomina *programa lineal* y consiste de una *función objetivo* y de al menos una *restricción*. La función objetivo consiste de al menos dos variables y las restricciones se definen sobre las mismas variables que la función objetivo. Así, para resolver un programa lineal, se deben encontrar los valores para todas las variables de tal forma que se maximiza (o minimiza) el resultado de la función objetivo al mismo tiempo que se cumplen las restricciones establecidas. Cuando un programa lineal tiene al menos una variable que está restringida a adquirir únicamente valores enteros, dicho programa lineal se denomina *programa lineal entero mixto* o MILP, por sus siglas en inglés.

A continuación se describe cómo se puede formular el problema del CIF máximo como un programa lineal y después se presenta una herramienta de software que se puede utilizar para resolver este problema.

1.4.1. Programa lineal para el problema del CIF máximo

El problema del CIF máximo se puede formular como un programa lineal de la siguiente manera:

$$\text{maximizar} \quad \sum_{i \in V_G} x_i \tag{1}$$

$$\text{sujeto a} \quad \sum_{j \in N[i]} x_j \leq 1 \text{ para toda } i \in V_G \tag{2}$$

$$\sum_{j \in N_2[i]} x_j \geq 1 \text{ para toda } i \in V_G \tag{3}$$

$$x_i \in \{0, 1\} \text{ para toda } i \in V_G \tag{4}$$

Esta formulación fue presentada por Christou y Vassilaras (2013). La expresión (1) es la función objetivo y las demás expresiones son las restricciones del problema. La restricción (3) es opcional y se utiliza para garantizar que cualquier solución factible encontrada sea maximal. La restricción (5) indica que todas las variables del problema pueden adquirir únicamente valores binarios, lo que implica que este programa lineal se categoriza como un *programa entero (0,1)*. Los programas enteros (0,1) son problemas NP-difíciles (Karp, 1972), por lo que expresar el problema del CIF máximo como un programa lineal no reduce la complejidad computacional del problema.

1.4.2. Solucionador Gurobi

Existen herramientas de software comerciales conocidas como *solucionadores de optimización* que pueden utilizarse para resolver problemas de optimización combinatoria con múltiples restricciones, incluyendo problemas MILP. En particular, en este trabajo de tesis se utilizó el solucionador Gurobi©.

Gurobi es una herramienta relativamente reciente, habiendo sido lanzada al público en 2009. Tiene la capacidad de resolver problemas de optimización convexos, no convexos, de programación lineal y de programación cuadrática. Anand *et al.* (2017) realizaron un estudio comparativo entre diferentes solucionadores de optimización que existen en el mercado. Según este trabajo, Gurobi se distingue de la competencia en

que puede correr utilizando múltiples unidades de cómputo de forma paralela. En este trabajo de tesis se decidió utilizar Gurobi porque puede programarse en el lenguaje Python y porque ofrece una licencia gratuita para uso académico.

Según la documentación oficial, Gurobi utiliza el algoritmo branch-and-cut (Balas *et al.*, 1996) para resolver problemas MILP. También se apoya de varias heurísticas y reglas de reducción para reducir el espacio de búsqueda y converger más rápidamente en una solución factible; Achterberg *et al.* (2020) describen detalladamente las reglas de reducción utilizadas por Gurobi. Durante su ejecución, Gurobi reporta varios resultados diferentes. Primero, reporta la solución inicial que encontró por medio de una heurística. Después, comienza a resolver un *programa lineal relajado*, el cual es idéntico al problema MILP original excepto que no posee restricciones enteras. Finalmente, una vez encontrada la solución a este problema, comienza la búsqueda branch-and-cut, partiendo de la solución encontrada. En este punto, Gurobi comienza a reportar las soluciones factibles que va encontrando hasta que encuentra la solución óptima o hasta que se agota el tiempo de ejecución disponible.

1.5. Justificación

El problema del CIF máximo tiene varias aplicaciones en el área de las telecomunicaciones inalámbricas. Por ejemplo, Hale (1980) explica que el problema de asignación de frecuencias en redes inalámbricas se puede reducir al problema del CIF máximo. En el problema de asignación de frecuencias, se tiene un conjunto de estaciones transreceptoras y un conjunto finito de frecuencias disponibles. El objetivo es asignar una frecuencia de transmisión a cada estación, cuidando de no asignar la misma frecuencia a dos estaciones que se encuentren separadas a cierta distancia, determinada por el alcance de transmisión de ambas estaciones. Este problema se puede modelar como un grafo de disco, utilizando la posición geográfica y el alcance de transmisión de las estaciones. Así, el problema de asignación de frecuencias se puede resolver encontrando el CIF máximo de dicho grafo.

Más recientemente, Christou y Vassilaras (2013) propusieron que el problema del CIF máximo se puede utilizar para determinar la cantidad máxima de dispositivos que pueden transmitir al mismo tiempo en una red inalámbrica ad-hoc sin que se generen

interferencias. Así mismo, Balakrishnan *et al.* (2004) explican que se puede desarrollar un protocolo de red que aprovecha lo anterior para determinar cuáles dispositivos en la red pueden suspenderse para ahorrar energía sin reducir la capacidad de transmisión de la red.

1.6. Hipótesis

En este trabajo de tesis se manejan dos hipótesis. La primera es que es posible diseñar un algoritmo cuyo orden de crecimiento es menor al del mejor algoritmo conocido para encontrar el CIF máximo en un familia particular de grafos si se restringe el problema a una subcategoría particular de dicha familia. La segunda es que se puede diseñar una heurística que corra en tiempo polinomial como una alternativa práctica a los solucionadores MIP que se utilizan actualmente para encontrar el CIF máximo de un grafo arbitrario.

1.7. Objetivo general

Evaluar el desempeño de diferentes algoritmos para resolver el problema del CIF maximal y máximo en familias particulares de grafos dispersos.

Para lograr este objetivo, primero se diseñaron dos algoritmos: uno para resolver el problema del CIF máximo en grafos cactus lineales y una heurística voraz para resolver el problema del CIF maximal en grafos arbitrarios. Ambos algoritmos corren en tiempo polinomial. Después, se generó una colección de grafos de prueba utilizando diferentes modelos generativos de grafos aleatorios y se ejecutaron estos dos algoritmos y el Gurobi sobre ellos. Se midió el tiempo de ejecución y la cardinalidad de la solución obtenida en cada una de estas ejecuciones. Finalmente, se realizó un análisis estadístico para detectar si había una diferencia significativa en el desempeño de estos algoritmos (comparados con Gurobi).

1.8. Objetivos específicos

1. Diseñar una heurística voraz para resolver el problema del CIF maximal en grafos arbitrarios en tiempo polinomial.

2. Diseñar un algoritmo voraz para resolver el problema del CIF máximo en grafos cactus lineales en tiempo lineal.
3. Generar una colección de grafos aleatorios que se puedan utilizar para evaluar el desempeño de los algoritmos propuestos.
4. Comparar experimentalmente el desempeño de los algoritmos propuestos con Gurobi.

Capítulo 2. Construcción de grafos de prueba

Como se mencionó anteriormente, en este trabajo de tesis se realizó una comparación experimental de los diferentes algoritmos diseñados y, para ello, fue necesario generar una colección de grafos de prueba utilizando diferentes modelos de grafos aleatorios. A continuación se explica qué son los modelos de grafos aleatorios y se describe el procedimiento que se utilizó para elegir aquellos que participaron en este trabajo de tesis. Después, se describe cómo funcionan los generadores de cada uno de los modelos elegidos.

2.1. Descripción de modelos de grafos aleatorios

Drobyshevskiy y Turdakov (2020) definen entre dos conceptos principales: el *modelo* y el *generador* de grafos aleatorios. Un modelo de grafos aleatorios es una distribución de probabilidad definida sobre un conjunto de grafos, mientras que un generador de grafos aleatorios es un algoritmo que produce como resultado un grafo extraído de una distribución particular. Los modelos de grafos aleatorios se idearon con el propósito de estudiar y replicar los patrones y fenómenos que surgen en la topología de redes y sistemas que se observan en la vida real, como son las redes sociales o el Internet.

Actualmente, existen en la literatura muchos modelos diferentes de grafos aleatorios. Sin embargo, la mayoría de ellos comparten un mismo conjunto de principios básicos, lo que permite categorizarlos de manera más o menos consistente. En particular, Drobyshevskiy y Turdakov (2020) proponen una taxonomía que consiste de tres categorías principales:

- Modelos generativos: son aquellos modelos donde primero se definió el algoritmo generador (utilizando reglas de construcción simples) y después se determinó qué características poseen los grafos generados por dicho algoritmo.
- Modelos definidos por características: denominados en inglés como *feature-driven models*, son aquellos donde primero se definió un conjunto de características deseadas y después se diseñó un generador que produce grafos que se ajustan lo mejor posible a dichas características.

- Modelos de dominio específico: son aquellos modelos que producen grafos con información o características particulares; por ejemplo, grafos con pesos en las aristas o con etiquetas en los vértices.

Para decidir cuáles modelos utilizar para generar la colección de grafos de prueba, se idearon los siguientes criterios de selección:

1. Se consideraron únicamente los modelos descritos en el trabajo de Drobyshevskiy y Turdakov (2020). Se decidió utilizar este trabajo como referencia debido a que contiene el catálogo de modelos de grafos aleatorios más completo de entre los trabajos consultados.
2. De la taxonomía presentada anteriormente, sólo se consideraron aquellos modelos que pertenecían a la categoría de modelos generativos. Esto se decidió porque no se necesitaban grafos especializados ni de características específicas para medir el desempeño de los algoritmos participantes.
3. Se procuró elegir los modelos más básicos, esto es, aquellos que fueron los primeros en introducir algún principio básico sobre la generación de grafos aleatorios y que no eran una modificación o extensión de algún modelo anterior.
4. Se evitó elegir dos modelos que siguieran el mismo principio básico.
5. Se dio preferencia a aquellos modelos que ya tenían una implementación disponible en NetworkX, una herramienta de software en el lenguaje de programación Python para el manejo de grafos.

Los modelos que se muestran en la Tabla 2 se eligieron utilizando estos criterios. Adicionalmente, se propusieron algoritmos para generar árboles, grafos conectados, grafos cactus lineales y grafos k-outerplanar aleatorios.

Modelo de Gilbert. Erdös y Rényi (1959) propusieron el primer modelo de grafos aleatorios, que también es uno de los más estudiados actualmente. En este modelo, se elije un grafo de n vértices y m aristas de forma aleatoria y con probabilidad uniforme de entre todos los $\binom{n(n-1)/2}{m}$ grafos que se pueden formar a partir de un mismo conjunto de vértices. Casi al mismo tiempo y de forma independiente, Gilbert (1959) propuso

Tabla 2. Los modelos generativos de grafos aleatorios que se fueron considerados. La columna “Principios” indica el principio básico sobre generación de grafos aleatorios que caracteriza cada modelo, los cuáles son: apego preferencial (AP), mundo pequeño (MP), copiado (C), reglas locales (RL), recursivo (R) y geométrico (G). La columna “NetworkX” indica si el modelo está implementado en NetworkX o no.

Modelo	**Año**	**Principios**	**NetworkX**
Erdős-Rényi	1960		
Gilbert	1960		✓
Barabási-Albert	1999	AP	✓
Watts-Strogatz	1998	MP	✓
Forest Fire	2007	C,RL,R	
Bollobás-Riordan	2003	AP	
Dorogovtsev-Mendes	2003	AP	
Kunegis-Blattner-Moser	2013	AP	
Kleinberg et al.	1999	C	
Crecimiento con copiado	2005	C	✓
Duplicado y divergencia	2003	C	Variante
Replicado de redes complejas (ReCoN)	2016	C	
Caminatas aleatorias	2003	RL	
Vecino más cercano	2003	RL	
Matríz recursiva (R-MAT)	2004	R	
Grafos estocásticos de Kronecker (SKG)	2005	R	
Generador de redes multi-fractales (MFNG)	2010	R	
Grafos geométricos aleatorios	2003	G	✓
Waxman	1988	G	✓
Yook-Jeong-Barabási	2002	G	
Wong-Pattison-Robins	2006	G	
Handcock-Raftery-Tantrum	2007	G	
Producto punto	2006	G	
Grafos hiperbólicos aleatorios	2012	G	
Grafos geométricos no homogéneos (GIRG)	2016	G	
Grafos basados en empotramiento (ERGG)	2017	C,G	

un modelo similar, donde, dado un conjunto de n vértices fijos y por cada una de todas las $n(n-1)/2$ aristas posibles, se decide aleatoriamente y con una probabilidad predefinida si la arista debe agregarse o no al grafo generado. Dadas las similitudes entre ambos modelos, en la literatura es común ver que se manejan como si fueran el mismo. En el caso particular de este trabajo de tesis, se decidió utilizar el modelo de Gilbert por ser el que estaba implementado en NetworkX.

Modelo de Barabási-Albert. Propuesto por Barabási y Albert (1999), este modelo introduce dos principios básicos sobre la generación de grafos aleatorios:

1. Crecimiento: (*growth* en inglés), se refiere a que, en lugar de comenzar con un

conjunto de vértices fijos, el grafo se construye agregando agregándole iterativamente nuevos vértices.

2. Apego preferencial: (*preferential attachment* en inglés), se refiere a que, entre mayor sea el grado de un vértice u presente en el grafo, mayor será la probabilidad de que un vértice v recién agregado al grafo se conecte con u.

El procedimiento para generar un grafo de este modelo consiste de los siguientes pasos: primero, se comienza con una cantidad pequeña de n_0 vértices, después, iterativamente se agrega un nuevo vértice que debe conectarse con $k \leq n_0$ vértices distintos ya presentes en el grafo, donde k es una constante predefinida. Estos k vértices se elijen de forma aleatoria y con una probabilidad que es directamente proporcional a su grado. Tras aplicar este procedimiento por τ iteraciones, se obtiene como resultado un grafo de $\tau + n_0$ vértices y $k\tau$ aristas. El grafo resultante se caracteriza en que es conectado y posee, relativamente, pocos vértices con grado alto y muchos vértices con grado pequeño.

Modelo de Watts-Strogatz. Propuesto por Watts y Strogatz (1998), este modelo se ideó con el propósito de simular un fenómeno conocido coloquialmente como “mundo pequeño” (*small world* en inglés). Dicho fenómeno se refiere a que, para cualquier par de vértices en un grafo conectado, la longitud del camino más corto que los conecta suele ser relativamente corta (Milgram, 1967). Para generar un grafo de este modelo, primero, se comienza con un anillo de n vértices, cada uno conectado con sus k vecinos más cercanos. Después, cada arista uv en dicho anillo se reemplaza con una arista uw, donde el vértice w se elige aleatoriamente de los $n-1$ vértices distintos a u con una probabilidad predefinida.

Grafos geométricos aleatorios. Drobyshevskiy y Turdakov (2020) explican que existen varios generadores de grafos geométricos aleatorios, pero todos siguen un mismo procedimiento general: se genera un conjunto de puntos distribuidos de forma aleatoria en el plano Euclidiano y se agregan aristas también de forma aleatoria pero utilizando una probabilidad que es proporcional a la distancia que separa los vértices extremos de dicha arista. Los modelos se distinguen en cómo se define la distribución de probabilidad con la cual se elijen las aristas del grafo generado. Waxman

(1988) propuso el primer modelo de grafos geométricos aleatorios, sin embargo, en este trabajo de tesis, se decidió utilizar el modelo propuesto por Penrose (2003). En este modelo, las aristas se agregan de forma determinística, donde dos vértices en el plano se conectan con una arista cuando la distancia entre ellos es menor o igual a alguna constante predefinida. Se eligió este modelo por ser el más simple que está implementado en NetworkX.

Grafos de duplicado y divergencia. Este modelo se ideó con el fin de generar grafos de relaciones proteína-proteína que permiten modelar la evolución de un genoma conforme este se reproduce y muta. Este modelo fue propuesto por Wagner (2003) y consiste de dos pasos principales:

1. Duplicado: se elije aleatoriamente un vértice u y se crea un nuevo vértice u' que se conecta a los vértices de $N(u)$. Después, se decide aleatoriamente y con una probabilidad predefinida si se debe agregar la arista uu' al grafo.
2. Divergencia: para cada vértice $v \in N(u)$, que está conectado a los vértices u y u', se elije una de las dos aristas uv y $u'v$ y se decide aleatoriamente y con una probabilidad predefinida si dicha arista debe removerse del grafo. El vértice u' se retiene en el grafo si, tras aplicar esta regla, posee al menos una arista incidente, de lo contrario u' se remueve del grafo.

Este procedimiento comienza con un grafo completo de orden dos y las reglas de duplicado y divergencia se aplican una cantidad determinada de veces. En particular, en este trabajo de tesis se decidió utilizar la variante de este generador propuesta por Ispolatov *et al.* (2005). Esta variante se distingue en dos aspectos: primero, el vértice replicado nunca se conecta con el vértice original y, segundo, se preservan todas las aristas incidentes al vértice original y únicamente se pueden eliminar aquellas incidentes al vértice duplicado. Se decidió utilizar esta variante por ser el que está disponible en NetworkX.

Grafos k-outerplanar. Los grafos *k-outerplanar* son un tipo particular de *grafos planos*. Un grafo plano es un grafo que puede dibujarse en el plano Euclidiano de tal forma

que las aristas se intersectan únicamente en los vértices. Un grafo *outerplanar* es un grafo plano donde todos los vértices tocan la *cara externa*; esto es, la región del plano que no está delimitada por las aristas del grafo. Un grafo k-outerplanar es un grafo plano que, tras repetir $k-1$ veces el procedimiento de remover aquellos vértices que tocan la cara externa (junto con sus aristas incidentes), el grafo resultante es outerplanar. En este trabajo de tesis se utiliza un algoritmo ideado por Yelbay *et al.* (2016), el cual consiste del siguiente procedimiento: primero, se generan k anillos concéntricos de n/k vértices cada uno, después, se agregan aristas de forma aleatoria para conectar cada anillo con aquél que se encuentra un nivel más profundo en la estructura concéntrica, cuidando que las aristas agregadas no provoquen que el grafo deje de ser plano.

Árboles aleatorios. El modelo de árboles aleatorios fue propuesto por Aldous (1990) y es igual al modelo de Erdős–Rényi con la única excepción de que no se agregan aristas al grafo si estas generan un ciclo. Es posible idear un generador más eficiente para este modelo si se hace uso de una *secuencia de Prüfer*. Todo árbol T de n vértices, donde cada vértice está etiquetado con un número entero único extraido del intervalo $[1, n]$ puede representarse por medio de una secuencia única de $n-2$ números enteros extraidos del mismo intervalo. Esta secuencia se conoce como la secuencia Prüfer del árbol T (Prüfer, 1918). La idea de utilizar secuencias de Prüfer para generar árboles y grafos conectados aleatorios fue propuesta por primera vez por Kumar *et al.* (1998). En particular, en este trabajo de tesis se utilizó el siguiente procedimiento: primero, se genera una secuencia de Prüfer eligiendo de forma aleatoria y con reemplazo $n-2$ números del intervalo $[1, n]$, después, se decodifica el árbol representado por dicha secuencia utilizando el algoritmo de Wang *et al.* (2009), el cual está implementado en NetworkX.

Grafos conectados aleatorios. El procedimiento descrito en el párrafo anterior también se puede aprovechar para generar grafos conectados aleatorios de n vértices y m aristas. Simplemente, se genera un árbol aleatorio de n vértices (utilizando el procedimiento anterior) y, después, se agregan $m-n-1$ aristas eligiéndolas de forma aleatoria del conjunto de $n(n-1)/2-n-1$ aristas posibles.

Grafos cactus lineales aleatorios. Por último, en este trabajo de tesis se ideó un procedimiento para generar grafos cactus lineales de forma aleatoria. Este procedimiento consiste de los siguientes pasos:

1. Se genera una secuencia de números enteros. Cada número en esta secuencia representa el tamaño de una subestructura en el grafo.
2. Por cada número en la secuencia, se decide de forma aleatoria y con probabilidad uniforme si se debe agregar un ciclo o un camino en el grafo.
3. Se agrega la subestructura elegida en el paso anterior y con el tamaño indicado por el número correspondiente en la secuencia.

2.2. Selección de los parámetros para los generadores

Una vez elegidos los generadores de grafos aleatorios que participarán en este trabajo de tesis, el siguiente paso fue decidir qué parámetros utilizar para generar la colección de grafos de prueba. El objetivo era que todos los grafos de prueba fueran conectados y dispersos. El problema era que cada generador puede admitir como entrada uno o varios parámetros diferentes y no siempre se puede predecir fácilmente cómo afecta cada uno al grafo resultante.

Para elegir los parámetros adecuados que generaban grafos conectados y dispersos, se decidió generar una colección de grafos preliminares utilizando diferentes parámetros y determinar qué efecto tenía cada parámetro a partir de diferentes métricas tomadas de los grafos en dicha colección. En particular, se generaron grafos preliminares para los modelos de Gilbert, Barabási-Albert, Watts-Strogatz, grafos geométricos aleatorios y grafos de duplicado y divergencia. No fue necesario generar grafos preliminares para los demás modelos elegidos ya que era trivial generar grafos conectados y dispersos usando estos modelos. Para cada modelo de la colección de grafos preliminares, se generaron varios grafos con creciente número de vértices y, para cada grafo generado, se midió el número de aristas, el grado promedio de sus vértices y el diámetro. Estas métricas después se organizaron en una gráfica para observar cómo eran afectadas por cada parámetro.

Fue a partir de esta información que se eligieron los parámetros que se describen a continuación:

Tabla 3. Métricas de los grafos de prueba. La columna “Modelo” indica el modelo de grafos aleatorios, el cuál corresponde a uno de los siguientes valores: Gilbert (GIL), Barabási-Albert (BA), Watts-Strogatz de 4 vecinos (WS4) y de 8 vecinos (WS8), grafos geométricos (GEO), grafos de duplicado y divergencia (DYD), grafos k-outerplanar (OUT), árboles (ARB), grafos conectados (CON) y grafos cactus lineales (CL). La columna n denota el número promedio de vértices, donde $\overline{k} = 1024$, la columna m denota el número promedio de aristas, la columna N_1 denota el grado promedio, las columnas N_2, N_3 y N_4 denotan el tamaño promedio del vecindario a distancia dos, tres y cuatro y D denota el diámetro de uno de los grafos. Se midió el diámetro de únicamente un grafo porque calcular el diámetro es una operación computacionalmente muy costoza.

