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

Maestría en Ciencias
en Ciencias de la Computación

Análisis de la factibilidad del diseño de algoritmos auto-estabilizantes para el conjunto independiente fuerte con peso máximo en uniciclos

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

David Zhou Tan

Ensenada, Baja California, México

2018

Tesis defendida por

David Zhou Tan

y aprobada por el siguiente Comité

Dr. José Alberto Fernández Zepeda

Codirector del Comité

Dr. Joel Antonio Trejo Sánchez

Codirector del Comité

Dr. Carlos Alberto Brizuela Rodríguez

Dr. Ricardo Arturo Chávez Pérez

Dr. Jesús Favela Vara

Coordinador del Posgrado en Ciencias de la Computación

Dra. Rufina Hernández Martínez

Directora de Estudios de Posgrado

Resumen de la tesis que presenta **David Zhou Tan** como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Análisis de la factibilidad del diseño de algoritmos auto-estabilizantes para el conjunto independiente fuerte con peso máximo en uniciclos

Resumen aprobado por:

Dr. José Alberto Fernández Zepeda	Dr. Joel Antonio Trejo Sánchez
Codirector de Tesis	Codirector de Tesis

En este trabajo se estudia el problema de encontrar el conjunto independiente fuerte (2-packing) con peso máximo, que suele aparecer en problemas de exclusión mutua, calendarización, etiquetado de mapas, entre otros. Este problema en grafos generales es NP-difícil, pero para grafos restringidos como el árbol, el unicicloy el cactus, se han encontrado soluciones eficientes. En esta investigación se trabaja en los casos del cactus y del unicicloPara el cactus se propone un algoritmo secuencial basado en un algoritmo reciente en la literatura que resuelve el mismo problema pero en cactus sin peso, y se implementa en un lenguaje de programación orientado a objetos (Java). Para el uniciclose diseña un algoritmo auto-estabilizante basado en la implementación anterior. Además se plantean los pasos para generalizar este último para que funcione en un cactus. Los algoritmos auto-estabilizantes se utilizan en sistemas distribuidos donde se necesita asegurar estabilidad sin necesidad de que haya interacción humana, incluso cuando ocurren fallos transitorios, tales como cortes de energía y pérdidas de memoria.

Palabras Clave: **Auto-estabilización, 2-packing, cactus, unicicloo**

Abstract of the thesis presented by **David Zhou Tan** as a partial requirement to obtain the Master of Science degree in Computer Science.

Feasibility analysis of the design of self-stabilizing algorithms for the maximum weighted independent set problem in unicycles

Abstract approved by:

Dr. José Alberto Fernández Zepeda	Dr. Joel Antonio Trejo Sánchez
Thesis Co-Director	Thesis Co-Director

In this thesis we study the problem of finding the strong independent set (2-packing) with maximum weight, usually appearing in problems of mutual exclusion, tasks scheduling, map labeling, among others. The 2-packing problem in general graphs is NP-hard, but in restricted graphs such as trees, unicycles and cacti, efficient solutions have been found. In this research we work on cacti and unicycles. For the cacti, we propose a sequential algorithm based on a recent algorithm in the literature that solves the same problem in unweighted cacti. This new algorithm is implemented in a object oriented programming languaje (Java). For the unicycles we design a self-stabilizing algorithm based on the previous implementation. Also, we present the steps to generalize it for the cacti. Self-stabilizing algorithms are used in distribuited systems where it is necessary to ensure stability without human interaction, even when transient faults occur, such as energy cuts and memory leaks.

Keywords: **Self-stabilization, 2-packing, cactus, unicycle**

Dedicatoria

A mis padres.

Agradecimientos

A mis directores de tesis: el Dr. José Alberto Fernández Zepeda y el Dr. Joel Antonio Trejo Sánchez, por el conocimiento y asesoría durante todo este tiempo.

A los miembros de mi comité: el Dr. Carlos Alberto Brizuela Rodríguez y el Dr. Ricardo Arturo Chávez Pérez, por sus observaciones y su valiosa retroalimentación durante el desarrollo de la tesis.

A mis compañeros de generación. Gracias por compartir tantas experiencias y apoyo mutuo en estos dos años.

A mis compañeros del Laboratorio de Algoritmos, por los momentos de plática y diversión, así como los de seriedad y trabajo.

A los compañeros del Departamento de Computación, por tantos momentos excepcionales.

A mi familia y amigos, por estar siempre conmigo.

Al Centro de Investigación Científica y de Educación Superior de Ensenada.

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

Tabla de contenido

Página

Tabla de contenido (continuación)

Lista de figuras

Figura Página

Lista de tablas

Tabla Página

Capítulo 1. Introducción

1.1 Auto-estabilización

En el entorno de los sistemas distribuidos, se busca que el sistema mantenga un estado estable, es decir, evitar que se realicen pasos no permitidos que desestabilicen el entorno global. Si los miembros de un sistema distribuido tienen acceso a una memoria compartida, alcanzar el estado estable es sencillo. Pero cuando cada nodo solamente puede utilizar información local, ya sea del mismo nodo o de los nodos vecinos, el problema se complica.

Un sistema distribuido se puede modelar como un grafo $G = (V, E)$, donde V es el conjunto de vértices, los cuales representan a cada uno de los nodos en el sistema, y E es el conjunto de aristas, que representan conexiones entre pares de vértices, es decir, si dos nodos tienen comunicación directa en el sistema. Se le conoce como *vecindario abierto* de un vértice i al conjunto de vértices con arista incidente a i (Goddard *et al.*, 2008). Si a este vecindario se agrega el vértice i, entonces se convierte en el *vecindario cerrado* de i. En este trabajo, cada vértice solamente tiene acceso a la información dentro de su vecindario cerrado.

Se definen a los privilegios (Dijkstra, 1974) como las funciones booleanas de los estados de los vértices y sus vecinos. Si alguna de estas funciones es verdadera, se dice que el vértice se encuentra privilegiado. Un calendarizador central es quien escoge uno de los vértices privilegiados, y en su turno, dicho vértice puede realizar un movimiento o cambio. Estas funciones y cambios tienen la forma: **si** (condición) **entonces** (movimiento), y en ocasiones se les llaman *reglas*. Un sistema es auto-estabilizante si, independientemente del estado inicial que adopte, puede alcanzar un estado estable en un tiempo finito a través de los turnos elegidos por el calendarizador. A esta propiedad se le conoce como *convergencia*. Una vez alcanzado el estado estable, si se mantiene en él, tiene la propiedad de la *cerradura*. Los algoritmos auto-estabilizantes son aquellos que operan bajo este tipo de sistemas distribuidos. Cada vértice en el sistema funciona bajo el mismo algoritmo, que consiste en una serie de reglas que permiten modificar las variables locales

para alcanzar un estado estable.

Existen diferentes tipos de calendarizadores (Tixeuil, 2010) además del central. Uno de ellos es el síncrono, en el cual todos los vértices privilegiados realizan un movimiento en el mismo instante. Otro tipo de calendarizador más general es el distribuido, el cual puede elegir cualquier cantidad de vértices privilegiados para mover en un instante. Si se elige uno solo, entonces es un calendarizador central; y si se eligen todos, es uno síncrono. Por lo tanto, un calendarizador distribuido es mucho más general que los otros.

El tiempo de ejecución de los algoritmos auto-estabilizantes se mide de forma diferente a los algoritmos secuenciales. En los algoritmos secuenciales el tiempo de ejecución es la cantidad máxima de operaciones que se necesitan realizar para proporcionar una solución, mientras que en los algoritmos auto-estabilizantes es la máxima cantidad de reglas que tienen que ejecutarse (movimientos) para llegar a un estado estable (Mjelde, 2004). Otra forma de calcular el tiempo de ejecución en algoritmos auto-estabilizantes es mediante las rondas. Una ronda es una secuencia de ejecución mínima para que cada vértice privilegiado se ejecute al menos una vez.

Los algoritmos auto-estabilizantes (Dolev, 2000) se utilizan principalmente en sistemas autónomos grandes, donde es difícil predecir cuándo o cómo ocurrirán fallas transitorias. Por ejemplo, en vehículos autónomos no es posible saber cuándo ocurrirá una falla que altere el sistema, por lo que debe estar preparado para estabilizarse por sí mismo antes de reanudar sus actividades.

1.2 Conjuntos independientes

El problema del conjunto independiente se define como sigue: dado un grafo $G = (V, E)$, se dice que $I \subseteq V$ es un conjunto independiente de G si no hay dos vértices en I que sean adyacentes (o que la distancia mínima, medida en aristas, entre ellos sea al menos dos). Un conjunto independiente maximal es aquel que no se le puede agregar un vértice más al conjunto sin violar la propiedad anterior. Un conjunto independiente máximo es un conjunto independiente maximal de cardinalidad máxima. El problema de encontrar un conjunto independiente máximo se encuentra en la clase NP-Difícil, pero

el de encontrar uno maximal se encuentra en la clase P, por lo que es mucho más fácil resolver el último. Además, no es posible encontrar un algoritmo de aproximación de factor constante al problema del conjunto independiente máximo a menos de que P = NP (Arora *et al.*, 1998).

Una versión más general del problema anterior es el de encontrar el conjunto k-*packing* (Meir y Moon, 1975). Se dice que $S \subseteq V$ es un conjunto k-*packing* de un grafo $G = (V, E)$ si la distancia, medida en aristas, entre cada par de vértices en S es mayor a k. En este sentido, el problema de encontrar el conjunto independiente es equivalente a encontrar el conjunto *1-packing*. Este trabajo se enfoca en encontrar el conjunto *2-packing*.

Un problema muy relacionado con los conjuntos independientes son los conjuntos dominantes. Dado un grafo $G = (V, E)$, se dice que $D \subseteq V$ es un conjunto dominante si para todo vértice de V no incluído en D, es adyacente a un vértice en D. Encontrar el conjunto dominante de menor cardinalidad es un problema NP-Completo (Garey y Johnson, 1990). Al igual que los conjuntos independientes, también existen los conjuntos dominantes minimales, que son los conjuntos dominantes tales que no existe algún subconjunto propio de ellos que también sea dominante. La relación con los conjuntos independientes es que todo conjunto independiente maximal es un conjunto dominante minimal.

Otro problema similar a los conjuntos independientes es el cubrimiento de vértices. Un cubrimiento de vértices de un grafo $G = (V, E)$ es un subconjunto $C \subseteq V$, tal que toda arista $e \in E$ es incidente a al menos un vértice en C. Encontrar el subconjunto de menor tamaño es un problema NP-Difícil (Karp, 1972). Los vértices que no se encuentran en un cubrimiento de vértices forman un conjunto independiente.

También existen variantes de cada uno de éstos y otros problemas con pesos. En la mayoría de los casos, se considera a los problemas con pesos como casos generalizados de los problemas sin pesos, ya que estos últimos pueden verse como pesos unitarios. En el caso del problema de encontrar el conjunto 2-packing con peso máximo, a cada vértice se le asigna un peso positivo, y el problema se convierte en encontrar el subconjunto de vértices cuya distancia entre ellos sea mayor a dos, y entre todos los subconjuntos posibles, escoger el que tenga mayor suma de pesos.

En el caso del problema de cubrimiento de vértices con peso, también se le asignan pesos a los vértices del grafo, y el problema consiste en encontrar el subconjunto de vértices que cubra todas las aristas, pero cuya suma de pesos sea la menor. Para este problema se han propuesto soluciones con programación lineal, utilizando el esquema primal dual para aproximar el resultado óptimo (Cormen *et al.*, 2009), (Vazirani, 2001).

1.2.1. Aplicación de los conjuntos independientes

Una posible aplicación de los conjuntos independientes es la siguiente. Suponga que en una universidad se está realizando un evento tipo olimpiada, donde cada estudiante se puede inscribir y participar en uno o más deportes. Una vez inscritos los estudiantes, el comité organizador calendariza los juegos o eventos de cada deporte, cuidando que no se programen dos juegos al mismo tiempo si existe algún estudiante que participe en ambos. Sin embargo, hay una hora en la que varios cazatalentos llegarán a visualizar a los estudiantes con mayor potencial, por lo que la universidad quiere programar la mayor cantidad de eventos a dicha hora.

Este problema se puede modelar como un grafo $G = (V, E)$, donde cada vértice $v \in V$ representa a un evento, y cada arista $e \in E$ conecta dos vértices si existe algún estudiante que participa en los eventos representados por ambos vértices. Si dos vértices comparten una arista, esos eventos contienen al menos un estudiante en común y no se pueden programar al mismo tiempo, por lo que el 1-packing máximo representa la mayor cantidad de eventos que se pueden programar sin que se violen las restricciones anteriores.

Por ejemplo, en la Tabla 1 se presenta el caso de cinco estudiantes y los deportes en los que participan. En la Figura 1a se muestra el grafo de estas relaciones. El vértice A tiene una arista a B porque Pedro participa en 100m y 200m, así como B se conecta a C porque Rodrigo participa en 200m y salto de longitud. En la Figura 1(b) se encuentra el conjunto 1-packing máximo (vértices pintados) de este grafo, lo cual significa que la mayor cantidad de eventos que se pueden programar al mismo tiempo son tres: 100 metros, salto de longitud y lanzamiento de bala.

	100m (A)	200m (B)	Salto de longitud (C)	Lanzamiento de bala (D)
Pedro	X	X		
Jimena	X			
Juan			X	
Rodrigo		X	X	
Karen				X

Tabla 1: Tabla de participación de los estudiantes.

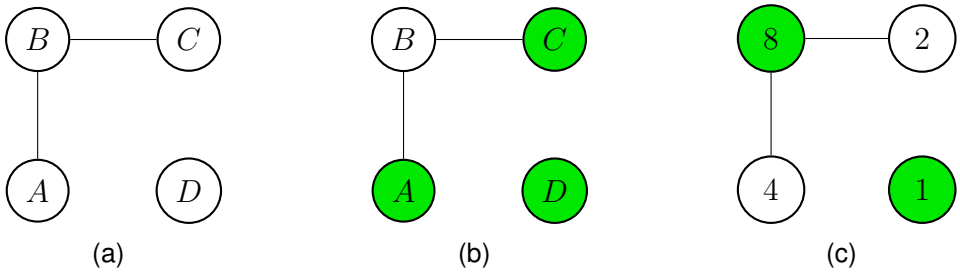


Figura 1: El vértice A corresponde al evento de 100 metros, B a 200 metros, C a salto de longitud y D a lanzamiento de bala. (a) grafo con la representación de los eventos y participación de estudiantes, (b) los vértices coloreados representan el conjunto 1-packing máximo, y (c) los vértices coloreados representan el conjunto 1-packing con peso máximo.

Este problema se puede extender al conjunto 1-packing con peso máximo si se le asignan pesos a los vértices, dependiendo de la cantidad de cazatalentos que existan de cada tipo de deporte. En la Figura 1(c) se muestra el caso cuando hay un cazatalentos de lanzamiento de bala, cuatro de 100 metros, ocho de 200 metros y dos de salto de altura. En este caso el conjunto 1-packing con peso máximo es el conjunto con los eventos de 200 metros y lanzamiento de bala, con un peso total de nueve. Note que el conjunto 1-packing con peso máximo no necesariamente es el conjunto 1-packing con mayor cardinalidad. Los pesos asignados a cada evento pudieran ser otros, por ejemplo, la importancia o popularidad del deporte, o cualquier otro criterio.

1.3 Antecedentes

Gairing *et al.* (2004) presentaron el primer algoritmo auto-estabilizante para encontrar un conjunto 2-packing maximal en tiempo exponencial. En este trabajo, ellos utilizan identificadores únicos en los vértices y dos variables locales simples: una para indicar la pertenencia al conjunto 2-packing y un apuntador para limitar la cantidad de vértices en el

conjunto 2-packing. La idea general del algoritmo es meter o sacar vértices del conjunto 2-packing dependiendo de los vecinos, que el tamaño de algunos grupos dentro del algoritmo nunca incrementan, y que éstos decrezcan paulatinamente con una cantidad finita de movimientos entre cada decremento. El algoritmo utiliza el calendarizador adversario central, y la complejidad espacial es de orden logarítmica en relación con el tamaño de los vértices, ya que requieren utilizar apuntadores a los vértices vecinos.

Para el conjunto k-packing maximal, Manne y Mjelde (2006) diseñaron un algoritmo auto-estabilizante, utilizando también identificadores únicos ordenables y el calendarizador adversario. Cada vértice tiene dos variables locales: la distancia y el identificador del vértice más cercano perteneciente al conjunto k-packing, y la distancia y el identificador al segundo vértice más cercano en el conjunto k-packing. Usando una función de soporte, se pueden calcular los valores adecuados de cada uno de los valores anteriores, convergiendo a un estado estable en tiempo exponencial. Aunque su idea no parece que se pueda expandir a redes anónimas (donde los vértices no tienen identificadores únicos), comienzan a explorar la compensación de la complejidad del tiempo contra el espacio.

Aún para el problema del conjunto k-packing maximal, Goddard *et al.* (2008) hacen uso de la *distancia*-k. A diferencia de los anteriores que utilizan información de un vértice y de sus vecinos inmediatos, en este caso se puede conocer la información de todos los vértices a una distancia k. Esta información se guarda en forma de grafo en cada vértice, así como dos variables locales muy similares a las de Gairing. La idea de este algoritmo es realizar conversiones desde un conocimiento de distancia k a uno de distancia $k/2$, y luego a $k/4$, y así sucesivamente. De esta manera, se puede llegar hasta la distancia 1, el cual es el esquema general del problema a tratar. Este algoritmo funciona en tiempo polinomial, ya que cada transformación tiene un costo polinomial en convertir de un modelo de distancia a otro. Pero su desventaja es que el espacio requerido en cada vértice para almacenar la información de sus vecinos a distancia k es del orden $n^{O(\log k)}$, donde n es la cantidad de vértices. En dicho artículo, ya existe una compensación de tiempo y espacio: se sacrifica almacenamiento de los vértices para que el algoritmo funcione de manera más rápida.

Shi (2012) comienza a trabajar con el calendarizador síncrono, el cual permite mover a

todos los vértices privilegiados en el mismo momento, aunque también realiza un análisis con cualquier calendarizador, como el distribuido. Utilizando reglas sencillas con información sobre los vértices más cercanos dentro del conjunto 2-packing, logra encontrar estabilidad en $O(mn)$ movimientos para cualquier calendarizador, y $O(n^2)$ rondas para el calendarizador síncrono, donde n y m son la cantidad de vértices y aristas, respectivamente. Este algoritmo auto-estabilizante funciona solamente con identificadores únicos ordenables. La complejidad espacial es de orden logarítmico.

Trejo-Sánchez y Fernández-Zepeda (2012) presentan un algoritmo para encontrar el conjunto 2-packing maximal en un grafo tipo cactus. El grafo tipo cactus es aquel en que cada arista a lo más puede pertenecer a un ciclo. Su algoritmo hace uso de la elección de líder para los grupos de vértices dentro del grafo y de un retraso para poder actualizar los vecinos de un vértice antes de cambiarlo, ya que el calendarizador utilizado es el síncrono. Este algoritmo corre en $O(D)$ rondas, donde D es el diámetro del cactus. El espacio requerido es de orden logarítmico.

Ding *et al.* (2014) presentan un algoritmo con convergencia segura para encontrar el conjunto 2-packing maximal, el cual asegura que la estabilidad no se pierde durante el intervalo de convergencia. En su algoritmo se encuentra un conjunto 2-packing en tres rondas (utiliza el calendarizador síncrono), pero no necesariamente maximal. Después de $O(n)$ rondas, encuentra un conjunto 2-packing maximal, sin romper la estabilidad en ningún momento.

Para el caso del k-packing máximo, Mjelde (2004) propone un algoritmo basado en programación dinámica que resuelve este problema en árboles en tiempo lineal. También diseña un algoritmo auto-estabilizante para el mismo problema, el cual se ejecuta en $O(n^3)$ movimientos. Estos algoritmos pueden modificarse para generar la solución a una variante con pesos en los vértices, y el tiempo de ejecución se mantiene.

Recientemente, Flores-Lamas *et al.* (2018) diseñaron un algoritmo secuencial para encontrar el conjunto 2-packing máximo en cactus, basado en el algoritmo de Mjelde. En éste se realiza un análisis para el caso del uniciclo y se extienden las propiedades a un cactus. Este algoritmo tiene complejidad $O(n^2)$ en unidades de tiempo.

1.4 Objetivo general

El objetivo general de este trabajo es analizar la factibilidad de diseñar algoritmos auto-estabilizantes para resolver el problema del conjunto independiente fuerte con peso máximo en uniciclos en tiempo polinomial.

1.4.1. Objetivos específicos

1. Analizar y diseñar un algoritmo secuencial para resolver en tiempo polinomial el problema del conjunto independiente fuerte con peso máximo en grafos cactus.
2. Implementar el algoritmo anterior en un lenguaje de programación.
3. Diseñar un algoritmo auto-estabilizante para resolver el problema en uniciclos en tiempo polinomial o justificar su dificultad.

1.5 Organización de la tesis

El resto del documento se organiza de la siguiente forma. En el Capítulo 2 se presenta un algoritmo secuencial que encuentra un conjunto 2-packing con peso máximo en un cactus. Este algoritmo se implementó en un lenguaje de programación orientado a objetos (Java). El Capítulo 3 presenta una versión auto-estabilizante del algoritmo anterior para un unicicloo, se demuestra su funcionalidad y se analiza su tiempo de ejecución. También se explican los pasos para extender el algoritmo a un cactus. Por último, el Capítulo 4 establece las conclusiones generales y el trabajo futuro.

Capítulo 2. 2-packing con peso máximo en cactus

A partir de este capítulo se hará referencia al algoritmo MAXIMUM-2-PACK-CACTUS presentado en (Flores-Lamas *et al.*, 2018) como $\mathcal{F}$, y al algoritmo con las adaptaciones para encontrar el conjunto 2-packing con peso máximo en cactus como $\mathcal{G}$. A continuación se presenta un esquema del funcionamiento y una breve descripción de cada uno de los pasos del algoritmo $\mathcal{F}$. Más adelante se explica con detalle cada uno de ellos.

El algoritmo $\mathcal{F}$ toma como entrada un grafo cactus $G = (V, E)$, donde V es un conjunto de vértices y E es un conjunto de aristas de la forma $e = (x, y)$, donde $x, y \in V$. Además, el grafo G es un cactus, es decir, cumple la propiedad de que cada par de ciclos en G comparten a lo más un vértice.

2.1 Esquema general del algoritmo $\mathcal{F}$

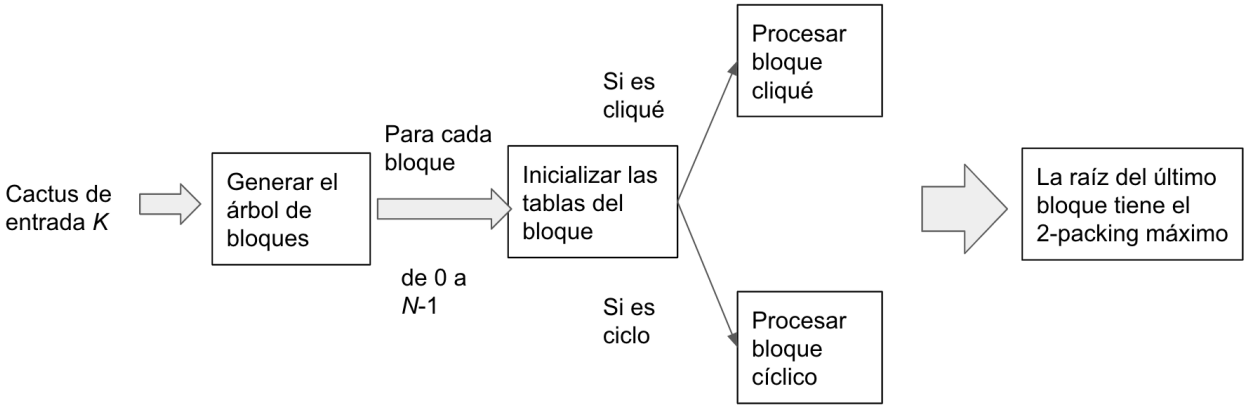


Figura 2: Diagrama general del algoritmo $\mathcal{F}$. N es la cantidad de bloques generados en el árbol de bloques.

- Se genera el árbol de bloques a partir del cactus G de entrada y se numera cada bloque de manera *post orden*. Un *bloque* es un componente biconectado maximal, es decir, un conjunto de vértices conectados por aristas, tales que al remover un vértice de ellos, no se crean más componentes conectados. En el caso de los cactus, los bloques solamente pueden ser de dos tipos: ciclos simples y puentes. Estos últimos equivalen a cliqués de tamaño dos que forman un puente, pero para simplicidad de este trabajo se les llama solamente *cliqués*. A los bloques ciclo se les llama solamente *ciclo*.

- Para cada uno de los bloques B_i, i comenzando desde 0 hasta $N-1$, se inicializan las tablas del bloque, y se procesa como cliqué o como ciclo, dependiendo del tipo de bloque que sea (ver Figura 2). Cada uno de los vértices v de un bloque tiene una tabla, en la que se guarda la mejor configuración hasta v, es decir, la mejor solución si solamente existieran v y sus vértices descendientes. Como el algoritmo funciona utilizando programación dinámica, la tabla tiene información de hasta distancia tres de v, lo que permite combinar soluciones actuales con las óptimas hasta ese punto.
    - Si el bloque es un cliqué de tamaño dos, se procesan las tablas del vértice más alejado de la raíz y después del vértice restante, en ese orden. El valor de las celdas de la tabla dependen de los bloques descendientes concurrentes a cada vértice.
    - Si el bloque no es cliqué de tamaño dos, entonces es un ciclo, por lo que se realiza un procedimiento para calcular el mejor árbol posible, removiendo una a una las aristas del ciclo, y verificando que no haya conflictos al regresar la arista removida a su lugar.
- Al terminar de procesar el bloque $N-1$ o bloque raíz, la solución al problema del conjunto 2-packing en el cactus se encuentra en el vértice raíz del bloque B_{N-1}.