Modelo	n	m	N_1	N_2	N_3	N_4	D
GIL	$2\overline{k}$	13099.70	12.79	156.27	1142.48	733.73	5
	$8\overline{k}$	52391.00	12.79	161.74	1771.61	5724.19	6
	$32\overline{k}$	209603.80	12.79	163.19	2005.99	16217.39	7
BA	$2\overline{k}$	6135.00	5.99	82.73	598.47	1105.20	6
	$8\overline{k}$	24567.00	6.00	99.18	1051.18	4049.14	7
	$32\overline{k}$	98295.00	6.00	119.23	1655.41	10593.68	8
WS4	$2\overline{k}$	4096.00	4.00	12.53	38.75	115.92	10
	$8\overline{k}$	16384.00	4.00	12.56	39.22	121.54	14
	$32\overline{k}$	65536.00	4.00	12.56	39.36	123.07	14
WS8	$2\overline{k}$	8192.00	8.00	51.15	303.57	1078.19	6
	$8\overline{k}$	32768.00	8.00	51.59	330.21	1834.35	7
	$32\overline{k}$	131072.00	8.00	51.69	337.22	2123.29	8
GEO	$2\overline{k}$	15753.83	15.38	30.31	46.39	61.12	34
	$8\overline{k}$	64412.63	15.72	31.83	49.96	67.78	68
	$32\overline{k}$	260869.46	15.92	32.59	51.80	71.27	137
DYD	$2\overline{k}$	4601.13	4.49	53.56	195.92	438.16	19
	$8\overline{k}$	20149.03	4.92	88.44	395.01	1208.92	16
	$32\overline{k}$	88249.16	5.39	121.33	884.98	3533.62	22
OUT	$2\overline{k}$	2137.03	2.09	2.36	2.77	3.26	197
	$8\overline{k}$	8563.20	2.09	2.37	2.80	3.30	552
	$32\overline{k}$	34266.50	2.09	2.38	2.80	3.31	2549
ARB	$2\overline{k}$	2047.00	2.00	2.99	3.96	4.93	163
	$8\overline{k}$	8191.00	2.00	3.00	3.99	4.98	380
	$32\overline{k}$	32767.00	2.00	3.00	3.99	4.99	781
CON	$2\overline{k}$	6149.06	6.00	34.67	187.55	730.06	8
	$8\overline{k}$	24571.60	6.00	34.88	199.41	1039.80	9
	$32\overline{k}$	98331.63	6.00	35.00	203.11	1151.00	11
CL	$2\overline{k}$	2097.93	2.05	2.22	2.40	2.60	1374
	$8\overline{k}$	8395.03	2.05	2.22	2.41	2.61	5381
	$32\overline{k}$	33585.86	2.05	2.22	2.41	2.61	20119

- Para todos los modelos, se generaron grafos de tres órdenes diferentes: de $2\overline{k}$, de $8\overline{k}$ y de $32\overline{k}$ vértices, donde $\overline{k} = 1024$.
- El modelo de Gilbert admite como único parámetro la probabilidad p de que una arista se agregue al grafo. Se decidió utilizar $p = 1.625 \times 10^{-3}$ para los grafos de $2\overline{k}$ y $8\overline{k}$ vértices y $p = 3.90625 \times 10^{-4}$ para los grafos de $32\overline{k}$ vértices.
- El modelo de Barabási-Albert admite como único parámetro el número de vecinos k con el cual debe conectarse cada nuevo vértice agregado al grafo. Se decidió utilizar el valor de $k = 3$ para todos los grafos generados.
- El modelo de Watts-Strogatz admite dos parámetros: el número de vecinos k con el cual debe conectarse cada vértice del anillo inicial y la probabilidad p con la que cada arista debe reemplazarse por otra. Se decidió generar dos colecciones de grafos de este mismo modelo, uno utilizando el valor de $k = 4$ y otro utilizando el valor de $k = 8$. Además, se decidió utilizar el valor de $p = 0.5$ para ambas colecciones.
- El modelo de grafos geométricos aleatorios admite como único parámetro la distancia máxima r a la que deben encontrarse dos vértices para conectarlos por medio de una arista. Se utilizó el valor de $r = 0.05$ para los grafos de $2\overline{k}$ vértices, $r = 0.025$ para los de $8\overline{k}$ vértices y $r = 0.0125$ para los de $32\overline{k}$ vértices.
- El modelo de grafos de duplicado y divergencia admite como único parámetro la probabilidad p para que una arista del vértice duplicado permanezca en el grafo generado. Se utilizó el valor de $p = 0.33$ para todos los grafos generados.
- El generador de grafos k-outerplanar admite como único parámetro el número de vértices del grafo. El generador utiliza este parámetro para calcular el número de anillos concéntricos, considerando que cada anillo consiste de 128 vértices.
- El generador de árboles aleatorios no admite ningún parámetro además del número de vértices del grafo resultante.
- El generador de grafos conectados aleatorios admite como único parámetro el número de aristas m del grafo resultante. Se decidió utilizar $m = 6\overline{k}$ para los grafos de $2\overline{k}$ vértices, $m = 24\overline{k}$ para los de $8\overline{k}$ vértices y $m = 96\overline{k}$ para los de $32\overline{k}$ vértices.

- El generador de grafos cactus lineales aleatorios admite como único parámetro el número máximo de vértices k que cada subestructura debe tener. Se decidió utilizar el valor de $k = 30$ para todos los grafos generados.

Una vez decididos los modelos y parámetros que se iban a utilizar, se generó la colección de grafos de prueba. Se generaron 90 grafos de cada modelo; 30 para cada uno de los tres órdenes descritos anteriormente. La Tabla 3 muestra las métricas promedio de los grafos de prueba.

Capítulo 3. Algoritmo secuencial para cactus lineales

En este capítulo se presenta el algoritmo LINEAR-CACTUS-2PACKING, el cual permite encontrar el CIF máximo en un tipo particular de grafo llamado *cactus lineal*. Un grafo cactus lineal es un tipo restringido del grafo cactus. Informalmente, un *grafo cactus* es un grafo conectado donde cualquier par de ciclos simples comparten a lo más un vértice. En contraste, un grafo cactus lineal tiene la propiedad adicional de que está conformado únicamente de caminos y ciclos conectados en serie. En la Figura 2 se muestran varios ejemplos de grafos cactus lineales. Hasta donde llega nuestro conocimiento, este trabajo de tesis es el primer trabajo académico donde se hace una referencia explícita al grafo cactus lineal, por lo que amerita introducir una definición formal para este concepto. Sin embargo, antes de ello, es necesario introducir algunas definiciones sobre teoría de grafos, las cuales se presentan en la siguiente sección.

3.1. Terminología

Un *subgrafo maximal* de G es un subgrafo que no está completamente contenido por algún otro subgrafo de G. Un *componente* de G es un subgrafo maximal conectado de G. Un *vértice de corte* de G es un vértice que, al removerse de G junto con sus aristas incidentes, se incrementa el número de componentes encontrados en G. Un *componente biconectado* o *bloque* de un grafo es un componente conectado que no tiene vértices de corte.

Un grafo cactus, K, puede descomponerse en un árbol denominado *árbol de bloques*, donde cada nodo representa un bloque de K y donde dos nodos están conectados por medio de una arista cuando comparten un vértice de corte en K. Flores-Lamas *et al.* (2018) define dos tipos de bloques: *bloques cíclicos*, que son aquellos que consisten de un ciclo simple, y *bloques clique*, que son aquellos que consisten de un “clique” de orden dos, esto es, un subgrafo completo de dos vértices. De esta forma, un grafo

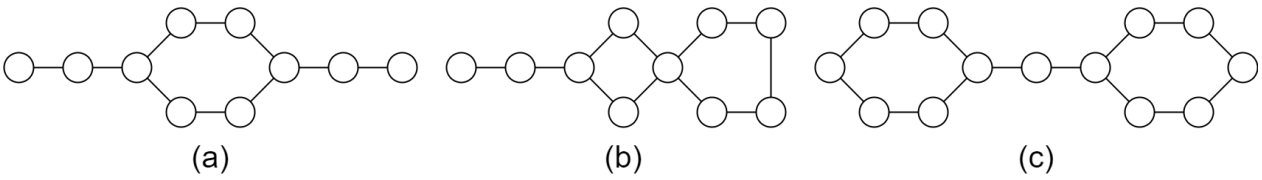


Figura 2. Tres grafos cactus lineales de ejemplo: (a) uno tipo LL, (b) uno tipo LC, y (c) uno tipo CC.

cactus se define formalmente como un grafo conectado cuyo árbol de bloques consiste únicamente de bloques cíclicos y bloques clique. Así mismo, en este trabajo de tesis, un grafo cactus lineal se define formalmente como un grafo cactus que tiene la propiedad adicional de que su árbol de bloques constituye un camino.

Aunado al concepto del grafo cactus lineal, en el presente trabajo también se introducen los conceptos de *vértices de cambio de tendencia* y *vértices extremos*. Estas son dos categorías en las que pueden distinguirse los vértices de un grafo cactus lineal. A continuación se presenta la definición formal de estos conceptos y se explica porqué esta categorización es relevante para el funcionamiento del algoritmo LINEAR-CACTUS-2PACKING.

Los vértices de cambio de tendencia (abreviado a partir de ahora por las siglas VCT) se definen formalmente como los vértices de grado tres o cuatro en un grafo cactus lineal. Es fácil ver que, en este tipo de grafos, ningún vértice tiene un grado mayor a cuatro. En particular, los vértices de grado tres y cuatro indican un cambio en la topología local del grafo en ese punto; esto es, los VCT indican en qué parte del grafo termina un camino y comienza un ciclo o vice versa, o en qué parte termina un ciclo y comienza otro.

Por otro lado, los vértices extremos (abreviado a partir de ahora por las siglas VEX) se definen formalmente de la siguiente manera. Sea K un grafo cactus lineal y sea P el árbol de bloques de K. Los VEX de K son el par de vértices conformado al elegir un vértice de cada bloque extremo de P por medio del siguiente procedimiento: si el bloque en cuestión es un bloque clique, entonces se elige el vértice en el bloque que es de grado uno en K. En cambio, si dicho bloque es un bloque cíclico, entonces se elige cualquiera de los dos vértices en el bloque que son adyacentes al VCT. Intuitivamente, se puede ver que si se elimina la arista que conecta el VCT con el VEX elegido en el bloque cíclico, se formaría un camino en este bloque y el VEX elegido sería el último vértice de dicho camino. Cabe mencionar que pueden haber múltiples pares posibles de VEX para un mismo grafo cactus lineal, pero, en el contexto del algoritmo LINEAR-CACTUS-2PACKING, basta con identificar sólo un par.

Los VCT y los VEX se denominan en conjunto como *vértices de partición*, nombre que se abrevia de ahora en adelante por las siglas VDP. Esto es, si un vértice particular

es un VCT o un VEX, entonces por definición también es un VDP. En cambio, si un vértice particular es un VDP, entonces debe ser ya sea un VCT o un VEX, pero no puede ser ambos al mismo tiempo.

El grafo cactus lineal se puede descomponer naturalmente en secciones delimitadas por los VDP. Se distinguen dos tipos de secciones: *secciones lineales*, denotadas por la letra L, y *secciones cíclicas*, denotadas por la letra C. Sea *P* el árbol de bloques de un grafo cactus lineal, una sección L es un camino maximal de bloques clique consecutivos en *P*, donde el primer y último bloque contienen cada uno un VDP distinto y donde ninguno de los bloques internos contienen un VDP. Se dice que una sección L es un camino maximal porque no está contenida por alguna otra sección L de mayor cardinalidad. Por otro lado, una sección C es un bloque cíclico en *P* que tiene la propiedad adicional de que contiene exactamente dos VDP. Es importante mencionar que una sección C también puede descomponerse en dos secciones L, donde ambos comienzan y terminan en el mismo par de VDP pero cuyos vértices internos correspondientes son distintos.

Dado un grafo cactus lineal, se puede construir un árbol de secciones al descomponer el grafo en secciones L y C y donde dos secciones están conectadas en el árbol por medio de una arista si éstas comparten un VDP. Se puede ver que el árbol de secciones de un grafo cactus lineal siempre constituye un camino, donde la primera y última sección contienen cada uno un VEX. Cabe mencionar que, debido a que una sección L se define como un camino maximal, no se pueden tener dos secciones L consecutivas en el árbol de secciones. Además, cualquier par de secciones L deben estar separadas por al menos una sección C.

Dependiendo de los tipos de secciones que se encuentren en los extremos del árbol de secciones de un grafo cactus lineal, se tiene que dicho grafo puede categorizarse de tres formas distintas. Cuando los extremos del árbol de secciones son ambos secciones L, se dice que el grafo correspondiente es un *grafo LL*. Si uno de los extremos es una sección L y el otro extremo es una sección C, entonces se dice que el grafo es *LC*. Por último, si ambos extremos son secciones C, entonces se dice que el grafo es *grafo CC*. La Figura 3 muestra cómo se puede descomponer un grafo cactus lineal de ejemplo en su árbol de bloques y su árbol de secciones. Cabe mencionar que la orientación de un grafo LC no es relevante; sin importar si la sección L se encuentra en el lado izquierdo

o derecho del grafo, el algoritmo GRAPH-MARKING, que se encarga de marcar los vértices del CIF máximo, siempre comienza a recorrer el grafo desde la sección L. Por ende, no es necesario hacer una distinción entre un grafo LC y uno CL.

La estrategia general del algoritmo LINEAR-CACTUS-2PACKING consiste en descomponer el grafo en su árbol de secciones y, después, realizar un marcado sobre los vértices de cada sección de tal forma que se encuentre el CIF maximal de mayor cardinalidad posible en dicha sección, considerando las restricciones introducidas por la sección anterior, y se introduzca la menor restricción posible a la sección siguiente.

La idea de descomponer el grafo en un árbol de sub-estructuras más simples no es nueva. Flores-Lamas *et al.* (2018) propusieron un algoritmo que descompone un grafo cactus en su árbol de bloques y utiliza programación dinámica a la vez que recorre dicho árbol para encontrar el CIF máximo del grafo. Por otro lado, Bern *et al.* (1987) propusieron que el problema general de encontrar un subgrafo óptimo en un grafo dado, *G*, puede resolverse en tiempo lineal si dicho grafo puede descomponerse en un "árbol de traducción" que describe el orden en que deben aplicarse un conjunto de "reglas de construcción" para construir el grafo *G*. Así, Flores Lamas (2018) propone la conjetura de que, utilizando el método del árbol de traducción, se puede diseñar

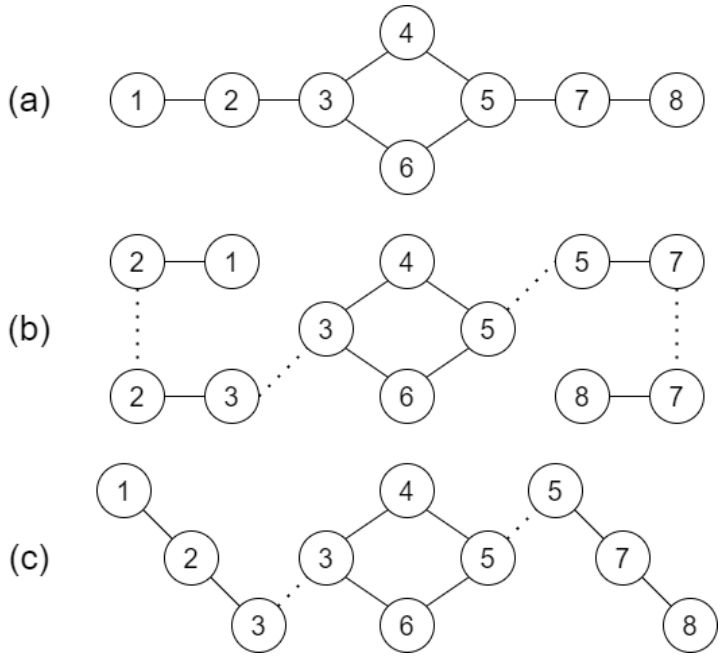


Figura 3. Diferentes formas en que se puede descomponer un mismo grafo cactus lineal de ejemplo. Las líneas punteadas indican las aristas de la descomposición en cuestión. (a) El grafo original, (b) su descomposición en un árbol de bloques y (c) su descomposición en un árbol de secciones.

un algoritmo que encuentra el CIF máximo en un grafo 1-outerplanar. El algoritmo LINEAR-CACTUS-2PACKING se distingue de los métodos anteriores en que no recurre a la programación dinámica.

3.2. Suposiciones

Sea $K = (V_K, E_K)$ un grafo cactus lineal, donde V_K denota el conjunto de vértices y E_K el conjunto de aristas. Se supone que V_K se representa por medio de un arreglo de listas de adyacencias que se denota como Adj_K. Así, $Adj_K[i]$ representa una lista enlazada que contiene los identificadores de los vértices adyacentes al vértice identificado por i.

Cuando una sub-rutina recibe como entrada una variable que representa un arreglo o cualquier otra estructura de datos no simple, se sigue por convención que cualquier modificación realizada sobre dicha estructura de datos dentro de la sub-rutina también se ve reflejada afuera de ella. Esto último se hace con el fin de compactar el pseudocódigo. Finalmente, el modelo computacional que se considera para definir el algoritmo LINEAR-CACTUS-2PACKING es el modelo RAM.

3.3. Descripción del algoritmo

El algoritmo LINEAR-CACTUS-2PACKING (Pseudocódigo 1) consiste de los siguientes pasos:

1. Se identifican los VDP del grafo y se elige uno de los VEX como el vértice raíz de la exploración del grafo.
2. Se explora el grafo sistemáticamente partiendo del vértice raíz elegido en el paso anterior para identificar las secciones L en las que puede descomponerse, incluyendo las secciones C.
3. Se realiza un marcado de vértices sobre cada sección identificada en el paso anterior para construir el CIF máximo del grafo.

En el pseudocódigo de este algoritmo se utiliza *r* para denotar el vértice elegido como la raíz de la exploración del grafo y *s* para denotar la cardinalidad del CIF encontrado. También se utiliza *part*, *dist*, *next*, *tail* y *color* para denotar otros datos relevantes sobre cada vértice del grafo, mismos que se explican más adelante. A continuación se describen detalladamente las sub-rutinas que conforman cada paso del algoritmo.

3.3.1. Identificación de los vértices de partición

El algoritmo PARTITION-VERTICES (Pseudocódigo 2) tiene dos propósitos: identificar los VDP en el grafo y elegir uno de los VEX como el vértice raíz. Para identificar los VDP, este algoritmo asocia una bandera booleana con cada vértice del grafo. Dicha bandera se denota como *part*[*i*] en el pseudocódigo y adquiere el valor de '1' si el vértice *i* es un VDP (esto es, un VCT o un VEX) y en caso contrario adquiere el valor de '0'.

El primer paso del algoritmo PARTITION-VERTICES consiste de identificar todos los VCT del grafo. Esto se logra recorriendo cada vértice y revisando si dicho vértice es de grado tres o cuatro (esto es, si el vértice en cuestión es un VCT o no) y asignando el valor apropiado para la bandera *part* de dicho vértice. Después, se deben identificar los VEX del grafo. El proceso que se lleva a cabo para lograr esto depende de la topología del grafo. Si el grafo es LL, entonces los VEX son los dos vértices de grado uno y cualquiera de ellos puede elegirse como la raíz de la exploración. El caso complicado ocurre cuando el grafo es LC o CC. En este caso, se debe realizar una exploración adicional del grafo, partiendo de un vértice raíz provisional, para encontrar la sección C en uno o ambos extremos y elegir un vértice de dicha sección como el VEX. El algoritmo ENDPOINT-CYCLE-TERMINALS lleva a cabo esta exploración. Una vez identificados los VEX, se le asigna el valor apropiado a la bandera *part* ambos.

Cuando el grafo de entrada es LC, el algoritmo ENDPOINT-CYCLE-TERMINALS parte del único vértice de grado uno en el grafo, que es un VEX. El segundo VEX es entonces el último vértice visitado por el algoritmo ENDPOINT-CYCLE-TERMINALS. Por otro lado, si el grafo de entrada es CC, entonces el algoritmo ENDPOINT-CYCLE-TERMINALS parte de algún VCT identificado previamente y se explora el grafo simultáneamente en ambos sentidos para encontrar las secciones C en los extremos. Después, se elige un vértice de cada sección C como los VEX.

Pseudocódigo 1: LINEAR-CACTUS-2PACKING(V_K, Adj_K)

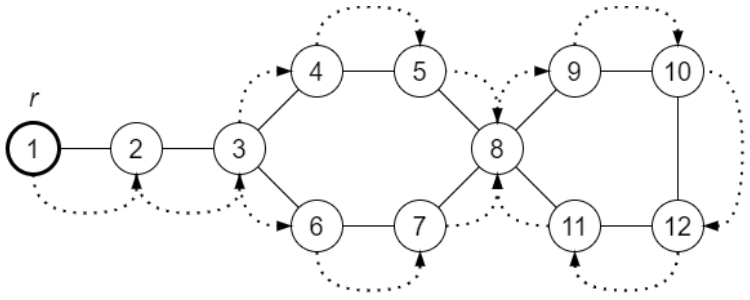
```
1  r, part ← PARTITION-VERTICES(V_K, Adj_K)
2  dist, next, tail ← PATH-DECOMPOSITION(r, part, V_K, Adj_K)
3  s, color ← GRAPH-MARKING(r, dist, next, tail, part, V_K, Adj_K)
4  return s, color
```

Pseudocódigo 2: PARTITION-VERTICES(V_K, Adj_K)

```
1  A[i] ← nil ∀i : 1 ≤ i ≤ 3                    ▷ Arreglo de vértices según su grado
2  c ← 0                                          ▷ Contador de vértices de grado 1
3  for each vertex i ∈ V_K do
4  |  switch δ(i) do
5  |  |  case 1 do                                ▷ El vértice i es de grado 1
6  |  |  |  c ← c + 1
7  |  |  |  part[i] ← true
8  |  |  |  A[1] ← i
9  |  |  end
10 |  |  case 2 do                                ▷ El vértice i es de grado 2
11 |  |  |  part[i] ← false
12 |  |  |  A[2] ← i
13 |  |  end
14 |  |  otherwise do                             ▷ El vértice i es de grado 3 o 4
15 |  |  |  part[i] ← true
16 |  |  |  A[3] ← i
17 |  |  end
18 |  end
19 end
20 if A[1] = nil and A[3] = nil then              ▷ El grafo es un ciclo
21 |  part[A[2]] ← part[Adj_K[A[2]][1]] ← true
22 |  return A[2], part
23 end
24 if A[1] = nil and A[3] ≠ nil then              ▷ El grafo es CC
25 |  r ← ENDPOINT-CYCLE-TERMINALS(A[3], part, V_K, Adj_K)
26 |  return r, part
27 end
28 if c = 1 then                                  ▷ El grafo es LC
29 |  ENDPOINT-CYCLE-TERMINALS(A[1], part, V_K, Adj_K)
30 end
31 return A[1], part                              ▷ El grafo es LL o LC
```

En cuanto al algoritmo ENDPOINT-CYCLE-TERMINALS, este realiza una exploración sobre el grafo de forma similar a como se realiza en el algoritmo PATH-DECOMPOSITION, el cual se explica más adelante. La diferencia entre estos dos algoritmos es que el algoritmo ENDPOINT-CYCLE-TERMINALS, además de recordar cuáles vértices ya fueron visitados, no registra información alguna conforme realiza la exploración. Dadas las similitudes entre estos dos algoritmos y con el propósito de compactar el pseudocódigo aquí presentado, se decidió omitir el pseudocódigo del algoritmo ENDPOINT-CYCLE-TERMINALS pero se muestra cómo funciona en la Figura 4 se muestra un ejemplo de cómo funciona. En dicha figura, tanto el vértice 9 como el 11 pueden elegirse como el VEX complementario al vértice 1 pero solo es necesario elegir uno de ellos. En este caso, se eligió el vértice 11 como el VEX por ser el último vértice que se visitó justo antes de regresar al vértice 8 que da origen a esa sección C.