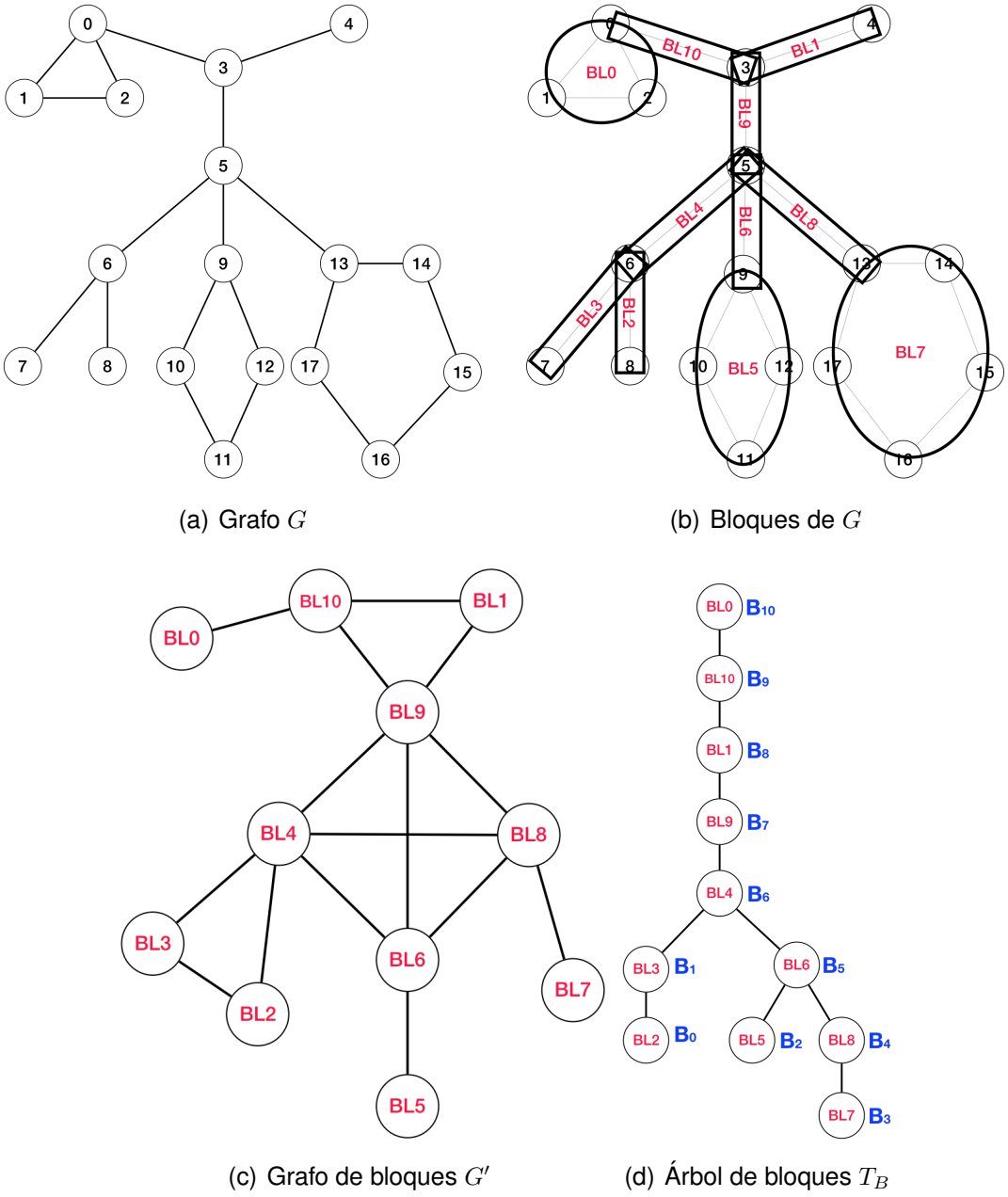
2.2 Pseudocódigos

El algoritmo $\mathcal{F}$ se compone de 11 rutinas o *pseudocódigos* (ver Apéndice A). Cada uno de ellos contiene una rutina que realiza una tarea específica en el algoritmo. Durante este trabajo se referencian dichos pseudocódigos y rutinas para la explicación del algoritmo $\mathcal{G}$, así como para el algoritmo auto-estabilizante, ya que muchas de las acciones son semejantes. A continuación se detallan cada uno de los procesos mencionados.

2.3 Creación del árbol de bloques

Para identificar los bloques del cactus, se realiza un recorrido por profundidad (DFS) en el grafo, manteniendo una lista de vértices visitados para crear los bloques adecuadamente (Figura 3(b)). Después se genera un grafo de bloques G', donde cada vértice

corresponde a un bloque, y dos vértices están unidos por una arista si los dos bloques correspondientes comparten un vértice en G. Por ello, cada vértice puede aparecer en dos o más bloques (ver Figura 3(c)).



(a) Grafo G (b) Bloques de G

(c) Grafo de bloques G' (d) Árbol de bloques T_B

Figura 3: Creación del árbol de bloques. (a) el grafo original; (b) se identifican los bloques, en rojo aparece el nombre que se les asigna, que es igual al orden en que fueron encontrados; (c) la creación del grafo de bloques; (d) el árbol de bloques donde se ve en azul la numeración de cada bloque.

Se escoge un bloque arbitrario BL y se genera un árbol de bloques T_B enraizado en BL, numerando los bloques en postorden. Este paso equivale a realizar un recorrido DFS

en G', donde el primer bloque recorrido es B_0, el siguiente B_1, etc., y la raíz del árbol de bloques es B_{N-1}, donde N es el número total de bloques (Figura 3(d)).

En cada bloque B_i se asigna un vértice raíz. Este vértice es el que se comparte con el bloque padre de B_i para $i < N-1$. Para $i = N-1$ (el bloque raíz), el vértice raíz es el que tiene el identificador menor. Por ejemplo, en la Figura 3(b), el bloque BL_6 se encuentra conformado por los vértices 5 y 9, BL_4 por los vértices 5 y 6, y BL_9 por los vértices 5 y 3. En la Figura 3(d), BL_4 (B_6) es el padre de BL_6 (B_5), por lo que el vértice raíz de BL_6 (B_5) es el vértice que tienen en común, en este caso, el vértice 5. Asimismo, BL_9 (B_7) es padre de BL_4 (B_6), por lo que el vértice en común entre ambos bloques es 5, el cual es el vértice raíz de BL_4 (B_6).

También dentro de cada bloque se numeran los vértices. Para un cliqué, el vértice raíz tiene numeración 0, y 1 para el vértice restante; estos vértices reciben el nombre de superior e inferior, respectivamente. Para un ciclo con k vértices, éstos se numeran en sucesión comenzando en 0 con el vértice raíz. Al vecino de la raíz en el ciclo con menor identificador se le asigna 1, y en esa dirección se continúa hasta terminar con el otro vecino de ciclo del vértice raíz, el de identificador mayor, que se numera $k-1$. La Figura 4 contiene un ejemplo de esta numeración.

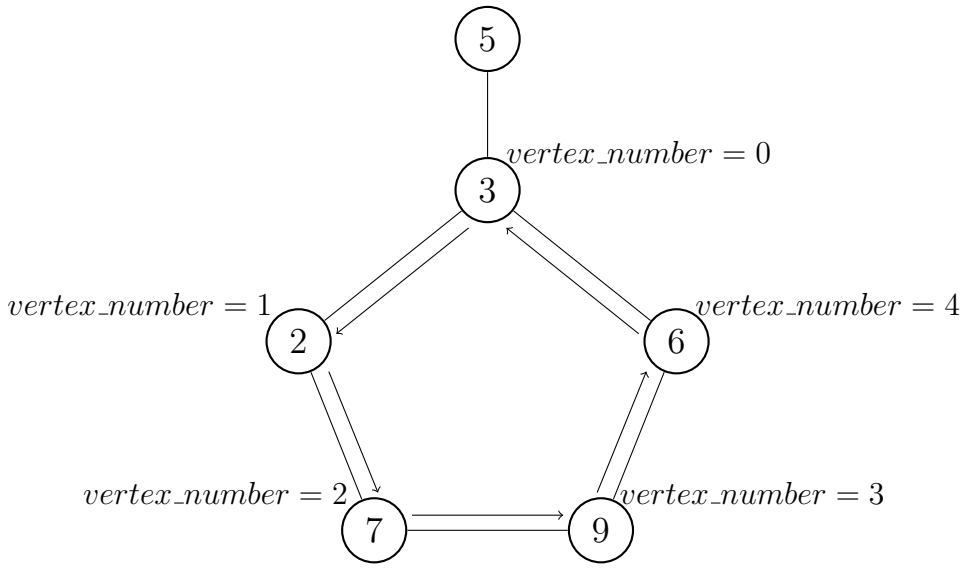


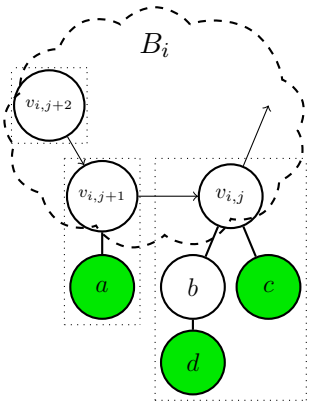
Figura 4: Ejemplo de numeración de los vértices de un ciclo. Los números dentro de cada vértice indican el identificador de los mismos. Las flechas indican la dirección en que se numeran los vértices.

2.4 Inicialización de las tablas

Existe una tabla auxiliar para cada vértice de cada bloque, de cuatro renglones y tres columnas para la raíz del bloque, y de tres renglones y dos columnas para los vértices restantes. La nomenclatura $\mathcal{T}_{v_{i,j}}[r][c]$ indica la celda en el renglón r y columna c en la tabla del vértice con numeración j dentro del bloque i, y en ella se mantiene la mejor solución posible hasta dicho vértice. Cada celda contiene tres campos, llamados key, set y $dist$. El primero indica la cardinalidad de la solución guardada, el segundo contiene el conjunto de vértices de dicha solución, y el tercero la distancia mínima desde $v_{i,j}$ hacia el vértice más cercano contenido en la solución.

En la celda $\mathcal{T}_{v_{i,j}}[r][c]$, r corresponde al renglón, que indica que la solución contenida se encuentra a una distancia de al menos r aristas del vértice $v_{i,j}$. Por ejemplo, si $r = 2$, los vértices contenidos en la solución de la celda están a distancia dos o más de $v_{i,j}$. En cambio, c corresponde a la columna de la tabla, que indica los descendientes que se toman en cuenta para la solución. La columna 0 considera al vértice $v_{i,j}$ y a todos los bloques descendientes de $v_{i,j}$. La columna 1 considera al vértice $v_{i,j}$, a todos los bloques descendientes de $v_{i,j}$, así como a los vértices descendientes de $v_{i,j}$ en el bloque i y sus bloques descendientes. Finalmente, la columna 2, que es exclusiva de las raíces de los bloques, considera la mejor solución del bloque i y todos los bloques descendientes a éste. En la Figura 5 se muestra un ejemplo de estas columnas y renglones.

La rutina de inicialización de las tablas (ver Pseudocódigo 2 del Apéndice A) se encarga de llenar las columnas 0 de cada una las tablas de los vértices de un bloque, sin importar qué tipo de bloque sea. Esta columna utiliza las tablas de las raíces de los bloques descendientes a cada vértice v, es decir, combina las soluciones de los bloques descendientes de v y las agrega en v. Por ejemplo, en la Figura 3, para inicializar la columna 0 del vértice 6 en el bloque B_4, se utiliza la tabla de la raíz del bloque descendiente en el vértice 6, en este caso el bloque B_3, cuya raíz es el mismo vértice 6, pero en el bloque B_3. Para inicializar la columna 0 del vértice 5 en B_4, se utiliza la tabla de la raíz del bloque descendiente, en este caso, la tabla del vértice 5 en el bloque B_6. Como ambos bloques B_3 y B_6 son descendientes del bloque B_4 en el árbol de bloques, tienen una



(a) Fragmento de grafo

	0	1
0	$\{2, (c, d), 1)$	$\{3, (a, c, d), 1\}$
1	$\{2, (c, d), 1)$	$\{3, (a, c, d), 1\}$
2	$\{1, (d), 2)$	$\{2, (a, d), 2\}$

(b) Tabla $\mathcal{T}$ del vértice $v_{i,j}$

Figura 5: Ejemplo extraído de (Flores-Lamas *et al.*, 2018); (a) en líneas punteadas se encuentran los subgrafos considerados para las columnas 0 de cada vértice en el ciclo B_i; (b) para el vértice $v_{i,j}$, su columna 0 considera a $v_{i,j}$, b, c y d, mientras que su columna 1 contiene a $v_{i,j+1}$, a, y $v_{i,j+2}$ también, así como a sus descendientes.

numeración menor, por lo que se asegura que sus tablas estén llenas en este momento.

2.5 Procesar cliqué

El procedimiento PROCESS-CLIQUE-BLOCK (ver Pseudocódigo 3 del Apéndice A) se encarga de procesar las columnas 1 de las tablas de los vértices de un cliqué, así como la columna 2 del vértice raíz.

La columna 1 del vértice inferior es idéntica a la columna 0, ya que éste no tiene descendientes en el bloque. La columna 1 del vértice superior se calcula combinando su columna 0 y la columna 1 del vértice inferior, de tal manera que las soluciones de ambas celdas no incurran en conflicto, es decir, que los vértices mantengan una distancia mínima de tres entre ellos.

Finalmente, la columna 2 del vértice superior es la misma que la columna 1, ya que se representa el mismo subgrafo en ambas columnas. Esta columna se utiliza en el proceso de inicialización de la tabla del bloque padre a este cliqué.

2.6 Procesar ciclo

El procedimiento PROCESS-CYCLIC-BLOCK (ver Pseudocódigo 7 del Apéndice A) se encarga de procesar las columnas 1 de las tablas de los vértices de un ciclo, así como la columna 2 del vértice raíz.

Sea k la cantidad de vértices en el ciclo actual i; este número es igual a la cantidad de aristas en dicho bloque. Se enumeran las aristas de la siguiente manera: la arista entre los vértices $v_{i,j}$ y $v_{i,j+1}$ es la arista j, y la arista entre los vértices $v_{i,k-1}$ y $v_{i,0}$ es la arista $k-1$.

Para llenar las tablas, es necesario realizar el procedimiento MAXIMUM-2-PACK-TREE (ver Pseudocódigo 4 del Apéndice A), el cual calcula el conjunto 2-packing para un árbol. Si se omite la existencia de una de las aristas, el ciclo se puede considerar como un árbol, y es posible utilizar el procedimiento mencionado. Sin embargo, este procedimiento puede encontrar una solución inválida para el ciclo, ya que al reconsiderar la arista omitida puede existir un par de vértices que se encuentren a distancia menor a tres. Lo anterior se arregla con otro procedimiento llamado CORRECT-CONFLICT (ver Pseudocódigo 9 del Apéndice A), el cual se detalla más adelante. Entonces, aplicar este método a cada una de las aristas, y de todas las soluciones encontradas se conserva la mejor, encontrando así la mejor solución al ciclo.

2.6.1. Maximum-2-Pack-Tree

Sea h la arista que se omite durante este proceso. Al quitar esta arista, se puede ver al ciclo como un árbol con dos ramas, donde la raíz es el vértice 0, y la rama "izquierda" comprende desde el vértice 1 al vértice h, y la rama "derecha" desde el vértice $k-1$ hasta el vértice $h+1$. Las excepciones son cuando $h = 0$, donde solamente existe la rama derecha, y $h = k-1$ donde solamente existe la rama izquierda. Enfoque su atención en el caso $0 < h < k-1$, donde existen dos ramas.

Cada rama se procesa por separado de manera ascendente, comenzando por el final de cada rama y terminando con la raíz. Al final de cada rama se encuentra una *hoja*,

un vértice sin descendientes dentro del mismo bloque. Esta hoja se puede considerar de la misma manera que al vértice inferior de un cliqué, por lo que calcular la columna 1 se realiza de manera similar, copiando la columna 0. Los demás vértices de las ramas, llamados vértices internos, se pueden considerar como los vértices superiores de un cliqué, por lo que la columna 1 también se calcula de forma parecida: utilizando su propia columna 0 y la columna 1 de su descendiente en el bloque. Cabe resaltar que si el vértice interno i se encuentra en la rama izquierda, su descendiente es $i + 1$, y es $i - 1$ si se encuentra en la rama derecha.

Finalmente, el vértice raíz calcula la mejor solución combinando ambas ramas y su propia columna 0. Ahora, si se reconsideran los casos $h = 0$ y $h = k - 1$ donde sólo hay una rama, la raíz simplemente combina la rama existente y su columna 0.

2.6.2. Adapt-For-Unicycle

El procedimiento ADAPT-FOR-UNICYCLE (ver Pseudocódigo 8 del Apéndice A) verifica si la solución encontrada por la raíz contiene vértices a distancia menor a tres entre ellos al considerar la arista h. Existen cinco tipos de conflictos, los cuales se muestran en la Figura 6.

Si existe alguno de estos casos, se utiliza la siguiente rutina, CORRECT-CONFLICT.

2.6.3. Correct-Conflict

En el procedimiento CORRECT-CONFLICT (ver Pseudocódigo 9 del Apéndice A) ya se tienen identificados los vértices que se encuentran en conflicto en ambas ramas. La forma de resolver el conflicto es analizando dos casos: en el primer caso *removiendo* de la solución al vértice de la rama izquierda que causa conflicto, y recalculando el mejor árbol, es decir, aplicando el procedimiento MAXIMUM-2-PACK-TREE; para el segundo caso se realiza el mismo procedimiento pero con el vértice conflictivo de la rama derecha. Esto genera dos nuevas soluciones, que pueden volver a tener conflicto. Si alguno o ambos tienen conflicto, se repite este proceso recursivamente hasta que no exista conflicto. Una

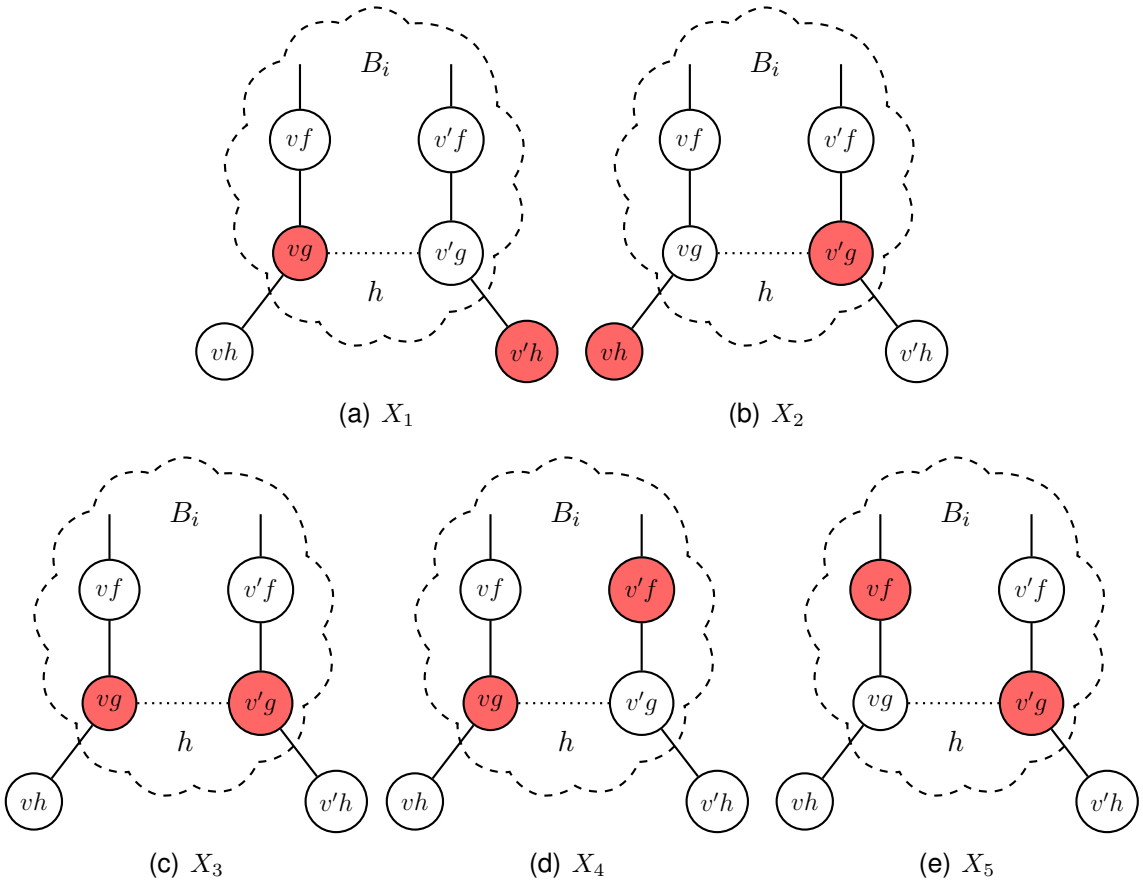
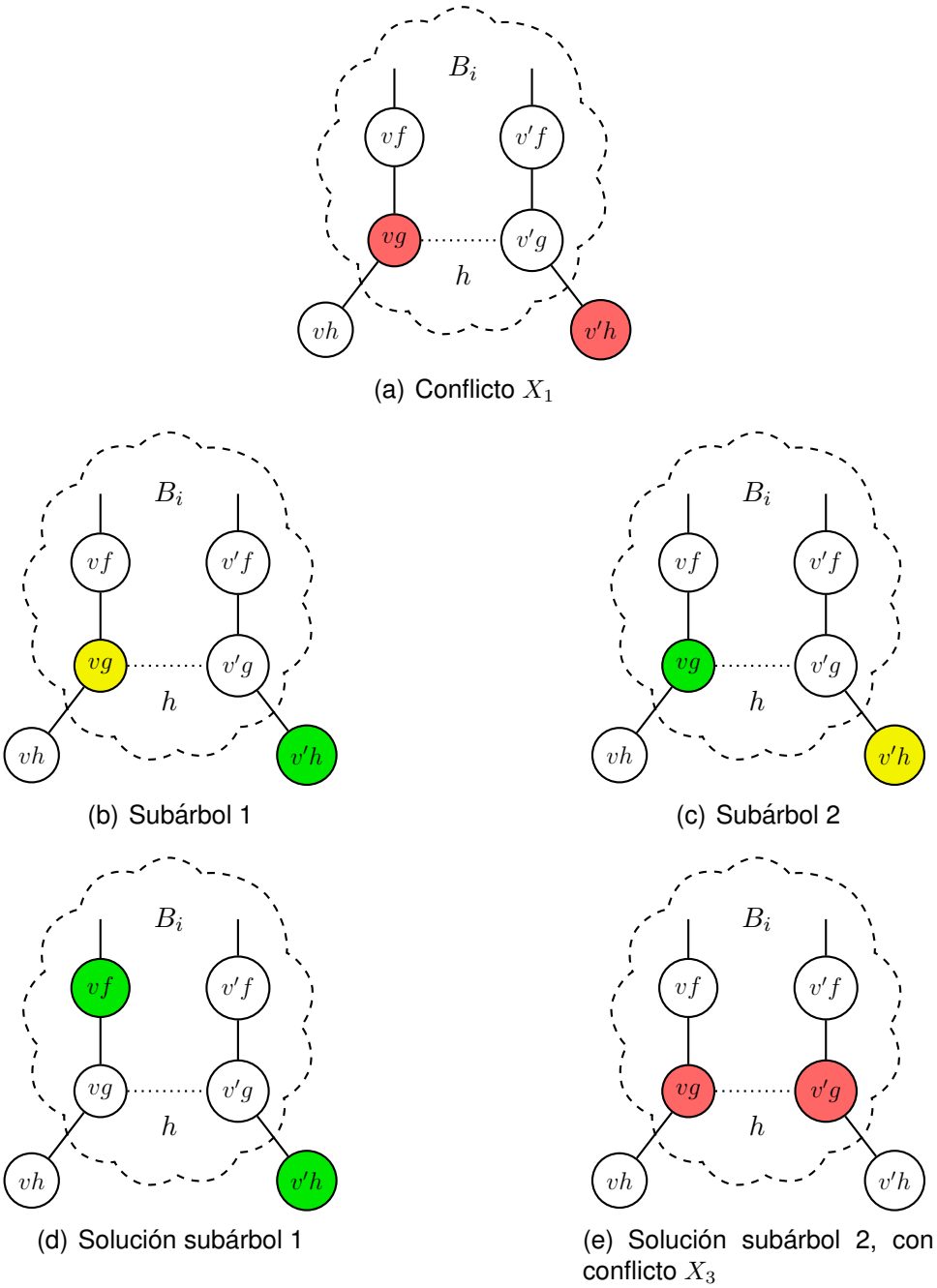


Figura 6: Los 5 diferentes conflictos, según el algoritmo $\mathcal{F}$.

vez que ambos casos se hayan resuelto, la mejor solución de ambas ramas se propone como solución a la recursión anterior. La Figura 7 muestra un ejemplo de este proceso.

En la Figura 7(a) se muestra uno de los conflictos, X_1. En la Figura 7(b) está el primer caso, que remueve el vértice conflictivo de la rama izquierda (en amarillo), y en la Figura 7(c), el segundo caso de manera similar. En la Figura 7(d) se muestra la solución del primer caso, mientras que en la Figura 7(e) la del segundo caso. Este último vuelve a tener conflicto (X_3), por lo que se debe repetir el proceso para dicha solución. Aquí se puede notar que al desactivar un vértice y aplicar nuevamente el algoritmo, la forma natural de reemplazar el vértice removido es utilizar al vértice padre como parte de la solución, o visto de otra manera, se “recorre” la solución un vértice hacia arriba, siempre y cuando esa sea la mejor opción disponible.



(a) Conflicto X_1

(b) Subárbol 1

(c) Subárbol 2

(d) Solución subárbol 1

(e) Solución subárbol 2, con conflicto X_3

Figura 7: Ejemplo de corrección de conflicto.

En (Flores-Lamas *et al.*, 2018) se muestra que solamente existen nueve estados posibles a los que pueden transitar los conflictos, y que la cantidad de recurrencias es a lo más tres, por lo que un conflicto se resuelve en $O(1)$ de recurrencias. También se demuestra que al aplicar este procedimiento a los bloques ciclos, se obtiene una solución equivalente en un cactus.

2.6.4. Solución del ciclo

Una vez aplicado el procedimiento MAXIMUM-2-PACK-TREE a cada una de las aristas del ciclo, así como sus respectivas correcciones de conflicto, se tienen k soluciones distintas almacenadas en la raíz del ciclo, cada una con cuatro renglones. Para cada uno de los renglones r, se calcula la mejor solución de todas y se guarda en la columna 2 del renglón r, como la mejor solución hasta el ciclo actual, a distancia r de la raíz.

Esta columna 2 es la que se utiliza en el procedimiento de inicialización de tablas del padre del ciclo, en caso de existir.

2.7 Solución global

Al terminar de procesar todos los bloques, el vértice raíz del último bloque o bloque raíz contiene en su columna 2 del renglón 0 la mejor solución encontrada en el cactus. Esta solución representa un conjunto 2-packing máximo en este grafo. La última rutina del algoritmo $\mathcal{F}$ es distribuir la solución, que le indica a todos los vértices su estado dentro de la mejor solución, ésto es, si son parte del conjunto 2-packing o no.

2.8 Contribución al algoritmo

Al algoritmo $\mathcal{F}$ se le hicieron algunas modificaciones, algunas de ellas como correcciones que necesitaba para funcionar correctamente, y otras como adaptaciones para resolver el problema del conjunto 2-packing con peso máximo. Las adaptaciones se mencionan a continuación:

2.8.1. Adaptaciones

- El Pseudocódigo 2 (ver apéndice A), INITIALIZE-BLOCK-TABLES, inicializa las tablas de los bloques a una configuración predeterminada, dependiendo de los bloques descendientes a cada vértice. Si el grafo de entrada tiene asignados pesos a los vértices, se puede cambiar en la línea 19 al 1 por el peso del vértice $v_{i,j}$, por lo que el algoritmo es capaz de resolver el problema del 2-packing con peso máximo.

- El campo *conjunto* de cada una de las celdas de las tablas contiene una lista de los vértices que se encuentran en la solución para dicha celda. Esta lista se transmite de una celda a otra, por lo que realizar operaciones con las celdas puede ser costoso, y además utilizar memoria adicional. Una alternativa es tener un número constante de apuntadores para cada celda, que indiquen las celdas de las cuales se obtuvo la solución, así como una variable booleana que indique si el vértice actual se encuentra contenido dentro de la solución. Esto permite reducir el orden del espacio de almacenamiento de $O(n)$ a $O(1)$. Para obtener la solución actual es necesario recorrer todos los vértices a través de los apuntadores, pero esta operación solamente se realiza en la rutina ADAPT-FOR-UNICYCLE, y los únicos vértices que se necesitan son los que se encuentran dentro del ciclo, por lo que el número de operaciones es $O(k)$, donde k es la cantidad de vértices del ciclo. Este trabajo adicional no afecta el tiempo de ejecución, ya que esta rutina se ejecuta una vez por cada ejecución de MAXIMUM-2-PACK-TREE, el cual también es $O(k)$.

2.9 Implementación

Al algoritmo $\mathcal{F}$ con las adaptaciones anteriores se le denomina $\mathcal{G}$. Este último se implementó en el lenguaje de programación Java. Se utilizó como base una versión mínima construida por el autor, de la cual se rescataron las estructuras para almacenar los grafos, los vértices y las aristas, así como los métodos de lectura de archivos y el formato de almacenamiento de archivos de entrada y soluciones. También se incluyó un método para resolver el problema del conjunto 2-packing máximo con búsqueda exhaustiva, es decir, revisando todas las soluciones posibles y rescatando la mejor de ellas. Este procedimiento realiza $O(2^n \cdot n^3)$ operaciones donde n es la cantidad de vértices del grafo, por lo que sólo es adecuado para valores pequeños de n.

Se implementaron desde cero las clases de Bloque, y Creación de árbol de bloques, los cuales contienen todas las instrucciones utilizadas dentro del algoritmo, como la inicialización de tablas, el procesamiento de bloques cliqué, el procesamiento de los bloques ciclo, así como las subrutinas internas que este último necesita. El número aproximado de líneas de código escritas fue de 2000.

2.10 Pruebas

Se realizaron pruebas para verificar que el algoritmo funcionara de manera correcta. Para ello, se programó un algoritmo generador de cactus aleatorio, que crea los grafos pegando "pedazos" de grafo de la siguiente manera:

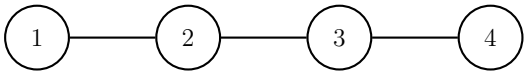
- Se escoge n, la cantidad de vértices que tendrá el cactus. Se crea un arreglo A de n elementos, numerados de 1 a n.
- Se genera un arreglo W de n enteros positivos, los pesos de los vértices. El elemento en la posición i corresponde al peso del vértice i, suponiendo que los vértices se numeran de 1 a n. Cada uno de los pesos se genera aleatoriamente, con distribución uniforme e independiente de los demás, en un intervalo entre 1 y 20.
- Se particiona el arreglo A en k distintos segmentos, cada segmento con al menos un elemento. Cada segmento representa a un "pedazo". Para cada pedazo se realizan las siguientes operaciones, en el orden de izquierda a derecha dentro del arreglo:
    - A cada pedazo se le asigna un bit aleatoriamente y con la misma probabilidad. Si el bit es 0, entonces el pedazo es un ciclo del tamaño del segmento más uno, de lo contrario es una cadena del tamaño del segmento. La excepción a esta operación es cuando el pedazo solamente contiene un elemento, y sólo puede ser una cadena, ya que no puede formar un ciclo.
    - Si el pedazo no es el primero, entonces se asigna i como el identificador del primer vértice del pedazo. Se elige arbitrariamente un entero k en el intervalo cerrado $[1, i-1]$. El entero k indica el identificador del vértice al que se adhiere el pedazo actual.

En la Figura 8 se muestra un ejemplo de un caso. El arreglo A de 10 elementos se divide en tres segmentos, el primero con los primeros cuatro vértices, el segundo con los siguientes tres, y el tercero con los últimos tres. Al primer segmento se le asigna el bit 1, por lo que se forma una cadena de cuatro elementos (Figura 8(b)). Al segundo segmento se le asigna el bit 0, por lo que forma un ciclo de tamaño cuatro. El entero k toma un

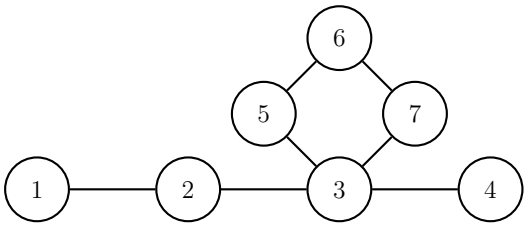
valor entre 1 e $i-1=4$, en este caso $k=3$, por lo que el ciclo se adjunta al vértice 3 (Figura 8(c)). Note que el ciclo contiene a todos los vértices del segmento actual y el vértice al que se adhieren. Por último, el tercer segmento obtiene un bit 1, por lo que es una cadena de tres vértices que se adjunta al vértice 5 (Figura 8(d)).

W	$[7,1,3,2,1,9,10,6,4,10]$		
A	$[1,2,3,4]$	$[5,6,7]$	$[8,9,10]$
Bit	1	0	1
i	-	5	8
k	-	3	5

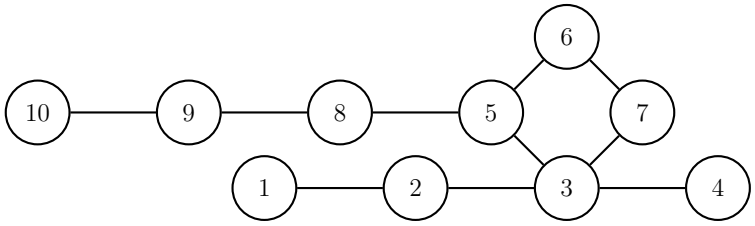
(a) Variables y valores asignados



(b) Primer pedazo



(c) Segundo pedazo



(d) Tercer pedazo

Figura 8: Ejemplo de la creación de un cactus aleatorio.

Se crearon 10 cactus para cada valor de n entre 8 y 24 vértices, para un total de 170 grafos distintos. A estos cactus se les calculó el conjunto 2-packing con peso máximo con el algoritmo de búsqueda exhaustiva para encontrar una solución correcta. A estos mismos se les calculó su conjunto 2-packing con peso máximo con el algoritmo $\mathcal{G}$, para verificar que las soluciones fueran equivalentes. Gracias a ello, se pudieron encontrar algunos errores de diseño en el algoritmo original, así como algunos errores de programación. Estas correcciones se muestran a continuación.

2.10.1. Correcciones

- En el Pseudocódigo 8, que contiene la rutina ADAPT-FOR-UNICYCLE, se revisa que la solución encontrada en el renglón 0 de la raíz esté libre de conflicto, pero esto se debe realizar para todos los renglones, comenzando desde el renglón 3. Por lo tanto, la rutina completa debe estar en un ciclo desde $r = 3 \rightarrow 0$.
- En el Pseudocódigo 9, CORRECT-CONFLICT, se remueve cada uno de los dos vértices conflictivos, uno a la vez, y la mejor solución de ambos se regresa. Si el vértice conflictivo no está dentro del árbol, es decir, es vh o $v'h$ (conflictos X_1 y X_2), primero se debe remover del renglón 1 y se compara con el renglón 2, y después se realiza la misma operación pero con el renglón 0, comparando con el renglón 1. Esta operación asegura que se remueva el vértice conflictivo y agregue su padre, como forma de *recorrer* la solución una unidad.

Se encontraron otros detalles en el algoritmo, pero éstos se discutieron con el autor y se corregieron directamente en el artículo original, por lo que ya no se muestran en este trabajo.

Capítulo 3. Algoritmo auto-estabilizante

En este capítulo se detalla el algoritmo auto-estabilizante $\mathcal{Z}$ propuesto para encontrar un conjunto 2-packing con peso máximo en uniciclos. Un uniciclo es un grafo conectado con n aristas, donde n es la cantidad de vértices, es decir, es un árbol con una arista adicional que forma exactamente un ciclo. La entrada al algoritmo $\mathcal{Z}$ es un uniciclo $G = (V, E)$, con las características mencionadas.

El algoritmo $\mathcal{Z}$ se basa en el algoritmo $\mathcal{G}$. El algoritmo $\mathcal{Z}$ se diseñó para simular los pasos que se siguen en $\mathcal{G}$, por lo que se necesitan hacer algunas adaptaciones en el enfoque auto-estabilizante. La estrategia general que sigue $\mathcal{Z}$ es la siguiente:

- Se elige un vértice líder y se crea un árbol de expansión con raíz en el líder.
- Se identifican las aristas puente del grafo, para saber si una arista es parte de un ciclo o forma un cliqué.
- Se numeran los vértices de ciclo de manera similar a $\mathcal{G}$.
- A partir del árbol enraizado, se calculan las tablas de los vértices de la siguiente manera:
    - Si el vértice es una hoja, es un vértice inferior en un cliqué, y se calcula la tabla del bloque al que pertenece.
    - Si el vértice no es hoja y tiene hijos, es un vértice superior de un conjunto de cliqués, y se calculan las tablas de dichos bloques.
        - Si dicho vértice tiene padre con arista puente, entonces también es vértice inferior de otro cliqué, y se calcula la tabla de dicho bloque.
    - Si el vértice es parte de un ciclo, se calculan las tablas de los diferentes árboles que se obtienen al remover cada una de las aristas del ciclo.
- Se distribuye la solución.

3.1 Elección de líder

La elección de líder y la creación del árbol de expansión se realiza utilizando el algoritmo de elección de líder de Datta *et al.* (2011b). Cada vértice tiene una variable *líder*, que indica al posible vértice raíz, y ésta se actualiza a lo largo del algoritmo utilizando las reglas propuestas. Al principio, cada vértice actualiza su variable líder con el identificador del vértice con menor identificador en todo su vecindario cerrado, que podría ser él mismo. Este proceso se repite progresivamente con algunas variantes, y al final del algoritmo, todos los vértices terminan con el mismo líder.

Este algoritmo de elección de líder tiene como entrada un grafo arbitrario conectado, y la salida es un árbol enraizado, donde cada vértice v conoce el identificador de la raíz, además de tener un apuntador hacia su padre en dicho árbol ($v.parent$). Como rutina adicional al algoritmo, se realiza un paso al final que permite a cada vértice tener una lista de sus vértices hijo. La cantidad de movimientos realizados por el algoritmo son $O(n^2)$ y la cantidad de rondas es $O(n)$, donde n es la cantidad de vértices del grafo.

Las reglas que utiliza este algoritmo tienen prioridad 1, lo que implica que si un vértice se encuentra habilitado por alguna regla de este algoritmo y alguna otra regla de otro algoritmo, ejecutará la primera. Esto evita que se ejecute el algoritmo posterior con información errónea.

3.2 Identificación de puentes

El algoritmo utilizado para identificar puentes es el de Karaata y Chaudhuri (1999), cuya estrategia general es utilizar las aristas que no forman parte del árbol de expansión, ya que éstas forzosamente forman ciclos. Al encontrar dos vértices unidos por dichas aristas, se propaga la información de ambos hacia sus padres mediante sus conjuntos S, donde se contienen los vértices en el ciclo, de tal manera que la misma información coincide en el ancestro común más bajo de ambos. Así, al final del algoritmo, dos vértices u y v se conectan por una arista puente si y solo si uno es padre del otro, y sus conjuntos S no comparten elementos.

Este algoritmo de identificación de puentes tiene como entrada un grafo con un árbol de expansión enraizado en algún vértice r, y como salida, para cada vértice, un conjunto S. Aunque los puentes no se identifican directamente, es posible realizar un post procesamiento que indique si alguna arista es puente o no sin realizar las operaciones anteriores. Si algún vértice contiene al menos una arista incidente que no es puente, entonces forma parte del ciclo, y se asigna una variable booleana en el vértice llamada $cycle\_vertex$ con valor verdadero, de lo contrario se le da un valor falso. También se supone que existe una función llamada $is\_bridge(u, v)$, que regresa un valor verdadero si la arista del vértice u al vértice v es un puente, y falso de lo contrario. Una vez realizado este procedimiento, se tiene a $v.children$ como una lista de vértices hijos de v tal que la arista que los conecta es un puente, así como una lista $v.cycle\_neighbors$ con los ids de los vértices con arista puente hacia v. La cantidad de movimientos requeridos es $O(dn)$, y la cantidad de rondas que necesita el algoritmo es $O(d)$, donde d es el diámetro del componente biconectado más grande del grafo. En este caso $d = O(n)$.

Las reglas de este algoritmo tienen prioridad 2, por lo que se ejecutan siempre y cuando no haya reglas habilitadas al mismo tiempo del algoritmo de elección de líder.

3.3 Maximum-2-Packing-Set $\mathcal{Z}$

3.3.1. Variables

Para la ejecución del algoritmo $\mathcal{Z}$, se necesitan algunas variables para cada vértice v, las cuales se describen a continuación:

- $v.error$: Variable booleana que indica si existe algún error en las tablas de v, o en alguno de los hijos de v.
- $v.children$: Arreglo de tamaño r con apuntadores hacia los r hijos de v con arista puente.
- $v.table\_children\_\{id\}$: Tabla auxiliar del vértice superior del cliqué conformado por los vértices v y id, donde id es el identificador de uno de sus hijos. El vértice v tiene r tablas de este tipo (la cardinalidad de $v.children$).

- $v.last\_child\_completed$: Entero que indica el índice de la última tabla $v.table\_children\_id$ llenada por v.
- $v.table\_up$: Tabla auxiliar del vértice inferior del cliqué formado por los vértices v y $v.parent$.
- $v.completed$: Variable booleana que indica si el vértice v ya está procesado, es decir, ya tiene la solución de acuerdo a los vértices adyacentes.
- $v.solution$: Apuntador a la celda de la solución contemplada por el algoritmo.

Nota: los vértices pueden ser vértice superior de varios cliqués y vértice inferior de otro cliqué distinto. La Figura 9 ilustra cinco vértices, tres de ellos hoja (c, d, e). El vértice b es vértice superior de tres cliqués distintos (b-c, b-d, b-e), y vértice inferior de otro cliqué (a-b), donde el vértice a es vértice superior.

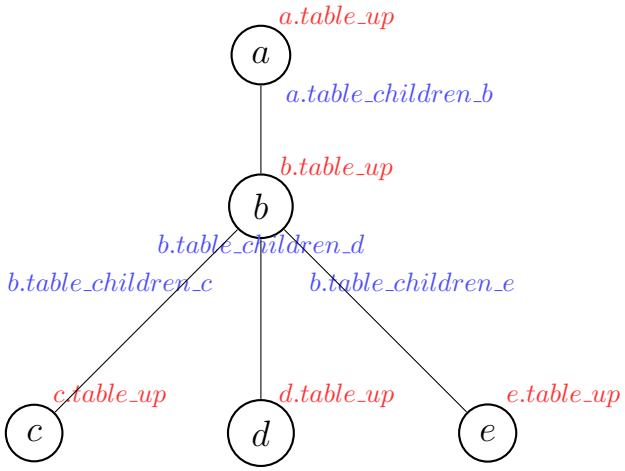


Figura 9: Ejemplo de las tablas de cliqués.

Las siguientes variables son de utilidad sólo para los vértices que pertenezcan a algún ciclo:

- $v.cycle\_neighbors$: Arreglo de tamaño x con apuntadores hacia los x vecinos de ciclo de v, o vecinos con arista no puente. En el caso para los vértices del ciclo, $x = 2$, ya que el uniciclo solamente tiene un ciclo, cada vértice en él tiene dos vecinos de ciclo, y por consecuencia dos aristas no puente.

- $v.cycle\_error\_\{h\}$: Variable booleana que indica si existe algún error en la tabla del ciclo en la recursión h. Nota: el valor de h es $O(n)$.
- $v.cycle\_number$: Entero que indica la numeración, a nivel del bloque, del vértice v dentro del ciclo al que pertenece.
- $v.numeration\_error$: Variable booleana que indica si existe algún error en la numeración del vértice v en el ciclo.
- $v.cycle\_successor$: Apuntador hacia el sucesor de v en el ciclo.
- $v.cycle\_antecessor$: Apuntador hacia el antecesor de v en el ciclo.
- $v.current\_tree$: Entero que indica la arista h que se está removiendo actualmente en el ciclo para crear el árbol de expansión durante la ejecución del procedimiento MAXIMUM-2-PACKING-TREE.
- $v.current\_h$: Expresión compuesta para indicar la recurrencia que se está calculando al corregir los conflictos del árbol de expansión durante el procedimiento ADAPT-FOR-UNICYCLE.
- $v.table\_cycle\_\{h\}$: Tabla auxiliar del vértice v en el árbol de expansión en la recurrencia h de corrección de conflictos.
- $v.\{h\}\_vg$, $v.\{h\}\_vf$, $v.\{h\}\_vh$: Identificadores de los vértices vg, vf y vh de la recurrencia h de corrección de conflictos, como indica la Figura 6.
- $v.\{h\}\_v'g$, $v.\{h\}\_v'f$, $v.\{h\}\_v'h$: Identificadores de los vértices $v'g$, $v'f$ y $v'h$ de la recurrencia h de corrección de conflictos, como indica la Figura 6.
- $v.\{h\}\_conflict$: Entero del 0 al 5 que indica el identificador del conflicto que se presentó en la recurrencia h de corrección de conflicto. El valor 0 indica sin conflicto, del 1 al 5 son los mismos presentados en la Figura 6.

3.3.2. Funciones

A continuación se presentan algunas funciones que permiten simplificar las fórmulas o expresiones booleanas en las reglas del algoritmo. La descripción completa de cada

una de ellas se encuentra en el Apéndice B.

- $error(v)$: Función que devuelve el valor verdadero si un vértice v se encuentra en estado de error; falso de lo contrario.
- $cycle\_error(v, h)$: Función que devuelve el valor verdadero si el vértice v del árbol h de ciclo se encuentra en estado de error.
- $is\_cycle\_root(v)$: Función que devuelve el valor verdadero si el vértice v es raíz del ciclo; falso de lo contrario.
- $is\_leaf(v)$: Función que devuelve el valor verdadero si el vértice v es una hoja en el árbol de expansión; falso de lo contrario.
- $table\_completed(v.table)$: devuelve el valor verdadero si una tabla del vértice v se encuentra llenada correctamente, de acuerdo a las tablas de los vértices de los que depende dicha tabla; falso de otra manera. Las instrucciones de esta función se omiten por la cantidad de condiciones con las que cuenta, ya que para cada tipo de tabla es una condición distinta.
- $table\_enabled(v)$: Función que devuelve el valor verdadero si el vértice v puede llenar alguna tabla de vértice superior de cliqué.
- $children\_blocks\_completed(v)$: Función que devuelve el valor verdadero si todos los bloques hijo de v ya terminaron de calcular sus tablas; falso de otra manera.
- $correct\_numeration(v)$: Función que devuelve el valor verdadero si el vértice v se encuentra numerado correctamente en el ciclo; falso de otra manera.
- $antecessor\_correct(v)$: Función que devuelve el valor verdadero si el vértice v tiene un antecesor en el ciclo correctamente numerado; falso de otra manera.
- $conflictive\_vertices\_correct(v, h)$: Función que regresa el valor verdadero si para el vértice v se encuentran correctamente indicados los identificadores de los vértices $vf, vg, vh, v'f, v'g, v'h$, para la recursión h actual; falso de otra manera.

- $check\_conflict\_left(v, h)$: Función que revisa si existe conflicto entre alguno de los vértices vf, vg y vh con alguno de los vértices $v'f$, $v'g$ y $v'h$ del vértice adyacente al otro lado de la arista removida h. Regresa un entero del 1 al 5 si existe algún error, conforme al algoritmo $\mathcal{G}$; regresa 0 en caso de que no exista error.

- $check\_conflict\_right(v, h)$: Función que revisa si existe conflicto entre alguno de los vértices $v'f$, $v'g$ y $v'h$ con alguno de los vértices vf, vg y vh del vértice adyacente al otro lado de la arista removida h. Regresa un entero del 1 al 5 si existe algún error, conforme al algoritmo $\mathcal{G}$; regresa 0 en caso de que no exista error.

- $vertex\_conflict(v, h)$: Función que regresa el número del conflicto, dependiendo de la ubicación de v en el árbol.

- $special\_case(v, h, conflict)$: Función que revisa si existe un caso especial al corregir el conflicto, en cuyo caso devuelve el valor verdadero; falso de lo contrario.

- $has\_to\_deactivate(v, h, conflict)$: Función que devuelve el valor verdadero si el vértice v debe desactivar algún vértice dependiendo del conflicto y la h actual; falso de lo contrario.

- $vertex\_deactivated(v, h, conflict)$: Función que devuelve el valor verdadero si el vértice v ya se desactivó en el conflicto y h actual; falso de lo contrario.

- $conflict\_correct(v)$: Función que devuelve el valor verdadero si el vértice v se encuentra habilitado para calcular la tabla de su conflicto correspondiente; falso de lo contrario.

3.3.3. Reglas

El algoritmo $\mathcal{Z}$ contiene 13 reglas en total. A continuación se mencionan las reglas y una breve descripción. Las guardias y las acciones de las reglas se encuentran en el Apéndice C. Estas reglas tienen prioridad 3, por lo que solamente se ejecutan en el vértice v si v no se encuentra habilitado por reglas de elección de líder ni de identificación de puentes.

Regla E. Regla de error: permite detectar errores de las tablas y de numeración de los vértices de ciclo, y restablece variables para su corrección. Considera los casos para errores en el ciclo, que propaga el error en el ciclo en caso de que exista; el caso de errores en las tablas, que propaga el error en alguno de los hijos del vértice v; y error en la numeración, en caso de que el vértice no se encuentre correctamente numerado.

Regla PC. Regla para procesar cliqué: calcula las tablas de los vértices inferior y superior de cliqués. Considera los casos cuando el vértice es una hoja, donde se calcula la tabla $table\_up$; cuando el vértice es un vértice inferior de cliqué y no es hoja, donde se calcula la tabla $table\_up$; y cuando es vértice superior de cliqué, donde se calculan las tablas $table\_children\_\{id\}$.

Regla NC. Regla de numeración de ciclo: asigna la numeración a los vértices que forman parte del ciclo. Considera el caso cuando el vértice es la raíz del ciclo, donde se le asigna el número 0 y se asignan el sucesor y antecesor del ciclo; y el caso de los demás vértices del ciclo, donde se asigna la numeración correspondiente, el sucesor y antecesor del ciclo.

Regla IC. Regla de inicialización de ciclo: calcula la primera columna de los vértices de ciclo, de forma similar a los vértices de cliqués.

Regla M2PT. Regla del MAXIMUM-2-PACKING-TREE: calcula la columna 1 de los vértices de ciclo, para los distintos árboles de expansión al omitir una arista. Considera los casos cuando v es la hoja de la rama izquierda (cuando $h = v.cycle\_number$); cuando es la hoja de la rama derecha ($h = v.cycle\_number - 1$); cuando es un vértice interno de la rama izquierda ($h > v.cycle\_number$); y cuando es vértice interno de la rama derecha ($h < v.cycle\_number - 1$)

Regla PR. Regla de procesamiento de raíz: combina las soluciones de ambas ramas del árbol de expansión de la rutina MAXIMUM-2-PACKING-TREE y calcula la columna 1 del vértice raíz del ciclo.

Regla AU. Regla de adaptación de uniciclo: revisa si la solución encontrada en la regla anterior contiene conflicto o no. Considera los casos cuando el vértice se encuentra en la

rama izquierda y cuando se encuentra en la rama derecha. En ambos casos se propaga la solución encontrada en la raíz hacia las hojas para revisar si existe conflicto.

Regla AVC. Regla de asignación de vértices conflictivos: asigna los vértices conflictivos vf, vg, vh, $v'f$, $v'g$ y $v'h$ correspondientes a h.

Regla AC. Regla de asignación de conflicto: asigna el conflicto correspondiente a las hojas de las ramas. Considera los casos para la hoja izquierda y para la hoja derecha, de acuerdo a la Figura 6.

Regla DV. Regla de desactivación de vértice: desactiva uno de los vértices conflictivos para encontrar una solución al conflicto.

Regla CC. Regla de corrección de conflicto: calcula las columnas 1 de los vértices de ambas ramas de manera similar a las regla $M2PT$, pero revisando los conflictos. Considera los casos cuando el vértice v es la hoja izquierda; cuando es la hoja derecha; cuando es un vértice interno de la rama izquierda; y cuando es un vértice interno de la rama derecha.

Regla PRC. Regla de procesamiento de raíz con conflicto: combina las soluciones de ambas ramas dependiendo del conflicto encontrado, de forma similar a la regla PR. Considera el caso cuando no existe conflicto; cuando existe un conflicto de caso general; cuando existe el conflicto especial 1; y cuando existe el conflicto especial 2 (ver sección 3.4.3.2).

Regla S. Regla de solución: propaga la solución encontrada por la raíz del árbol de expansión del grafo original a los demás vértices y les indica si son parte del conjunto 2-packing. Considera el caso cuando el vértice v es la raíz del árbol de expansión del grafo original y cuando no lo es.

3.4 Explicación del algoritmo $\mathcal{Z}$

El primer paso del algoritmo $\mathcal{G}$ consiste en identificar los bloques, construir un árbol de bloques con un recorrido en el grafo y numerarlos en post-orden. En el algoritmo $\mathcal{Z}$,

este procedimiento se omite, aprovechando las características del grafo obtenido a partir de los algoritmos de elección de líder y de identificación de puentes:

- Utilizando las aristas puente, se sabe si la arista es parte de un ciclo o de un cliqué. De esta forma todos los bloques (el ciclo y los cliqués) ya se encuentran definidos.
- La numeración de los bloques del grafo sirve para indicar el orden en que se procesan. La numeración ya está implícita, dependiendo de cómo se tengan almacenados los hijos dentro de un vértice. Por ejemplo, suponiendo que el vértice 7 tiene a sus hijos ordenados como $9, 1, 4$ en la Figura 10(a), el cliqué de los vértices 7-9 se tiene que procesar primero, después el cliqué 7-1, y después el cliqué 7-4, y por último el cliqué 5-7. En la Figura 10(b) se muestra la equivalencia del grafo anterior en un grafo de bloques.