Por último, cuando el grafo consiste únicamente de un ciclo, se puede elegir cualquier par de vértices adyacentes como los VEX del grafo. Una vez elegidos los VEX, se elige uno de ellos como la raíz de la exploración. Cabe mencionar que en todos los casos descritos anteriormente se asigna el valor de “verdadero” a la variable *part* de los VEX elegidos.



i	1	2	3	4	5	8	6	7	8	9	10	12	11	8
Origen	-	1	1	3	3	3	3	3	3	8	8	8	8	8
***part*[*i*]**	1	0	1	0	0	1	0	0	1	0	0	0	0	1

Figura 4. Proceso de exploración del algoritmo PARTITION-VERTICES para identificar los VDP en un grafo cactus lineal de ejemplo. Las flechas punteadas indican el orden en que se explora el grafo. Debajo del grafo, se ilustran tres arreglos. El arreglo denotado como *i* representa los identificadores de cada vértice, listados en el mismo orden en que fueron explorados. El arreglo denotado como *part*[*i*] representa las banderas booleanas que indican si el vértice *i*, en la columna correspondiente, es un VDP (1) o no (0). El arreglo denotado como *Origen* contiene el identificador del último VDP visitado antes de visitar el vértice *i*. La columna que está resaltada con un rectángulo indica que ahí se detectó la sección C en el extremo del grafo, ya que el valor de *i* y de *Origen* son iguales. El vértice resaltado con un contorno grueso y denotado con el símbolo *r* es el vértice elegido como la raíz de la exploración.

La Figura 5 muestra un grafo de ejemplo después de aplicar el algoritmo PARTITION-VERTICES. Se puede observar que se eligió el vértice 1 como la raíz de la exploración por ser el único vértice de grado uno. Además, el valor de la bandera *part* permite distinguir cuáles vértices son VDP y cuáles no.

3.3.2. Descomposición en secciones lineales

Después de identificar los VDP y de elegir un vértice raíz, el siguiente paso es descomponer el grafo, incluyendo las secciones C, en secciones L. Por ejemplo, en la Figura 7, los vértices marcados de negro son los VDP de ese grafo. Dicho grafo puede entonces descomponerse en las siguientes secciones L: $(1, 2, 3)$, $(3, 4, 5, 8)$, $(3, 6, 7, 8)$, $(8, 9, 10, 12)$ y $(8, 11)$.

El algoritmo que se usa para descomponer el grafo se denomina PATH-DECOMPOSITION, y se apoya de la sub-rutina PATH-EXPLORATION (Pseudocódigos 3 y 4). Este algoritmo en sí no descompone el grafo, sino que más bien lo explora para recabar información relevante a partir de la cual se puede deducir posteriormente en qué punto del grafo comienza y termina cada sección L en que puede descomponerse. Así, este algoritmo consiste de visitar cada VDP, i, y por cada vértice j adyacente a él, se recorre el camino p que comienza en la arista ij y termina en el VDP más cercano. El camino recorrido constituye entonces una sección L en el grafo. Por cada sección L recorrida, se registra el vértice j que completa la primera arista de esa sección, el VDP donde terminó la sección y la distancia recorrida. Esta información se denota respectivamente como $next[i][p]$, $tail[i][p]$ y $dist[i][p]$ en el pseudocódigo de este algoritmo.

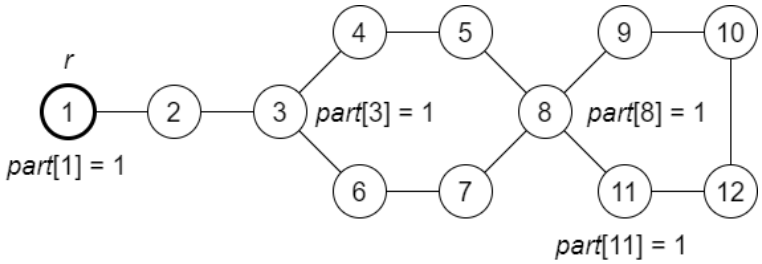


Figura 5. Un grafo cactus lineal de ejemplo después de identificar sus VDP con el algoritmo PARTITION-VERTICES. Los vértices identificados como VDP tienen asignado el valor de '1' en su bandera *part* correspondiente. El resto de los vértices, cuyo valor *part* no se muestra en la imagen, tienen asignado el valor de '0'. El vértice recalcado con un contorno grueso y denotado con el símbolo *r* es el vértice elegido como la raíz de la exploración.

El algoritmo PATH-DECOMPOSITION explora el grafo de tal forma que no avanza al siguiente VDP hasta no terminar de recorrer todos las secciones L adyacentes al VDP que se está visitando en ese momento. Además, mantiene un registro de cuáles vértices del grafo ya fueron visitados para evitar visitarlos más de una vez. Esto último se logra por medio de un marcado de vértices que consiste de los siguientes colores:

1. Blanco: indica que el vértice en cuestión no se ha visitado todavía. Este color se denota como WHITE.
2. Gris: indica que el vértice en cuestión ya se visitó, pero puede visitarse de nuevo por tratarse de un VDP que da cierre a una sección C en el grafo. Este color se denota como GRAY.
3. Negro: indica que el vértice en cuestión ya se visitó y, por ende, no debe volverse a visitar. Este color se denota como BLACK.

Pseudocódigo 3: PATH-DECOMPOSITION$(r, part, V_K, Adj_K)$

```
1  C[i] ← WHITE ∀i ∈ V_K                          ▷ El color de marcado de cada vértice
2  tail[i][j] ← nil ∀i ∈ V_K ∀j : 1 ≤ j ≤ 2
3  C[r] ← BLACK
4  Q ← ∅                                          ▷ Cola de los VDP a ser visitados
5  ENQUEUE(Q, r)
6  while Q ≠ ∅ do
7      i ← DEQUEUE(Q)
8      p ← 0                                      ▷ Contador de arreglos lineales adyacentes a i
9      for each vertex j ∈ Adj_K[i] do
10         if C[j] =WHITE or C[j] =GRAY then
11             p ← p + 1
12             t, d ← PATH-EXPLORATION(j, C, part, Adj_K)
13             next[i][p] ← j                     ▷ El vecino de i donde comienza el arreglo lineal p
14             dist[i][p] ← d                     ▷ La longitud del arreglo lineal p
15             tail[i][p] ← t                     ▷ El VDP donde termina el arreglo lineal p
16             if C[t] = WHITE then
17                 C[t] ← GRAY
18                 ENQUEUE(Q, t)
19             end
20         end
21     end
22     C[i] ← C[tail[i][p]] ←BLACK
23 end
24 return dist, next, tail
```

Pseudocódigo 4: PATH-EXPLORATION($i, C, part, Adj_K$)

```
1 t ← i
2 d ← 1
3 while not part[t] do
4 |   C[t] ← BLACK
5 |   t ← j ∈ Adj_K[t] : C[j] = WHITE or C[j] = GRAY
6 |   d ← d + 1
7 end
8 return t, d
```

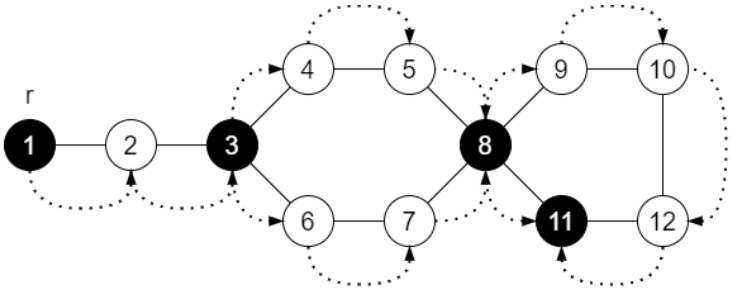
En los Pseudocódigos 3 y 4 se utiliza $C[i]$ para denotar el color asignado al vértice i durante la exploración del grafo. Cabe mencionar que al inicio de la exploración, todos los vértices comienzan marcados de blanco.

Al terminar la exploración del grafo, cada VDP tiene registrados a lo más dos secciones L adyacentes; esto es, cada una de las variables *next*, *dist* y *tail* correspondientes a dicho vértice tiene registrado uno o dos valores. Esto se debe a que cada VDP tiene un máximo de cuatro vecinos y a lo más dos de ellos estarían marcados de negro al momento que el algoritmo PATH-DECOMPOSITION visita dicho vértice. Así mismo, el último VDP visitado por este algoritmo no tiene ningún valor registrado en *next*, *dist* y *tail*, pues al momento que se visita dicho vértice ya no quedan secciones L que hagan falta recorrer. El número de secciones L adyacentes que tiene registrado cada VDP es una característica que se aprovecha más adelante, en la construcción del CIF, para distinguir si un VDP determinado da origen a una sección L o a una sección C en el grafo y para distinguir si ya se terminó de recorrer el grafo completo o no.

La Figura 6 muestra un grafo de ejemplo y el orden en que el algoritmo PATH-DECOMPOSITION explora los vértices. La Figura 7 muestra el mismo grafo de ejemplo después de aplicar este algoritmo.

3.3.3. Marcado del CIF máximo

Una vez terminada la exploración del grafo, se cuenta con suficiente información para construir el CIF máximo y el algoritmo que se encarga de ello se denomina GRAPH-MARKING (Pseudocódigo 5). Este algoritmo marca los vértices utilizando los colores que se presentan a continuación. Cada color puede representarse como un número entero.



			(a)				(b)				(c)					(d)		(e)
i	1	2	3	3	4	5	8	3	6	7	8	8	9	10	12	11	8	11
***part*[*i*]**	1	0	1	1	0	0	1	1	0	0	1	1	0	0	0	1	1	1
Origen	-	1	1	-	3	3	3	-	3	3	3	-	8	8	8	8	-	8
***dist*[*i*]**	0	1	2	0	1	2	3	0	1	2	3	0	1	2	3	4	0	1

Figura 6. Proceso de exploración del algoritmo PATH-DECOMPOSITION para identificar todas las secciones L en que puede descomponerse un grafo cactus lineal de ejemplo. Las flechas punteadas indican el orden en que se explora el grafo. Debajo del grafo se ilustran cuatro arreglos. El arreglo denotado como *i* representa los identificadores de cada vértice, listados en el mismo orden en que se exploraron. El arreglo *part*[*i*] representa las banderas booleanas que indican si el vértice *i*, en la columna correspondiente, es un VDP (1) o no (0). El arreglo *Origen* contiene el identificador del último VDP visitado antes de visitar el vértice *i*. El arreglo *dist*[*i*] representa la distancia de separación entre el vértice *Origen* y el vértice *i*. Las columnas (a), (b), (c), (d) y (e) indican los diferentes puntos en que el algoritmo detectó que terminó de recorrer un arreglo lineal en el grafo.

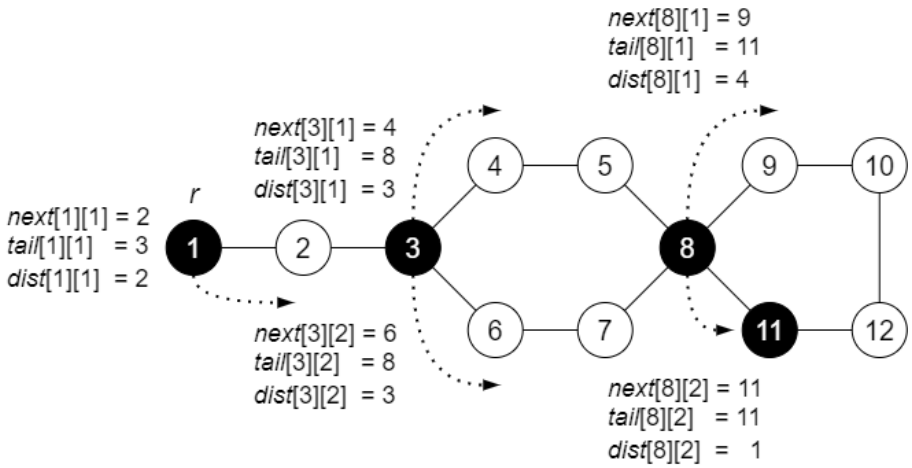


Figura 7. Un grafo cactus lineal de ejemplo después de identificar todas las secciones L en que puede descomponerse con el algoritmo PATH-DECOMPOSITION. Los vértices negros son los VDP del grafo. Cada uno de ellos tiene registrados no más de dos secciones L adyacentes, representados por las flechas punteadas y por los valores de *next*, *tail* y *dist* correspondientes. El vértice denotado con el símbolo *r* es el vértice raíz de la exploración del grafo. El vértice 11, que es un VEX y, por ende, no tiene registrado ninguna sección L.

Pseudocódigo 5: GRAPH-MARKING($r, dist, next, tail, part, V_K, Adj_K$)

```
1  color[i] ← GREEN ∀i ∈ V_K
2  i ← r
3  while tail[i][1] ≠ nil do
4  |  if tail[i][2] = nil then                    ▷ El vértice i origina una sección L
5  |  |  MARK-PATH(i, next[i][1], dist[i][1] − 1, color, Adj_K)
6  |  |  c ← (color[i] + dist[i][1]) mód 3
7  |  |  if c ≠ RED then color[tail[i][1]] ← c end
8  |  else                                        ▷ El vértice i origina una sección C
9  |  |  MARK-CYCLE(i, dist, next, color, part, Adj_K)
10 |  end
11 |  i ← tail[i][1]
12 end
```

- Verde: indica que el vértice está disponible. Este color se representa con ∞ y se denota como GREEN.
- Naranja: indica que el vértice está restringido y es adyacente a algún otro vértice marcado de rojo. Este color se representa con el número '1' y se denota como ORANGE.
- Amarillo: indica que el vértice está restringido y además se encuentra a una distancia de al menos dos de algún otro vértice marcado de rojo. Este color se representa con el número '2' y se denota como YELLOW.
- Rojo: indica que el vértice está empaquetado. Este color se representa con el número '0' y se denota como RED.

Se utiliza $color[i]$ en el pseudocódigo de este algoritmo para denotar el color asignado al vértice con identificador i. Al inicio del algoritmo GRAPH-MARKING, todos los vértices comienzan marcados de verde.

Antes de describir detalladamente en qué consiste el algoritmo GRAPH-MARKING, es importante mencionar que cada vez que un vértice se marca de rojo, se deben realizar dos tareas adicionales:

(a) Se debe marcar de naranja cualquier vértice adyacente que esté marcado de amarillo.

(b) Se debe incrementar en uno el valor del contador de vértices rojos s que se describe en el pseudocódigo del algoritmo LINEAR-CACTUS-2PACKING.

La tarea (a) es importante para marcar correctamente una sección L en el grafo. En este caso, cuando un vértice se marca de amarillo, el siguiente vértice en la secuencia puede marcarse de rojo. Al hacerlo, el vértice que originalmente era amarillo ya no cumple con la definición de ese color puesto que ahora es adyacente a un vértice rojo. Para que el marcado sea correcto, dicho vértice debe marcarse de naranja. Por otro lado, el propósito de la tarea (b) es simplemente llevar un registro de la cardinalidad del CIF encontrado hasta ese momento. Aunque importantes, estas dos tareas se omiten en el pseudocódigo para hacerlo más compacto; se supone a partir de ahora que estas tareas se llevan a cabo de forma implícita.

En cuanto al funcionamiento detallado del algoritmo GRAPH-MARKING , éste recorre los VDP del grafo en el mismo orden que lo hizo el algoritmo PATH-DECOMPOSITION y aplica diferentes reglas de marcado dependiendo si dicho vértice tiene registradas uno o dos secciones L; esto es, si dicho vértice da origen a un sección L (en cuyo caso se aplica la sub-rutina MARK-PATH, Pseudocódigo 6) o a una sección C (en cuyo caso se aplica la sub-rutina MARK-CYCLE, Pseudocódigo 7). Si el VDP visitado no tiene ningún arreglo lineal registrado, entonces eso significa que ya se marcaron todos los vértices del grafo y el algoritmo GRAPH-MARKING puede terminar su ejecución.

El algoritmo MARK-PATH se encarga de marcar los vértices del CIF máximo en una sección L. Este algoritmo consiste simplemente de recorrer la sección L de un extremo al otro y marcar cada vértice dependiendo del color del vértice anterior de la siguiente manera:

- Si el vértice anterior al vértice visitado está marcado de rojo, entonces el vértice visitado se marca de naranja.

- Si el vértice anterior está marcado de naranja, entonces el vértice visitado se marca de amarillo.
- Si el vértice anterior está marcado de amarillo, entonces el vértice visitado se marca de rojo. Además, implícitamente se marca el vértice anterior de naranja y se incrementa en uno el contador *s* del número de vértices marcados de rojo.

Pseudocódigo 6: MARK-PATH$(x, j, d_{max}, color, Adj_K)$

```
1  if color[x] = GREEN then color[x] ← RED end
2  i ← x
3  d ← 1
4  while d ≤ d_max do
5  |  color[j] ← (color[i] + 1) mód 3
6  |  i ← j
7  |  j ← k ∈ Adj_K[j] : color[k] = GREEN
8  |  d ← d + 1
9  end
10 return i, j
```

Pseudocódigo 7: MARK-CYCLE$(x, next, dist, color, part, Adj_K)$

```
1  if dist[x][1] = 1 then d_A ← 1* else d_A ← dist[x][1] mód 3
2  if dist[x][2] = 1 then d_B ← 1* else d_B ← dist[x][2] mód 3
3  if color[x] =GREEN then
4  |  if d_A = d_B = 2 then color[x] ←RED else color[x] ←YELLOW end
5  end
6  p_1, p_2 ← CYCLE-PATH-ORDER(color[x], d_A, d_B)
7  if dist[x][1] + dist[x][2] = 3 then                    ▷ Ciclo de tamaño 3
8  |  a ← b ← x
9  |  a' ← next[x][p_2]
10 |  b' ← next[x][p_1]
11 else
12 |  if dist[x][1] = 1 or dist[x][2] = 1 then            ▷ Ciclo con VDP adyacentes
13 |  |  a ← x
14 |  |  a' ← next[x][p_2]
15 |  |  b, b' ← MARK-PATH (x, dist[x][p_1] − 2, next[x][p_1], color, Adj_K)
16 |  else                                               ▷ Ciclo con VDP no adyacentes
17 |  |  a, a' ← MARK-PATH (x, dist[x][p_1] − 1, next[x][p_1], color, Adj_K)
18 |  |  b, b' ← MARK-PATH (x, dist[x][p_2] − 2, next[x][p_2], color, Adj_K)
19 |  end
20 end
21 color[a'], color[b'] ← MARK(color[a], color[b])
```

Se puede aprovechar el hecho de que cada color se representa por medio de un número entero para calcular el color de cada vértice visitado a partir del color del vértice anterior utilizando la operación módulo.

En la mayoría de los casos, cuando se marca una sección L, el VDP x que da origen a esa sección ya tiene asignado un color, debido a que se marcó en la sección previa del grafo. Sin embargo, en caso de que x esté marcado de verde, este vértice se puede marcar de rojo, pues es el color que eventualmente lleva a encontrar el CIF de mayor cardinalidad en esa sección.

Otro detalle de este algoritmo es que, si el VDP al final de la sección L debe marcarse de rojo, entonces se deja marcado de verde. Hay que recordar que este vértice da inicio a una sección C en la siguiente parte el grafo. Por ende, si se marca de rojo, se estarían restringiendo muchos vértices de esa sección, lo que a su vez reduciría la cardinalidad del CIF que podría construirse. Por esta razón, es preferible dejar ese vértice sin marcar de tal forma que, cuando sea turno de marcar esa sección C, ese vértice estaría disponible y podría ser marcado del color que más conviene en ese momento.

En la Figura 8 se muestra cómo se aplica el algoritmo MARK-PATH sobre la sección L en el extremo izquierdo del grafo que se ha usado como ejemplo.

El algoritmo MARK-CYCLE se encarga de marcar los vértices en una sección C del grafo. Como se explicó anteriormente, las secciones C pueden descomponerse en dos secciones L que comienzan y terminan en el mismo par de VDP. Por ende, se puede utilizar el algoritmo MARK-PATH para marcar los vértices en ambas secciones L de forma individual. El problema con este procedimiento es que un mismo vértice compartido por ambas secciones puede marcarse de dos colores diferentes. Para resolver este conflicto, las secciones L se marcan parcialmente y los vértices que quedan pendientes se pueden marcar de forma óptima dependiendo del tamaño de la sección C y del último color asignado en cada sección L. Este procedimiento produce el CIF de mayor cardinalidad posible en la sección C que además restringe la menor cantidad de vértices posibles en el resto del grafo.