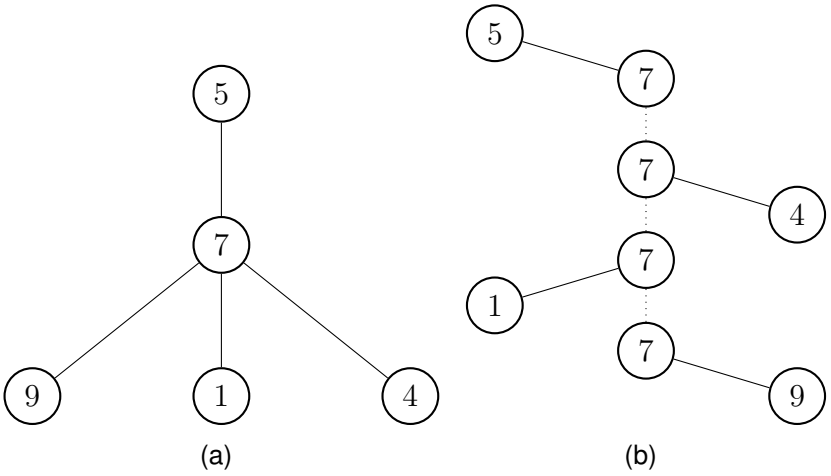
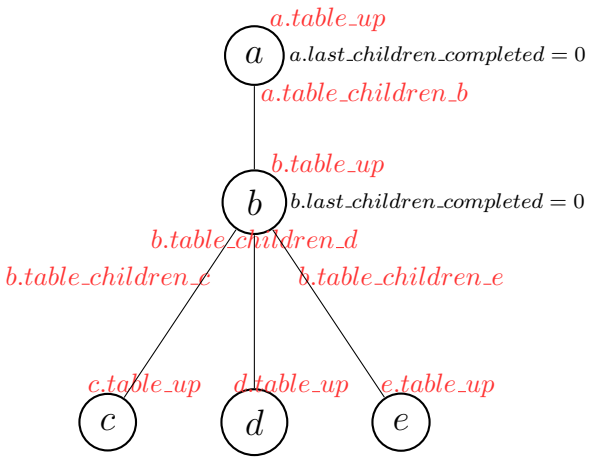


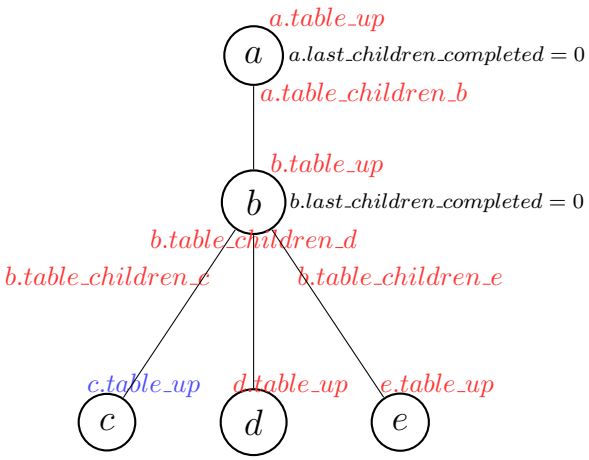
Figura 10: Ejemplo de orden de bloques.

3.4.1. Procesamiento de cliqués

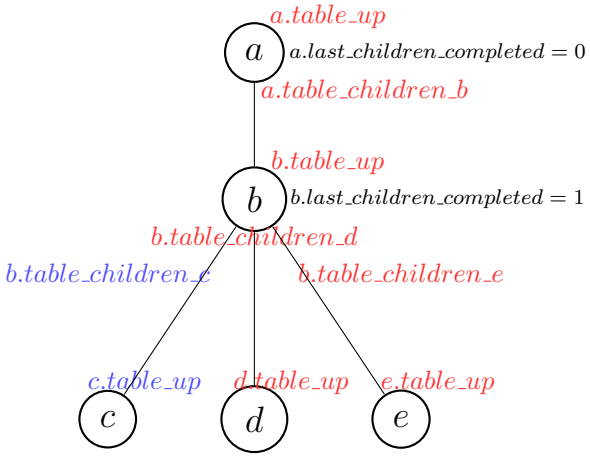
En el algoritmo $\mathcal{Z}$, se considera un cliqué al conjunto de dos vértices y una arista entre ellos, tal que la arista es un puente dentro del grafo. El procesamiento de las tablas de los vértices que forman parte de los cliqués se realiza en la regla PC. Recordar que los cliqués se forman por dos vértices: superior e inferior, siendo superior el que se encuentra más cercano a la raíz del árbol, y el restante inferior. Los vértices inferiores se dividen en dos casos: cuando son vértices hoja del árbol de expansión o cuando no lo son.


a.table_up
a
a.last_children_completed = 0
a.table_children_b
b.table_up
b
b.last_children_completed = 0
b.table_children_d
b.table_children_c
b.table_children_e
c.table_up
d.table_up
e.table_up
c
d
e

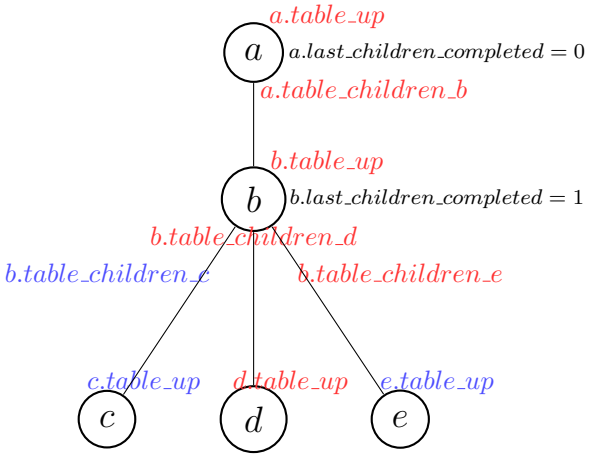
(a)


a.table_up
a
a.last_children_completed = 0
a.table_children_b
b.table_up
b
b.last_children_completed = 0
b.table_children_d
b.table_children_c
b.table_children_e
c.table_up
d.table_up
e.table_up
c
d
e

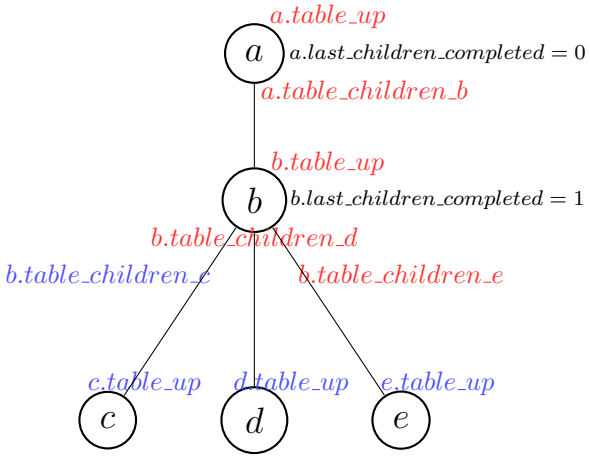
(b)


a.table_up
a
a.last_children_completed = 0
a.table_children_b
b.table_up
b
b.last_children_completed = 1
b.table_children_d
b.table_children_c
b.table_children_e
c.table_up
d.table_up
e.table_up
c
d
e

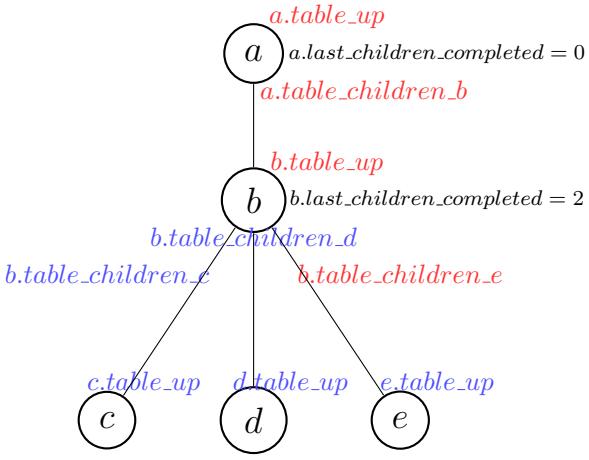
(c)


a.table_up
a
a.last_children_completed = 0
a.table_children_b
b.table_up
b
b.last_children_completed = 1
b.table_children_d
b.table_children_c
b.table_children_e
c.table_up
d.table_up
e.table_up
c
d
e

(d)


a.table_up
a
a.last_children_completed = 0
a.table_children_b
b.table_up
b
b.last_children_completed = 1
b.table_children_d
b.table_children_c
b.table_children_e
c.table_up
d.table_up
e.table_up
c
d
e

(e)


a.table_up
a
a.last_children_completed = 0
a.table_children_b
b.table_up
b
b.last_children_completed = 2
b.table_children_d
b.table_children_c
b.table_children_e
c.table_up
d.table_up
e.table_up
c
d
e

(f)

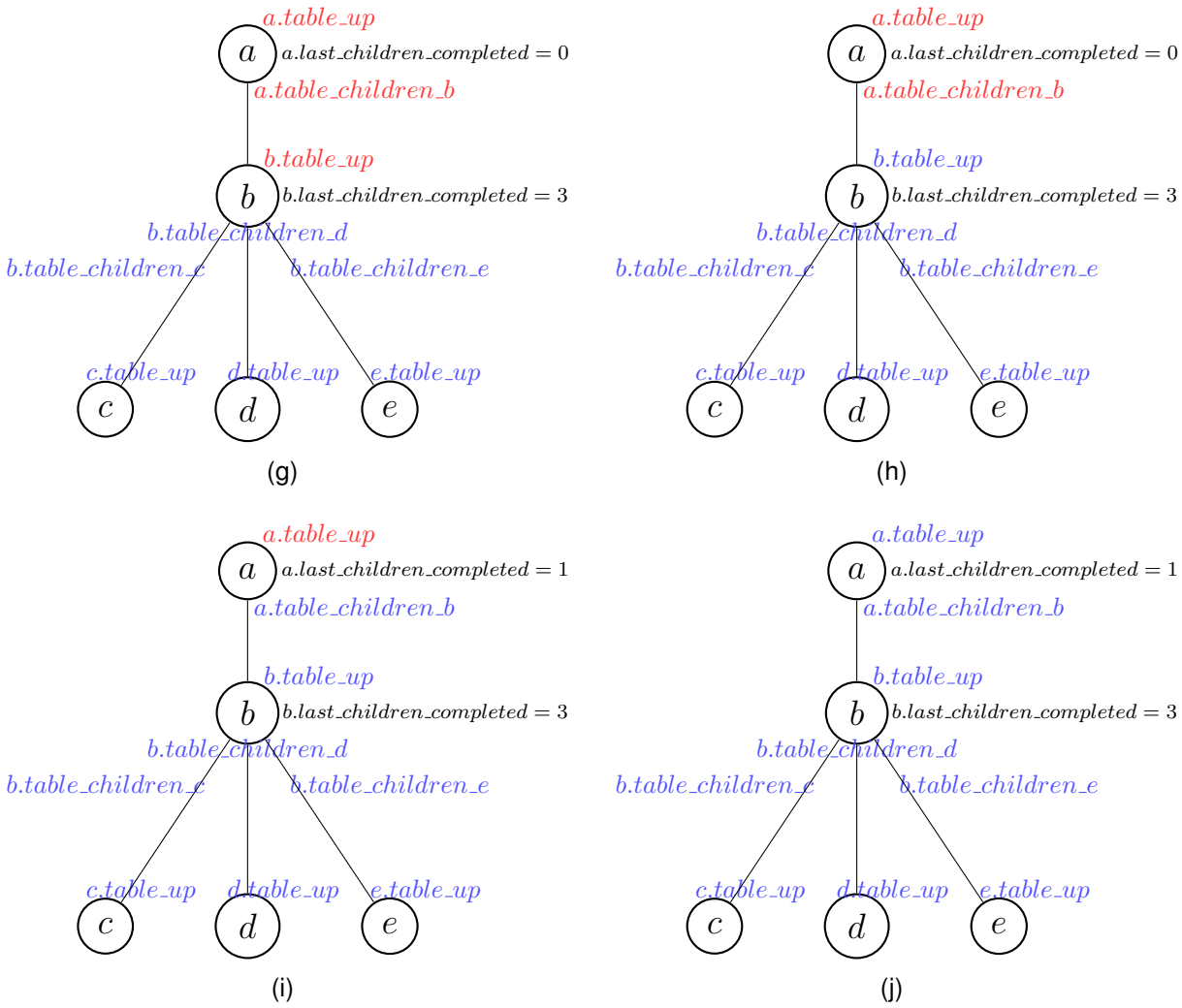


Figura 11: Ejemplo de procesamiento de cliqués.

La regla PC contempla tres casos: en el primer caso se encarga de las tablas de los vértices hoja, y se calculan todas las columnas con la configuración nula que se presenta en $\mathcal{G}$. El segundo caso calcula las tablas de los vértices inferiores no hoja, con instrucciones similares a las mostradas en el Pseudocódigo 2 (ver Apéndice A). En ambos casos, las tablas que se calculan son $v.table\_up$. El tercer caso calcula las tablas de los vértices superiores, una tabla a la vez, de todas las que forman parte de $v.table\_children\_\{id\}$. Utiliza la función $table\_enabled$, que permite saber si la última tabla ya se ha calculado para continuar con la siguiente.

En la Figura 11, el vértice b tiene tres hijos: c, d, e, en ese orden. De 11(a) a 11(b) se llena la tabla $c.table\_up$ con el primer caso de la regla PC. De 11(b) a 11(c) se llena la

tabla $b.table\_children\_c$ con el tercer caso de la regla PC, ya que su primer hijo ya llenó su tabla $table\_up$. De 11c) a 11(d) se llena $e.table\_up$ con el primer caso de la regla PC. La tabla $b.table\_children\_d$ no se puede llenar porque $d.table\_up$ aún no se ha llenado. De 11(d) a 11(e) se llena $d.table\_up$ con el primer caso de la regla PC. De 11(e) a 11(f) ya es posible llenar $b.table\_children\_d$, lo cual se hace con el tercer caso de la regla PC. De 11(f) a 11(g) se llena de manera similar $b.table\_children\_e$. En 11(h) ya están llenas todas las tablas de b como vértice superior de cliqué, por lo que se llena su tabla de vértice inferior en el cliqué a-b, la tabla $b.table\_up$, con el segundo caso de la regla PC. En 11(i) se llena la tabla $a.table\_children\_b$ con el tercer caso de la regla PC. Por último, en 11(j) se llena $a.table\_up$ con el segundo caso de la regla PC.

3.4.2. Numeración del ciclo

Los vértices que se encuentran en el ciclo deben estar numerados para poder calcular las tablas de los distintos árboles. Esta numeración se realiza de manera similar al mencionado en $\mathcal{G}$, asignando un entero no negativo a la variable $cycle\_number$ cada uno de los vértices, comenzando en cero para la raíz del ciclo, e incrementando hacia el lado del vecino del ciclo con el menor identificador, terminando con el otro vecino.

La raíz del ciclo se encuentra con la función $is\_cycle\_root(v)$. Un vértice es raíz del ciclo si pertenece a un ciclo, y la arista a su padre es un puente (es el vértice de mayor altura en el árbol) o si no tiene padre, en cuyo caso tal vértice también es la raíz del árbol de expansión del grafo original.

La numeración se realiza con la regla NC, que contempla dos casos. El primer caso numera al vértice raíz del ciclo, mientras que el segundo caso numera a los restantes. Estos dos casos utilizan la función $correct\_numeration$, que verifica que un vértice se encuentre correctamente numerado; el segundo caso utiliza una función adicional llamada $antecessor\_correct$, que indica si existe un único vértice vecino numerado correctamente, cuyo apuntador de sucesor sea hacia v.

3.4.3. Procesamiento del ciclo

Las columnas 0 de los vértices pertenecientes al ciclo se calculan usando la regla IC, la cual es muy parecida al segundo caso de la regla PC, pero adaptada para vértices de ciclo.

Como se menciona en $\mathcal{G}$, el algoritmo remueve cada arista del ciclo una a la vez, generando un árbol, y calculando la mejor solución de dicho árbol. Después se verifica que no existan conflictos al conectar el árbol, en cuyo caso se resuelve recorriendo los vértices. El algoritmo $\mathcal{Z}$ funciona de una manera similar. Los diferentes árboles se calculan de manera secuencial, es decir, primero cuando se remueve la primera arista, se corrigen los errores en caso de que existan, se calcula la mejor solución, y después prosigue a remover la segunda arista y repetir el proceso hasta la última arista.

3.4.3.1. Maximum 2-packing tree

Sea h el índice de la arista que se remueve, siendo 0 la arista entre la raíz del ciclo y el vértice con numeración 1 en el ciclo, la arista 1 entre los vértices 1 y 2, y así sucesivamente hasta terminar con la arista $k-1$ entre el último vértice y la raíz del ciclo, donde k es el número de vértices en el ciclo. Cuando se remueve la arista h, se llama *rama izquierda* a los vértices $0, 1, 2, ..., h$, con sus respectivas aristas intermedias, y *rama derecha* a los vértices $h+1, h+2, ..., k-1, 0$, también incluyendo las aristas entre dichos vértices.

El cálculo del árbol, al remover la arista h, se realiza con la regla $M2PT$, mientras que la regla PR procesa al vértice raíz para combinar las soluciones de ambas ramas.

La regla $M2PT$ contempla cuatro casos. El primer caso realiza el cálculo de los vértices internos de la rama izquierda; el segundo caso procesa el vértice hoja de la rama izquierda; el tercer caso procesa el vértice hoja de la rama derecha; el cuarto caso realiza el cálculo de los vértices internos de la rama derecha.

El segundo y tercer caso procesan a las hojas sin necesidad de la información de otro vértice, solamente de su propia columna 0. El primer y cuarto caso necesitan que los vértices antecesor y sucesor hayan calculado su columna 1, respectivamente.

Una vez que todos los vértices, excepto la raíz, hayan calculado su columna 1, es el turno del último vértice de procesar ambas ramas. Esta operación se realiza con la regla PR, calculando el mejor árbol con operaciones similares a las del Pseudocódigo 5 de $\mathcal{F}$.

En la Figura 12 se muestra el proceso para un ciclo con seis vértices y removiendo la arista $h = 3$. Para facilitar la explicación, los números dentro de los vértices indican la numeración $vertex\_number$ dentro del ciclo, no son los identificadores de los vértices. Se supone que la columna 0 de cada vértice ya se calculó, así como las tablas de todos los demás vértices descendientes de ellos. Un vértice coloreado verde indica que ya se calculó su tabla en la columna 1, un vértice amarillo indica que está habilitado para calcularse, y un vértice rojo indica que no está habilitado por ninguna regla del cálculo del mejor árbol y no se ha procesado. Cuando un vértice está amarillo, se indica la regla que lo habilita, con un número que indica el caso. El orden en que se procesan los vértices en este caso es: $4, 3, 2, 1, 5, 0$, pero es posible cualquier otro orden donde se procesen los vértices de las hojas hacia arriba.

3.4.3.2. Adapt for unicycle

Una vez calculado el mejor árbol, se debe asegurar que no exista conflicto al colocar la arista h de vuelta al ciclo. La mejor solución hasta el momento se encuentra en la raíz, por lo que se debe de distribuir a los vértices restantes del ciclo, para que los vértices incidentes a la arista h puedan identificar si existe conflicto. Como se menciona en $\mathcal{G}$, la rutina de adaptar para el uniciclo puede tener algunos niveles de recursión, pero la cantidad siempre es de orden constante $O(1)$ para cada arista h removida. En el caso del algoritmo $\mathcal{Z}$ es necesario tener una tabla en cada vértice por cada nivel de recursión con el fin de mantener la información requerida para calcular la solución. Dichas tablas utilizan una nomenclatura especial, la cual se explica a continuación:

- Las tablas del mejor árbol hasta el momento, propagadas por la raíz del ciclo hacia los demás vértices sin conocimiento de haber conflicto o no, se denotan con terminación "$\_rx$", donde x es un entero del 0 al 3. Por ejemplo, cuando se calcula $h = 3$ por primera vez y la raíz procesa el mejor árbol, se tienen que revisar los cuatro

renglones por errores. Comenzando por el renglón $r = 3$, se propaga dicha solución hacia los vértices como $current\_h = h3\_r3$. En otras palabras, las tablas que requieren revisar si existe conflicto se envían un renglón a la vez, de la raíz a los demás vértices del ciclo. Hay que notar que, por simplicidad, cuando se realiza una comparación de la numeración de un vértice con la variable $current\_h$, sólo se toma el número después de la h, por ejemplo, si $v.cycle\_number = v.current\_h$, donde $v.current\_h = h3\_r3$, se toma a $v.current\_h = 3$.

- Cuando se calculan los dos subárboles al existir conflicto, los vértices de la rama izquierda agregan a su tabla la terminación "$\_I$", mientras que los vértices de la rama derecha la terminación "$\_D$". Por ejemplo, una vez propagada la tabla $h3\_r3$ a los vértices del ciclo y se encuentra un conflicto, la hoja de la rama izquierda (vértice 3) comienza a calcular la rama sin el vértice conflictivo, y le pone el nombre $h3\_r3\_I$. Esta rama continúa calculando dicha tabla hasta llegar al vértice 1. De igual manera, la hoja derecha calcula $h3\_r3\_D$, y se sigue calculando a lo largo de la rama derecha hasta llegar al último vértice antes de la raíz.

En resumen, si el flujo del algoritmo es de raíz a hojas, la terminación es $\_rx$, y si es de hoja a raíz, es $\_D$ o $\_I$, dependiendo de la rama en la que se encuentre el vértice del ciclo.

La regla que se encarga de la rutina de adaptar el uniciclo es AU, que contempla dos casos. El primer caso propaga la solución de la raíz a los vértices del ciclo por la rama izquierda, mientras que el segundo caso lo hace por la rama derecha. La regla AVC asigna los vértices conflictivos ($vg, vf, vh, v'g, v'f, v'h$) a los vértices hoja para la verificación de conflicto. Desde las hojas es posible verificar si existe conflicto o no, ya que todos los vértices marcados que pueden causarlo están a distancia uno o cero de las hojas.

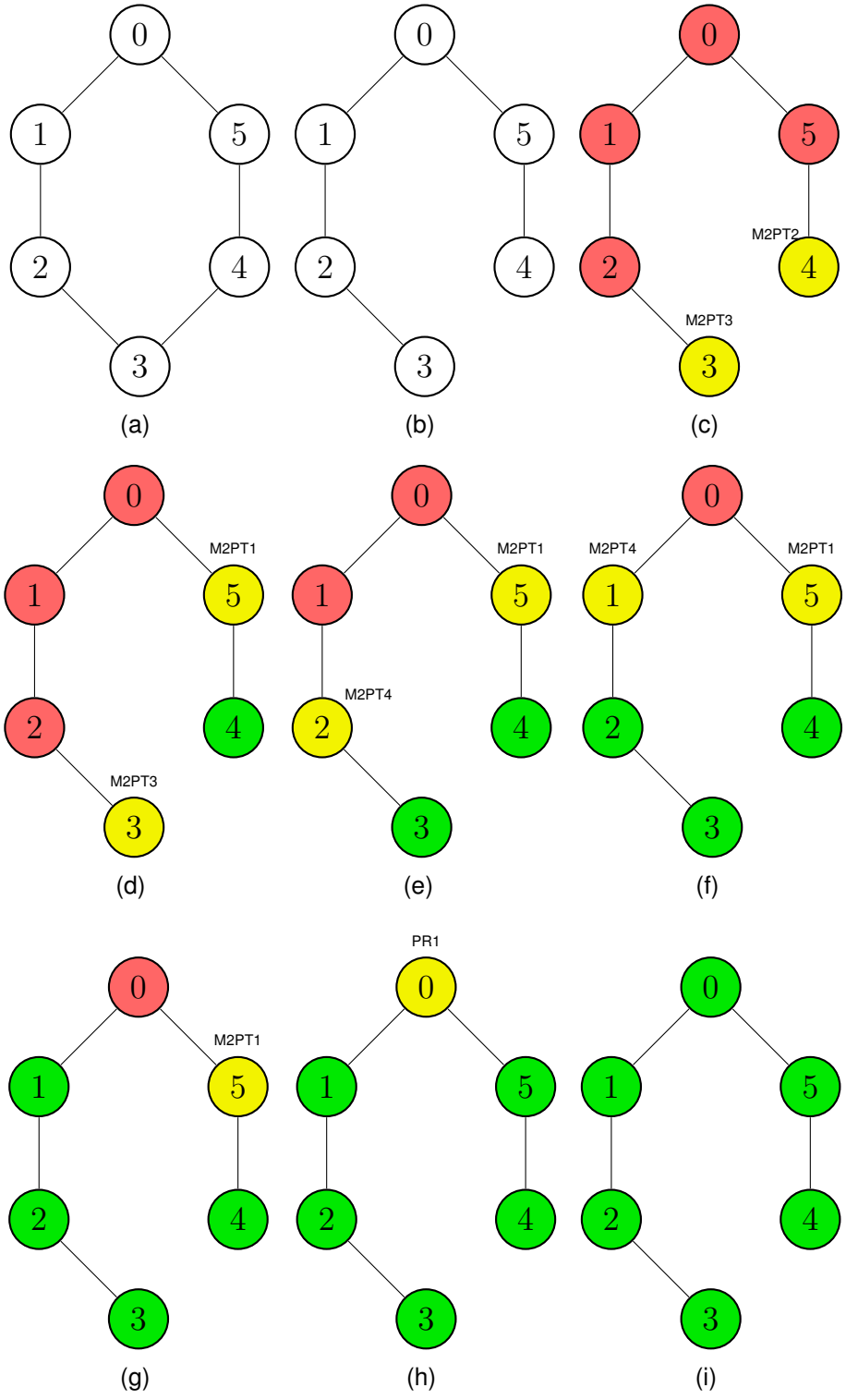


Figura 12: Ejemplo de secuencia para el cálculo del mejor árbol.

La regla AC agrega el conflicto correspondiente a los dos vértices adyacentes a la arista h. Si existe conflicto, se le asigna a la variable $v.\{current\_h\}\_conflict$ un número

del 1 al 5, dependiendo del error. Estos errores corresponden a los asignados $X1$, $X2$, $X3$, $X4$ o $X5$ en $\mathcal{G}$. El primer caso de la regla asigna el error en la hoja izquierda, mientras que el segundo caso lo hace en la hoja derecha. Finalmente, la regla DV se encarga de remover el vértice conflictivo de las tablas en caso de existir un conflicto. Esta regla se apoya de cinco funciones, por lo que se describe el funcionamiento de cada una de ellas.

La función $conflict\_correct(v)$ indica si el vértice v tiene asignado el conflicto correcto, de acuerdo a la variable $v.\{current\_h\}$. Los conflictos se calculan en las hojas y se transmiten hacia la raíz. Hay cuatro casos para v: hoja izquierda, hoja derecha, vértice interno izquierdo y vértice interno derecho. Para las hojas izquierda y derecha sólo es necesario que el conflicto sea igual al de la función $check\_conflict\_left$ y $check\_conflict\_right$, respectivamente. Para los vértices internos izquierdos, es necesario que su hijo, en este caso su sucesor, tenga llena su tabla de ciclo de la h actual con terminación $\_I$. Lo mismo sucede con los vértices internos derechos, pero con la terminación $\_D$. Para todos los vértices es necesario que la tabla del vértice hijo se encuentre llena, o en el caso de las hojas, que el conflicto se haya resuelto correctamente.

La función $vertex\_conflict(v, h)$ regresa el conflicto dependiendo de la ubicación de v en el árbol. Si v es una hoja, se regresa directamente $v.\{h\}\_conflict$. Si v es vértice interno izquierdo, se regresa el conflicto de su sucesor, y si es vértice interno derecho, se regresa el conflicto de su antecesor.

La función $special\_case(v, h, conflict)$ revisa el siguiente caso: al momento de desactivar un vértice, la mayoría de las veces se puede hacer directamente con el número del vértice, el valor de h y conociendo el conflicto actual. Si se necesitan desactivar vh o vg, se puede hacer directamente en la hoja izquierda, y $v'h$ o $v'g$ en la hoja derecha, véase la Figura 13(a). Los vértices vf y $v'f$ son los padres de las hojas, por lo que se deben desactivar en un momento distinto a las hojas. Cuando $0 < h < k - 1$ no existe problema, ya que dichos padres son vértices internos. Sin embargo, para $h = 0$, si el conflicto es 5, vg es la raíz, y uno de los vértices que debe desactivarse es el vértice con numeración $k - 1$ (vf), que técnicamente no es el padre de la hoja, por lo que se debe manejar como un caso especial (Figura 13(b)). El otro caso es el simétrico: si $h = k - 1$, el conflicto es 4, $v'g$ es la raíz, y uno de los vértices conflictivos es el vértice 1 ($v'f$), por lo que debe

atenderse de un modo similar (Figura 13(c)). Éstas son las condiciones que verifica la función $special\_case$.

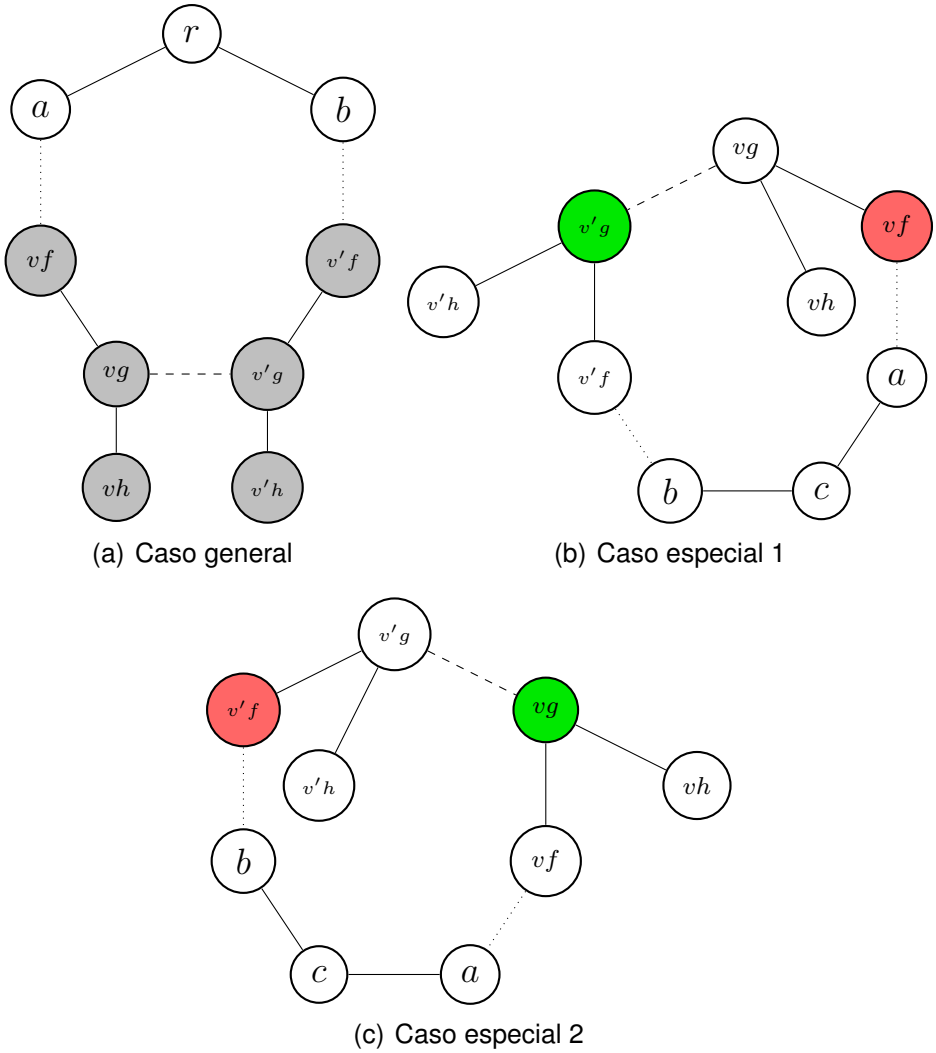


Figura 13: Casos general y especiales de corregir conflicto.

La función $has\_to\_deactivate(v, h, conflict)$ regresa un valor verdadero si el vértice v debe desactivarse o desactivar a uno de sus hijos, en el caso de que exista el conflicto 1 o 2 y se necesite remover a vh o $v'h$ de la solución. Primero pregunta si es alguno de los casos especiales mencionados anteriormente, y después analiza los casos generales. En caso de que tenga que desactivar un vértice, la siguiente función indica si el vértice a remover se encuentra desactivado.

Una vez que se detecta que el vértice v debe desactivar alguno de los vértices, la función $vertex\_deactivated(v, h, conflict)$ devuelve el valor verdadero si dicho vértice ya se encuentra desactivado, de lo contrario hay que removerlo. Primero se revisa si es

alguno de los casos especiales, y después para los casos generales.

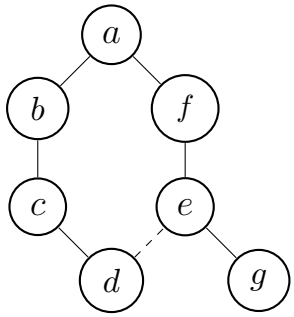
La regla CC es equivalente a la regla $M2PT$, pero para el momento cuando se encuentra resolviendo conflictos. Las operaciones son similares, pero no se incrementa el contador del árbol, solamente se propaga el conflicto.

Finalmente, la regla PRC procesa al vértice raíz. Esta regla contempla cuatro casos: el primer caso es cuando no existe conflicto en el árbol calculado; el segundo caso cuando sí existe conflicto, pero no es un caso especial; el tercer caso maneja el caso especial 1, cuando la arista removida es adyacente a la raíz por la izquierda ($h = 0$), y el vértice conflictivo es la misma raíz (Figura 13(b)); y el cuarto caso se encarga del caso especial 2, que es similar al anterior, pero la arista removida se encuentra en el lado derecho ($h = n - 1$) (Figura 13(c)).

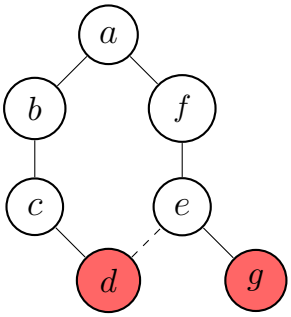
En el caso con conflicto se calculan los dos subárboles correspondientes. El subárbol 1 utiliza la rama derecha original y la rama izquierda procesada sin el vértice conflictivo, y el subárbol 2 utiliza la rama izquierda original y la rama derecha procesada sin el vértice en conflicto. Después manda la siguiente recursión: agrega $\_I\_R3$ al $current\_h$, es decir, comienza a revisar si existe conflicto en el subárbol 1.

Si no existe conflicto, entonces se pasa a la siguiente recursión, dependiendo del caso de $current\_h$:

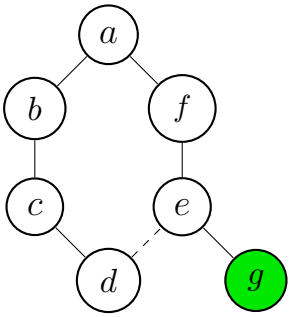
- Si termina con $r3$, $r2$ o $r1$, este índice se decrementa en uno.
- Si termina con $\_I\_r0$, pasa a $\_D\_r3$.
- Si termina con $\_D\_r0$, se calcula el mejor árbol combinando las soluciones de los subárboles 1 y 2. Se eliminan los últimos dos términos de $current\_h$, y se continúa con la siguiente recursión, de acuerdo a estas mismas condiciones. Por ejemplo, $h3\_r3\_D\_r0$ pasaría a $h3\_r2$.
- Si termina con $r0$ pero no tiene indicador I o D, entonces es la recursión inicial, y se pasa al siguiente h. Por ejemplo, $h3\_r0$ pasa a $h4\_r3$.



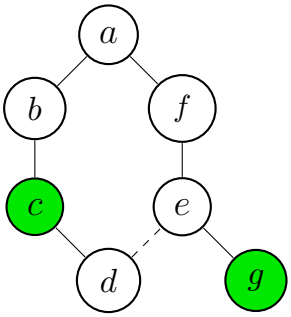
(a) h3



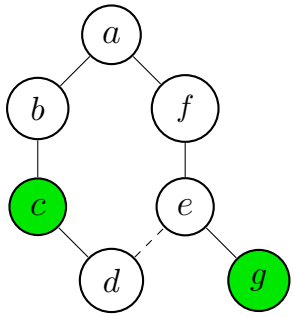
(b) h3_r3



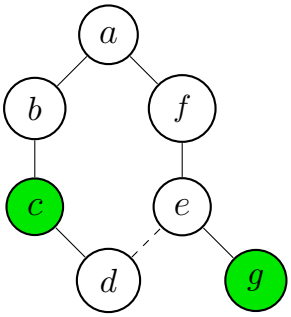
(c) h3_r3_l_r3



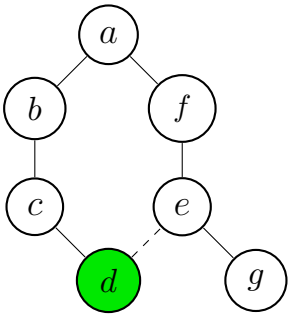
(d) h3_r3_l_r2



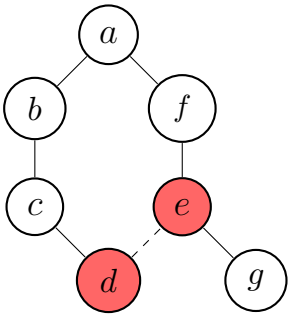
(e) h3_r3_l_r1



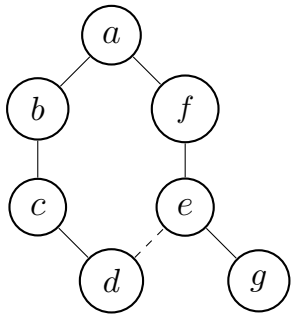
(f) h3_r3_l_r0



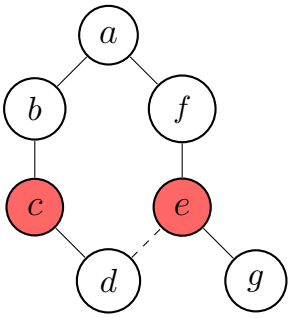
(g) h3_r3_D_r3



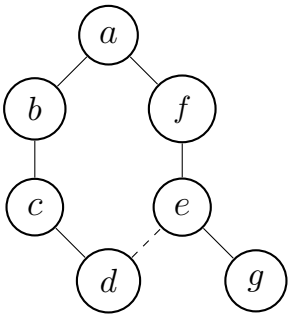
(h) h3_r3_D_r2



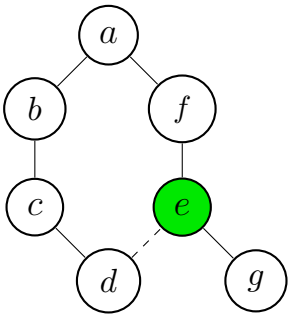
(i) h3_r3_D_r2_l_r3



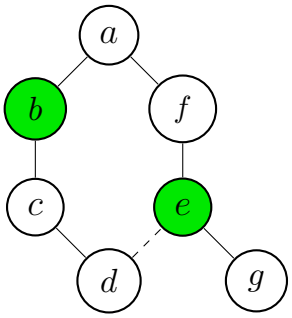
(j) h3_r3_D_r2_l_r2



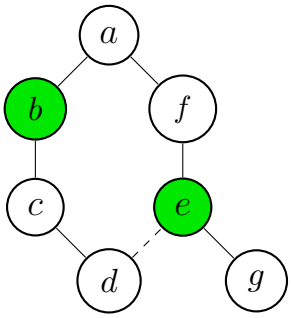
(k) h3_r3_D_r2_l_r2_l_r3



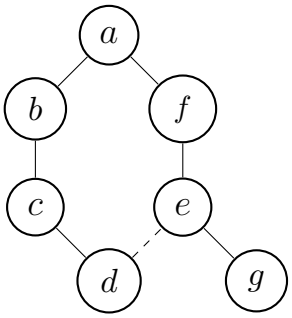
(l) h3_r3_D_r2_l_r2_l_r2



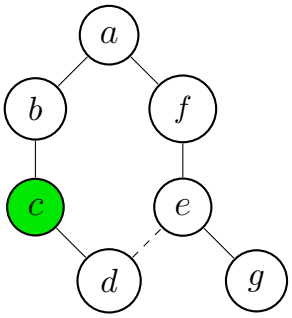
(m) h3_r3_D_r2_l_r2_l_r1



(n) h3_r3_D_r2_l_r2_l_r0



(ñ) h3_r3_D_r2_l_r2_D_r3



(o) h3_r3_D_r2_l_r2_D_r2

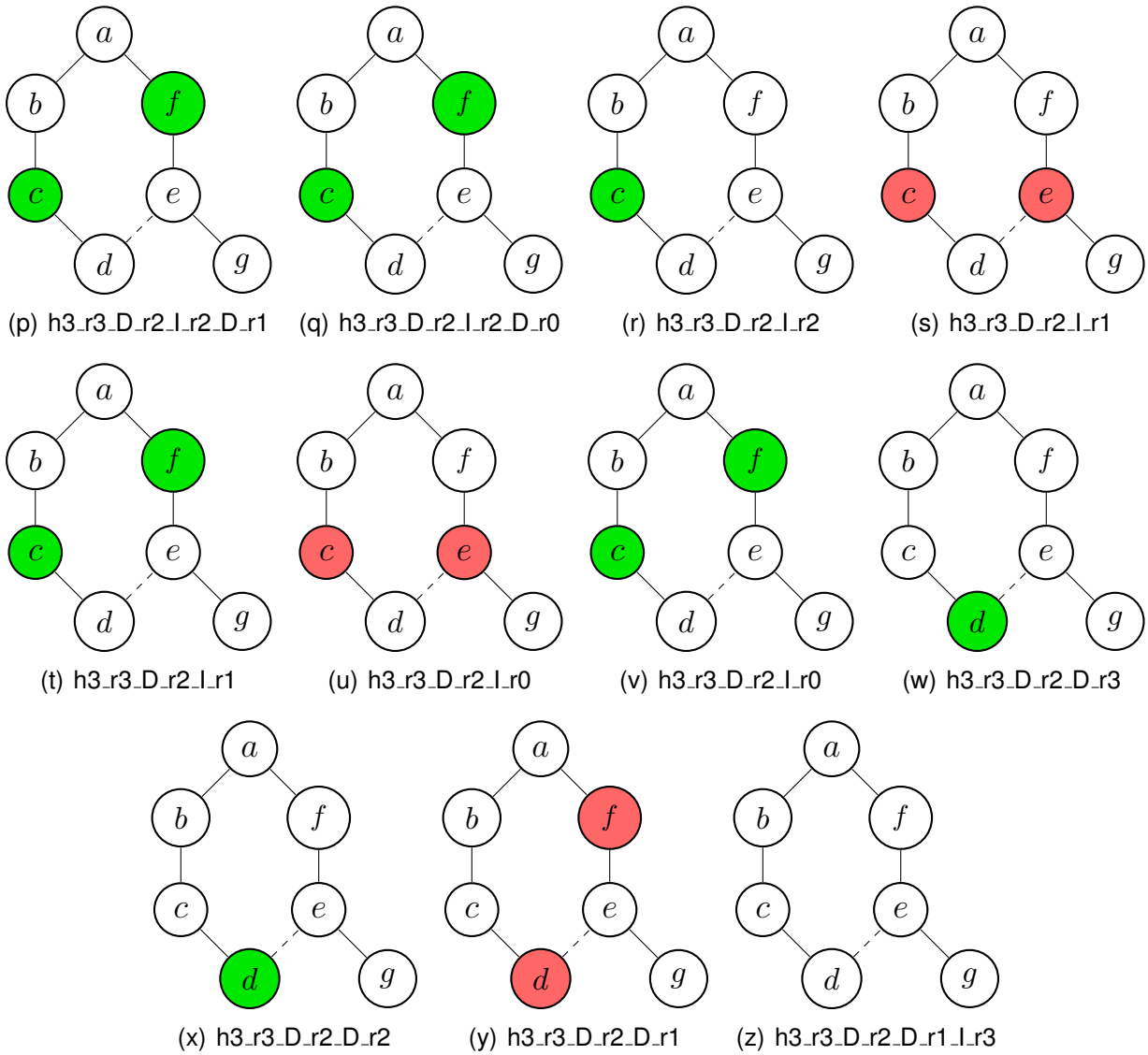
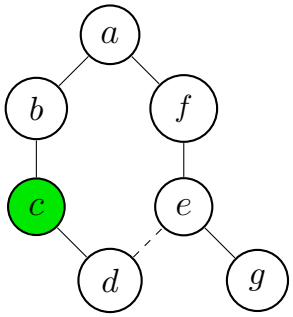
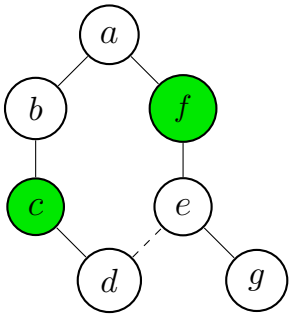


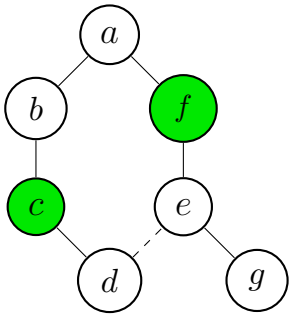
Figura 14: Ejemplo de corrección de conflictos.



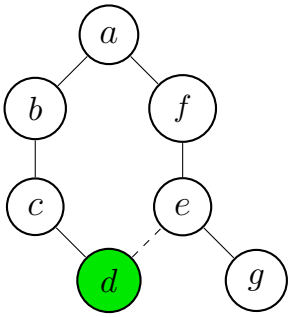
(a) h3_r3_D_r2_D_r1_I_r2



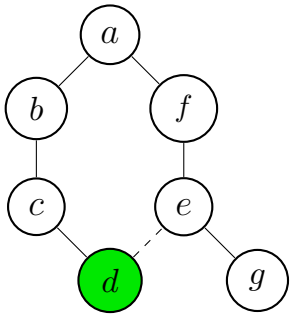
(b) h3_r3_D_r2_D_r1_I_r1



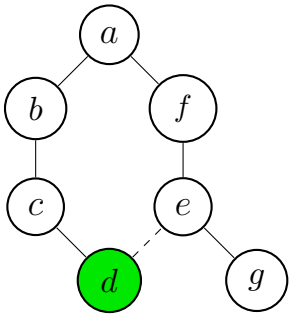
(c) h3_r3_D_r2_D_r1_I_r0



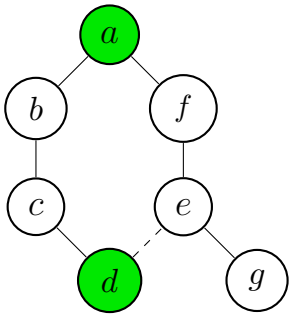
(d) h3_r3_D_r2_D_r1_D_r3



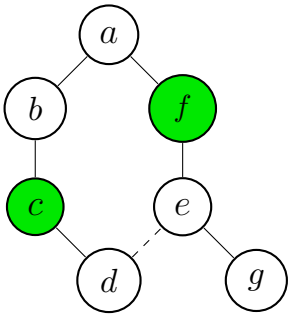
(e) h3_r3_D_r2_D_r1_D_r2



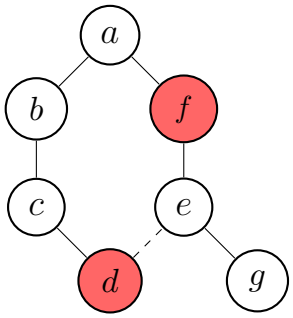
(f) h3_r3_D_r2_D_r1_D_r1



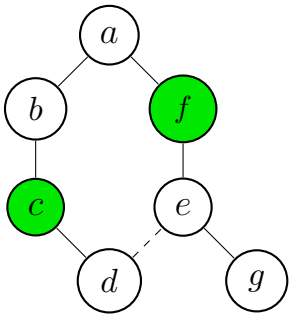
(g) h3_r3_D_r2_D_r1_D_r0



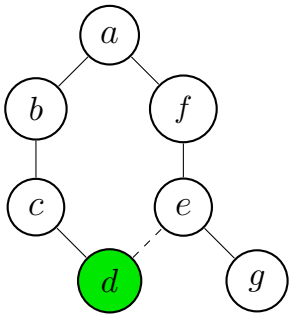
(h) h3_r3_D_r2_D_r1



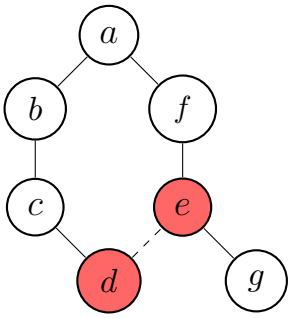
(i) h3_r3_D_r2_D_r0



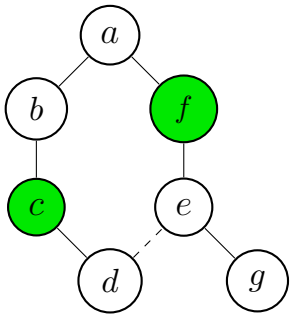
(j) h3_r3_D_r2_D_r0



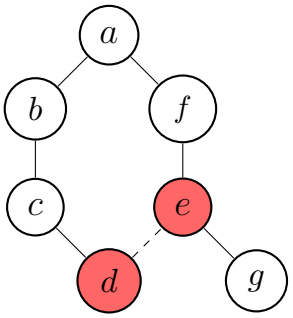
(k) h3_r3_D_r2



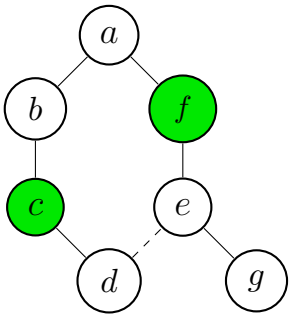
(l) h3_r3_D_r1



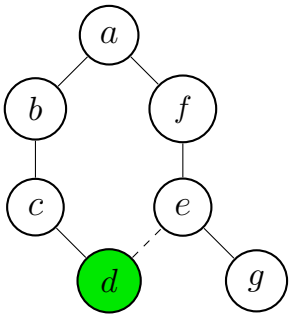
(m) h3_r3_D_r1



(n) h3_r3_D_r0



(ñ) h3_r3_D_r0



(o) h3_r3

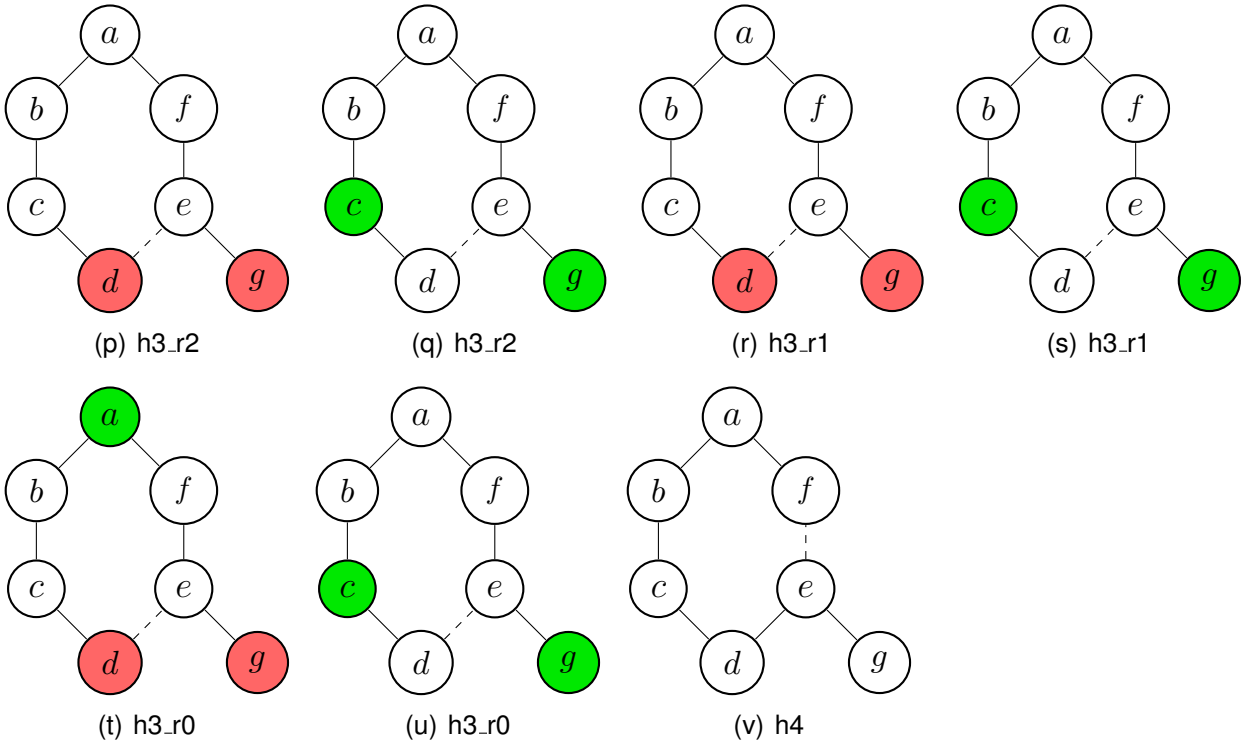


Figura 15: Ejemplo de corrección de conflictos (continuación).

En las figuras 14 y 15 se muestra un ejemplo del orden en que se calculan los árboles y sus respectivos conflictos, al momento de utilizar alguno de los casos de la regla PRC en la raíz del ciclo, que en este caso es el vértice a. El cliqué e-g y las columna 0 de cada uno de los vértices del ciclo ya se han calculado, así como la columna 1 para $h = 3$. Los vértices coloreados son los que se encuentran activos dentro de la solución indicada. Los vértices rojos indican conflicto.

- Figura 14(a). Se calcula $h = 3$ usando la regla $M2PT$.
- Figura 14(b). Se encuentra conflicto en $h3\_r3$. Se calculan $h3\_r3\_I$ y $h3\_r3\_D$. Se verifica primero $h3\_r3\_I$, comenzando por su renglón 3.
  - Figuras 14(c) - 14(f). $h3\_r3\_I\_r3$ a $h3\_r3\_I\_r0$ no tienen conflicto. Se prosigue a $h3\_r3\_D$.
  - Figura 14(g). $h3\_r3\_D\_r3$ no tiene conflicto.
  - Figura 14(h). $h3\_r3\_D\_r2$ tiene conflicto. Se calcula $h3\_r3\_D\_r2\_I$ y $h3\_r3\_D\_r2\_D$, y se verifica primero $h3\_r3\_D\_r2\_I$.