Concretamente, el algoritmo MARK-CYCLE consiste de los siguientes pasos:

1. Se calcula el módulo 3 de la longitud de las secciones L que conforman la sección C. Si la longitud de uno de ellos es exactamente 1, se utiliza el símbolo 1* para distinguirlo de otras longitudes cuyo módulo 3 también resultan en un valor de 1. Los resultados obtenidos por estos cálculos se denotan d_A y d_B.

2. Si el VDP que da origen a la sección C, denotado como x, está marcado de verde, entonces se marca con el color más conveniente, según el tamaño del ciclo.

3. Dependiendo de los valores de d_A y d_B y del color del vértice x, se decide qué sección L debe marcarse primero y cuál después. La sección que se marca primero se denota p_1 y la que se marca después se denota p_2.

4. Se marcan todos los vértices de la sección p_1, utilizando el algoritmo MARK-PATH, con la excepción del último vértice, esto es, el VDP que cierra la sección C. El último vértice marcado por MARK-PATH se denota a y el VDP que cierra la sección C se denota a'.

5. Se marcan todos los vértices de la sección p_2, utilizando el algoritmo MARK-PATH, con la excepción de los últimos dos vértices, esto es, el VDP que cierra la sección C y el vértice que antecede a dicho vértice en la sección p_2. El último vértice marcado por MARK-PATH se denota b y el vértice que le sigue en la sección p_2 se denota b'.

6. Note que al concluir los dos pasos anteriores, los vértices a y b están marcados y los vértices a' y b' quedan pendientes de marcar. El color con el que se deben marcar a' y b' se eligen dependiendo de los colores asignados a los vértices a y b.

La Figura 9 muestra cómo se aplica este procedimiento para marcar una de las secciones C del grafo de ejemplo. Note que en el inciso (a) de esta figura, el vértice 3, que da origen a la sección C, está marcado de amarillo ya que éste es el color que le asignó el algoritmo MARK-PATH en la Figura 8.

En el paso 3 del procedimiento anterior, el orden del marcado de las secciones L que conforman la sección C se decide dependiendo de los valores de d_A y d_B y del color del VDP que da origen a dicha sección. El orden óptimo de marcado para todos los posibles casos se muestra en la Tabla 4. Así, la sub-rutina CYCLE-PATH-ORDER puede decidir en qué orden se deben marcar las secciones L consultando dicha tabla. En el ejemplo de la Figura 9, dado que ambas secciones L son de la misma longitud, se marcan en el mismo orden en que se exploraron.

De forma similar, en el paso 4 del procedimiento anterior, se tiene que los colores de a' y b' se deciden dependiendo de los colores de a y b. Los colores óptimos para cada posible caso también están precalculados y se muestran en la Tabla 5. Así, la subrutina MARK puede decidir los colores de a' y b' consultando dicha tabla. En el ejemplo de la Figura 9, dado que el vértice a se marcó de naranja y el vértice b se marcó de amarillo, los vértices a' y b' se marcan de naranja y rojo respectivamente.

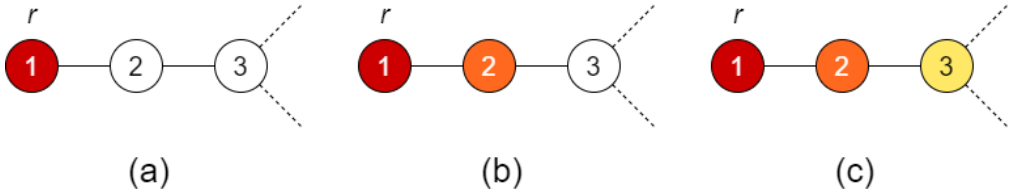


Figura 8. Proceso de marcado del CIF en una sección L de ejemplo utilizando el algoritmo MARK-PATH. Sin importar la longitud del camino o el color inicial de *r*, los vértices se marcan siguiendo esta misma secuencia de colores. (a) El vértice *r* se marca de rojo porque es el color que más conviene para este caso. (b)-(c) Marcado del resto de los vértices de la sección.

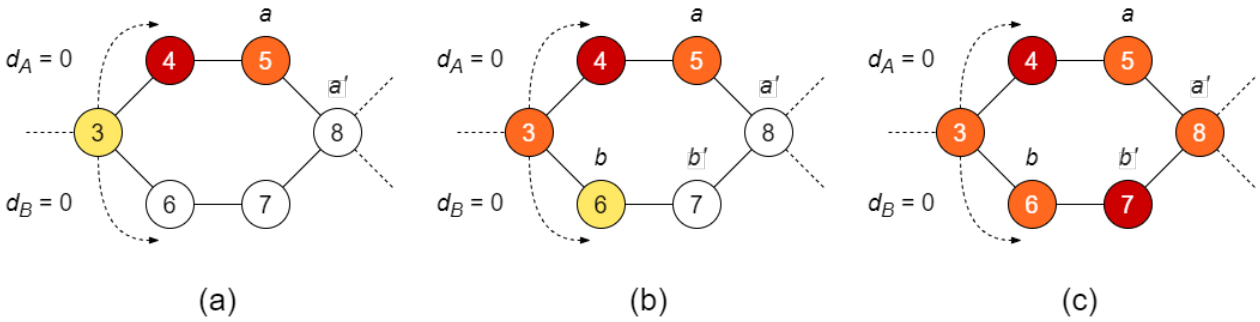


Figura 9. Proceso de marcado del CIF en una sección C de ejemplo utilizando el algoritmo MARK-CYCLE. El orden en que se debe marcar cada arreglo lineal que conforma esta sección se decide utilizando la Tabla 4. (a) Se marca la sección L superior utilizando el algoritmo MARK-PATH y deteniéndose un vértice antes de llegar al VDP que está al final de dicha sección. (b) Se marca la sección L inferior utilizando el mismo algoritmo, pero deteniéndose dos vértices antes de llegar al VDP al final de dicha sección. (c) Se marcan los dos vértices que faltan de la sección C utilizando la Tabla 5.

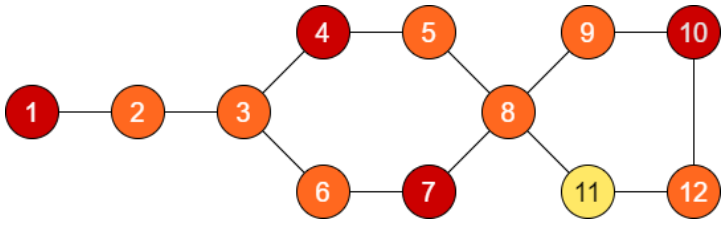


Figura 10. Ejemplo del CIF máximo en un grafo cactus lineal después de aplicar el algoritmo LINEAR-CACTUS-2PACKING.

Tabla 4. Orden óptimo en que deben marcarse las secciones L que conforman un ciclo en un grafo cactus lineal. Los renglones de esta tabla se interpretan de la siguiente manera: dadas dos secciones L, A y B, si d_A y d_B tienen los valores indicados en la primera y segunda columna y el VDP que da origen a la sección C posee uno de los colores indicados en la parte superior de la tabla, entonces las secciones L deben marcarse en el orden que se indica en las dos columnas debajo del color correspondiente, donde p_1 denota la sección L que debe marcarse primero y p_2 la que debe marcarse después.

d_A	d_B	YELLOW		ORANGE		RED	
		p_1	p_2	p_1	p_2	p_1	p_2
0	0	*A*	*B*	*A*	*B*	*A*	*B*
0	1	*A*	*B*	*B*	*A*	*B*	*A*
0	2	*A*	*B*	*B*	*A*	*B*	*A*
0	1*	*A*	*B*	*A*	*B*	*A*	*B*
1	0	*B*	*A*	*A*	*B*	*A*	*B*
1	1	*A*	*B*	*A*	*B*	*A*	*B*
1	2	*B*	*A*	*B*	*A*	*B*	*A*
1	1*	*A*	*B*	*A*	*B*	*A*	*B*
2	0	*B*	*A*	*A*	*B*	*A*	*B*
2	1	*A*	*B*	*A*	*B*	*A*	*B*
2	2	*A*	*B*	*A*	*B*	*A*	*B*
2	1*	*A*	*B*	*A*	*B*	*A*	*B*
1*	0	*B*	*A*	*B*	*A*	*B*	*A*
1*	1	*B*	*A*	*B*	*A*	*B*	*A*
1*	2	*B*	*A*	*B*	*A*	*B*	*A*

Tabla 5. Colores para marcar de forma óptima los dos vértices que cierran una sección C en un grafo cactus lineal. Los renglones de esta tabla se interpretan de la siguiente manera: si el vértice a está marcado con el color indicado en la primera columna y b está marcado con el color de la segunda, entonces el vértice a' se debe marcar con el color indicado en la tercera columna y b' con el color indicado en la cuarta.

a	**b**	$\boldsymbol{a'}$	$\boldsymbol{b'}$
RED	RED	ORANGE	ORANGE
	ORANGE	ORANGE	YELLOW
	YELLOW	ORANGE	YELLOW
ORANGE	RED	YELLOW	ORANGE
	ORANGE	YELLOW	YELLOW
	YELLOW	ORANGE	RED
YELLOW	RED	YELLOW	ORANGE
	ORANGE	RED	YELLOW
	YELLOW	ORANGE	RED

Una característica que comparten los algoritmos MARK-PATH y MARK-CYCLE es que ambos marcan los vértices de cada sección de tal forma que el conjunto de vértices rojos constituye un CIF maximal de la mayor cardinalidad posible al mismo tiempo que se restringen la menor cantidad de vértices de las secciones posteriores. Esto permite que el marcado de una sección interfiera lo mínimo posible con el marcado de la sección siguiente y, como consecuencia, al unir el marcado de todas las secciones, se obtiene el marcado óptimo del grafo completo. En la Figura 10 se muestra cómo quedaría marcado el grafo de ejemplo después de aplicar el algoritmo LINEAR-CACTUS-2PACKING. Se puede ver que para este grafo, el CIF consiste de cuatro vértices: $\{1, 4, 7, 10\}$.

3.4. Análisis

En esta sección se presenta el análisis del algoritmo LINEAR-CACTUS-2PACKING, el cual consiste de varios lemas que culminan por demostrar el Teorema 3.1. En cada lema, se demuestra que cada sub-rutina que conforma el algoritmo es correcta. Cabe mencionar que, con el fin de compactar las demostraciones aquí presentadas, las demostraciones por invariante de lazo se movieron al Anexo .

Lema 3.1. *Sea K un grafo cactus lineal de n vértices y n aristas. El algoritmo* PARTITION-VERTICES*, identifica todos los VDP de K y entrega como resultado un VEX, r.*

Demostración. Se debe demostrar que PARTITION-VERTICES identifica correctamente los VCT y los VEX del grafo. Este algoritmo comienza por recorrer todos los vértices del grafo para marcar como VDP aquellos que son de grado 1, 3 y 4 y registra el último vértice visitado de grado 1, 2 y 3 o 4. Este procedimiento se describe en el ciclo de las líneas 3-19 del Pseudocódigo 2. Dado que, por definición, un VCT es un vértice de grado 3 o 4, se tiene que PARTITION-VERTICES logra identificar correctamente todos los VCT del grafo.

A continuación, el algoritmo utiliza el registro que tomó del último vértice visitado de cada grado para decidir si el grafo K se trata de un ciclo, un grafo LL, uno LC o uno CC. Posteriormente, se aplica el procedimiento correspondiente para encontrar correctamente los VEX del grafo. Este procedimiento se describe en las líneas 20-31.

Una vez encontrados los VEX, se elige uno de ellos como el vértice resultante *r*. Cabe mencionar que, cuando el grafo de entrada es LC o CC, se debe aplicar el algoritmo ENDPOINT-CYCLE-TERMINALS, el cual es una adaptación del algoritmo PATH-DECOMPOSITION, que más adelante se demuestra que es correcto.

Dado que el algoritmo PARTITION-VERTICES identifica correctamente los VCT y los VEX del grafo, el algoritmo es correcto. □

Lema 3.2. *Dado un VEX, r, la rutina* PATH-DECOMPOSITION *recorre todos los VDP de K siguiendo una secuencia determinada, comenzando por r y continuando con el VDP más cercano, t. Por cada VDP denotado como i, se registra la siguiente información sobre a lo más dos secciones L que conectan i con t: el vértice adyacente a i que completa la primera arista de dicha sección, su longitud y el VDP t donde termina.*

Demostración. Para demostrar que esta rutina es correcta, se utiliza el método de la invariante de lazo que se presenta a continuación.

Antes de comenzar cada iteración del ciclo de las líneas 5-23 del Pseudocódigo 3, se tiene lo siguiente:

(a) La cola *Q* contiene únicamente el VDP *i*.

(b) Todos los vértices desde *r* hasta *i* están marcados de negro.

(c) Todos los VDP marcados de negro, excepto *i*, tienen registradas la información de a lo más dos secciones L adyacentes a ellos, de la misma forma que se describe en el lema anterior.

(d) El vértice *i* es el VDP donde terminan las secciones L registradas en el VDP visitado en la iteración anterior.

(e) El VDP *i* tiene a lo más dos vecinos marcados de blanco.

Una vez demostrado que la invariante de lazo se cumple en la inicialización, mantenimiento y finalización, el resultado entregado por el algoritmo PATH-DECOMPOSITION cumple con el Lema 3.2. □

Lema 3.3. *Sea P una sección L de longitud d_{max} que comienza en la arista* (x, j)*, donde x es un VDP y donde todos los vértices, excepto quizás x, están marcados de verde. El algoritmo* MARK-PATH *marca los vértices de P de tal forma que el conjunto de vértices rojos en P constituye un CIF que es de la mayor cardinalidad posible considerando la restricción establecida por el color de x.*

Demostración. La correctitud de este algoritmo se demuestra por medio de la siguiente invariante de lazo: antes de comenzar cada iteración del ciclo de las líneas 3-4 del Pseudocódigo 6, el marcado del camino que comienza en el vértice x y termina en el vértice i constituye el CIF de mayor cardinalidad posible considerando la restricción establecida por el color de x, además, el vértice j está marcado de verde.

Después de demostrar que la invariante de lazo se cumple en la inicialización, mantenimiento y finalización, el algoritmo MARK-PATH cumple el Lema 3.3. □

Lema 3.4. *Dada una sección C donde todos los vértices están disponibles, tal vez con excepción del vértice x, y que está constituida por dos secciones L que comienzan ambas en el VDP denotado x y terminan en el mismo VDP denotado t, el algoritmo* MARK-CYCLE *marca los vértices de ambas secciones L de tal forma que los vértices rojos constituyen el CIF de mayor cardinalidad posible, considerando el color de x, y el vértice t tiene el color más cercano posible a amarillo.*

Demostración. Para demostrar la correctitud de este algoritmo, se deben considerar todos los posibles casos que pueden presentarse en la entrada de este algoritmo. Cada sección C se distingue por medio de tres variables: el color de x, el módulo tres de la longitud de cada sección L y el orden en que se marcan ambas secciones L (esto es, cuál sección se marca primero). Así, se tienen en total 45 configuraciones diferentes en que puede presentarse la sección C recibida como la entrada del algoritmo. Si se desglosan todos estos casos, se puede elegir el marcado óptimo de cada uno que satisface las condiciones descritas en el lema anterior.

Suponiendo que x está disponible, MARK-CYCLE comienza por determinar el color que más conviene para x. Para ello, primero se calcula el módulo tres de la longitud de cada sección L que conforma la sección C de entrada. El resultado de este cálculo para cada sección se denota d_A y d_B. Después, si se tiene que $d_A = d_B = 2$, el color que

más conviene para x es rojo, ya que, de esta forma, el vértice t quedará marcado de amarillo y el CIF encontrado para la sección C será el máximo. Por otro lado, si al menos uno de los valores d_A o d_B es distinto de dos, entonces el color que más conviene para x es amarillo, ya que, de esta forma, t quedará marcado de naranja o amarillo (dependiendo del caso) y el CIF resultante será máximo. Esto está garantizado porque los colores con los cuales se debe marcar x en cada posible caso se determinaron tras analizar exhaustivamente todos los posibles casos de entrada.

El siguiente paso es determinar el orden en que se deben marcar ambas secciones L. El orden es importante porque afecta el color de t. Se puede decidir cuál es el orden apropiado analizando todos los posibles casos de entrada. Los resultados de este análisis exhaustivo se describen en la Tabla 4. Al codificar esta tabla en MARK-CYCLE se puede decidir en tiempo constante el orden en que deben marcarse las secciones L.

Luego, se deben marcar las secciones L en el orden establecido y tomando en cuenta el color de x. Sin importar la longitud de las secciones L, la estrategia que se sigue aquí es la misma: se marcan ambas secciones L excepto por $t = a'$ y su vecino, b', que pertenece a la sección opuesta. Dependiendo de los colores de los vértices adyacentes a éstos dos en la sección correspondiente, denotados respectivamente como a y b, se puede decidir el color que más conviene para a' y b'. Dichos colores se pueden obtener al analizar exhaustivamente todos los posibles casos de entrada. Los colores más convenientes para estos vértices se describen en la Tabla 5. La codificación de esta tabla en MARK-CYCLE permite decidir en tiempo constante cuáles colores asignar a cada vértice.

Por último, cabe mencionar que, por el Lema 3.3, se tiene que cada sección L se marcó desde x hasta a y b de forma óptima, tomando en cuenta la restricción establecida por el color de x. Así, al sumar los colores de a' y b', el marcado resultante constituye el CIF máximo de la sección C a la vez que el color de t es lo más cercano posible a amarillo. □

Lema 3.5. *Dado un grafo cactus lineal, K, donde cada VDP tiene registrada información relevante sobre a lo más dos secciones L adyacentes y dado un VEX, r, el algoritmo* GRAPH-MARKING *marca los vértices de K de tal forma que el conjunto de vértices rojos constituye el CIF máximo de K.*

Demostración. Por el Lema 3.2, los vértices de K se recorrieron en una secuencia determinada que comienza en r y continúa hacia el VDP más cercano. Por la demostración de este lema, este orden se ve reflejado en las secciones L registradas en cada VDP; cada sección L termina en el VDP que sigue en la secuencia. Así, el algoritmo GRAPH-MARKING recorre los VDP de K en este mismo orden.

También por el Lema 3.2, cada VDP tiene registradas a los más dos secciones L. Esta información se utiliza para determinar si el VDP i visitado en cada iteración da origen a una sección L o una sección C en el grafo K. Dependiendo de cuál sea el caso, se aplica entonces el algoritmo MARK-PATH o MARK-CYCLE. Por los Lemas 3.3 y 3.4, estos algoritmos marcan la sección correspondiente de forma óptima, procurando introducir la menor cantidad de restricciones posibles en la siguiente sección en el grafo.

En caso de que el vértice i inicie una sección L, el algoritmo MARK-PATH se aplica a toda la sección excepto por el último vértice. Si dicho vértice fuera a marcarse de rojo, podría restringir hasta cuatro vértices de la siguiente sección del grafo, lo que podría reducir considerablemente la cardinalidad del CIF encontrado en esa sección. Así, si dicho vértice se va a marcar de rojo, se deja marcado de verde. Esto reduce en uno la cardinalidad del CIF construido para la sección L visitada en ese momento en uno, pero, por el Lema 3.3, el resto de la sección está marcada de tal forma que el CIF construido tiene la mayor cardinalidad posible. Además, dado que la sección que se origina en ese vértice es una sección C, por el Lema 3.4, ese vértice se va a marcar de forma óptima, de tal forma que el CIF construido para esa sección C será un CIF máximo. Así, la cardinalidad del CIF construido por el algoritmo para todo el grafo K no se ve reducida si el último vértice de la sección L no se marca de rojo.

Finalmente, dado que el algoritmo GRAPH-MARKING marca cada sección del grafo K de tal forma que el CIF construido en cada uno tiene la mayor cardinalidad posible, al finalizar este algoritmo, se construyó el CIF máximo de K. □

Teorema 3.1. *Dado un grafo cactus lineal K el algoritmo* LINEAR-CACTUS-2PACKING *aplica un marcado de vértices en tiempo lineal de tal forma que el conjunto de vértices rojos constituye el CIF máximo de K.*

Demostración. En el Pseudocódigo 1, este algoritmo consiste de tres pasos generales, representados por los algoritmos PARTITION-VERTICES, PATH-DECOMPOSITION y GRAPH-MARKING. Por el Lema 3.1, el algoritmo PARTITION-VERTICES identifica correctamente todos los VDP del grafo K y elige uno de los VEX como la raíz de la exploración. Después, por el Lema 3.2, el algoritmo PATH-DECOMPOSITION explora el grafo, partiendo de la raíz, y registra información sobre cada sección L en la que K puede descomponerse en cada VDP de este mismo grafo. Finalmente, por el Lema 3.5, se recorren los VDP comenzando por el vértice raíz y siguiendo el mismo orden en que se llevó a cabo la exploración. Además, por cada sección que comienza en cada VDP, dicha sección se marca de tal forma que el CIF construido tiene la mayor cardinalidad posible considerando las restricciones establecidas en la sección anterior y minimizando la cantidad de restricciones introducidas en la siguiente sección. Así, al unir el marcado de todas las secciones del grafo, se tiene que el CIF resultante es máximo. Por lo tanto, el algoritmo LINEAR-CACTUS-2PACKING resuelve correctamente el problema del CIF máximo en grafos cactus lineales.

En cuanto al tiempo de ejecución, se puede ver que este algoritmo tarda $O(n)$ tiempo, ya que cada vértice se visita a lo más una cantidad constante de veces. □

Capítulo 4. Heurística voraz para grafos arbitrarios

En este capítulo se presenta el algoritmo GREEDY-MAXIMAL-2PACKING, que es una heurística voraz que resuelve el problema del CIF maximal. Este algoritmo admite como entrada un grafo arbitrario y corre en tiempo polinomial. Además, se puede convertir fácilmente en un algoritmo aleatorio, de tal forma que se puede mejorar la cardinalidad de la solución encontrada a expensas de ejecutar el algoritmo varias veces sobre el mismo grafo de entrada.

En el Capítulo 5 se muestra que este algoritmo produce soluciones de mejor calidad que una heurística aleatoria ingenua. Además, en general, el algoritmo requiere un menor tiempo de ejecución que Gurobi. Dado que este algoritmo admite como entrada cualquier tipo de grafo, es atractivo desde un punto de vista práctico a pesar de resolver sólamente el problema del CIF maximal. Aunado al hecho de que este algoritmo admite como entrada cualquier tipo de grafo, esto hace que el uso del algoritmo GREEDY-MAXIMAL-2PACKING sea atractivo desde un punto de vista práctico a pesar de no resolver formalmente el problema del CIF máximo.

4.1. Suposiciones generales y notación a utilizar

Sea $G = (V_G, E_G)$ un grafo arbitrario, donde V_G denota el conjunto de vértices y E_G el conjunto de aristas. Se supone que G se representa por medio de un arreglo de listas de adyacencias que se denota como Adj_G. También se supone que cada vértice de G se identifica por medio de un número entero único, extraído del conjunto $\{1, 2, \ldots, n\}$. Así, $Adj_G[i]$ denota una lista enlazada que contiene los identificadores de los vértices adyacentes al vértice identificado por i.

Cuando una sub-rutina recibe como entrada una variable que representa un arreglo o cualquier otra estructura de datos no simple, se sigue por convención que cualquier modificación realizada sobre dicha estructura de datos dentro de la sub-rutina también se ve reflejada afuera de ella. Ésto se hace con el fin de compactar el pseudocódigo.

Durante la ejecución del algoritmo, se utiliza el siguiente conjunto de colores para categorizar los vértices de G. Cabe mencionar que estos colores pueden representarse como números enteros diferentes.

- Verde: indica que el vértice en cuestión está disponible. Se denota como GREEN.
- Amarillo: indica que el vértice en cuestión está restringido. Se denota como YELLOW.
- Rojo: indica que el vértice en cuestión está empaquetado. Se denota como RED.

Se utiliza $color[i]$ para denotar el color asignado al vértice i. Se utiliza una notación similar para representar otros valores relevantes asociados a cada vértice de G.

4.2. Descripción del algoritmo

El algoritmo GREEDY-MAXIMAL-2PACKING (Pseudocódigo 8) consiste de los siguientes pasos:

1. Cada vértice de V_G se marca de verde.
2. Por cada vértice verde i de V_G:
   a) Se cuentan cuántos vértices verdes se encuentran en $N_2[i]$. Esta cantidad se denota como $grn[i]$.
   b) Se cuentan cuántos vértices amarillos se encuentran en $N_2(i)$. Esta cantidad se denota como $yelw[i]$.
3. De los vértices verdes de V_G, se selecciona al vértice j tal que $grn[j]$ sea el menor de todos.
   a) En caso de empate, se elije aquél vértice cuyo valor de $yelw[j]$ sea el mayor entre los vértices empatados.
   b) Si aún así hay un empate, entonces se elije aquél vértice cuyo identificador sea el menor de los vértices empatados. Alternativamente, se pueden desempatar los vértices eligiendo uno de forma aleatoria.
4. El vértice j se marca de rojo y todos los vértices verdes que se encuentren en $N_2(j)$ se marcan de amarillo.
5. Se repiten los pasos del 2 al 4 mientras aún existan vértices verdes en V_G.
6. Finalmente, el CIF que encontró el algoritmo consiste de todos los vértices marcados de rojo.

En el pseudocódigo de este algoritmo se utiliza s para denotar la cardinalidad del CIF encontrado por el algoritmo.

La característica principal del algoritmo GREEDY-MAXIMAL-2PACKING es el conteo de vértices verdes y amarillos en el vecindario extendido de cada vértice i del grafo. Intuitivamente el conteo de vértices verdes indica cuántos vértices van a restringirse al empaquetarse el vértice i. Por otro lado, el conteo de vértices amarillos indica el tamaño de la intersección entre el conjunto de vértices restringidos en el grafo y el vecindario extendido del vértice i. Así, la estrategia voraz de este algoritmo consiste en empaquetar, en cada iteración, el vértice que minimice el número de restricciones introducidas al grafo y que, al mismo tiempo, maximice la compactación entre los vértices empaquetados en el CIF resultante.

La Figura 11 muestra cómo se aplica el algoritmo GREEDY-MAXIMAL-2PACKING sobre un grafo de ejemplo. En él se muestra el conteo de vértices verdes y amarillos para cada vértice. A pesar de que no se muestra cómo quedan ordenados los vértices dentro de la cola de prioridad, sí se resalta el vértice que se empaquetó, según los criterios descritos anteriormente; minimizando el conteo de verdes, maximizando el conteo de amarillos y, en caso de empate, eligiendo el vértice con el identificador más pequeño. El algoritmo logra encontrar el CIF máximo para este ejemplo particular, pero no está garantizado que este algoritmo encuentre el CIF máximo para un grafo arbitrario.

4.2.1. Conteo inicial de vértices disponibles

En la primera iteración del algoritmo GREEDY-MAXIMAL-2PACKING, se deben contar los vértices verdes que se encuentran en $N_2[i]$ para cada vértice i. El procedimiento que se encarga de hacer esto se denomina COUNT-NEIGHBORS-BY-COLOR (Pseudocódigo 9) y consiste de realizar una búsqueda en anchura (breadth-first-search o BFS, en inglés) acotada a una distancia de dos, partiendo del vértice origen. El pseudocódigo de este algoritmo es una adaptación del algoritmo BFS presentado por Cormen *et al.* (2009).

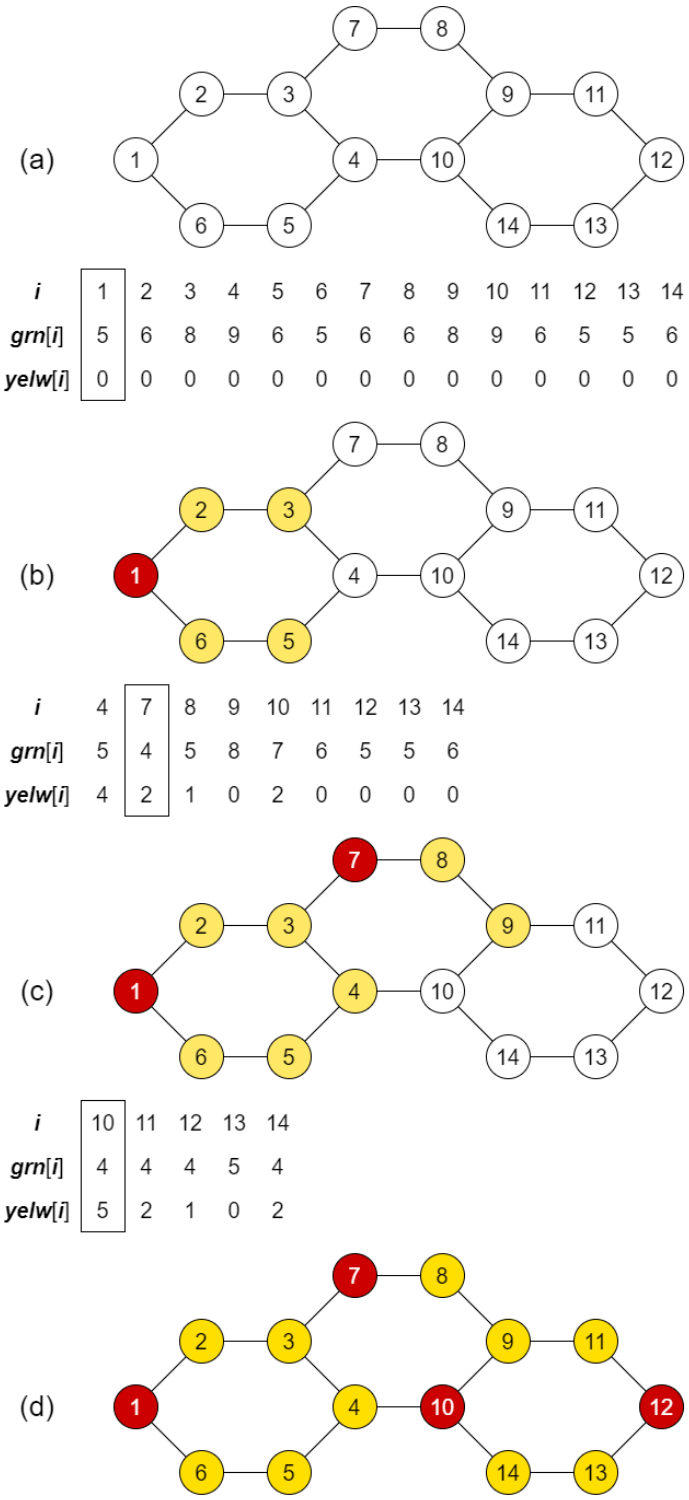
Se utiliza una cola ordinaria Q para almacenar los vértices cuyo vecindario falta por explorarse y se utiliza un marcado de vértices para evitar que un vértice se cuente más de una vez. Los colores utilizados por este marcado de vértices son los siguientes:

Pseudocódigo 8: GREEDY-MAXIMAL-2PACKING(V_G, Adj_G)

```
1  H[i] ← grn[i] ← yelw[i] ← pos[i] ← nil ∀i ∈ V_G
2  dist[j][k] ← 0 ∀j : 1 ≤ j ≤ 2 ∀k ∈ V_G
3  vis[j][k] ← WHITE ∀j : 1 ≤ j ≤ 2 ∀k ∈ V_G
4  color[i] ← GREEN ∀i ∈ V_G
5  s ← 0
6  for each vertex i ∈ V_G do
7  |  yelw[i] ← 0
8  |  COUNT-NEIGHBORS-BY-COLOR(i, Adj_G, grn, yelw, dist[1], vis[1])
9  |  j ← LENGTH(H)+1
10 |  H[j] ← i
11 |  pos[i] ← LENGTH(H)← j
12 end
13 COUNTING-SORT(H, pos, V_G, grn)
14 while LENGTH(H) > 0 do
15 |  i ← HEAP-EXTRACT-MAX(H, pos, grn, yelw)
16 |  color[i] ← RED
17 |  s ← s + 1
18 |  MARK-NEIGHBORS(i, Adj_G, color, H, pos, grn, yelw, dist, vis)
19 end
20 return s, color
```

Pseudocódigo 9: COUNT-NEIGHBORS-BY-COLOR($i, Adj_G, grn, yelw, dist, vis$)

```
1  grn[i] ← 1
2  yelw[i] ← 0
3  vis[i] ← GRAY
4  Q ← P ← ∅
5  ENQUEUE(Q, i)
6  ENQUEUE(P, i)
7  while Q ≠ ∅ do
8  |  j ← DEQUEUEQ
9  |  for each vertex k ∈ Adj_G[j] do
10 |  |  if vis[k] = WHITE then
11 |  |  |  vis[k] ← GRAY
12 |  |  |  ENQUEUE(P, k)
13 |  |  |  dist[k] ← dist[j] + 1
14 |  |  |  if color[k] = GREEN then grn[i] ← grn[i] + 1 end
15 |  |  |  if color[k] = YELLOW then yelw[i] ← yelw[i] + 1 end
16 |  |  |  if dist[k] < 2 then ENQUEUE(Q, k) end
17 |  |  end
18 |  end
19 |  vis[j] ← BLACK
20 end
21 dist[i] ← 0 ∀i ∈ P
22 vis[i] ← WHITE ∀i ∈ P
```



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
***grn*[*i*]**	5	6	8	9	6	5	6	6	8	9	6	5	5	6
***yelw*[*i*]**	0	0	0	0	0	0	0	0	0	0	0	0	0	0

i	4	7	8	9	10	11	12	13	14
***grn*[*i*]**	5	4	5	8	7	6	5	5	6
***yelw*[*i*]**	4	2	1	0	2	0	0	0	0

i	10	11	12	13	14
***grn*[*i*]**	4	4	4	5	4
***yelw*[*i*]**	5	2	1	0	2

Figura 11. Proceso de marcado sobre un grafo de ejemplo utilizando el algoritmo GREEDY-MAXIMAL-2PACKING. Debajo de cada figura se muestran tres arreglos. El arreglo denotado como i representa los identificadores de cada vértice, el arreglo denotado como $grn[i]$ representa el número de vértices verdes en $N_2[i]$ y el arreglo denotado como $yelw[i]$ representa el número de vértices amarillos en $N_2[i]$. En esta figura, los vértices blancos representan los vértices verdes del algoritmo, los cuales están disponibles. Los vértices rojos están empaquetados y los amarillos están restringidos. (a) Primero se elige el vértice 1 para empaquetarlo, (b) luego se elige el vértice 7, (c) después se elige el vértice 10 (d) y, finalmente, se elige vértice 12, resultando así el CIF máximo para este grafo particular.

- Blanco: indica que el vértice en cuestión no se ha contado todavía. Se denota como WHITE.
- Gris: indica que el vértice en cuestión ya se contó pero aún no se han contado todos sus vecinos. Se denota como GRAY.
- Negro: indica que el vértice en cuestión ya se contó y ya se contaron todos sus vecinos. Se denota como BLACK.

Se utiliza $vis[j]$ para denotar el color asignado al vértice j por la búsqueda en anchura realizada por el algoritmo COUNT-GREEN-NEIGHBORS. También se utiliza $dist[j]$ para denotar la distancia entre el vértice j y el vértice i. Esto permite detectar si se llegó a la frontera de $N_2(i)$ y, de ser así, truncar la BFS en ese punto.

Por último, al concluir la ejecución de este algoritmo, es necesario deshacer el marcado de vértices realizado, ya que los arreglos vis y $dist$ volverán a utilizarse en BFS posteriores. Para lograr esto sin incrementar el tiempo de ejecución, cada vez que se marca un vértice, éste se almacena en una cola ordinaria denotada como P. Esta cola se recorre al final de la búsqueda para reasignar el color inicial de cada vértice contenido en ella.

4.2.2. La cola de prioridad

En el algoritmo GREEDY-MAXIMAL-2PACKING, los vértices verdes del grafo se almacenan en una cola de prioridad, denotada como H en el pseudocódigo. Esta cola se utiliza para acelerar el proceso de elegir el siguiente vértice a ser empaquetado. En particular, este algoritmo hace uso de las siguientes operaciones básicas sobre la cola de prioridad: extraer el elemento de mayor prioridad, eliminar un elemento particular de la cola y mantener los elementos ordenados según su prioridad cada vez que se modifica su estado o el estado de la cola. La implementación de estas operaciones es similar a la descrita por Cormen *et al.* (2009), donde la cola de prioridad se maneja como un "max heap". Así, las operaciones del "max heap" se adaptaron del trabajo de estos autores para emplearse en el algoritmo GREEDY-MAXIMAL-2PACKING.

La operación más básica utilizada por la cola de prioridad es PRIORITY-COMPARISON (Pseudocódigo 10) y representa la implementación directa de los criterios para elegir el siguiente vértice a empaquetar y para desempatar los vértices cuando hay más de uno que cumple dichos criterios. En particular, esta rutina compara el conteo de vértices verdes y amarillos de dos vértices, i y j, y decide si i tiene mayor prioridad que j o no. Implícitamente, cada vértice tiene una prioridad que se deduce de su conteo de vértices verdes y amarillos, recordando que el criterio voraz del algoritmo GREEDY-MAXIMAL-2PACKING es empaquetar el vértice que minimiza el conteo de verdes y maximiza el conteo de amarillos. Así, la cola de prioridad se utiliza para mantener ordenados los vértices de mayor a menor prioridad de tal forma que el vértice que se debe empaquetar en cada iteración es aquél que se encuentra en la primera posición de la cola.

Otro componente fundamental para la cola de prioridad es la rutina MAX-HEAPIFY (Pseudocódigo 11). Como se explica en Cormen *et al.* (2009), cuando se modifica el contenido de la cola de prioridad (e.g., cuando se elimina un elemento o cuando se modifica el conteo de verdes y/o de amarillos), la prioridad de los elementos puede cambiar, por lo que es necesario volverlos a ordenar en la cola. Dicho reordenamiento se lleva a cabo aplicando recursivamente la rutina MAX-HEAPIFY, desde la posición del elemento modificado hasta el final de la cola. Cabe mencionar que, en el pseudocódigo de esta rutina, se tiene que LEFT(i) $\triangleq 2i$ y que RIGHT(i) $\triangleq 2i + 1$, igual a como se describe en Cormen *et al.* (2009). Así mismo, se utiliza *L*ENGTH(H) para denotar el número de elementos almacenados en la cola de prioridad H y se supone que cambia implícitamente conforme se agregan y eliminan vértices en H.

También se debe mencionar que se utiliza $pos[i]$ en el pseudocódigo para denotar la posición en la que se encuentra almacenado el vértice i en la cola de prioridad. Conocer esta posición es útil cuando se desea eliminar un vértice de la cola de prioridad (por haberse restringido). El problema con almacenar la posición de cada vértice es que debe corregirse cada vez que se cambia el orden de los elementos en la cola de prioridad.

Las rutinas PRIORITY-COMPARISON y MAX-HEAPIFY se utilizan para mantener ordenados los elementos contenidos en la cola de prioridad; sin embargo, es muy costoso (en el tiempo de ejecución) aplicar estas rutinas al inicio del algoritmo, cuando todos los

Pseudocódigo 10: PRIORITY-COMPARISON$(i, j, grn, yelw)$

```
1  if grn[i] < grn[j] then  return true end
2  if grn[i] > grn[j] then  return false end
3  if yelw[i] > yelw[j] then  return true end
4  if yelw[i] < yelw[j] then  return false end
5  if i < j then  return true end
6  return false
```

Pseudocódigo 11: MAX-HEAPIFY$(H, i, grn, yelw)$

```
1  ℓ ← LEFT(i)
2  r ← RIGHT(i)
3  if ℓ ≤ LENGTH(H) and PRIORITY-COMPARISON(H[ℓ], H[i], grn, yelw) then
4  |  max ← ℓ
5  else
6  |  max ← i
7  end
8  if r ≤ LENGTH(H) and PRIORITY-COMPARISON(H[r], H[max], grn, yelw) then
9  |  max ← r
10 end
11 if max ≠ i then
12 |  Intercambiar H[i] con H[max]
13 |  Intercambiar pos[i] con pos[max]
14 |  MAX-HEAPIFY(H, max, grn, yelw)
15 end
```

Pseudocódigo 12: COUNTING-SORT(H, pos, V_G, grn)

```
1  C[i] ← 0  ∀i ∈ V_G
2  for each vertex i ∈ V_G do
3  |  C[grn[i]] ← C[grn[i]] + 1
4  end
5  for i ← 2 to n do
6  |  C[i] ← C[i] + C[i − 1]
7  end
8  for i ← n down to 1 do
9  |  H[C[grn[i]]] ← i
10 |  pos[i] ← C[grn[i]]
11 |  C[grn[i]] ← C[grn[i]] − 1
12 end
```

Pseudocódigo 13: HEAP-EXTRACT-MAX$(H, pos, grn, yelw)$

```
1  max ← H[1]
2  H[1] ← H[LENGTH(H)]
3  pos[1] ← 1
4  LENGTH(H) ← LENGTH(H) − 1
5  MAX-HEAPIFY(H, 1, grn, yelw)
6  return max
```

vértices del grafo se deben insertar en la cola uno por uno. Dado que el orden de cada vértice en dicha cola depende del conteo de vértices verdes y amarillos de cada elemento, y aunado al hecho de que dicho conteo nunca es mayor a n, se puede utilizar un algoritmo de ordenamiento en tiempo lineal, como COUNTING-SORT (Pseudocódigo 12), para ordenar los vértices en la cola de prioridad de forma decreciente, según su prioridad, sin necesidad de recurrir a la aplicación repetitiva de la rutina MAX-HEAPIFY. El pseudocódigo del algoritmo COUNTING-SORT se adaptó del libro de Cormen *et al.* (2009) para ser utilizado de esta forma por el algoritmo GREEDY-MAXIMAL-2PACKING.

Por último, la rutina HEAP-EXTRACT-MAX (Pseudocódigo 13) se utiliza para recuperar el primer elemento de la cola de prioridad, esto es, el elemento de mayor prioridad. De nuevo, la implementación de esta rutina se adaptó del libro de Cormen *et al.* (2009) para el algoritmo GREEDY-MAXIMAL-2PACKING.

4.2.3. Restricción de vértices en el vecindario extendido

Cuando un vértice se empaqueta, se deben restringir todos los vértices de su vecindario abierto extendido. Esta tarea se lleva a cabo por medio de la rutina MARK-NEIGHBORS (Pseudocódigo 14), la cual es similar a la rutina COUNT-GREEN-NEIGHBORS, ya que también consiste de una búsqueda en anchura. La diferencia radica en que la búsqueda realizada por MARK-NEIGHBORS está acotada a una distancia de cuatro, en lugar de dos; se restringen todos los vértices verdes a distancia uno y dos del último vértice empaquetado y se actualiza el conteo de vértices verdes y amarillos de aquellos vértices a distancia tres y cuatro de dicho vértice.

Cada vez que se restringe un vértice, éste se debe remover de la cola de prioridad. Para ello, se utiliza la rutina HEAP-REMOVE (Pseudocódigo 15), que se adaptó del libro de Cormen *et al.* (2009). Como se mencionó anteriormente, esta rutina aprovecha que ya se conoce en qué posición se encuentra el vértice a eliminar en la cola de prioridad, la cual está registrada en el arreglo *pos*.

También se requiere actualizar el conteo de vértices verdes y amarillos de los vértices a distancia tres y cuatro del último vértice empaquetado, ya que a que se encuentran suficientemente cerca a dicho vértice como para que se vea alterado el marcado de sus vecindarios abiertos extendidos correspondientes. Los demás vértices del grafo

Pseudocódigo 14: MARK-NEIGHBORS($i, Adj_G, color, H, pos, grn, yelw, dist, vis$)

```
1  vis[1][i] ← GRAY
2  Q ← P ← ∅
3  ENQUEUE(Q, i)
4  ENQUEUE(P, i)
5  while Q ≠ ∅ do
6      j ← DEQUEUE(Q)
7      for each vertex k ∈ Adj_G[j] do
8          if vis[1][k] = WHITE then
9              vis[1][k] ← GRAY
10             ENQUEUE(P, k)
11             dist[1][k] ← dist[1][j] + 1
12             if dist[1][k] ≤ 2 then
13                 color[k] ← YELLOW
14                 HEAP-REMOVE(H, pos[k], pos, grn, yelw)
15             end
16             if dist[1][k] > 2 and dist[1][k] ≤ 4 then
17                 COUNT-NEIGHBORS-BY-COLOR(k, Adj_G, grn, yelw, dist[2], vis[2])
18                 HEAP-PRIORITY-INCREASE(H, pos[k], pos, grn, yelw)
19             end
20             if dist[1][k] < 4 then ENQUEUE(Q, k) end
21         end
22     end
23     vis[1][j] ← BLACK
24 end
25 dist[1][i] ← 0 ∀i ∈ P
26 vis[1][i] ← WHITE ∀i ∈ P
```

Pseudocódigo 15: HEAP-REMOVE($H, i, pos, grn, yelw$)

```
1  grn[i] ← ∞
2  yelw[i] ← −∞
3  Intercambiar H[i] con H[LENGTH(H)]
4  Intercambiar pos[i] con pos[LENGTH(H)]
5  LENGTH(H) ← LENGTH(H) − 1
6  MAX-HEAPIFY(H, i, grn, yelw)
```

Pseudocódigo 16: HEAP-PRIORITY-INCREASE($H, i, pos, grn, yelw$)

```
1  while i ≥ 1 and PRIORITY-COMPARISON(H[i], H[PARENT(i)], grn, yelw) do
2      Intercambiar H[i] con H[PARENT(i)]
3      Intercambiar pos[i] con pos[PARENT(i)]
4      i ← PARENT(i)
5  end
```

no sufren cambios en sus vecindarios y, por ende, no se requiere actualizar su conteo. De nuevo, se utiliza la rutina COUNT-NEIGHBORS-BY-COLOR (Pseudocódigo 9) para realizar la actualización del conteo de estos vértices.