- Figura 14(i). $h3\_r3\_D\_r2\_I\_r3$ no tiene vértice activo, así que no tiene conflicto.
- Figura 14(j). $h3\_r3\_D\_r2\_I\_r2$ tiene conflicto entre los vértices c y e, así que se calculan sus respectivos subárboles.
  - Figuras 14(k) - 14(n). Todos los renglones de $h3\_r3\_D\_r2\_I\_r2\_I$ están libres de conflicto, se procede al subárbol derecho.
  - Figuras 14(ñ) - 14(q). Todos los renglones de $h3\_r3\_D\_r2\_I\_r2\_D$ tampoco tienen conflicto. Se combinan soluciones.
- Figura 14(r). $h3\_r3\_D\_r2\_I\_r2$ toma el valor de la mejor solución entre $h3\_r3\_D\_r2\_I\_r2\_I\_r2$ (Figura 14(l)) y $h3\_r3\_D\_r2\_I\_r2\_D\_r2$ (Figura 14(o)).
- Figura 14(s). $h3\_r3\_D\_r2\_I\_r1$ tiene el mismo conflicto que $h3\_r3\_D\_r2\_I\_r2$ (Figura 14(j)); los subárboles generados son los mismos (Figuras 14(k) - 14(q)), por lo que se omiten.
- Figura 14(t). $h3\_r3\_D\_r2\_I\_r1$ toma la mejor solución entre $h3\_r3\_D\_r2\_I\_r1\_I\_r1$ y $h3\_r3\_D\_r2\_I\_r1\_D\_r1$. Estos son los mismos que $h3\_r3\_D\_r2\_I\_r2\_I\_r1$ (Figura 14(m)) y $h3\_r3\_D\_r2\_I\_r2\_D\_r1$ (Figura 14(p)).
- Figura 14(u). $h3\_r3\_D\_r2\_I\_r0$ tiene el mismo error anterior, se vuelven a omitir los subárboles.
- Figura 14(v). $h3\_r3\_D\_r2\_I\_r0$ toma la mejor solución entre $h3\_r3\_D\_r2\_I\_r0\_I\_r0$ y $h3\_r3\_D\_r2\_I\_r0\_D\_r0$. Estos son los mismos que $h3\_r3\_D\_r2\_I\_r2\_I\_r0$ (Figura 14(n)) y $h3\_r3\_D\_r2\_I\_r2\_D\_r0$ (Figura 14(q)). Se prosigue al subárbol derecho.
- Figura 14(w). $h3\_r3\_D\_r2\_D\_r3$ no tiene conflicto.
- Figura 14(x). $h3\_r3\_D\_r2\_D\_r2$ no tiene conflicto.
- Figura 14(y). $h3\_r3\_D\_r2\_D\_r1$ tiene conflicto entre d y f. Se calculan $h3\_r3\_D\_r2\_D\_r1\_I$ y $h3\_r3\_D\_r2\_D\_r1\_D$, comenzando con el subárbol izquierdo.
  - Figura 14(z). $h3\_r3\_D\_r2\_D\_r1\_I\_r3$ no tiene conflicto.
  - Figuras 15(a) - 15(c). Los demás renglones de $h3\_r3\_D\_r2\_D\_r1\_I$ tampoco tienen conflicto.

        - ◇ Figuras 15(d) - 15(g). Los renglones de $h3\_r3\_D\_r2\_D\_r1\_D$ tampoco tienen conflicto.
    - Figura 15(h). $h3\_r3\_D\_r2\_D\_r1$ toma la mejor solución entre $h3\_r3\_D\_r2\_D\_r1\_I\_r1$ (Figura 15(b)) y $h3\_r3\_D\_r2\_D\_r1\_D\_r1$ (Figura 15(f)).
    - Figura 15(i). $h3\_r3\_D\_r2\_D\_r0$ presenta el mismo conflicto que $h3\_r3\_D\_r2\_D\_r1$. Se omiten los subárboles.
    - Figura 15(j). $h3\_r3\_D\_r2\_D\_r0$ toma la mejor solución entre $h3\_r3\_D\_r2\_D\_r0\_I\_r0$ y $h3\_r3\_D\_r2\_D\_r0\_D\_r0$. Estos son los mismos que $h3\_r3\_D\_r2\_D\_r1\_I\_r0$ (Figura 15(c)) y $h3\_r3\_D\_r2\_D\_r1\_D\_r0$ (Figura 15(g)).
- Figura 15(k). $h3\_r3\_D\_r2$ toma la mejor solución entre $h3\_r3\_D\_r2\_I\_r2$ (Figura 14(r)) y $h3\_r3\_D\_r2\_D\_r2$ (Figura 14(x)).
- Figura 15(l). $h3\_r3\_D\_r1$ tiene el mismo conflicto que $h3\_r3\_D\_r2$. Se omiten los subárboles.
- Figura 15(m). $h3\_r3\_D\_r1$ toma la mejor solución entre $h3\_r3\_D\_r1\_I\_r1$ y $h3\_r3\_D\_r1\_D\_r1$. Estos son los mismos que $h3\_r3\_D\_r2\_I\_r1$ (Figura 14(t)) y $h3\_r3\_D\_r2\_D\_r1$ (Figura 15(h)).
- Figura 15(n). $h3\_r3\_D\_r0$ tiene el mismo conflicto que $h3\_r3\_D\_r2$. Se omiten los subárboles.
- Figura 15(ñ). $h3\_r3\_D\_r0$ toma la mejor solución entre $h3\_r3\_D\_r0\_I\_r0$ y $h3\_r3\_D\_r0\_D\_r0$. Estos son los mismos que $h3\_r3\_D\_r2\_I\_r0$ (Figura 14(v)) y $h3\_r3\_D\_r2\_D\_r0$ (Figura 15(j)).

- Figura 15(o). $h3\_r3$ toma la mejor solución entre $h3\_r3\_I\_r3$ (Figura 14(c)) y $h3\_r3\_D\_r3$ (Figura 14(g)).
- Figura 15(p). $h3\_r2$ tiene el mismo conflicto que $h3\_r3$. Se omiten los subárboles.
- Figura 15(q). $h3\_r2$ toma la mejor solución entre $h3\_r2\_I\_r2$ y $h3\_r2\_D\_r2$. Estos son los mismos que $h3\_r3\_I\_r2$ (Figura 14(d)) y $h3\_r3\_D\_r2$ (Figura 15(k)).
- Figura 15(r). $h3\_r1$ tiene el mismo conflicto anterior y se omiten los subárboles.

- Figura 15(s). $h3\_r1$ toma la mejor solución entre $h3\_r1\_I\_r1$ y $h3\_r1\_D\_r1$. Estos son los mismos que $h3\_r3\_I\_r1$ (Figura 14(e)) y $h3\_r3\_D\_r1$ (Figura 15(m)).
- Figura 15(t). $h3\_r0$ tiene el mismo conflicto que los renglones anteriores. Se omiten los subárboles.
- Figura 15(u). $h3\_r0$ toma la mejor solución entre $h3\_r0\_I\_r0$ y $h3\_r0\_D\_r0$. Estos son los mismos que $h3\_r3\_I\_r0$ (Figura 14(f)) y $h3\_r3\_D\_r0$ (Figura 15(ñ)).
- Figura 15(v). Se continúa con $h4$.

3.5 Demostración

La demostración se realiza por partes, suponiendo en cada una que las anteriores a ella ya se ha demostrado correctamente. Las partes que se necesitan demostrar son el llenado correcto de las tablas de cliqués y de ciclo, así como la numeración de los vértices del ciclo. Este algoritmo se basa en el algoritmo $\mathcal{G}$, por lo que la demostración indica equivalencia entre ambos.

Vértices de cliqués

Como se menciona anteriormente, hay dos tipos de vértices en los cliqués: vértice inferior y vértice superior. A su vez, los vértices inferiores se clasifican como vértices hoja y vértice no hoja.

Lema 3.1 *Todos las tablas de los vértices de cliqués se llenan correctamente con la regla* PC.

Demostración. Se divide en tres casos, dependiendo del vértice v: cuando v es hoja, cuando v es vértice inferior pero no es hoja, y cuando v es vértice superior.

Los vértices hoja en el árbol enraizado son aquellos que no forman parte de un ciclo y no tienen hijos en el árbol. Estas condiciones las verifica la función $is\_leaf(v)$. Como las hojas solamente tienen una tabla, $table\_up$, se pueden llenar en cualquier momento

sin depender de ningún otro vértice. Los datos con los que se llena la tabla son los de la configuración nula en las columnas 0 y 1. Este proceso se realiza en el primer caso de la regla PC.

Los vértices inferiores que no son hojas también tienen tienen una tabla $table\_up$, además de tablas hacia sus hijos. Dichas tablas deben calcularse antes de poder procesar la tabla $table\_up$, que se llena con los datos del último hijo, ya que éste contiene la información de los hijos anteriores. El segundo caso de la regla PC identifica a los vértices inferiores de cliqués no hoja. Esta regla verifica que la última tabla hijo ya se haya llenado para procesar su propia tabla $table\_up$, que se llena con los datos de esta última, ya que de acuerdo a la equivalencia de bloques de la Figura 10, es su único bloque hijo. La excepción se da cuando el vértice forma parte de un ciclo; pero esto implica que tal vértice es la raíz del ciclo, y su tabla $table\_up$ se calcula directamente con las reglas del ciclo.

Los vértices superiores contienen las tablas $table\_children\_id$ hacia cada uno de los hijos. Estas tablas se llenan en el orden en que se tienen asignados los hijos de cada vértice, como se muestra en la Figura 10, ya que cada tabla contiene la información de la tabla anterior.

Sean $v.children\_1, v.children\_2, ..., v.children\_k$ los k hijos de arista puente de v. Si las tablas $v.children\_1.table\_up, v.children\_2.table\_up, ..., v.children\_k.table\_up$ ya se han llenado, entonces las tablas $table\_children\_\{id\}$ se llenan correctamente con la tercera condición de la regla PC, ya que las primeras dos condiciones del tercer caso de esta regla aseguran que los vértices que se pueden habilitar son los vértices superiores de cliqués. La siguiente parte se demuestra por inducción.

Caso base. Si $v.last\_child\_completed = 0$, se calcula $v.table\_children\_\{v.children\_1\}$ con la información de $v.children\_1.table\_up$, y se incrementa $v.last\_child\_completed$ a 1.

Hipótesis inductiva. Suponga que $v.last\_child\_completed = i$, entonces la tabla $v.table\_children\_\{v.children\_i\}$ ya se ha calculado.

Paso inductivo. Sea $v.last\_child\_completed = i{+}1$. $v.table\_children\_\{v.children\_(i{+}1)\}$

se calcula con $v.children\_(i+1).table\_up$ y $v.table\_children\_\{v.children\_i\}$, la primera como precondición y la segunda como hipótesis. Se incrementa $v.last\_child\_completed$ a $i+2$.

Así, todas las tablas $table\_children\_\{id\}$ se llenan correctamente con el tercer caso de la regla PC. ■

Lema 3.2 *Los vértices del ciclo se numeran correctamente con la regla NC.*

Demostración. Se divide en dos casos: cuando el vértice es la raíz del ciclo y cuando no lo es.

Cuando el vértice es la raíz, se utiliza el primer caso de la regla NC. Este caso tiene dos condiciones: que el vértice sea la raíz del ciclo y que no se encuentre correctamente numerado. La función $correct\_numeration$, para el caso de la raíz, debe cumplir que tenga la numeración 0, su sucesor debe ser el vecino en $cycle\_neighbors$ con menor id, y antecesor el id del vecino con mayor id. Si no cumple estas condiciones, la regla los asigna correctamente.

Para el caso de un vértice diferente de la raíz, primero se demuestra la siguiente propiedad: sea r la raíz del ciclo y v otro vértice del ciclo. Si v se encuentra correctamente numerado, entonces todos los vértices desde r hacia v, en dirección de los sucesores, se encuentran correctamente numerados.

Esta propiedad se demuestra por contradicción. Suponga que existe un vértice w entre r y v que no se encuentra correctamente numerado. Entonces $w.numeration\_error$ tiene un valor verdadero. Por lo tanto, el siguiente vértice en la secuencia, suponga x, al llamar la función $correct\_numeration(x)$ regresa un valor falso, por lo que $x.numeration\_error = verdadero$. Usando el mismo argumento para los siguientes vértices, se llega a que $v.numeration\_error = verdadero$, lo cual es una contradicción, ya que v se encuentra correctamente numerado. Hay que notar que el proceso de propagación de los errores no es instantáneo, ya que se va aplicando el tercer caso de la regla E para cada vértice afectado.

Para demostrar que los vértices diferentes de la raíz se numeran correctamente se

considera la propiedad anterior y el segundo caso de la regla NC: sea v el último vértice correctamente numerado a partir de la raíz (v puede ser la raíz) y w el siguiente vértice en el ciclo, entonces w se numera correctamente con el segundo caso de la regla NC.

El segundo caso de la regla NC verifica que w sea parte del ciclo pero no raíz, que no se encuentre correctamente numerado, y que tenga sólo un vecino en $w.cycle\_neighbors$ que se encuentre correctamente numerado (el antecesor) y que apunte su sucesor a él. La última condición se cumple porque uno de sus vecinos de ciclo (v) se encuentra correctamente numerado y apunta a w, y el otro vecino en el ciclo no puede estar correctamente numerado si w aún no lo está, por la propiedad anterior. Entonces, el segundo caso de la regla NC se aplica a w y lo numera correctamente. Aplicando ésto a cada uno de los vértices del ciclo de forma secuencial, se numeran todos de manera correcta. ■

Columna 0 de vértices de ciclo

La columna 0 de los vértices de ciclo se llenan de una manera similar a la columna 0 de la table $table\_up$ de los vértices inferiores de cliqués (segundo caso de la regla PC). La única diferencia es la condición de qué tipo de vértice es.

Lema 3.3 *Los vértices de ciclo calculan correctamente su columna 0 con la regla IC.*

Demostración. De manera similar al segundo caso del Lema 3.1, se llena solamente la columna 0 de las tablas $table\_up$ de los vértices del ciclo. ■

Maximum 2-packing tree

En esta sección se llama k al número de vértices en el ciclo, y se supone que todos ellos se encuentran correctamente numerados. También por practicidad, los identificadores que se usan son los de la numeración del ciclo, a diferencia de los identificadores en el grafo original. Como el procedimiento es similar para todos los distintos árboles que se pueden generar, se realiza la demostración para $h = i$, donde $0 \leq i < k$, es decir, cuando se remueve la arista entre el vértice i y el vértice $i + 1$, y resulta una estructura de árbol. En caso de que $i = k - 1$, la arista removida es la arista entre el vértice $k - 1$ y la raíz.

También se supone que la columna 0 de las tablas $table\_up$ de todos los vértices ya se ha llenado.

Lema 3.4 *Los vértices distintos de la raíz llenan la columna* 1 *de su tabla* $table\_up\_\{i\}$ *correctamente con la regla* $M2PT$*.*

Demostración. Se divide en dos casos: cuando el vértice es una hoja de alguna de las ramas, y cuando es un vértice interno.

El caso de los vértices hoja, se divide en dos casos: cuando la hoja es adyacente a la arista removida por el lado izquierdo, y cuando lo es por el lado derecho. Cuando es una hoja izquierda, se aplica el primer caso de la regla $M2PT$. En ésta, se revisa que el vértice sea del ciclo pero no raíz, esté correctamente numerado, tenga llena su columna 0, que la columna por llenar aún no esté llena, y que tenga la numeración adecuada, es decir, que su numeración coincida con la hoja en el árbol actual. La numeración para la hoja de la rama izquierda es i, por lo que este caso se resuelve correctamente. Análogamente para la hoja derecha, cuya numeración es $i+1$, se aplica el segundo caso de la regla $M2PT$.

El caso de los vértices internos también se divide en dos casos: cuando es vértice interno de la rama izquierda, y cuando lo es de la rama derecha. Sea v un vértice interno que no es hoja ni raíz, esto es, $v \neq 0, v \neq i, v \neq i+1$. Si $i > v$, v es un vértice interno de la rama izquierda. Los vértices internos de la rama izquierda necesitan que la columna 1 de la tabla del vértice sucesor esté llena, así como su propia columna 0. Estas condiciones las satisface el tercer caso de la regla $M2PT$. Para $i < v - 1$, v es un vértice interno de la rama derecha. Las condiciones son similares, pero necesitan que el vértice antecesor haya llenado su columna 1. Estas condiciones las satisface el cuarto caso de la regla $M2PT$. ■

Lema 3.5 *El vértice raíz llena correctamente su columna* 1 *con la regla* PR*.*

Demostración. Para calcular la columna 1 del vértice raíz es necesario que ambas ramas hayan llenado su columna 1. Estas condiciones son las que necesita la regla PR. Hay que

notar que cuando $i = 0$ o $i = k - 1$, la raíz solamente necesita una de las dos ramas, ya que la otra no existe. Lo anterior no afecta en las condiciones, ya que el vértice antecesor es la hoja de la rama izquierda para $i = k - 1$, y el vértice sucesor es la hoja para la rama derecha cuando $i = 0$, y para que las ramas se llenen correctamente, las hojas también deben estarlo. ■

Adaptar el uniciclo

La solución potencial se encuentra en la columna 1 de la raíz, pero no es posible detectar los posibles conflictos desde ahí, así que la solución se propaga hasta los vértices i e $i + 1$, que verifican si existe conflicto.

Lema 3.6 *La solución se propaga correctamente desde la raíz a los vértices i e $i + 1$ con la regla AU.*

Demostración. La solución se propaga por las dos ramas por separado, primero se analiza la rama izquierda. Sea v un vértice de la rama izquierda, entonces $v \leq i$. La solución del vértice $v - 1$ se propaga hacia v solamente si $v - 1$ es correcto, es decir, contiene la solución que la raíz le transmitió. Esta solución se obtiene al verificar los apuntadores de la celda que marcaba el vértice $v - 1$ como solución potencial. Lo anterior es correcto con el primer caso de la regla AU. Para un vértice v de la rama derecha, $v > i$, y la solución la obtiene del vértice $v + 1$, o directamente de la raíz, si $v = k - 1$. Esto aplica para el segundo caso de la regla AU. ■

Lema 3.7 *Una vez que los vértices adyacentes a la arista removida i tienen la solución potencial, la regla AVC les asigna las variables vf, vg, vh a la hoja izquierda, y $v'f, v'g, v'h$ a la hoja derecha, indicando los vértices coloreados en la solución.*

Demostración. En caso de que el vértice i esté marcado en la solución potencial, se asigna $vg = i$. Los vértices vf y vh se encuentran a distancia uno del vértice i, por lo que es posible asignarlos desde i. Cabe resaltar que de estos tres vértices, a lo más hay uno en la solución potencial. Lo mismo pasa con el vértice $i + 1$ y las variables $v'g, v'f$ y $v'h$. ■

Asignación de conflicto

Una vez que la hoja de la rama izquierda y la hoja de la rama derecha tienen los vértices vf, vg, vh y $v'f, v'g, v'h$ asignados, respectivamente, es posible revisar cuál de los cinco conflictos existe, o si no existe tal.

Lema 3.8 *Las hojas identifican el conflicto correspondiente con la regla AC, y lo asigna a sus variables $\{current\_h\}\_conflict$.*

Demostración. Para la hoja izquierda, las condiciones del primer caso de la regla AC son que el vértice v sea vértice de ciclo, que esté correctamente numerado, que sea la hoja izquierda, que tenga asignado los vértices conflictivos correctamente, así como el de su sucesor, en este caso, la hoja derecha. La última condición es que si el entero que tiene en su variable de conflicto es diferente al de la función de $check\_conflict\_left$, entonces la variable toma el valor que la última regresa. La función $check\_conflict\_left$ recibe como entrada un vértice v y la numeración h actual ($v.current\_h$). Como lo muestra la Figura 6, si vg se encuentra activado y $v'h$ también, entonces es el conflicto 1. Estos vértices equivalen a $v.h\_vg$ y $v.sucessor.h\_v'h$, que son los que indican la función. Lo mismo para vh ($v.h\_vh$) y $v'g$ ($v.successor.h\_v'g$) en el conflicto 2, vg ($v.h\_vg$) y $v'g$ ($v.successor.h\_v'g$) en el conflicto 3, vg ($v.h\_vg$) y $v'f$ ($v.successor.h\_v'f$) en el conflicto 4, y vf ($v.h\_vf$) y $v'g$ ($v.successor.h\_v'g$) en el conflicto 5. Si ninguno de estos casos sucede, entonces el conflicto es 0.

Lo mismo sucede con la función $check\_conflict\_right$ y el segundo caso de la regla AC para la hoja derecha. ■

Corregir conflicto

Lema 3.9 *Si un vértice v se encarga de remover algún vértice conflictivo, éste se remueve correctamente con la regla DV antes de que se calcule su tabla en la iteración h hacia la raíz.*

Demostración. Las reglas que calculan las tablas en la rutina de adaptar el uniciclo y corregir conflicto son CC para los vértices internos, mientras que PRC se encarga

de la raíz. Recordar que cualquiera de los vértices del ciclo pueden ser responsables de desactivar algún vértice en conflicto, incluyendo a la raíz. Sin embargo, en todas las reglas anteriormente mencionadas, hay una condición que asegura que el vértice ya haya desactivado al vértice conflictivo en caso de necesitarlo, excepto en el primer caso de la regla PRC para la raíz, ya que ésta se utiliza cuando el conflicto es 0, es decir, no hay conflicto y por lo tanto, no es necesario remover ningún vértice de la solución. La regla DV se activa si el conflicto es el correcto, y el vértice está habilitado para remover el vértice conflictivo. ■

Lema 3.10 *La regla CC calcula correctamente las tablas de los vértices distintos de la raíz en la rutina de corregir conflicto.*

Demostración. De forma similar al Lema 3.4, se divide en cuatro casos: hoja izquierda, hoja derecha, vértice interno izquierdo y vértice interno derecho.

Suponiendo que existe conflicto y el vértice encargado de remover al vértice conflictivo en la solución es la hoja izquierda, éste lo debe realizar primero con la regla DV. Una vez removido, no necesita que otro vértice se haya calculado, por lo que el primer caso de la regla CC aplica el mismo procedimiento que el primer caso de la regla $M2PT$ para calcular su columna 1. Lo mismo sucede para la hoja derecha, donde el segundo caso de la regla CC aplica el procedimiento del segundo caso de $M2PT$. Recordar que corregir el conflicto es el mismo procedimiento que el Maximum 2-packing tree, pero removiendo el vértice conflictivo y sin incrementar la variable del contador de árbol, los cuales se demuestran en el Lema 3.4.

De manera similar, el tercer y cuarto caso de la regla CC son similares al tercer y cuarto caso de la regla $M2PT$, respectivamente (Lema 3.4). Calcular los vértices internos de las ramas necesitan que sus hijos hayan llenado su tabla correctamente, lo que se verifica en la función $conflict\_correct$. Las demás condiciones son similares a los dos primeros casos de CC. En estos últimos dos casos hay una asignación adicional de transmitir el número de conflicto. ■

Una vez que los vértices internos hayan calculado sus tablas, toca a la raíz verificar si

hay que corregir o no. Como se describe en la explicación de la regla PRC, hay cuatro casos para la corrección de conflictos: cuando el conflicto es 0 o no hay conflicto; cuando es el caso general (Figura 13(a)), donde ambas ramas deben haber terminado y coincidir en el conflicto distinto de cero; cuando es el caso específico 1 (Figura 13(b)), donde $h = 0$, solamente se tiene rama derecha, y como la raíz es la hoja izquierda, se revisa que esta última y el vértice antecesor coincidan en el conflicto y que haya terminado la rama $\_D$; cuando es el caso específico 2 (Figura 13(c)) sucede lo mismo, pero con la rama izquierda $\_I$ y $h = k - 1$.

Lema 3.11 *Si el conflicto es 0, la raíz pasa a la siguiente recursión con el primer caso de la regla PRC.*

Demostración. Como no existe conflicto, el árbol pasa a la recursión siguiente, las acciones que realiza la regla son idénticas a la forma mostrada en el algoritmo:

- Si $v.current\_h$ termina en $r3, r2$, o $r1$, $v.current\_h$ pasa al siguiente renglón, $r2, r1, r0$, respectivamente.
- Si $v.current\_h$ termina en I_r0, $v.current\_h$ pasa a D_r3.
- Si $v.current\_h$ termina en D_r0, calcula el mejor árbol mezclando el subárbol 1 y el subárbol 2, se guarda y se elimina D_r0 de $v.current\_h$ y se pasa a la siguiente recursión con esta misma regla.
- Si $v.current\_h$ termina en $r0$ sin tener $\_D$ o $\_I$ antes, es decir, termina con hx_r0, donde $0 \leq x \leq k - 1$, entonces se calcula el mejor árbol hasta el momento y se guarda en $v.table\_up$, columna 2, y se pasa a la siguiente x, renglón 3.

■

Lema 3.12 *Si el conflicto es diferente de 0, la raíz pasa a la corrección de conflictos con los casos restantes de la regla PRC.*

Demostración. Primero se analiza el caso general (Figura 13(a)). Este caso se asigna al segundo caso de la regla PRC, que revisa que $h \neq 0$ y $h \neq k-1$. En tal caso se revisa que el primer vértice de ambas ramas tengan el mismo conflicto (diferente de cero), y que en caso de que la raíz tenga que desactivar un vértice, lo haya hecho ya. El subárbol 1 se calcula de la misma forma que las líneas 17 - 23 del Pseudocódigo 5, pero con las tablas $v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I$ y $v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}$, $v.cycle\_successor.table\_cycle\_\{v.current\_h\}$ y $v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D$ para el subárbol 2. Se agrega $\_I\_r3$ a $v.current\_h$ y se prosigue a revisar si existe conflicto en dicha rama.

Para el primer caso especial, cuando $h = 0$, se revisa que el primer vértice de la rama derecha esté lleno, que su conflicto coincida con el de la raíz, y si este último debe desactivar algún vértice. El cálculo de los árboles es similar a las líneas 3 - 9 del Pseudocódigo 5, utilizando $v.table\_cycle\_\{v.current\_h\}\_I$ columna 0 como vértice raíz y $v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}$ para el primero y $v.table\_cycle\_\{v.current\_h\}$ como raíz y $v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D$ para el segundo. Lo anterior se realiza en el tercer caso de la regla PRC.

Para el segundo caso especial, cuando $h = k-1$, se revisa que el primer vértice de la rama izquierda esté lleno, que su conflicto coincida con el de la raíz, y si fue necesario desactivar algún vértice. El cálculo de los árboles es similar a las líneas 10 - 16 del Pseudocódigo 5, utilizando $v.table\_cycle\_\{v.current\_h\}\_D$ columna 0 como vértice raíz y $v.cycle\_successor.table\_cycle\_\{v.current\_h\}$ para el subárbol 2 y $v.table\_cycle\_\{v.current\_h\}$ como raíz y $v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I$ para el subárbol 1. Esto se realiza en el cuarto caso de la regla PRC. ■

Lema 3.13 *Al terminar de calcular $h = i$, en la columna 2 de $r.table\_up$ se encuentra la mejor solución hasta el árbol i.*

Demostración. El cálculo del árbol $h = i$ termina cuando se llega a la recursión $current\_h = hi\_r0$ y no existe conflicto, es decir, todos los renglones se corrigieron. Esto sucede en la última condición de la acción del primer caso de la regla PRC, en la cual se compara lo

que se tiene en $r.table\_up$ columna 2 con $r.hi\_r3, r.hi\_r2, r.hi\_r1$ y $r.hi\_r0$, y se guarda el máximo en cada renglón de $r.table\_up$ columna 2. ■

Lema 3.14 *Al terminar de calcular $h = k - 1$, la columna 2 de $r.table\_up$ contiene la mejor solución del ciclo y sus descendientes.*

Demostración. Directo del Lema 3.13, aplicando $i = k - 1$. ■

Distribución de la solución

Teorema 3.1 *Sea r el vértice raíz del árbol enraizado generado en el algoritmo de elección de líder de Datta* et al. *(2011b), es decir, el vértice cuya variable $parent = null$. Si r pertenece a un ciclo, la tabla $r.table\_up$ contiene la solución al conjunto 2-packing con peso máximo en el uniciclo. Si r no pertenece a un ciclo, la tabla $r.table\_children\_\{x\}$, donde x es el identificador del último hijo de r en su lista de hijos, contiene la solución mencionada.*

Demostración. Si r pertenece a un ciclo, la tabla $r.table\_up$ se llena de acuerdo al Lema 3.14, que contiene la mejor solución del ciclo y de los descendientes de los vértices en él, los cuales comprenden el uniciclo completo y por lo tanto el 2-packing con peso máximo. Si r no pertenece a un ciclo, entonces es un vértice superior de cliqué. En los vértices superiores de los cliqués, la tabla $v.table\_children\_\{x\}$, donde x es el identificador del último hijo en la lista de hijos de v contiene la mejor solución hasta v. Esta información normalmente se pasa a la tabla $v.table\_up$, ya sea porque es parte de un ciclo utilizando la regla IC, o porque es vértice inferior de otro cliqué, usando el tercer caso de la regla PC (Lema 3.1). Como r no tiene padre, no es vértice inferior de otro cliqué y no se habilita por esta regla, así que la solución queda en la tabla $r.table\_children\_\{x\}$. ■

Una vez que el vértice raíz contenga la solución del uniciclo en su última tabla, se necesita propagar la solución hacia los demás vértices del grafo.