El detalle con el procedimiento anterior es que incrementa la prioridad de los vértices cuyo conteo se modifica, puesto que su conteo de vértices verdes disminuye y su conteo de vértices amarillos incrementa. Como consecuencia, es necesario aplicar la rutina HEAP-PRIORITY-INCREASE (Pseudocódigo 16) inmediatamente después de aplicar la rutina COUNT-NEIGHBORS-BY-COLOR para reorganizar correctamente los elementos de la cola de prioridad. El pseudocódigo de la rutina HEAP-PRIORITY-INCREASE también se adaptó del libro de Cormen *et al.* (2009) y cabe señalar que PARENT(i) $\triangleq \lfloor i/2 \rfloor$.

4.3. Análisis

A continuación se presenta el análisis del algoritmo GREEDY-MAXIMAL-2PACKING. Para compactar las demostraciones aquí presentadas, las demostraciones por invariante de lazo se movieron al Anexo .

Teorema 4.1. *El algoritmo* GREEDY-MAXIMAL-2PACKING *resuelve el problema del CIF maximal para grafos arbitrarios en tiempo polinomial.*

Demostración. Comenzando por demostrar que GREEDY-MAXIMAL-2PACKING es correcto, dado que todas las rutinas que conforman este algoritmo fueron adaptadas de algoritmos correctos que se explican en el libro de Cormen *et al.* (2009), se puede suponer que estas rutinas también son correctas. Además, cabe mencionar que estas rutinas tienen el mismo costo computacional que los algoritmos originales de donde se adaptaron.

Entonces, el siguiente paso es demostrar que se cumple la invariante de lazo que se presenta a continuación sobre el bucle de las líneas 14 a la 19 del Pseudocódigo 8.

Antes de comenzar cada iteración i, se tiene que:

1. El conjunto de vértices empaquetados S constituye un CIF válido.

2. El número de vértices empaquetados antes de comenzar la iteración anterior, s_{i-1}, es estrictamente menor al número de vértices empaquetados antes de comenzar la iteración actual, s_i.

3. El número de vértices disponibles antes de comenzar la iteración anterior, a_{i-1}, es estrictamente mayor al número de vértices disponibles antes de comenzar la iteración actual, a_i.

Una vez demostrado que algoritmo es correcto, se puede proceder a calcular el costo computacional del mismo. Primero se tiene que las operaciones de inicialización de las líneas 1 a la 4 y el algoritmo COUNTING-SORT en la línea 13 del Pseudocódigo 8 requieren $O(n)$ tiempo cada uno. Luego, en el bucle de las líneas 6 a la 12, se visitan todos los vértices del grafo y, por cada uno, se realiza una búsqueda en anchura a profundidad dos. Por lo tanto, este procedimiento requiere $O(n(n_2 + m_2))$ tiempo, donde n_2 y m_2 denotan respectivamente el número de vértices y de aristas del grafo inducido por el vecindario cerrado extendido de mayor cardinalidad en el grafo.

Después, se tiene el bucle principal en las líneas 14 a la 19. Este bucle se ejecuta tantas veces como la cardinalidad del CIF encontrado por el algoritmo, el cual se denota como s. En cada iteración, se realizan dos búsquedas BFS, una anidada dentro de la otra. La búsqueda principal está acotada a una distancia de cuatro a partir del último vértice empaquetado. Se supone que esta búsqueda requiere $O(n_4 + m_4)$ tiempo, donde n_4 y m_4 denotan respectivamente el número de vértices y aristas del grafo inducido por el vecindario extendido a distancia cuatro de mayor cardinalidad en el grafo. La segunda búsqueda se aplica únicamente partiendo de los vértices a distancia tres y cuatro del último vértice empaquetado y está acotada a una distancia de dos. Sin embargo, para simplificar el análisis, se puede suponer que esta búsqueda también se aplica a los vértices a distancia uno y dos del último vértice empaquetado. Así, se tiene que esta parte del algoritmo requiere en total $O(n_4(n_2+m_2)+m_4)$ tiempo por cada iteración del bucle.

Aunado a lo anterior, cada vez que se aplica la búsqueda a distancia dos sobre un vértice, la prioridad de ese vértice cambia y, por ende, se requiere reordenar la cola de prioridad. Esta operación requiere $O(\log h)$ tiempo, donde h denota el número de elementos de la cola de prioridad al momento de ejecutar la operación. Sin embargo,

no es fácil calcular cuántas veces se ejecuta dicha operación sobre cada vértice del grafo. Así, para facilitar el análisis, se supone que esta operación se aplica sobre cada vértice explorado por la búsqueda a distancia cuatro, lo que resulta en $O(n_4 \log n)$ tiempo por cada iteración del bucle.

Por último, se deben considerar todas las veces que se elimina un vértice de la cola de prioridad. Un vértice se elimina de cola de prioridad ya sea porque se empaquetó (en cuyo caso se utiliza la rutina HEAP-EXTRACT-MAX) o porque se restringió (en cuyo caso se utiliza la rutina HEAP-REMOVE). Estas dos rutinas tienen el mismo costo computacional que las demás rutinas de manipulación de la cola de prioridad: $O(\log h)$ tiempo. Dado que no se puede restringir un vértice después de haber sido empaquetado y vice versa, se tiene que cada vértice se elimina de la cola de prioridad exactamente una vez. Por lo tanto, el costo total por eliminar todos los vértices de la cola de prioridad es $O(n \log n)$ tiempo.

Finalmente, el costo computacional del algoritmo GREEDY-MAXIMAL-2PACKING está dado por

$$O(n(n_2 + m_2 + \log n) + sn_4(n_2 + m_2 + m_4/n_4 + \log n)). \tag{5}$$

□

Cabe mencionar que el orden de crecimiento presentado en el análisis anterior representa una cota que quizás es muy holgada para el verdadero costo computacional del algoritmo. Además, se puede observar que depende de múltiples variables y no únicamente del número de vértices y de aristas, como es común en otros algoritmos de grafos. Sin embargo, esta representación tiene la ventaja de que puede simplificarse dependiendo de la estructura del grafo de entrada que se esté considerando, revelando un orden de crecimiento más ajustado.

Por ejemplo, considérese un ciclo de n vértices. En este caso, el vecindario cerrado extendido de cada vértice contiene 5 vértices y 4 aristas y el vecindario extendido a distancia cuatro contiene 9 vértices y 8 aristas (suponiendo que n es suficientemente grande). Entonces, se tiene que $n_2 = m_2 = n_4 = m_4 = O(1)$. Además, se sabe que la cardinalidad del CIF máximo de un ciclo es $\lfloor n/3 \rfloor = O(n)$. Por lo tanto, sustituyendo en

(5), se tiene que el orden de crecimiento del algoritmo GREEDY-MAXIMAL-2PACKING para un ciclo es $O(n \log n)$. Cabe mencionar que este mismo análisis también aplica cuando el grafo de entrada es un camino de n vértices (para conocer cómo se define un grafo camino, consulte el Capítulo 3).

Por otro lado, considérese un grafo completo. En este caso, se tiene que $n_2 = n_4 = n$ y $m_2 = m_4 = O(n^2)$. Además, se sabe que la cardinalidad del CIF máximo para un grafo completo es 1. Por lo tanto, sustituyendo en (5), el orden de crecimiento del algoritmo GREEDY-MAXIMAL-2PACKING para un grafo completo es $O(n^3)$.

Teorema 4.2. *El algoritmo GREEDY-MAXIMAL-2PACKING resuelve el problema del CIF máximo en grafos camino.*

Demostración. En un grafo camino de n vértices, se tiene que el vecindario cerrado extendido de todos los vértices es de a lo más cinco vértices, donde los vértices extremos tienen el vecindario cerrado extendido de menor tamaño, consistiendo de solo tres vértices. Por lo anterior, los vértices extremos siempre tienen la mayor prioridad para ser empaquetados por el algoritmo GREEDY-MAXIMAL-2PACKING, sin importar la longitud del camino. Esto implica que, en cada iteración de este algoritmo, el conjunto de vértices disponibles sigue constituyendo un camino, aunque de menor cardinalidad que en la iteración anterior. Así, el algoritmo GREEDY-MAXIMAL-2PACKING marca los vértices extremos del camino de vértices disponibles en cada iteración (sin importar el orden) hasta llegar a un camino de a lo más tres vértices. En este caso, no importa cuál vértice se empaquete, la cardinalidad del CIF construido siempre crece en uno y ya no quedan vértices disponibles, en cuyo momento el algoritmo termina su ejecución.

Dado que el procedimiento anterior siempre empaqueta cada tercer vértice en el camino, se tiene que la cardinalidad del CIF construido está dada por $\lfloor n/3 \rfloor + 1$, lo que corresponde al tamaño del CIF máximo. Por lo tanto, el algoritmo GREEDY-MAXIMAL-2PACKING encuentra el CIF máximo de un grafo camino. □

Teorema 4.3. *El algoritmo GREEDY-MAXIMAL-2PACKING resuelve el problema del CIF máximo en ciclos.*

Demostración. En un ciclo de $n \geq 3$ vértices, se tiene que, en la primera iteración del algoritmo GREEDY-MAXIMAL-2PACKING, todos los vértices tienen un vecindario cerrado

extendido del mismo tamaño: a lo más cinco vértices. Por lo anterior, se tiene que el algoritmo puede empaquetar cualquiera de ellos. Después de empaquetar el primer vértice, se tiene que el conjunto de vértices que quedan disponibles ya no constituyen un ciclo, sino un camino de a lo más $n-k$ vértices, donde $3 \leq k \leq 5$. Así, por el Teorema 4.2, se tiene que GREEDY-MAXIMAL-2PACKING correctamente encuentra el CIF máximo del camino resultante. Por lo tanto, el algoritmo GREEDY-MAXIMAL-2PACKING encuentra el CIF máximo de un ciclo. □

Capítulo 5. Diseño del experimento

En este capítulo se compara el desempeño del algoritmo GREEDY-MAXIMAL-2PACKING (la heurística voraz propuesta en el Capítulo 4) y el algoritmo LINEAR-CACTUS-2PACKING (el algoritmo secuencial propuesto en el Capítulo 3) con Gurobi (el solucionador de optimización elegido como referencia para este trabajo de tesis). Concretamente, el objetivo de este experimento es refutar o corroborar las siguientes hipótesis:

- El algoritmo GREEDY-MAXIMAL-2PACKING requiere menos tiempo de ejecución que Gurobi y encuentra soluciones que, en la práctica, son de buena calidad.
- El algoritmo LINEAR-CACTUS-2PACKING obtiene la solución óptima y requiere menos tiempo de ejecución que los otros dos algoritmos.

5.1. Metodología

El experimento realizado consistió de generar una colección de grafos de prueba sobre los cuales posteriormente se ejecutaron los algoritmos mencionados anteriormente. En cada prueba, se midió el tiempo de ejecución (en segundos) que requirió el algoritmo y la cardinalidad del CIF que encontró. Después, se aplicó un análisis estadístico sobre los datos recabados para determinar si había alguna diferencia significativa en la calidad de las soluciones encontradas por los algoritmos participantes. Todos los algoritmos se ejecutaron en la misma computadora, la cual estaba equipada con un procesador Intel© Xeon Gold 5222, 16 GiB de memoria RAM y que corría el sistema operativo Ubuntu 18.04 LTS. Todos los algoritmos participantes se programaron utilizando el lenguaje de programación Python. Ningún algoritmo utilizó cómputo paralelo de ningún tipo.

Además de los tres algoritmos mencionados anteriormente, en el experimento también participó una heurística aleatoria ingenua que consistía de iterativamente empaquetar un vértice disponible, elegido de forma aleatoria, y restringir su vecindario abierto a distancia dos. Gurobi se utilizó como el algoritmo de referencia para establecer una métrica para la calidad de las soluciones encontradas, mientras que la heurística ingenua se utilizó para determinar cuánto fue el incremento en la calidad de las soluciones debido a las decisiones voraces utilizadas por la heurística voraz. Así, la

calidad de la solución se define en este experimento como la razón entre la solución encontrada por un algoritmo y la solución más grande encontrada por Gurobi. En este sentido, si la solución encontrada por Gurobi es la solución óptima, entonces la calidad describe qué tan cercano a la solución óptima se encuentra la solución del algoritmo que se está comparando.

5.2. Análisis estadístico

Una vez ejecutados los algoritmos sobre la colección de grafos de prueba, se realizó un análisis estadístico para comparar el desempeño de los algoritmos participantes.

El análisis estadístico se dividió en dos partes. En la primera parte se aplican varias pruebas de hipótesis para determinar si hay o no alguna diferencia estadísticamente significativa entre los resultados de dos algoritmos que se están comparando. En la segunda parte se calcula un intervalo de confianza para describir la magnitud de dicha diferencia (si la hubo). Los Pseudocódigos 17 y 18 describen el procedimiento que se sigue en ambas partes del análisis estadístico.

La entrada de la primera parte del análisis estadístico consiste de dos muestras de datos, A y B. Cada muestra consiste de las cardinalidades de las soluciones encontradas por uno algoritmos que se desean comparar para un mismo modelo de grafos aleatorios. La salida de este análisis es uno de tres símbolos posibles:

- “A=B” indica que no se detectó una diferencia significativa entre los datos comparados.
- “A>B” indica que los datos de A son significativamente mayores a los de B.
- “B>A” indica que los datos de B son significativamente mayores a los de A.

Cabe mencionar que se supone que los datos de A y B están ordenados de tal forma que el par de datos en una misma posición ordinal corresponden al mismo grafo.

Esta parte del análisis comienza por determinar si tanto A como B constituyen una distribución normal. Para lograr esto, se aplica la prueba de Lilliefors sobre cada muestra. Si resulta que ambas muestras forman una distribución normal, entonces se aplica

Pseudocódigo 17: Análisis estadístico (primera parte)

```
1  p_A ← LILLIEFORS(A)
2  p_B ← LILLIEFORS(B)
3  if p_A > 0.05 or p_B > 0.05 then
4  |  p ← TWOTAILEDSTUDENTREPSAMPLES(A,B)
5  |  if p ≥ 0.05 then return "A=B"
6  |  p ← RIGHTTAILEDSTUDENTREPSAMPLES(A,B)
7  else
8  |  p ← TWOTAILEDWILCOXONREPSAMPLES(A,B)
9  |  if p ≥ 0.05 then return "A=B"
10 |  p ← RIGHTTAILEDWILCOXONREPSAMPLES(A,B)
11 end
12 if p < 0.05 then  return "A>B"
13 return "B>A"
```

Pseudocódigo 18: Análisis estadístico (segunda parte)

```
1  p ← LILLIEFORS(X)
2  if p > 0.05 then  x, y ← CONFIDENCEINTERVAL(X, 0.95)
3  else  x, y ← BCABOOTSTRAPCI(X, 1000, 0.95)
4  return x, y
```

la prueba *t* de Student de dos colas para muestras repetidas. Si se retiene la hipótesis nula de esta prueba, entonces el análisis estadístico concluye con "A=B" como resultado. En cambio, si se rechaza la hipótesis nula, entonces se vuelve a aplicar la misma prueba *t* de Student pero de una cola en lugar de dos. Si se retiene la hipótesis nula de esta prueba, entonces el análisis estadístico concluye con "B>A" como resultado. En cambio, si se rechaza la hipótesis nula, entonces el análisis concluye con "A>B". En caso de que A y/o B no constituyan una distribución normal, se utiliza la prueba de Wilcoxon en lugar de la prueba *t* de Student. Cabe mencionar que el grado de significancia (*level of significance* en inglés) utilizado en todas las pruebas de hipótesis fue 0.05.

La segunda parte del análisis estadístico recibe como única entrada una muestra de datos, X, que contiene la calidad de todas las soluciones encontradas por un algoritmo para un mismo modelo de grafos aleatorios. La salida es el límite inferior y superior del intervalo de confianza (Neyman y Jeffreys, 1937) para la media poblacional de X. Se utiliza un grado de confianza (*level of confidence* en inglés) del 95 por ciento para calcular el intervalo de confianza. Esta parte del análisis también comienza aplicando la prueba de Lilliefors sobre X para determinar si constituye una distribución normal

o no. Si resulta que *X* es una distribución normal, entonces se calcula el intervalo de confianza a partir de la razón *t* de la distribución. En cambio, si *X* no es una distribución normal, entonces se genera un *bootstrap* (Efron, 1979) de 1000 muestras y se calcula el intervalo de confianza a partir de ellas utilizando corrección de sesgo y aceleración (Efron, 1987), técnica conocida como BCa por sus siglas en inglés.

Finalmente, es importante mencionar que el análisis estadístico descrito en esta sección es una adaptación del análisis presentado en el trabajo de Zatarain Aceves *et al.* (2011). Así, se puede consultar la publicación de Zatarain *et al*. si se desea saber en qué consisten, de dónde provienen y cuáles son las hipótesis que manejan las pruebas de Lilliefors, *t* de Student y Wilcoxon.

Capítulo 6. Resultados y discusión

6.1. Resultados de los experimentos

En las Figuras 12 y 13 se muestra el intervalo de confianza para la calidad de las soluciones encontradas por la heurística voraz, la heurística ingenua y del primer CIF maximal encontrado por Gurobi para cada modelo de grafos aleatorios, según como fue calculado por el análisis estadístico descrito en el Capítulo anterior. Por otro lado, en las Figuras 14 y 15 se muestra el intervalo de confianza para el tiempo de ejecución requerido por la heurística voraz para encontrar el CIF maximal y por Gurobi para encontrar la primera solución de igual o mayor cardinalidad a la solución correspondiente de la heurística voraz. Por último, en la Figura 16 se muestra el intervalo de confianza para el tiempo de ejecución requerido por la heurística voraz y el algoritmo LINEAR-CACTUS-2PACKING. En todas estas figuras, la longitud de las barras indican el tamaño del intervalo de confianza. Las tablas que contienen los valores numéricos que corresponden a estas figuras se encuentran en el Anexo .

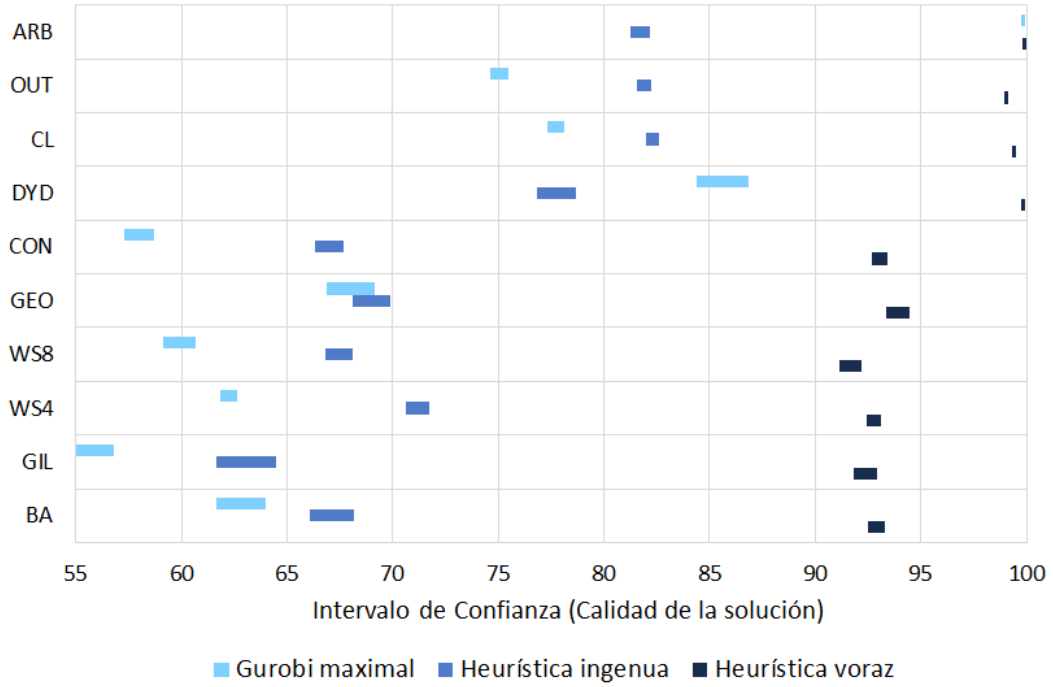


Figura 12. Intervalo de confianza de la calidad de las soluciones encontradas por diferentes algoritmos para los grafos de $2\overline{k}$ vértices. Entre mayor sea la calidad, es mejor. El eje vertical denota cada modelo de grafos aleatorios: árboles (ARB), k-outerplanar (OUT), cactus lineal (CL), duplicado y divergencia (DYD), conectados (CON), geométricos (GEO), Watts-Strogatz de cuatro (WS4) y ocho (WS8) vecinos, Gilbert (GIL) y Barabási-Albert (BA).

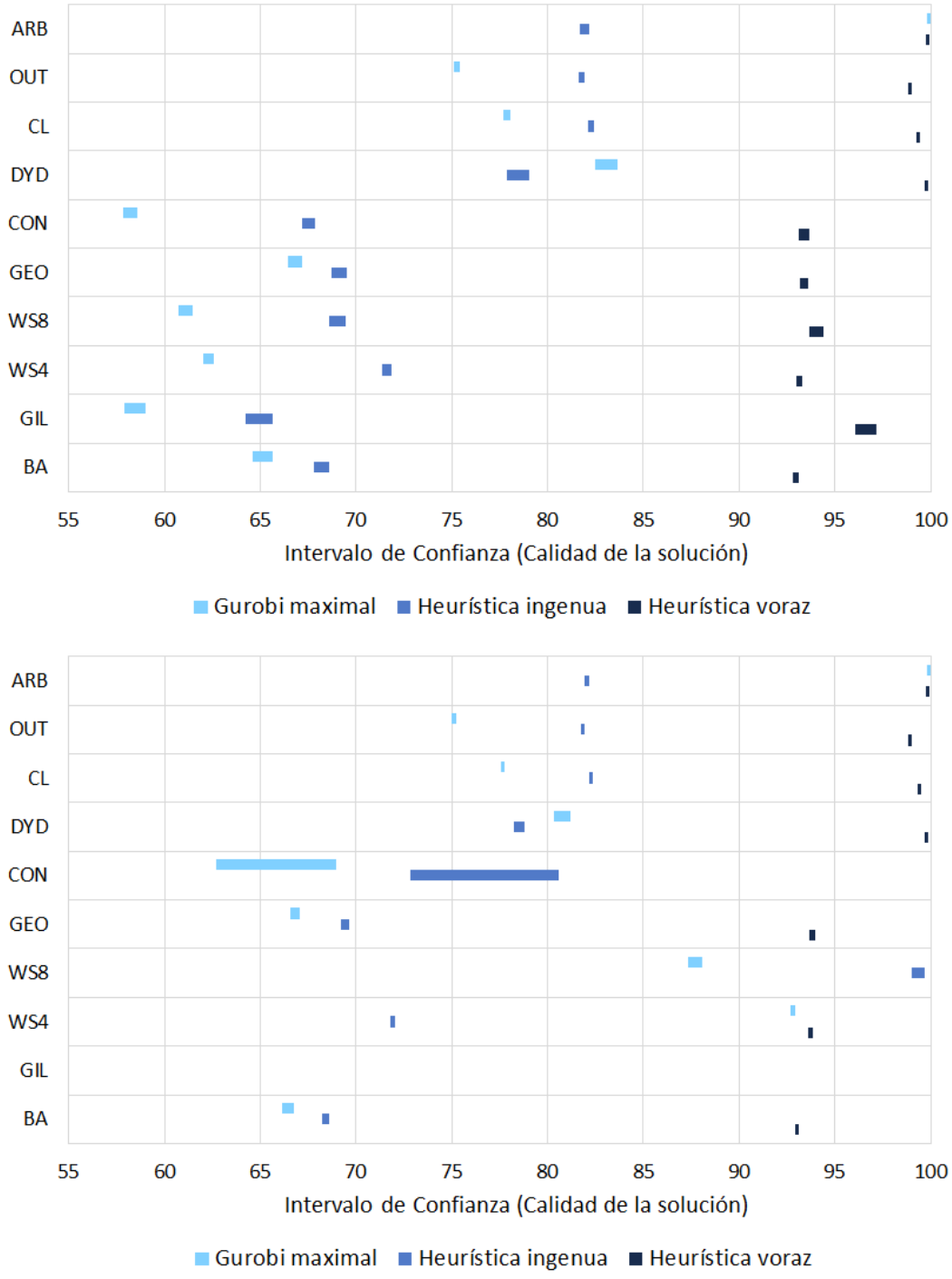


Figura 13. Intervalo de confianza de la calidad de las soluciones encontradas por diferentes algoritmos para los grafos de $8\overline{k}$ (superior) y $32\overline{k}$ vértices (inferior). Entre mayor sea la calidad, es mejor. El eje vertical denota cada modelo de grafos aleatorios: árboles (ARB), k-outerplanar (OUT), cactus lineal (CL), duplicado y divergencia (DYD), conectados (CON), geométricos (GEO), Watts-Strogatz de cuatro (WS4) y ocho (WS8) vecinos, Gilbert (GIL) y Barabási-Albert (BA).

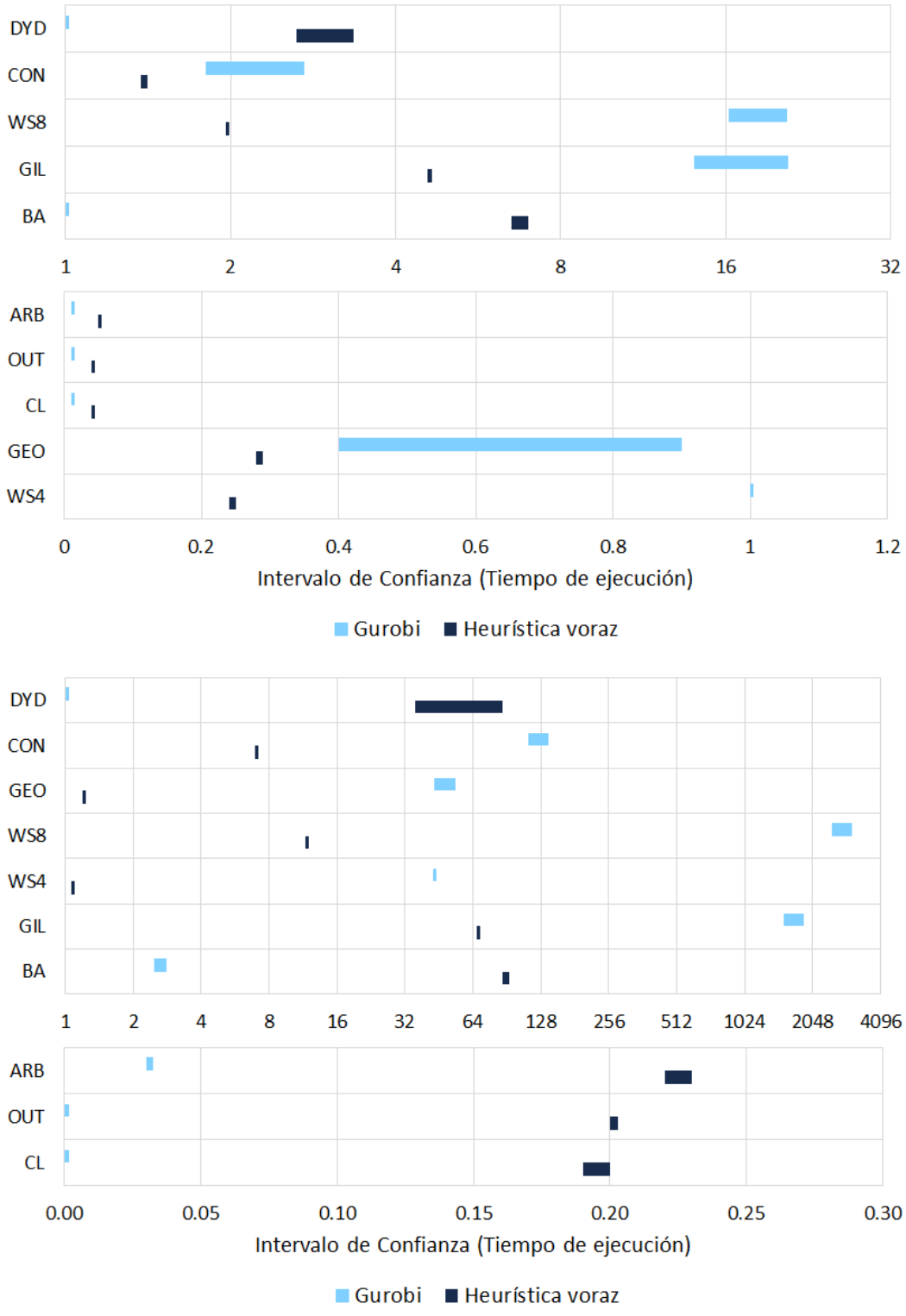


Figura 14. Intervalo de confianza del tiempo de ejecución requerido por diferentes algoritmos para los grafos de $2\overline{k}$ (superior) y $8\overline{k}$ vértices (inferior). Entre menor sea el tiempo, es mejor. El eje vertical denota cada modelo de grafos aleatorios: árboles (ARB), k-outerplanar (OUT), cactus lineal (CL), duplicado y divergencia (DYD), conectados (CON), geométricos (GEO), Watts-Strogatz de cuatro (WS4) y ocho (WS8) vecinos, Gilbert (GIL) y Barabási-Albert (BA).

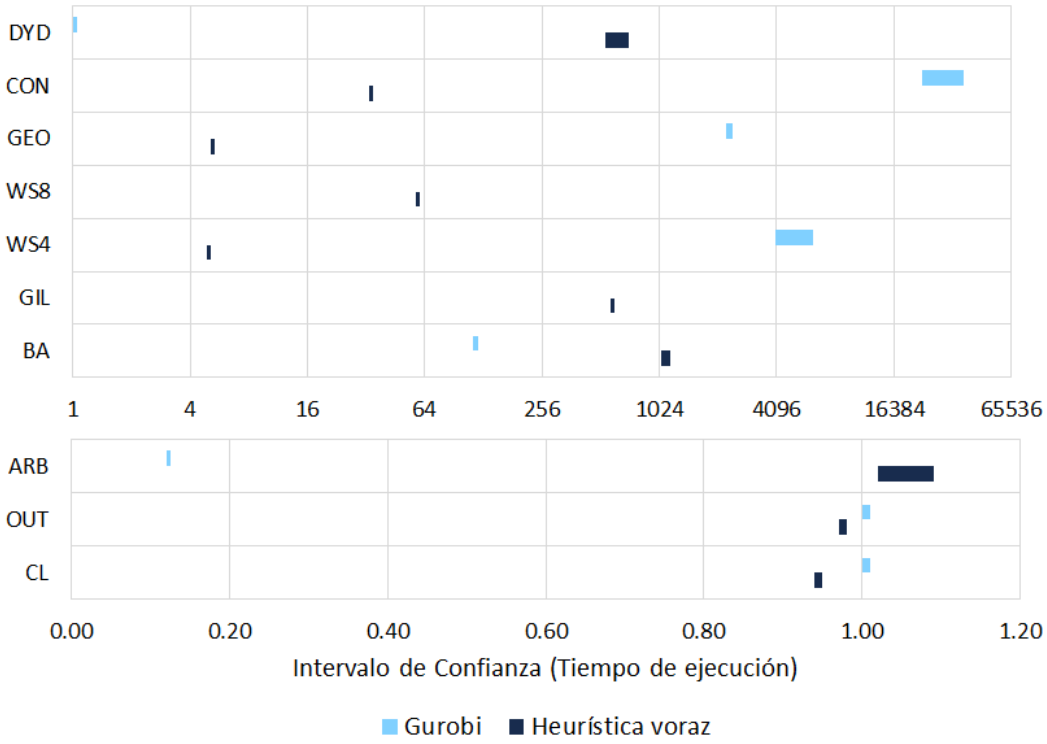


Figura 15. Intervalo de confianza del tiempo de ejecución requerido por diferentes algoritmos para los grafos de $32\overline{k}$ vértices. Entre menor sea el tiempo, es mejor. El eje vertical denota cada modelo de grafos aleatorios: árboles (ARB), k-outerplanar (OUT), cactus lineal (CL), duplicado y divergencia (DYD), conectados (CON), geométricos (GEO), Watts-Strogatz de cuatro (WS4) y ocho (WS8) vecinos, Gilbert (GIL) y Barabási-Albert (BA).

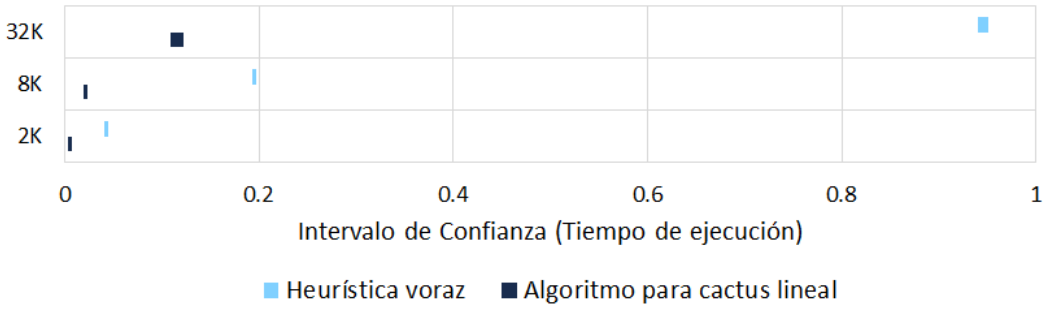


Figura 16. Intervalo de confianza del tiempo de ejecución requerido por diferentes algoritmos para los grafos cactus lineales. Entre menor sea el tiempo, es mejor. El eje vertical denota el número de vértices de cada grafo, donde $K = 1024$.

6.2. Discusión de los resultados

Comenzando por la calidad de las soluciones, los resultados del análisis estadístico que se muestran en las Figuras 12 y 13 indican claramente que, en la mayoría de los casos, la heurística voraz encontró soluciones cuya calidad está por encima del 90 por ciento; esto es, la heurística voraz encontró soluciones que son no más de 10 por

ciento más pequeñas que las mejores soluciones encontradas por Gurobi. En el caso particular de los grafos Watts-Strogatz ($k = 8$) y Gilbert de $32\overline{k}$ vértices, la heurística voraz encontró soluciones que superaron en cardinalidad a las de Gurobi. No se detectó una diferencia estadísticamente significativa entre las soluciones encontradas por estos dos algoritmos para los grafos conectados aleatorios de $32\overline{k}$ vértices.

Comparando la calidad de las soluciones encontradas por la heurística voraz con las de la heurística aleatoria ingenua y con la calidad de las primeras soluciones maximales encontradas por Gurobi, se tiene que la heurística voraz supera a estos dos algoritmos en casi todos los casos. En el caso de los árboles aleatorios de todos los órdenes, no se detectó una diferencia estadísticamente significativa entre las soluciones de la heurística voraz y las primeras soluciones maximales de Gurobi.

Comparando la calidad de las soluciones con las del algoritmo para grafos cactus lineales, es evidente que la heurística voraz no puede encontrar la solución óptima para estos grafos, pero las soluciones que encuentra tienen una calidad del 99 por ciento.

En cuanto al tiempo de ejecución, los resultados que se muestran en las Figuras 14 y 15 indican que, en la mayoría de los casos, la heurística voraz requiere menos tiempo de ejecución que Gurobi para encontrar una solución de igual o mayor cardinalidad. En el caso de los grafos Barabási-Albert, duplicado y divergencia y árboles, la heurística voraz requirió más tiempo que Gurobi. Esto también sucedió para los grafos k-outerplanar y cactus lineales de $2\overline{k}$ y $8\overline{k}$ vértices. Esto implica que las heurísticas y reglas de reducción que aplica Gurobi para estos modelos son más eficientes que la heurística voraz.

Finalmente, comparando el tiempo de ejecución de la heurística voraz con el del algoritmo para grafos cactus lineales, se tiene que este último corre aproximadamente diez veces más rápido que la heurística voraz y, por extensión, también es más rápido que Gurobi. Esto implica que el algoritmo LINEAR-CACTUS-2PACKING es el mejor de los tres algoritmos para resolver el problema del CIF máximo en grafos cactus lineales.

Capítulo 7. Conclusiones y trabajo futuro

En este trabajo de tesis, se diseñó un algoritmo denominado LINEAR-CACTUS-2PACKING que resuelve el problema del CIF máximo en grafos cactus lineales en tiempo lineal. También se diseñó una heurística voraz denominada GREEDY-MAXIMAL-2PACKING que resuelve el problema del CIF maximal en grafos arbitrarios en tiempo polinomial.

Se demostró de forma experimental que la heurística voraz es, en la práctica, una buena alternativa a Gurobi y a una heurística aleatoria ingenua, tanto en la calidad de la solución como en el tiempo de ejecución. El experimento mostró que la heurística voraz encuentra soluciones cuya cardinalidad está por encima del 90 % de la cardinalidad de las soluciones de Gurobi. En particular, cuando el grafo es de orden grande, la heurística voraz encuentra soluciones de igual o mayor cardinalidad a las de Gurobi para los grafos de Gilbert, de Watts-Strogatz de 8 vecinos y grafos conectados aleatorios. También se mostró que, en la mayoría de los casos, Gurobi requiere igual o mayor tiempo de ejecución para encontrar una solución de igual cardinalidad al de la heurística voraz. Finalmente, se mostró que las soluciones encontradas por la heurística voraz siempre superan en cardinalidad a las soluciones de la heurística ingenua.

7.1. Trabajo futuro

De los resultados obtenidos, hay varias vertientes por donde se puede realizar mayor investigación y llegar a resultados interesantes. La primera involucra la optimización del algoritmo GREEDY-MAXIMAL-2PACKING. Existe una ineficiencia en este algoritmo que ocurre en el escenario que se explica a continuación. En cada iteración del algoritmo, lo más probable es que se empaquete un vértice j que se encuentra a distancia tres o cuatro del último vértice empaquetado, i. Después, se aplica una búsqueda en anchura a profundidad cuatro que comienza en j y, como consecuencia, esta búsqueda eventualmente visita el vértice i y algunos de sus vértices adyacentes. Sin embargo, dado que i ya está empaquetado y sus vecinos están restringidos, no es necesario visitar estos vértices. Para evitar que esto suceda, se pueden eliminar las aristas incidentes al último vértice empaquetado. Esto debería reducir el tiempo de ejecución requerido por el algoritmo, aunque la magnitud de dicha reducción dependerá de la densidad del grafo.

Otra vertiente que se puede explorar en cuanto a la heurística voraz es demostrar o refutar la conjetura de que el CIF construido por este algoritmo es *1-maximal*. Un CIF es 1-maximal cuando, al desempaquetar uno de los vértices empaquetados (no importa cuál), de los vértices que pueden hacerse disponibles en su vecindario abierto a distancia dos, no es posible empaquetar dos o más vértices sin que el conjunto resultante de vértices empaquetados deje de ser un CIF. Si se logra demostrar que esta conjetura es cierta, esto podría ser una propiedad particular de la heurística voraz lo haría más atractivo de utilizar en algunos escenarios que otras heurísticas existentes para resolver el problema del CIF.

Otra cosa que se podría estudiar sobre la heurística voraz es que se podría comparar experimentalmente el desempeño de este algoritmo con el algoritmo de Christou y Vassilaras (2013). Se espera que el algoritmo de estos autores entregue soluciones de mayor cardinalidad a cambio de un mayor tiempo de ejecución. Sin embargo, se conjetura que, al igual que como ocurre con Gurobi, hay grafos para los cuales, debido a su densidad y/o topología, el algoritmo de Christou y Vassilaras tardaría demasiado tiempo en encontrar una solución factible, en cuyo caso, la heurística voraz podría ser una alternativa viable a dicho algoritmo. También se podría explorar la posibilidad de combinar la heurística voraz con el algoritmo de Christou y Vassilaras. El algoritmo de estos autores utiliza una heurística más simple para construir las soluciones factibles que va explorando con branch-and-bound. Es posible que, si esta heurística se sustituye por el algoritmo GREEDY-MAXIMAL-2PACKING, se podría converger más rápidamente en un óptimo local.

Por último, en cuanto al algoritmo LINEAR-CACTUS-2PACKING, se conjetura que este algoritmo se puede extender para encontrar el CIF máximo de un grafo cactus general en tiempo lineal. De demostrarse verdadera esta conjetura, este algoritmo sería más rápido que el algoritmo de Flores-Lamas *et al.* (2018). Para extender el algoritmo LINEAR-CACTUS-2PACKING a los grafos cactus, sería conveniente extenderlo primero al grafo unicicilo. En el uniciclo, se puede aplicar el algoritmo de Mjelde (2004) para encontrar el CIF máximo en los árboles adyacentes al ciclo. De esta forma, se pueden marcar los vértices de estos árboles, procurando no empaquetar el vértice raíz del árbol, pues este vértice también pertenece al ciclo. Después, se tiene que las restricciones introducidas por el procedimiento anterior parte el ciclo en varios caminos, por

lo que se conjetura que se puede aplicar el algoritmo MARK-PATH sobre cada uno de estos caminos para obtener el CIF de mayor cardinalidad del ciclo considerando las restricciones establecidas por los árboles.

De demostrarse que la conjetura anterior es cierta, el algoritmo para grafos uniciclo podría extenderse a grafos cactus de una manera similar a como lo hicieron Flores-Lamas *et al*. La diferencia sería ahora que el algoritmo resultante podrá marcar los vértices de los ciclos del grafo en tiempo lineal, cuando el algoritmo original de estos autores tarda tiempo cuadrático para realizar este mismo paso.

Literatura citada

Achterberg, T., Bixby, R. E., Gu, Z., Rothberg, E., y Weninger, D. (2020). Presolve reductions in mixed integer programming. *INFORMS J. Comput.*, **32**(2): 473–506.

Aldous, D. J. (1990). A random tree model associated with random graphs. *Random Struct. Algorithms*, **1**(4): 383–402.

Anand, R., Aggarwal, D., y Kumar, V. (2017). A comparative analysis of optimization solvers. *Journal of Statistics and Management Systems*, **20**(4): 623–635.

Awerbuch, B. (1987). Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems (detailed summary). En: A. V. Aho (ed.), *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. ACM, pp. 230–240.

Balakrishnan, H., Barrett, C. L., Kumar, V. S. A., Marathe, M. V., y Thite, S. (2004). The distance-2 matching problem and its relationship to the mac-layer capacity of ad hoc wireless networks. *IEEE J. Sel. Areas Commun.*, **22**(6): 1069–1079.

Balas, E., Ceria, S., y Cornuéjols, G. (1996). Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Manage. Sci.*, **42**(9): 1229–1246.

Barabási, A. y Albert, R. (1999). Emergence of scaling in random networks. *Science*, **286**(5439): 509–512.

Bern, M. W., Lawler, E. L., y Wong, A. L. (1987). Linear-time computation of optimal subgraphs of decomposable graphs. *J. Algorithms*, **8**(2): 216–235.

Blair, J. y Manne, F. (2003). Efficient self-stabilizing algorithms for tree networks. En: *23rd International Conference on Distributed Computing Systems*. pp. 20–26.

Christou, I. T. y Vassilaras, S. (2013). A parallel hybrid greedy branch and bound scheme for the maximum distance-2 matching problem. *Comput. Oper. Res.*, **40**(10): 2387–2397.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, tercera edición. Cambridge, Mass.

Diestel, R. (2000). *Graph theory*. Número 173 en: Graduate texts in mathematics. Springer, segunda edición. New York.