Lema 3.15 *La regla S le indica a los vértices la solución del conjunto 2-packing con peso máximo cuando se termina de calcular el uniciclo.*

Demostración. Se divide en dos casos: cuando el vértice es la raíz r del árbol de expansión o líder, y cuando no lo es.

Para el caso del vértice raíz, si $r.error = false$ y es vértice de ciclo y no tiene errores de ciclo, entonces todas sus tablas ya se encuentran correctamente llenadas, por lo que la solución global ya se encuentra completa. Si no es parte de un ciclo no necesita estas condiciones. El primer caso de la regla S verifica estas condiciones y asigna la solución a la tabla correspondiente, según el Teorema 3.1, y cambia la variable $r.completed$ a verdadero.

Para el caso de un vértice distinto de la raíz, sea p el vértice padre del vértice $v \neq r$. Si $p.completed = true$, entonces la solución ya se encuentra en p, ya que la variable $completed$ solamente se cambia en la regla S a verdadero, y siempre que se habiliten con la regla E, cambia a falso. Las condiciones del segundo caso de la regla S son similares al primer caso, pero con la adición de que el padre se encuentre completado, y las acciones que realiza son copiar la solución del vértice padre, y cambiar su propio estado a completado. ■

3.6 Corrección de errores

Detectar una falla transitoria en $\mathcal{Z}$ no es fácil por la cantidad de variables de los que depende la solución de alguna tabla. Por ejemplo, el cambio en una celda de la tabla de un vértice de ciclo puede afectar el resultado de la tabla siguiente, y de la siguiente, incluso hasta la mejor solución al ciclo, y posiblemente hasta la solución global. Encontrar el cambio, resultado de un fallo transitorio, necesita verificar que todas las tablas se encuentran llenas con información de acuerdo a las tablas de los vértices descendientes. Sin embargo, verificar todas las tablas requiere al menos la misma cantidad de tiempo que el algoritmo completo, por lo que no es eficiente. Una manera de resolver este problema es con la regla de error E, que contempla tres casos.

El primer caso de la regla verifica que las tablas de los vértices de ciclo se encuentren correctas y revisa todas las recursiones posibles en las tablas. En caso de encontrar algún error en alguna $h < current\_h$, regresa la variable $current\_h$ a la h con error pa-

ra que se corrija y se continúe con el proceso. El segundo caso verifica que las tablas de los cliqués se hayan llenado correctamente. De lo contrario se reestablece la variable $last\_child\_completed$ a 0 para que reinicie el proceso de cálculo de tablas. El restablecimiento de las variables permite calcular la solución correcta y que el cambio se propague correctamente hacia las siguientes tablas. El tercer caso verifica que los vértices de ciclo se encuentren numerados correctamente. Esta regla se relaciona con la regla NC, que se encarga de numerar a los vértices de ciclo, y con la función $correct\_numeration$, que se utiliza en todas las reglas de las tablas de ciclo.

Esta regla se diferencia de las reglas regulares en que no se verifican de forma tan frecuente como las demás, es decir, puede pasar más tiempo en que se verifiquen las reglas de error en cada vértice, para no sobrecargar el trabajo de cada vértice.

3.7 Cerradura

En este apartado se demuestra la cerradura del algoritmo, asegurando que al llegar a la solución en el vértice raíz y propagándola a los demás vértices, no hay vértice alguno privilegiado con alguna regla. En este caso, todas las tablas se encuentran correctamente procesadas, por lo que cualquier caso de la función $table\_completed$ resulta verdadero. La mayoría de las reglas requieren que alguna tabla no esté llena, mediante el argumento $\neg table\_completed$, y por la razón anterior, no se habilitan. Las reglas que no contienen esta expresión son: el tercer caso de PC, NC, AVC, AC, DV, PRC y S.

El tercer caso de la regla PC no se habilita porque $v.last\_child\_completed = |v.children|$, ya que todas las tablas hijo se han llenado. Entonces la función $table\_enabled$ devuelve un valor falso.

La regla de numeración NC solamente numera los vértices del ciclo de manera correcta, por lo que no se habilita.

La regla AVC asigna los vértices conflictivos, pero dichos vértices deben de estar correctos ya que no se modifican en ninguna otra regla. La función $conflictive\_vertices\_correct$ verifica ésto. Las reglas AC y DV tienen una función parecida a la regla anterior, ya que

su procedimiento realiza las acciones que verifica la función en la guardia.

La regla PRC contiene la condición $v.current\_h \leq v.cycle\_antecessor.cycle\_number$, y al haber calculado todas las recurrencias de los árboles del ciclo, $v.current\_h$ termina con un valor una unidad mayor a la numeración del último vértice, es decir, si el ciclo contiene k vértices, el último vértice está numerado $k-1$, y $v.current\_h = k$, por lo que tampoco se habilita.

Por último, la regla S requiere que los vértices tengan la variable $completed$ en falso, que no es posible ya que ésta se cambia al distribuir la solución. Por lo tanto, ninguna regla queda habilitada al llegar a la solución global y que dicha solución se distribuya a todos los vértices.

3.8 Análisis de complejidad

3.8.1. Espacio

Cada celda de una tabla contiene apuntadores hacia las celdas que se utilizaron para calcular la solución, un entero o punto flotante para el peso de la solución, y un entero que indica la distancia mínima hacia los vértices de dicha solución. La cantidad de apuntadores es $O(1)$, ya que la forma en la que se crea el árbol de expansión y el orden en que se calculan las tablas, cada bloque tiene a lo más dos bloques descendientes, por lo que cada celda utiliza $O(1)$ en almacenamiento, suponiendo que los apuntadores requieren $O(1)$ en memoria. Cada tabla tiene a lo más 12 celdas, por lo tanto cada tabla también necesita $O(1)$ espacio.

Para cada vértice v, las variables $v.children$ y $v.table\_children\_\{id\}$ dependen de la cantidad de hijos que tenga v, que es $O(n)$, y cada elemento tiene tamaño constante. Las demás variables que no son variables de ciclo necesitan $O(1)$ espacio.

La variable de ciclo $v.cycle\_neighbors$ depende de la cantidad de vecinos de ciclo que puede tener, en este caso solamente puede tener dos. Cada una de las variables $v.cycle\_error\_\{h\}$, $v.table\_cycle\_\{h\}$, $v.\{h\}\_vg$, $v.\{h\}\_vf$, $v.\{h\}\_vh$, $v.\{h\}\_v'g$, $v.\{h\}\_v'f$, $v.\{h\}\_v'h$ y $v.\{h\}\_conflict$ se utiliza para cada recursión de h y son $O(1)$. Hay a lo más tres

recursiones por renglón de tabla, entonces por cada tabla también es un orden constante. Hay $O(n)$ tablas distintas, una por cada arista removida para calcular los árboles, por lo que estas variables comprenden $O(n)$ de espacio.

En total, cada vértice necesita $O(n)$ de almacenamiento.

3.8.2. Tiempo

El algoritmo $\mathcal{Z}$ necesita que los algoritmos de elección de líder y de identificación de puentes se ejecuten previamente. Cada uno de estos algoritmos toma $O(n)$ rondas mediante el calendarizador adversario.

La ventaja del algoritmo $\mathcal{Z}$ es que los movimientos están diseñados para que se ejecuten en orden, así como lo realiza el algoritmo secuencial. Esto es, ningún vértice está habilitado para realizar alguna acción hasta que los vértices descendientes hayan procesado sus tablas (con excepción de las reglas de numeración de ciclo, ya que no dependen de ninguna tabla). Esto permite que el tiempo de ejecución sea igual al tiempo que toman todos los vértices en calcular sus tablas, que es equivalente al algoritmo $\mathcal{G}$, el cual toma $O(n^2)$ rondas utilizando un calendarizador distribuido síncrono, es decir, donde todos los nodos habilitados pueden realizar alguna acción por ronda.

Por lo tanto, el tiempo de ejecución total es de $O(n) + O(n) + O(n^2) = O(n^2)$ rondas en el calendarizador síncrono.

3.8.3. Recuperación de errores

Suponga que durante algún momento del algoritmo, L es el conjunto de vértices *legales*, es decir, que tengan hasta el momento todas sus variables y tablas con valores correctos. Sea $x \in L$ un vértice que sufre una falla transitoria y se recupera con valores arbitrarios. Las reglas de error detectan esta inconsistencia en el vértice x, así como en el vértice $x.parent$, por lo que se reinician algunas variables para volver a realizar los cálculos. Ambos deben recalcular sus tablas, pero durante el proceso, $x.parent$ cambia sus valores, por lo que $x.parent.parent$ podría detectar error, y propagarlo hacia arriba en

el árbol de expansión, potencialmente procesando de nuevo el grafo entero.

Por lo tanto, el tiempo de recuperación de errores en el algoritmo $\mathcal{Z}$ es $O(n^2)$ rondas.

3.9 Generalización a cactus

En $\mathcal{G}$ se diseña un algoritmo que funciona para un uniciclo, y se demuestra que dicho algoritmo genera una solución equivalente al marcado de un cactus, por lo que los ciclos y cliqués se procesan con las mismas instrucciones, sin realizar operaciones especiales por ciclos adicionales. En $\mathcal{Z}$ las reglas simulan el mismo comportamiento que los Pseudocódigos de $\mathcal{F}$, por lo que para funcionar en cactus solamente necesitarían pequeñas modificaciones:

- La numeración de los ciclos debe ser independiente, por lo que se tendría que identificar las aristas no puente que pertenecen a distintos ciclos. Esto es necesario cuando existen ciclos que comparten vértice, ya que dicho vértice tendría más de dos aristas no puente.
- Todas las operaciones del ciclo deben implementar alguna función que les indique si forman parte del mismo ciclo. Nuevamente, lo anterior es importante para el vértice en común de los ciclos, ya que puede tener varios vecinos de distintos ciclos.

Estas modificaciones no suponen gran cambio en las reglas, pero quizá sí trabajo manual de varias condiciones adicionales, por lo que quedan fuera del alcance de este trabajo.

Capítulo 4. Conclusiones

En este capítulo se presentan los resultados y conclusiones de este trabajo, las complicaciones encontradas durante la elaboración de los algoritmos, así como las propuestas de trabajo futuro.

Adaptar el algoritmo $\mathcal{F}$ para un cactus con peso en los vértices y encontrar el conjunto 2-packing con peso máximo fue sencillo y no involucró muchos cambios en el algoritmo. Lo anterior debido a que al usar programación dinámica en este problema se exploran todas las soluciones relevantes de forma exhaustiva, por lo que calcular las mejores soluciones parciales se realiza de la misma manera en el caso de vértices sin y con pesos. No todos los problemas con variantes con y sin peso admiten este tipo de soluciones, por ejemplo, si en el problema de la mochila (*Knapsack*) los objetos que se introducen tienen beneficio unitario, se puede encontrar una solución óptima mediante una metodología voraz en tiempo polinomial; cuando los beneficios de los objetos tienen valores distintos, se vuelve mucho más complicado.

En el caso del algoritmo $\mathcal{Z}$, más que diseñar un algoritmo desde cero, se adaptó el algoritmo anterior para que funcionara en el contexto distribuido. Se diseñaron reglas que realizan operaciones semejantes al algoritmo secuencial, y se agregaron funciones especiales para que el sistema pueda recuperarse en caso de que ocurra una falla transitoria y reconfigurarse para llegar a un estado global estable. Estas reglas deben contemplar los diferentes tipos de fallos que pueden ocurrir, como el cambio del valor de las variables por una falla transitoria, debido a por ejemplo, un sobrevoltaje.

4.1 Limitaciones

Se encontraron pocos artículos en la literatura de algoritmos auto-estabilizantes con soluciones óptimas y usando programación dinámica. En especial porque verificar que el estado actual de un vértice sea el correcto puede tomar mucho tiempo, por la cantidad de tablas que el algoritmo tiene que leer, y por la cantidad de vértices de los que depende, lo cual se puede ver en la descripción de la función $table\_completed$, lo que lleva al siguiente punto.

Se tuvo que elegir una compensación entre tiempo de recuperación y espacio en memoria en los vértices del ciclo. Al principio se había contemplado utilizar una sola tabla para todas las recursiones de los distintos árboles, pero sobreescribir los valores anteriores podría causar problemas de consistencia. Además si ocurre alguna falla transitoria en alguno de los árboles anteriores, no es posible recalcular la tabla sin procesar por completo el ciclo. Por esta razón se usó una tabla por cada recursión, lo que llevó a $O(n)$ de almacenamiento por vértice.

Además, la recuperación de errores no es óptima, ya que verificar que todas se encuentren en estado legal implica verificar todas las tablas, lo que lleva a una lenta corrección de errores.

La falta de un simulador de algoritmos auto-estabilizantes no permitió detectar errores fácilmente en el algoritmo $\mathcal{Z}$. En el caso del algoritmo secuencial, programarlo fue de gran ayuda para encontrar detalles que se omitieron durante el diseño del mismo. En el caso del algoritmo auto-estabilizante, la demostración se basa en $\mathcal{F}$.

4.2 Trabajo futuro

Como se comentó al final del Capítulo 3, se puede modificar el algoritmo $\mathcal{Z}$ para que funcione en un cactus. Aunque las reglas no necesiten tantos cambios, hay que tener cuidado en cómo se manejan las aristas de ciclo.

Siguiendo la última limitante, crear un simulador para algoritmos auto-estabilizantes sería de ayuda para los futuros algoritmos, así como para verificar que los existentes funcionen de manera correcta, o al menos de apoyo para encontrar errores.

Durante el proceso de cálculo del mejor árbol, se encontró que si se trabajaba con un calendarizador distribuido, muchas de las reglas se podían ejecutar al mismo tiempo, ya que si se ve de manera secuencial, solamente un vértice del ciclo trabaja durante una unidad de tiempo, mientras que los demás vértices esperan a que llegue su turno de procesar información. Debido a esto, es posible crear un *pipelining* de instrucciones, de tal manera que se ejecuten varias operaciones al mismo tiempo en diferentes vértices.

Por ejemplo, el cálculo de las tablas de las hojas del árbol del ciclo, puede ser simultáneo para diferentes aristas removidas, es decir, se podría calcular las hojas para $h = 2$, $h = 3$, $h = 4$... al mismo tiempo, ya que cada una de esas operaciones no depende de las otras, y se realizan en vértices distintos. Los vértices siguientes a ellos también serían distintos, por lo que se ejecutarían en la segunda ronda de instrucciones, y así sucesivamente. Faltaría hacer el análisis del tiempo de ejecución en tal caso.

Asimismo, el algoritmo $\mathcal{Z}$ puede funcionar para un calendarizador adversario, pero la forma de demostrar este tipo de calendarizador es más estricto que el actual, por lo que no es trivial hacer la conversión.

Se puede modificar la estructura del árbol para incluir solamente una cantidad de orden constante de apuntadores y evitar tener $O(h)$ de ellos, donde h es la cantidad de hijos que tiene un vértice en el árbol. Así, la complejidad espacial queda acotada por el tamaño del ciclo más grande. Sin embargo, esta modificación podría afectar algunas de las condiciones con la relación entre vértices padre e hijo, por lo que algunas reglas y funciones cambiarían.

Literatura citada

Arora, S., Lund, C., Motwani, R., Sudan, M., y Szegedy, M. (1998). Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, **45**(3): 501–555.

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

Datta, A. K., Larmore, L. L., y Vemula, P. (2011a). An o(n)-time self-stabilizing leader election algorithm. *J. Parallel Distrib. Comput.*, **71**(11): 1532–1544.

Datta, A. K., Larmore, L. L., y Vemula, P. (2011b). Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, **412**(40): 5541–5561.

Dijkstra, E. W. (1974). Self-stabilizing systems in spite of distributed control. *Communications of the 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*, May. pp. 747–754.

Dolev, S. (2000). *Self-stabilization*. MIT press.

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.

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

Garey, M. R. y Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co. New York, NY, USA.

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

Hedetniemi, S. M., Hedetniemi, S. T., Jacobs, D. P., y Srimani, P. K. (2003). Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers and Mathematics with Applications*, **46**(5-6): 805–812.

Karaata, M. H. y Chaudhuri, P. (1999). A self-stabilizing algorithm for bridge finding. *Distributed Computing*, **12**(1): 47–53.

Karp, R. M. (1972). *Reducibility among Combinatorial Problems*, pp. 85–103. Springer US, Boston, MA.

Manne, F. y Mjelde, M. (2006). A memory efficient self-stabilizing algorithm for maximal k-packing. En: *Symposium on Self-Stabilizing Systems*. Springer, 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).

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

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

Tixeuil, S. (2010). *Algorithms and Theory of Computation Handbook: general concepts and techniques*. Chapman & Hall, segunda edición. pp. 26.1–26.46.

Trejo-Sánchez, J. A. y Fernández-Zepeda, J. A. (2012). A self-stabilizing algorithm for the maximal 2-packing in a cactus graph. En: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, pp. 863–871.

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

Vazirani, V. V. (2001). *Approximation Algorithms*. Springer-Verlag. Berlin, Heidelberg.

Yen, L.-H., Huang, J.-Y., y Turau, V. (2016). Designing self-stabilizing systems using game theory. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, **11**(3): 18.

Apéndice A. Pseudocódigos del algoritmo $\mathcal{F}$

Pseudocódigo 1 ROOTED-BLOCK-TREE-POSTORDER(K)

Entrada: Un cactus conectado, no dirigido K.

Salida: Un árbol de bloques T_B enraizado en algún bloque arbitrario y una etiqueta en cada bloque que representa su numeración postorden.

1: **inicio**
2: Identificar los bloques de K mediante el algoritmo DFS
3: Elegir un bloque arbitrario B_x como el bloque raíz
4: Aplicar un algoritmo DFS iniciando en B_x para generar un árbol enraizado de bloques T_B
5: Etiquetar cada bloque de T_B usando su numeración en postorden
6: **regresar** T_B
7: **fin**

Pseudocódigo 2 INITIALIZE-BLOCK-TABLES(B_i, C_{B_i})

Entrada: Para cada vértice $v_{i,j} \in B_i$, $\mathcal{T}_{v_{s,0}}[r][2]$ de cada bloque $B_s \in C_{v_{i,j}}$, $\forall r$.
Salida: $\mathcal{T}_{v_{i,j}}[r][0], \forall r$ y $\forall j$.

inicio
para todo $v_{i,j} \in B_i$ **hacer**
  si $|C_{v_{i,j}}| = 0$ **entonces**
    Asignar a $\mathcal{T}_{v_{i,j}}[r][0]$ la configuración nula $\forall r$
  si no
    $\alpha_1 \leftarrow \{0, (\emptyset), \infty\}$
    $\alpha_0 \leftarrow \{0, (\emptyset), \infty\}$
    para todo $v_{s,0} \in C_{v_{i,j}}$ **hacer**
      $\alpha_1 \leftarrow \alpha_1 \bullet \mathcal{T}_{v_{s,0}}[2][2]$
      $\alpha_0 \leftarrow \alpha_0 \bullet \mathcal{T}_{v_{s,0}}[3][2]$
    fin para
    $\mathcal{T}_{v_{i,j}}[2][0] \leftarrow \alpha_1$
    si $j = 0$ **entonces**
      $\mathcal{T}_{v_{i,0}}[3][0] \leftarrow \alpha_0$
      $\mathcal{T}_{v_{i,0}}[2][0] \leftarrow \max(\mathcal{T}_{v_{i,0}}[2][0], \mathcal{T}_{v_{i,0}}[3][0])$
    fin si
    $\mathcal{T}_{v_{i,j}}[1][0] \leftarrow \max_{\forall B_s \in C_{v_{i,j}}}(\alpha_1 \circ \mathcal{T}_{v_{s,0}}[2][2] \bullet \mathcal{T}_{v_{s,0}}[1][2])$
    $\mathcal{T}_{v_{i,j}}[1][0] \leftarrow \max(\mathcal{T}_{v_{i,j}}[2][0], \mathcal{T}_{v_{i,j}}[1][0])$
    $\mathcal{T}_{v_{i,j}}[0][0] \leftarrow \alpha_0 \bullet \{1, (v_{i,j}), 0\}$
    $\mathcal{T}_{v_{i,j}}[0][0] \leftarrow \max(\mathcal{T}_{v_{i,j}}[1][0], \mathcal{T}_{v_{i,j}}[0][0])$
  fin si
fin para
regresar $\mathcal{T}_{v_{i,j}}[r][0], \forall r$ and $\forall j$
fin

Pseudocódigo 3 PROCESS-CLIQUE-BLOCK(B_i)

Entrada: Un clique de tamaño 2 del bloque B_i.
Salida: Un marcado $M_{i,0}[1]$ de $\mathcal{T}_{v_{i,0}}$.

inicio
para $r \leftarrow 2$ a 0 **hacer**
  $\mathcal{T}_{v_{i,1}}[r][1] \leftarrow \mathcal{T}_{v_{i,1}}[r][0]$
fin para
$\mathcal{T}_{v_{i,0}}[3][1] \leftarrow \mathcal{T}_{v_{i,0}}[3][0] \bullet \mathcal{T}_{v_{i,1}}[2][1]$
$\mathcal{T}_{v_{i,0}}[2][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,0}}[3][1])$
$\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][0] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,1}}[0][1])$
$\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][1], \mathcal{T}_{v_{i,0}}[2][1])$
$\mathcal{T}_{v_{i,0}}[0][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[0][0] \bullet \mathcal{T}_{v_{i,1}}[2][1], \mathcal{T}_{v_{i,0}}[1][1])$
regresar $\mathcal{T}_{v_{i,0}}[r][1], \forall r$
fin

Pseudocódigo 4 MAXIMUM-2-PACK-TREE(t_i^h, h)

Entrada: El árbol de expansión t_i^h de B_i. El índice h de la arista eliminada de B_i.
Salida: Un árbol actualizado t_i^h y un marcado $M_{i,0}^h[1]$.

inicio
si $h > 0$ **entonces**
  para $r \leftarrow 2$ a 0 **hacer**
    $\mathcal{T}_{v_{i,h}}[r][1] \leftarrow \mathcal{T}_{v_{i,h}}[r][0]$
  fin para
  para $k = h - 1$ a 1 **hacer**
    $\mathcal{T}_{v_{i,k}}[2][1] \leftarrow \mathcal{T}_{v_{i,k}}[2][0] \bullet \mathcal{T}_{v_{i,k+1}}[1][1]$
    $\mathcal{T}_{v_{i,k}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,k}}[1][0] \bullet \mathcal{T}_{v_{i,k+1}}[1][1], \mathcal{T}_{v_{i,k}}[2][0] \bullet \mathcal{T}_{v_{i,k+1}}[0][1])$
    $\mathcal{T}_{v_{i,k}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,k}}[1][1], \mathcal{T}_{v_{i,k}}[2][1])$
    $\mathcal{T}_{v_{i,k}}[0][1] \leftarrow \max(\mathcal{T}_{v_{i,k}}[0][0] \bullet \mathcal{T}_{v_{i,k+1}}[2][1], \mathcal{T}_{v_{i,k}}[1][1])$
  fin para
fin si
si $h < |B_i| - 1$ **entonces**
  para $r \leftarrow 2$ a 0 **hacer**
    $\mathcal{T}_{v_{i,h+1}}[r][1] \leftarrow \mathcal{T}_{v_{i,h+1}}[r][0]$
  fin para
  para $k = h + 2$ a $|B_i| - 1$ **hacer**
    $\mathcal{T}_{v_{i,k}}[2][1] \leftarrow \mathcal{T}_{v_{i,k}}[2][0] \bullet \mathcal{T}_{v_{i,k-1}}[1][1]$
    $\mathcal{T}_{v_{i,k}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,k}}[1][0] \bullet \mathcal{T}_{v_{i,k-1}}[1][1], \mathcal{T}_{v_{i,k}}[2][0] \bullet \mathcal{T}_{v_{i,k-1}}[0][1])$
    $\mathcal{T}_{v_{i,k}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,k}}[1][1], \mathcal{T}_{v_{i,k}}[2][1])$
    $\mathcal{T}_{v_{i,k}}[0][1] \leftarrow \max(\mathcal{T}_{v_{i,k}}[0][0] \bullet \mathcal{T}_{v_{i,k-1}}[2][1], \mathcal{T}_{v_{i,k}}[1][1])$
  fin para
fin si
$\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ PROCESS-SOURCE-VERTEX(t_i^h, h)
regresar t_i^h
fin

Pseudocódigo 5 PROCESS-SOURCE-VERTEX(t_i^h, h)

Entrada: El árbol de expansión t_i^h de B_i. El índice h de la arista eliminada de B_i.
Salida: Un marcado $M_{i,0}^h[1]$.