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

Ding, Y., Wang, J. Z., y Srimani, P. K. (2014). Self-stabilizing algorithm for maximal 2-packing with safe convergence in an arbitrary graph. En: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE Computer Society, pp. 747–754.

Drobyshevskiy, M. y Turdakov, D. (2020). Random graph modeling: A survey of the concepts. *ACM Comput. Surv.*, **52**(6): 131:1–131:36.

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *Annals of Statistics*, **7**(1): 1–26.

Efron, B. (1987). Better bootstrap confidence intervals. *Journal of the American Statistical Association*, **82**(397): 171–185.

Eppstein, D. (2016). Simple recognition of Halin graphs and their generalizations. *J. Graph Algorithms Appl.*, **20**(2): 323–346.

Erdös, P. y Rényi, A. (1959). On random graphs I. *Publicationes Mathematicae*, **6**: 290–297.

Feo, T. A. y Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, **6**(2): 109–133.

Flores Lamas, A. (2018). *Estudio de factibilidad para resolver el problema 2-packing máximo en tiempo polinomial en grafos outerplanares*. Tesis de Doctorado en Ciencias, Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California.

Flores-Lamas, A., Fernández-Zepeda, J. A., y Trejo-Sánchez, J. A. (2018). Algorithm to find a maximum 2-packing set in a cactus. *Theoretical Computer Science*, **725**: 31–51.

Flores-Lamas, A., Fernández-Zepeda, J. A., y Trejo-Sánchez, J. A. (2020). A distributed algorithm for a maximal 2-packing set in Halin graphs. *J. Parallel Distributed Comput.*, **142**: 62–76.

Gairing, M., Geist, R., Hedetniemi, S. T., y Kristiansen, P. (2004a). A self-stabilizing algorithm for maximal 2-packing. *Nord. J. Comput.*, **11**(1): 1–11.

Gairing, M., Goddard, W., Hedetniemi, S. T., Kristiansen, P., y McRae, A. A. (2004b). Distance-two information in self-stabilizing algorithms. *Parallel Process. Lett.*, **14**(3-4): 387–398.

Gilbert, E. N. (1959). Random Graphs. *The Annals of Mathematical Statistics*, **30**(4): 1141–1144.

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

Hale, W. (1980). Frequency assignment: Theory and applications. *Proceedings of the IEEE*, **68**(12): 1497–1514.

Hochbaum, D. S. y Shmoys, D. B. (1985). A best possible heuristic for the k-center problem. *Math. Oper. Res.*, **10**(2): 180–184.

Ispolatov, I., Krapivsky, P. L., y Yuryev, A. (2005). Duplication-divergence model of protein interaction network. *Phys. Rev. E*, **71**: 61911.

Karp, R. M. (1972). Reducibility among combinatorial problems. *Complexity of computer computations*, pp. 85–103.

Kumar, V., Deo, N., y Kumar, N. (1998). Parallel generation of random trees and connected graphs. *Congressus Numerantium*, pp. 7–18.

Land, A. H. y Doig, A. G. (1960). An automatic method for solving discrete programming problems. *Econometrica*, **28**(3): 105–132.

Lynch, N. A. (1996). *Distributed Algorithms*. Data Management Systems. Morgan Kaufmann Publishers, Inc. San Francisco, Cal.

Manne, F. y Mjelde, M. (2006). A memory efficient self-stabilizing algorithm for maximal k-packing. En: A. K. Datta y M. Gradinariu (eds.), *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium*. Springer, Vol. 4280 de *Lecture Notes in Computer Science*, 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.

Milgram, S. (1967). The small world problem. *Psychology Today*, **1**(1): 60–67.

Mjelde, M. (2004). *k-Packing and k-Domination on tree graphs*. Tesis de maestría, The University of Bergen.

Neyman, J. y Jeffreys, H. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, **236**(767): 333–380.

Penrose, M. (2003). *Random Geometric Graphs*. Oxford University Press.

Prüfer, H. (1918). Neuer beweis eines satzes über permutationen. *Arch. Math. Phys*, **27**(1918): 742–744.

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

Trejo-Sánchez, J. A. y Fernández-Zepeda, J. A. (2012). A self-stabilizing algorithm for the maximal 2-packing in a cactus graph. En: *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 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. *J. Parallel Distributed Comput.*, **74**(3): 2193–2202.

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

Wagner, A. (2003). How the global structure of protein interaction networks evolves. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, **270**(1514): 457–466.

Wang, X., Wang, L., y Wu, Y. (2009). An optimal algorithm for Prufer codes. *J. Softw. Eng. Appl.*, **2**(2): 111–115.

Watts, D. J. y Strogatz, S. H. (1998). Collective dynamics of ‘small-world’ networks. *Nature*, **393**(6684): 440–442.

Waxman, B. M. (1988). Routing of multipoint connections. *IEEE J. Sel. Areas Commun.*, **6**(9): 1617–1622.

Yelbay, B., Birbil, S. I., Bülbül, K., y Jamil, H. M. (2016). Approximating the minimum hub cover problem on planar graphs. *Optim. Lett.*, **10**(1): 33–45.

Zatarain Aceves, H., Fernández Zepeda, J. A., y Brizuela Rodriguez, C. A. (2011). *Resolivendo el problema de asignación de guardaespaldas utilizando algoritmos genéticos*. Tesis de maestría, Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California.

Anexos

A. Demostraciones por invariante de lazo

A continuación se presentan las demostraciones por invariante de lazo que fueron omitidas en los análisis de los algoritmos presentados en los Capítulos 3 y 4 por razones de espacio.

A.1. Demostración del Lema 3.2

Invariante de lazo. Antes de comenzar cada iteración del ciclo de las líneas 5-23 del Pseudocódigo 3, se tiene lo siguiente:

(a) La cola Q contiene únicamente el VDP i.

(b) Todos los vértices desde r hasta i están marcados de negro.

(c) Todos los VDP marcados de negro, excepto i, tienen registradas la información de a lo más dos secciones L adyacentes a ellos, de la misma forma que se describe en el lema anterior.

(d) El vértice i es el VDP donde terminan las secciones L registradas en el VDP visitado en la iteración anterior.

(e) El VDP i tiene a lo más dos vecinos marcados de blanco.

Inicialización. Antes de comenzar la primera iteración del algoritmo, se tiene que el único elemento contenido en Q es el VDP r (por la línea 5 del pseudocódigo), por lo que se cumple la condición (a). Por otro lado, r está marcado de negro (por la línea 3). Aunado a esto, como $r = i$, se tiene que se cumple la condición (b) y como i no tiene registrada ningún información todavía, también se cumple la condición (c). Dado que r es un VEX (por el resultado entregado por el algoritmo PARTITION-VERTICES; véase el Lema 3.1), se tiene que $\delta(r) \leq 2$, dependiendo del tipo de grafo cactus lineal que es K. Esto implica que r tiene a lo más dos vecinos. Aunado a esto, se tiene que el resto de los vértices están marcados de blanco (por la línea 1), por lo que se cumple la condición (e). Por último, debido a que no se ha ejecutado ninguna iteración todavía,

la condición (d) se cumple por inconstrucción. Por lo tanto, se cumple la invariante de lazo para este caso.

Mantenimiento Supóngase que la invariante de lazo se cumple antes de comenzar alguna iteración arbitraria k. Esto implica que la cola Q contiene únicamente el VDP i, que todos los vértices desde r hasta i están marcados de negro, que todos los VDP negros, excepto i, tienen registrada información relevante sobre a lo más dos secciones L adyacentes, que el vértice i es el VDP donde terminan las secciones L registradas en el VDP visitado en la iteración $k-1$ y que i tiene a lo más dos vecinos blancos. Durante la ejecución del algoritmo, se tiene que i se extrae de la cola Q, quedando está vacía. Luego, se aplica la sub-rutina PATH-EXPLORATION sobre cada vecino blanco de r. Esta sub-rutina recorre cada vértice blanco en la sección L que comienza en i hasta encontrar el primer VDP que no está marcado de negro, el cual se denota como t. Al mismo tiempo, la sub-rutina cuenta cuántos vértices se han visitado y los va marcando de negro. El número de vértices visitados se denota como d y, al finalizar la ejecución de PATH-EXPLORATION , su valor corresponde a la longitud de la sección L que comienza en i y termina en t. Esta sub-rutina entrega como resultado t y d y estos valores se registran en i junto con el vértice sobre el cuál comenzó la exploración de PATH-EXPLORATION como la información relevante de la sección L recién explorada. Después, si t está marcado de blanco, se agrega a la cola Q y se marca de gris. Si el vértice i tiene dos vecinos blancos, esto implica que este vértice da origen a una sección C, por lo que deben explorarse dos secciones L y ambas terminan en el mismo VDP, t. Sin embargo, dado que t se inserta en Q únicamente cuando está marcado de blanco, t se inserta exactamente una vez en Q. Al finalizar la ejecución de esta iteración, los vértices t e i se marcan de negro. Además, debido a que $\delta(t) \leq 4$ y dado que se exploraron a lo más dos secciones L que terminaron en t, a t le quedan a lo más dos vecinos blancos. Así, antes de comenzar la iteración $k+1$, Q contiene únicamente el VDP t, que todos los vértices desde r hasta t están marcados de negro, que todos los VDP negros, excepto t, tienen registrada información relevante sobre a lo más dos secciones L adyacentes, que t es el VDP donde terminan las secciones registradas en i y que t tiene a lo más dos vecinos blancos. Por lo tanto, la invariante de lazo se mantiene en cada iteración.

Finalización. Al terminar la ejecución del algoritmo PATH-DECOMPOSITION, el vértice i es el VEX opuesto a r, que Q está vacía, que todos los vértices desde r hasta i (esto es, todos los vértices del grafo) están marcados de negro, que todos los VDP del grafo tienen registradas a lo más dos secciones L adyacentes, que i es el VDP donde terminan las secciones L registradas al vértice visitado en la iteración anterior y que i no tiene ningún vecino blanco. Cabe mencionar que, dado que en cada iteración se visita el vértice donde terminan las secciones registradas al VDP visitado en la iteración anterior, los vértices se visitaron en un orden secuencial que comienza en r y continua con el VDP más cercano hasta terminar en el VEX opuesto. Así, al finalizar la ejecución del algoritmo PATH-DECOMPOSITION, el resultado entregado por este algoritmo cumple con el Lema 3.2.

A.2. Demostración del Lema 3.3

Invariante de lazo Antes de comenzar cada iteración del cicle de las líneas 3-4 del Pseudocódigo 6, el marcado del camino que comienza en el vértice x y termina en el vértice i constituye el CIF de mayor cardinalidad posible considerando la restricción establecida por el color de x, además, el vértice j está marcado de verde.

Inicialización. Antes de comenzar la primera iteración del algoritmo, el vértice $i = x$ y todos los vértices, incluyendo j, están marcados de verde por la línea 1 del Pseudocódigo 5. El camino que comienza en x y termina en i está constituido únicamente por el vértice x el cual puede estar marcado de verde o cualquier otro color. Si está marcado de verde, entonces, por la línea 1 del Pseudocódigo 6, x se marca de rojo. Esto resulta en el CIF de mayor cardinalidad posible para este caso particular. Si, por el contrario, x está marcado de algún otro color, el color de este vértice no cambia, pues esa fue la restricción establecida por el marcado de la sección anterior, que se supone es un marcado óptimo. Por ende, no importa qué color tiene asignado el vértice x, siempre constituye el marcado que resulta en el CIF de mayor cardinalidad posible considerando las restricciones introducidas por la sección anterior. Por lo tanto, se cumple la invariante de lazo para este caso.

Mantenimiento. supóngase que la invariante de lazo se cumple antes de comenzar alguna iteración arbitraria q. Esto implica que el marcado del camino que comienza en x y termina en i constituye el CIF de mayor cardinalidad considerando el color de x. Además el vértice j está marcado de verde por ser el vértice que se marcará en esta iteración. Durante la ejecución del algoritmo, el vértice j se marca de algún color que depende del color del vértice i. Así, dependiendo del marcado de j, la cardinalidad del CIF encontrado puede quedarse igual o puede incrementarse en uno. Supóngase que j se marcó de un color de tal forma que la cardinalidad del CIF se mantiene igual a pesar de que pudo haberse incrementado. Si la cardinalidad del CIF puede incrementarse en ese momento, quiere decir que j se encuentra a una distancia de al menos tres aristas del último vértice marcado de rojo, en el camino que comienza en x y termina en i. Esto a su vez implica que i está marcado de amarillo. Así, aplicando el cálculo de la línea 5 en el Pseudocódigo 6, j se marcó de rojo en este caso, lo que contradice la suposición anterior. Así, antes de comenzar la iteración $q+1$, se tiene que el marcado del camino que comienza en x y termina en $i=j$ constituye un CIF de la mayor cardinalidad posible considerando el color de x. Además, j es ahora el vértice verde adyacente a i. Por lo tanto, se mantiene la invariante de lazo en este caso.

Finalización. Al finalizar el algoritmo, el marcado del camino que comienza en x y termina en el vértice a distancia d_{max} de x constituye la sección P en su totalidad y el marcado de ésta constituye un CIF de la mayor cardinalidad posible considerando el color de x. Por lo tanto, se cumple el Lema 3.3 sin importar el color con el que está marcado el vértice x.

A.3. Demostración del Lema 4.1

Invariante de lazo. Antes de comenzar cada iteración i, se tiene que:

1. El conjunto de vértices empaquetados S constituye un CIF válido.
2. El número de vértices empaquetados antes de comenzar la iteración anterior, s_{i-1}, es estrictamente menor al número de vértices empaquetados antes de comenzar la iteración actual, s_i.

3. El número de vértices disponibles antes de comenzar la iteración anterior, a_{i-1}, es estrictamente mayor al número de vértices disponibles antes de comenzar la iteración actual, a_i.

Inicialización. Antes de ejecutarse la primera iteración del algoritmo, se tiene que todos los vértices están disponibles, lo que significa que $S = \emptyset$. Además no hay ninguna iteración anterior con la cuál comparar el número de vértices disponibles o empaquetados. Por lo tanto, la invariante de lazo se cumple por inconstrucción.

Mantenimiento. Supóngase que la invariante de lazo se cumple antes de comenzar alguna iteración arbitraria k. Esto implica que S es un CIF válido y que se tienen $s_k = |S|$ vértices empaquetados y a_k vértices disponibles. Durante la ejecución de esta iteración, se elige un vértice disponible, v, para empaquetarlo. Como v está disponible, quiere decir que el vértice empaquetado más cercano a v se encuentra a una distancia de al menos tres aristas, por lo que S sigue siendo un CIF válido después de empaquetar v. También se tiene que $s_{k+1} = s_k + 1 > s_k$. Por otro lado, se restringen los vértices de $N_2(v)$, lo que reduce el número de vértices disponibles a $a_{k+1} = a_k - |N_2(v)| < a_k$. Por lo tanto, se cumple la invariante de lazo antes de comenzar la iteración $k + 1$.

Finalización. Al finalizar la última iteración del algoritmo, se tiene que S constituye un CIF válido. Además, debido a que el número de vértices disponibles disminuyó en cada iteración, se tiene que ya no quedan vértices disponibles; es decir, todos los vértices del grafo están empaquetados o restringidos. Por lo anterior, se tiene que ya no es posible construir un vértice de mayor cardinalidad. Por lo tanto la solución encontrada por el algoritmo constituye un CIF maximal.

B. Resultados numéricos del análisis estadístico

Las tablas aquí presentadas muestran los resultados numéricos que corresponden a las figuras presentadas en el Capítulo 6. Las Tablas 6 y 7 muestran el tiempo de ejecución y la Tabla 8 muestra la calidad de las soluciones de varios algoritmos.

Tabla 6. Intervalo de confianza para el tiempo de ejecución promedio requerido por varios algoritmos. Las columnas se dividen en tres pares, uno para cada tamaño del grafo que se denota como n en la parte superior del primer renglón. La columna "Inf" indica el límite inferior del intervalo y la columna "Sup" el límite superior. Todos los valores representan el número de segundos.

(a) Tiempo de ejecución de la heurística voraz

Modelo	$n = 2\kappa$		$n = 8\kappa$		$n = 32\kappa$	
	Inf	**Sup**	**Inf**	**Sup**	**Inf**	**Sup**
Barabási-Albert	6.51	6.98	86.97	93.15	1051.54	1167.36
Gilbert	4.58	4.66	67.27	68.56	584.83	596.12
Watts-Strogatz ($k = 4$)	0.24	0.25	1.07	1.08	4.92	4.94
Watts-Strogatz ($k = 8$)	1.96	1.99	11.76	11.84	58.77	59.05
Geométricos	0.28	0.29	1.21	1.22	5.17	5.22
Conectados	1.37	1.41	7.04	7.16	33.84	34.18
Duplicado y divergencia	2.64	3.36	35.47	86.33	541.61	713.10
Cactus lineales	0.04	0.04	0.19	0.20	0.94	0.95
k-outerplanar	0.04	0.04	0.20	0.20	0.97	0.98
Árboles	0.05	0.05	0.22	0.23	1.02	1.09

(b) Tiempo de ejecución de Gurobi

Modelo	$n = 2\kappa$		$n = 8\kappa$		$n = 32\kappa$	
	Inf	**Sup**	**Inf**	**Sup**	**Inf**	**Sup**
Barabási-Albert	0.00	0.00	2.47	2.80	113.48	121.32
Gilbert	13.97	20.82	1531.69	1880.21	N/A (B>A)	
Watts-Strogatz ($k = 4$)	1.00	1.00	42.67	44.35	4052.98	6299.43
Watts-Strogatz ($k = 8$)	16.22	20.71	2482.09	3049.13	N/A (B>A)	
Geométricos	0.40	0.90	42.94	53.40	2269.15	2447.99
Conectados	1.80	2.73	113.30	137.62	(22782)	(37116)
Duplicado y divergencia	(0.01)	(0.01)	0.05	0.06	0.31	0.33
Cactus lineales	0.00	0.00	0.00	0.00	1.00	1.00
k-outerplanar	0.00	0.00	0.00	0.00	1.00	1.00
Árboles	0.01	0.01	0.03	0.03	0.12	0.12

Tabla 7. Comparación del tiempo de ejecución de la heurística voraz y el algoritmo secuencial para los grafos cactus lineales. Las columnas se dividen en tres pares, uno para cada tamaño del grafo, el cuál se denota con la letra n en la parte superior del primer renglón. La columna "Inf" indica el límite inferior del intervalo y la columna "Sup" el límite superior. Todos los valores representan el número de segundos.

Algoritmo	$n = 2\kappa$		$n = 8\kappa$		$n = 32\kappa$	
	Inf	**Sup**	**Inf**	**Sup**	**Inf**	**Sup**
Greedy-Maximal-2Packing	0.042	0.044	0.195	0.197	0.941	0.951
Linear-Cactus-2Packing	0.004	0.005	0.020	0.022	0.109	0.122

Tabla 8. Intervalo de confianza para la calidad promedio de las soluciones encontradas por varios algoritmos. Las columnas se dividen en tres pares, uno para cada tamaño del grafo que se como n en la parte superior del primer renglón. La columna "Inf" indica el límite inferior del intervalo y la columna "Sup" el límite superior. Todos los valores representan un porcentaje con respecto a la mejor solución correspondiente encontrada por Gurobi.

(a) Calidad de los resultados obtenidos por la heurística voraz

Modelo	$n = 2\kappa$		$n = 8\kappa$		$n = 32\kappa$	
	Inf	**Sup**	**Inf**	**Sup**	**Inf**	**Sup**
Barabási-Albert	92.50	93.32	92.72	93.12	92.95	93.12
Gilbert	91.83	92.94	96.09	97.16	N/A (A>B)	
Watts-Strogatz ($k = 4$)	92.44	93.11	93.00	93.29	93.58	93.83
Watts-Strogatz ($k = 8$)	91.15	92.21	93.68	94.39	N/A (A>B)	
Geométricos	93.35	94.44	93.19	93.60	93.67	93.99
Conectados	92.66	93.40	93.11	93.54	N/A (A=B)	
Duplicado y divergencia	99.75	99.93	99.78	99.87	99.81	99.85
Cactus lineales	99.29	99.49	99.37	99.44	99.41	99.45
k-outerplanar	98.93	99.13	98.88	98.98	98.88	98.95
Árboles	99.90	99.97	99.87	99.93	99.87	99.89

(b) Calidad de los resultados obtenidos por la heurística aleatoria ingenua

Modelo	$n = 2\kappa$		$n = 8\kappa$		$n = 32\kappa$	
	Inf	**Sup**	**Inf**	**Sup**	**Inf**	**Sup**
Barabási-Albert	66.10	68.17	67.82	68.62	68.24	68.60
Gilbert	61.66	64.48	64.26	65.63	N/A (A>B)	
Watts-Strogatz ($k = 4$)	70.64	71.73	71.39	71.88	71.82	72.04
Watts-Strogatz ($k = 8$)	66.80	68.11	68.59	69.46	99.02	99.70
Geométricos	68.09	69.86	68.70	69.55	69.24	69.62
Conectados	66.33	67.70	67.17	67.87	72.86	80.57
Duplicado y divergencia	76.84	78.66	77.86	79.05	78.27	78.71
Cactus lineales	81.97	82.60	82.12	82.41	82.22	82.37
k-outerplanar	81.59	82.21	81.63	81.91	81.80	81.93
Árboles	81.24	82.20	81.71	82.18	81.93	82.20

(c) Calidad de las primeras soluciones maximales obtenidas por Gurobi

Modelo	$n = 2\kappa$		$n = 8\kappa$		$n = 32\kappa$	
	Inf	**Sup**	**Inf**	**Sup**	**Inf**	**Sup**
Barabási-Albert	61.65	64.01	64.61	65.62	66.14	66.74
Gilbert	54.93	56.78	57.90	59.02	N/A (A=B)	
Watts-Strogatz ($k = 4$)	61.83	62.62	62.03	62.57	62.69	62.91
Watts-Strogatz ($k = 8$)	59.16	60.67	60.73	61.50	87.32	88.08
Geométricos	66.87	69.18	66.43	67.17	66.57	67.09
Conectados	57.27	58.70	57.85	58.60	62.71	68.96
Duplicado y divergencia	84.36	86.85	82.47	83.64	80.34	81.18
Cactus lineales	77.30	78.10	77.69	78.03	77.63	77.78
k-outerplanar	74.63	75.46	75.09	75.41	74.99	75.22
Árboles	99.89	99.96	99.98	99.99	99.99	100.00