inicio
$d \leftarrow |B_i| - 1$
si $h = 0$ **entonces**
  $\mathcal{T}_{v_{i,0}}[3][1] \leftarrow \mathcal{T}_{v_{i,0}}[3][0] \bullet \mathcal{T}_{v_{i,d}}[2][1]$
  $\mathcal{T}_{v_{i,0}}[2][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,d}}[1][1], \mathcal{T}_{v_{i,0}}[3][1])$
  $\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][0] \bullet \mathcal{T}_{v_{i,d}}[1][1], \mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,d}}[0][1])$
  $\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][1], \mathcal{T}_{v_{i,0}}[2][1])$
  $\mathcal{T}_{v_{i,0}}[0][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[0][0] \bullet \mathcal{T}_{v_{i,d}}[2][1], \mathcal{T}_{v_{i,0}}[1][1])$
fin si
si $h = d$ **entonces**
  $\mathcal{T}_{v_{i,0}}[3][1] \leftarrow \mathcal{T}_{v_{i,0}}[3][0] \bullet \mathcal{T}_{v_{i,1}}[2][1]$
  $\mathcal{T}_{v_{i,0}}[2][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,0}}[3][1])$
  $\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][0] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,1}}[0][1])$
  $\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][1], \mathcal{T}_{v_{i,0}}[2][1])$
  $\mathcal{T}_{v_{i,0}}[0][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[0][0] \bullet \mathcal{T}_{v_{i,1}}[2][1], \mathcal{T}_{v_{i,0}}[1][1])$
fin si
si $h \neq 0 \wedge h \neq d$ **entonces**
  $\mathcal{T}_{v_{i,0}}[3][1] \leftarrow \mathcal{T}_{v_{i,0}}[3][0] \bullet \mathcal{T}_{v_{i,d}}[2][1] \bullet \mathcal{T}_{v_{i,1}}[2][1]$
  $\mathcal{T}_{v_{i,0}}[2][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[2][0] \bullet \mathcal{T}_{v_{i,d}}[1][1] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,0}}[3][1])$
  $\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \mathcal{T}_{v_{i,0}}[2][0] \bullet \max(\mathcal{T}_{v_{i,d}}[0][1] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,d}}[1][1] \bullet \mathcal{T}_{v_{i,1}}[0][1])$
  $\mathcal{T}_{v_{i,0}}[1][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}[1][0] \bullet \mathcal{T}_{v_{i,d}}[1][1] \bullet \mathcal{T}_{v_{i,1}}[1][1], \mathcal{T}_{v_{i,0}}[1][1], \mathcal{T}_{v_{i,0}}[2][1])$
  $\mathcal{T}_{v_{i,0}}[0][1] \leftarrow \max(\mathcal{T}_{v_{i,d}}[2][1] \bullet \mathcal{T}_{v_{i,1}}[2][1] \bullet \mathcal{T}_{v_{i,0}}[0][0], \mathcal{T}_{v_{i,0}}[1][1])$
fin si
regresar $\mathcal{T}_{v_{i,0}}[r][1], \forall r$
fin

Pseudocódigo 6 MAXIMUM-2-PACK-UNICYCLE(U)

Entrada: Un grafo uniciclo no dirigido U.
Salida: Un marcado $M_{N,0}[2]$ de U que representa un conjunto 2-packing máximo.

inicio
$T_B \leftarrow$ROOTED-BLOCK-TREE-POSTORDER(U)
para $i \leftarrow 1$ a $N - 1$ **hacer**
  $\mathcal{T}_{v_{i,j}}[r][0] \leftarrow$INITIALIZE-BLOCK-TABLES($B_i, C_{B_i}$), $\forall j$
  $\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$PROCESS-CLIQUE-BLOCK($B_i$)
  $\mathcal{T}_{v_{i,0}}[r][2] \leftarrow \mathcal{T}_{v_{i,0}}[r][1]$
fin para
$\mathcal{T}_{v_{N,j}}[r][0] \leftarrow$INITIALIZE-BLOCK-TABLES($B_N, C_{B_N}$), $\forall j$
$\mathcal{T}_{v_{N,0}}[r][2] \leftarrow$PROCESS-CYCLIC-BLOCK($B_N$)
$U \leftarrow$ DISTRIBUTE-MAXIMUM-2-PACK-SOLUTION($U, \mathcal{T}_{v_{N,0}}[0][2]$)
regresar U
fin

Pseudocódigo 7 PROCESS-CYCLIC-BLOCK(B_i)

Entrada: Un bloque cíclico B_i cuyo vértice origen es $v_{i,0}$.
Salida: Un marcado $M_{i,0}[2]$.

inicio
para $h = 0$ a $|B_i| - 1$ **hacer**
  $t_i^h \leftarrow B_i \setminus e_{i,h}$
  $t_i^h \leftarrow$ MAXIMUM-2-PACK-TREE(t_i^h, h)
  $\mathcal{T}^h_{v_{i,0}}[r][1] \leftarrow$ ADAPT-FOR-UNICYCLE(t_i^h, h)
fin para
$max\_sol \leftarrow \mathcal{T}^0_{v_{i,0}}[r][1]$
para $h = 1$ a $|B_i| - 1$ **hacer**
  $max\_sol \leftarrow \max(max\_sol, \mathcal{T}^h_{v_{i,0}}[r][1]), \forall r$
fin para
$\mathcal{T}_{v_{i,0}}[r][2] \leftarrow max\_sol$
regresar $\mathcal{T}_{v_{i,0}}[r][2], \forall r$
fin

Pseudocódigo 8 ADAPT-FOR-UNICYCLE(t_i^h, h)

Entrada: El árbol de expansión t_i^h de B_i. El índice h de la arista eliminada de B_i.
Salida: Un marcado $M^h_{i,0}[1]$.

inicio
$v'_f \leftarrow v_{i,h-1 \bmod |B_i|}$
$v'_g \leftarrow v_{i,h}$
$v_g \leftarrow v_{i,(h+1) \bmod |B_i|}$
$v_f \leftarrow v_{i,(h+2) \bmod |B_i|}$
si $(v_g \wedge v'_h) \in \mathcal{T}_{v_{i,0}}[0][1].set \mid v'_h \in C_{v'_g} \wedge dist(v'_g, v'_h) = 1$ **entonces**
  $\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ CORRECT-CONFLICT(t_i^h, h, v_g, v'_h)
si no
  si $(v_h \wedge v'_g) \in \mathcal{T}_{v_{i,0}}[0][1].set \mid v_h \in C_{v_g} \wedge dist(v_g, v_h) = 1$ **entonces**
    $\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ CORRECT-CONFLICT(t_i^h, h, v_h, v'_g)
  si no
    si $(v_g \wedge v'_g) \in \mathcal{T}_{v_{i,0}}[0][1].set$ **entonces**
      $\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ CORRECT-CONFLICT(t_i^h, h, v_g, v'_g)
    si no
      si $(v_g \wedge v'_f) \in \mathcal{T}_{v_{i,0}}[0][1].set$ **entonces**
        $\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ CORRECT-CONFLICT(t_i^h, h, v_g, v'_f)
      si no
        si $(v_f \wedge v'_g) \in \mathcal{T}_{v_{i,0}}[0][1].set$ **entonces**
          $\mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ CORRECT-CONFLICT(t_i^h, h, v_f, v'_g)
        fin si
      fin si
    fin si
fin si
regresar $\mathcal{T}_{v_{i,0}}[r][1], \forall r$
fin

Pseudocódigo 9 CORRECT-CONFLICT(t_i^h, h, v, v')

Entrada: Un árbol de expansión t_i^h de B_i, el índice h, y dos vértices conflictivos v y v'.
Salida: Un marcado libre de conflictos $M_{i,0}^h[1]$.

inicio
$t^1 \leftarrow t_i^h$
$t^2 \leftarrow t_i^h$
si $v \in t^1$ **entonces**
  $\mathcal{T}_v^1[0][0] \leftarrow \mathcal{T}_v^1[0][0] \circ \{1, (v), 0\}$
  $\mathcal{T}_v^1[0][0] \leftarrow \max(\mathcal{T}_v^1[0][0], \mathcal{T}_v^1[1][0])$
si no
  si $\mathcal{T}_{p(v)}^1[0][0] = \mathcal{T}_{p(v)}^1[1][0]$ **entonces**
    $\mathcal{T}_{p(v)}^1[0][0] \leftarrow \mathcal{T}_{p(v)}^1[2][0]$
  fin si
  $\mathcal{T}_{p(v)}^1[1][0] \leftarrow \mathcal{T}_{p(v)}^1[2][0]$
  $\mathcal{T}_{p(v)}^1[0][0] \leftarrow \max(\mathcal{T}_{p(v)}^1[1][0], \mathcal{T}_{p(v)}^1[0][0])$
fin si
$t^1 \leftarrow$ MAXIMUM-2-PACK-TREE(t^1, h)
$\mathcal{T}_{v_{i,0}}^1[r][1] \leftarrow$ ADAPT-FOR-UNICYCLE(t^1, h)
si $v' \in t^2$ **entonces**
  $\mathcal{T}_{v'}^2[0][0] \leftarrow \mathcal{T}_{v'}^2[0][0] \circ \{1, (v'), 0\}$
  $\mathcal{T}_{v'}^2[0][0] \leftarrow \max(\mathcal{T}_{v'}^2[0][0], \mathcal{T}_{v'}^2[1][0])$
si no
  si $\mathcal{T}_{p(v')}^2[0][0] = \mathcal{T}_{p(v')}^2[1][0]$ **entonces**
    $\mathcal{T}_{p(v')}^2[0][0] \leftarrow \mathcal{T}_{p(v')}^2[2][0]$
  fin si
  $\mathcal{T}_{p(v')}^2[1][0] \leftarrow \mathcal{T}_{p(v')}^2[2][0]$
  $\mathcal{T}_{p(v')}^2[0][0] \leftarrow \max(\mathcal{T}_{p(v')}^2[1][0], \mathcal{T}_{p(v')}^2[0][0])$
fin si
$t^2 \leftarrow$ MAXIMUM-2-PACK-TREE(t^2, h)
$\mathcal{T}_{v_{i,0}}^2[r][1] \leftarrow$ ADAPT-FOR-UNICYCLE(t^2, h)
$\mathcal{T}_{v_{i,0}}[r][1] \leftarrow \max(\mathcal{T}_{v_{i,0}}^1[r][1], \mathcal{T}_{v_{i,0}}^2[r][1])$
regresar $\mathcal{T}_{v_{i,0}}[r][1], \forall r$
fin

Pseudocódigo 10 UNICYCLE-TO-TREE(U)

Entrada: Un grafo uniciclo no dirigido U enraizado en $v_{N,0}$ del bloque cíclico B_N y un marcado $M_{N,0}[2]$ de U que representa un conjunto 2-packing máximo.
Salida: Un árbol $T(U)$ con un marcado $M_{N,0}[2] = M_{N,0}[1] = \mathcal{M}_{N,0}[1]$, el cual representa un conjunto 2-packing máximo.

1: **inicio**
2: Aplicar BFS($U, v_{N,0}$) e identificar $v_x, p(v_x), v_z, v_y$, la arista e_1
3: $T(U) \leftarrow U$
4: Borrar todas las aristas de P_1 en $T(U)$
5: Agregar una arista entre cada vértice $f(v_{N,m})$ a $f(p(v_x))$, tal que $v_{N,m} \in P_1$ en U, $v_{N,m} \neq v_y$, y $v_{N,m} \neq v_z$
6: **regresar** $T(U)$
7: **fin**

Pseudocódigo 11 MAXIMUM-2-PACK-CACTUS(K)

Entrada: Un grafo cactus conectado $K = (V_K, E_K)$, donde $|V_K| \geq 2$.
Salida: Un conjunto $S_{N,0}[2] \subseteq V_K$ que representa un conjunto 2-packing máximo para K.

1: **inicio**
2: $T_B \leftarrow$ROOTED-BLOCK-TREE-POSTORDER(K)
3: **para** $i \leftarrow 1$ a N **hacer**
4: $\quad \mathcal{T}_{v_{i,j}}[r][0] \leftarrow$ INITIALIZE-BLOCK-TABLES(B_i, C_{B_i}), $\forall j$
5: $\quad$ **si** B_i es un bloque cliqué **entonces**
6: $\quad\quad \mathcal{T}_{v_{i,0}}[r][1] \leftarrow$ PROCESS-CLIQUE-BLOCK(B_i)
7: $\quad\quad \mathcal{T}_{v_{i,0}}[r][2] \leftarrow \mathcal{T}_{v_{i,0}}[r][1]$
8: $\quad$ **fin si**
9: $\quad$ **si** B_i es un bloque ciclo **entonces**
10: $\quad\quad \mathcal{T}_{v_{i,0}}[r][2] \leftarrow$ PROCESS-CYCLIC-BLOCK(B_i)
11: $\quad$ **fin si**
12: **fin para**
13: $K \leftarrow$ DISTRIBUTE-MAXIMUM-2-PACK-SOLUTION($K, \mathcal{T}_{v_{N,0}}[0][2]$)
14: **regresar** K
15: **fin**

Apéndice B. Funciones del algoritmo $\mathcal{Z}$

- $error(v)$:

  return $\exists t \in v.children$ tal que $\neg table\_completed(v.table\_children\{t\}) \vee \neg table\_completed(v.table\_up)$

- $cycle\_error(v, h)$:

  return $\neg table\_completed(v.table\_cycle\_\{h\})$

- $is\_cycle\_root(v)$:

  return $v.cycle\_vertex \wedge (v.parent = null \vee is\_bridge(v, v.parent))$

- $is\_leaf(v)$:

  return $\neg is\_cycle\_vertex(v) \wedge v.children = \emptyset$

- $table\_enabled(v)$:

  if $v.last\_child\_completed = |v.children|$ **then** return false
  return $table\_completed(v.childred[v.last\_child\_completed + 1].table\_up)$

- $children\_blocks\_completed(v)$:

  return $v.last\_child\_completed = |v.children|$

- $correct\_numeration(v)$:

  return $(is\_cycle\_root(v) \wedge v.cycle\_number = 0 \wedge v.cycle\_successor =$ id del vecino con menor id en $v.cycle\_neighbors \wedge v.cycle\_antecessor =$ id del otro vecino en $v.cycle\_neighbors) \vee (\neg is\_cycle\_root(v) \wedge v.cycle\_vertex = true \wedge v.cycle\_number \neq 0 \wedge \neg v.cycle\_antecessor.numeration\_error \wedge v.cycle\_antecessor.vertex\_number = v.vertex\_number - 1 \wedge v.cycle\_successor =$ id del otro vecino diferente de $v.cycle\_antecessor$ en $v.cycle\_neighbors$)

- $antecessor\_correct(v)$:

  return $(\exists! w \in v.cycle\_neighbors$ tal que $correct\_numeration(w) \wedge w.cycle\_successor = v.id)$

- $conflictive\_vertices\_correct(v, h)$:

  return $(v.vertex\_number = h \wedge v.\{h\}\_vf = v.cycle\_antecessor \wedge v.\{h\}\_vg = v \wedge v.\{h\}\_vh =$ vértice marcado en $v.table\_cycle\_\{h\}$ a distancia uno$) \vee$ $(v.vertex\_number = h + 1 \wedge v.\{h\}\_v'f = v.cycle\_successor \wedge v.\{h\}\_v'g = v \wedge v.\{h\}\_v'h =$ vértice marcado en $v.table\_cycle\_\{h\}$ a distancia uno$)$

- $check\_conflict\_left(v, h)$:

  if $v.h\_vg \wedge v.successor.h\_v'h$ **then** return 1
  else if $v.h\_vh \wedge v.successor.h\_v'g$ **then** return 2
  else if $v.h\_vg \wedge v.successor.h\_v'g$ **then** return 3
  else if $v.h\_vg \wedge v.successor.h\_v'f$ **then** return 4
  else if $v.h\_vf \wedge v.successor.h\_v'g$ **then** return 5
  else return 0

- $check\_conflict\_right(v, h)$:

  if $v.h\_v'h \wedge v.antecessor.h\_vg$ **then** return 1
  else if $v.h\_v'g \wedge v.antecessor.h\_vh$ **then** return 2
  else if $v.h\_v'g \wedge v.antecessor.h\_vg$ **then** return 3
  else if $v.h\_v'f \wedge v.antecessor.h\_vg$ **then** return 4
  else if $v.h\_v'g \wedge v.antecessor.h\_vf$ **then** return 5
  else return 0

- $vertex\_conflict(v, h)$:

  if $(v.cycle\_number = h \vee v.cycle\_number = h + 1)$ **then** return $v.\{h\}\_conflict$
  else if $v.cycle\_number < h$ **then** return $v.cycle\_antecessor.\{h\}\_conflict$
  else if $v.cycle\_number > h + 1$ **then** return $v.cycle\_successor.\{h\}\_conflict$

- $special\_case(v, h, conflict)$:

  return $(h = 0 \wedge conflict = 5 \wedge v.cycle\_successor.cycle\_number = 0) \vee (v.cycle\_number = 1 \wedge conflict = 4 \wedge v.cycle\_antecessor.cycle\_antecessor.cycle\_number)$

- $has\_to\_deactivate(v, h, conflict)$:

  return $(special\_case(v, h, conflict)) \vee$
  $(v.cycle\_number = h \wedge conflictive\_vertices\_correct(v, h) \wedge conflict = 1, 3, 4) \vee$
  $(v.cycle\_number = h \wedge conflictive\_vertices\_correct(v, h) \wedge conflict = 2) \vee$
  $(v.cycle\_number = h - 1 \wedge conflict = 5) \vee$
  $(v.cycle\_number = h + 1 \wedge conflictive\_vertices\_correct(v, h) \wedge conflict = 2, 3, 5) \vee$
  $(v.cycle\_number = h + 1 \wedge conflictive\_vertices\_correct(v, h) \wedge conflict = 1) \vee$
  $(v.cycle\_number = h + 2 \wedge conflict = 4)$

- $vertex\_deactivated(v, h, conflict)$:

  return $(h = 0 \wedge conflict = 5 \wedge v.cycle\_successor.cycle\_number = 0 \wedge$
  v está desactivado en $v.table\_cycle\_\{h\}\_D) \vee$
  $(v.cycle\_number = 1 \wedge conflict = 4 \wedge$
  $h = v.cycle\_antecessor.cycle\_antecessor.cycle\_number \wedge$
  v está desactivado en $v.table\_cycle\_\{h\}\_I) \vee$
  $((\neg special\_case(v, h, conflict) \wedge$
  $((v.cycle\_number = h \wedge conflictive\_vertices\_correct(v, h) \wedge conflict = 1, 3, 4 \wedge$
  $v.\{h\}\_vg$ está desactivado en $v.table\_cycle\_\{h\}\_I) \vee$

$(v.cycle\_number = h \wedge conflictive\_vertices\_correct(v,h) \wedge conflict = 2 \wedge$
$v.\{h\}\_vh$ está desactivado en $v.table\_cycle\_\{h\}\_I) \vee$
$(v.cycle\_number = h-1 \wedge conflict = 5 \wedge v$ está desactivado en $v.table\_cycle\_\{h\}\_I) \vee$
$(v.cycle\_number = h+1 \wedge conflictive\_vertices\_correct(v,h) \wedge conflict = 2,3,5 \wedge$
$v.\{h\}\_v'g$ está desactivado en $v.table\_cycle\_\{h\}\_D) \vee$
$(v.cycle\_number = h+1 \wedge conflictive\_vertices\_correct(v,h) \wedge conflict = 1 \wedge$
$v.\{h\}\_v'h$ está desactivado en $v.table\_cycle\_\{h\}\_D) \vee$
$(v.cycle\_number = h+2 \wedge conflict = 4 \wedge v$ está desactivado en $v.table\_cycle\_\{h\}\_D)))$

- $conflict\_correct(v)$:

  return $(v.cycle\_number = v.current\_h \wedge$
  $v.\{vcurrent\_h\}\_conflict = check\_conflict\_left(v, v.current\_h)) \vee$
  $(v.cycle\_number = v.current\_h + 1 \wedge$
  $v.\{v.current\_h\}\_conflict = check\_conflict\_right(v, v.current\_h)) \vee$
  $(v.cycle\_number < v.current\_h \wedge$
  $table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I)) \vee$
  $(v.cycle\_number > v.current\_h + 1 \wedge$
  $table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D))$

Apéndice C. Reglas del algoritmo $\mathcal{Z}$

E:

if $((cycle\_error(v,h) \neq v.cycle\_error\_\{h\}) \vee$
$(cycle\_error(v,h) \neq v.cycle\_error\_\{h\}) \vee$
$(correct\_numeration(v) = v.numeration\_error))$ **then**
{

if $cycle\_error(v,h) \neq v.cycle\_error\_\{h\}$ **then**
$v.cycle\_error\_\{h\} \leftarrow cycle\_error(v,h)$
if $v.cycle\_error\_\{h\} = true$ **then**
if $v.current\_h > h$ **then**
$v.current\_h \leftarrow h$
if $v.current\_tree > h$ **then**
$v.current\_tree \leftarrow h$
$v.completed \leftarrow false$
else if $cycle\_error(v,h) \neq v.cycle\_error\_\{h\}$ **then**
$v.error \leftarrow error(v)$
if $v.error = true$ **then**
$k \leftarrow$ índice del primer hijo con error
if $v.last\_child\_completed > k$ **then**
$v.last\_child\_completed \leftarrow k$
$v.current\_tree \leftarrow 0$
$v.completed \leftarrow false$
else if $correct\_numeration(v) = v.numeration\_error$ **then**
$v.numeration\_error \leftarrow \neg correct\_numeration(v)$
}

PC:

if $(is\_leaf(v) \wedge \neg table\_completed(v.table\_up)) \vee$
$(\neg is\_leaf(v) \wedge v.cycle\_vertex = false \wedge is\_bridge(v, v.parent) \wedge$
$\neg table\_completed(v.table\_up) \wedge children\_blocks\_completed(v)) \vee$
$(\neg is\_leaf(v) \wedge |v.children| \neq 0 \wedge table\_enabled(v))$ **then**
{

if $is\_leaf(v) \wedge \neg table\_completed(v.table\_up)$ **then**
$v.table\_up \leftarrow$ *null configuration*
$v.table\_up$ columna $1 \leftarrow v.table\_up$ columna 0
$v.error \leftarrow false$
else if $(\neg is\_leaf(v) \wedge v.cycle\_vertex = false \wedge is\_bridge(v, v.parent) \wedge$
$\neg table\_completed(v.table\_up) \wedge children\_blocks\_completed(v))$ **then**
Calcular columna 0 de $v.table\_up$ con el Pseudocódigo 2
Copiar columna 0 a columna 1 (Pseudocódigo 3)
$v.error \leftarrow false$
else if $(\neg is\_leaf(v) \wedge |v.children| \neq 0 \wedge table\_enabled(v))$
Calcular $v.table\_children\_\{v.children[v.last\_child\_completed + 1]\}$
columna 0 de acuerdo a Pseudocódigo 2
Calcular columna 1 de acuerdo a Pseudocódigo 3
Copiar columna 1 a columna 2
$v.last\_child\_completed \leftarrow v.last\_child\_completed + 1$
}

NC: **if** $(is\_cycle\_root(v) \wedge \neg correct\_numeration(v)) \vee$
$(v.cycle\_vertex \wedge \neg is\_cycle\_root(v) \wedge \neg correct\_numeration(v) \wedge$
$antecessor\_correct(v))$ **then**
{

if $is\_cycle\_root(v)$ **then**
$v.cycle\_number \leftarrow 0$
$v.cycle\_successor \leftarrow \text{máx}_{w.id}[w \in v.cycle\_neighbors]$
$v.cycle\_antecessor \leftarrow \text{mín}_{w.id}[w \in v.cycle\_neighbors]$
$v.current\_tree \leftarrow -1$
else
Sea w el vecino en $v.cycle\_neighbors$ cuyo $w.cycle\_successor = v.id$
y x el otro vecino
$v.cycle\_number \leftarrow w.cycle\_number + 1$
$v.cycle\_successor \leftarrow x.id$
$v.cycle\_antecessor \leftarrow w.id$
}

IC: **if** $v.cycle\_vertex \wedge \neg table\_completed(v.table\_up)$ columna $0 \wedge$
$children\_blocks\_completed(v)$ **then**
{

Calcular columna 0 de $v.table\_up$ de acuerdo al Pseudocódigo 2
$v.current\_tree \leftarrow 0$
$v.current\_h \leftarrow h0\_r3$
}

M2PT: **if** $v.cycle\_vertex \wedge \neg is\_cycle\_root(v) \wedge correct\_numeration(v) \wedge$
$table\_completed(v.table\_up[columna\,0]) \wedge$
$\neg table\_completed(v.table\_up\_\{v.current\_tree\}[columna\,1]) \wedge$
$((v.current\_tree < v.cycle\_number - 1 \wedge$
$table\_completed(v.cycle\_antecessor.table\_up\_\{v.current\_tree\}[columna\,1])) \vee$
$(v.current\_tree = v.cycle\_number - 1) \vee$
$(v.current\_tree = v.cycle\_number) \vee$
$(v.current\_tree > v.cycle\_number \wedge$
$table\_completed(v.cycle\_successor.table\_up\_\{v.current\_tree\}[columna\,1]))$ **then**
{

if $v.current\_tree < v.cycle\_number - 1$ **then**
Calcular la columna 1 de la tabla $v.table\_up\_\{v.current\_tree\}$
de acuerdo al Pseudocódigo 4, líneas 17-22
else if $v.current\_tree = v.cycle\_number - 1$ **then**
Calcular la columna 1 de la tabla $v.table\_up\_\{v.current\_tree\}$
copiando la columna 0, Pseudocódigo 4 líneas 14-16.

else if $v.current\_tree = v.cycle\_number$ **then**
Calcular la columna 1 de la tabla $v.table\_up\_\{v.current\_tree\}$
copiando la columna 0, Pseudocódigo 4 líneas 3-5.
else
Calcular la columna 1 de la tabla $v.table\_up\_\{v.current\_tree\}$
de acuerdo al Pseudocódigo 4, líneas 6-11
$v.current\_tree \leftarrow v.current\_tree + 1$
}

PR: **if** $is\_cycle\_root(v) \wedge correct\_numeration(v) \wedge$
$\neg table\_completed(v.table\_up\_\{v.current\_tree\}[columna1]) \wedge$
$table\_completed(v.cycle\_antecessor.table\_up\_\{v.current\_tree\}[columna1]) \wedge$
$table\_completed(v.cycle\_successor.table\_up\_\{v.current\_tree\}[columna1])$ **then**
{
Calcular la columna 1 de la tabla $v.table\_up\_\{v.current\_tree\}$
de acuerdo al Pseudocódigo 5, líneas 1-25
}

AU: **if** $v.cycle\_vertex \wedge \neg is\_cycle\_root(v) \wedge correct\_numeration(v) \wedge$
$\neg table\_completed(v.table\_cycle\_\{v.current\_h\}) \wedge$
$((v.cycle\_number \leq v.current\_h \wedge$
$table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\})) \vee$
$(v.cycle\_number > v.current\_h \wedge$
$table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}))$ **then**
{
if $v.cycle\_number \leq v.current\_h$ **then**
Indicar cuál es la celda correspondiente a v que usa la solución
de la raíz, dependiendo de la celda del padre (antecesor)
$v.current\_h \leftarrow v.cycle\_antecessor.current\_h$
else
Indicar cuál es la celda correspondiente a v que usa la solución
de la raíz, dependiendo de la celda del padre (sucesor)
$v.current\_h \leftarrow v.cycle\_successor.current\_h$
$v.cycle\_error\_\{h\} \leftarrow false$
}

AVC: **if** $v.cycle\_vertex \wedge correct\_numeration(v) \wedge (v.cycle\_number = v.current\_h \vee$
$v.cycle\_number = v.current\_h + 1) \wedge table\_completed(v.table\_cycle\_\{v.current\_h\})$
$\wedge \neg conflictive\_vertices\_correct(v, v.current\_h)$ **then**
{
Asignar los vertices conflictivos como en el Pseudocódigo 8
}

AC: **if** $v.cycle\_vertex \wedge correct\_numeration(v) \wedge$
$conflictive\_vertices\_correct(v, v.current\_h) \wedge$
$((v.cycle\_number = v.current\_h \wedge$
$conflictive\_vertices\_correct(v.cycle\_successor, v.current\_h) \wedge$
$v.\{current\_h\}\_conflict \neq check\_conflict\_left(v, v.current\_h)) \vee$
$(v.cycle\_number = v.current\_h + 1 \wedge$
$conflictive\_vertices\_correct(v.cycle\_antecessor, v.current\_h) \wedge$
$v.\{current\_h\}\_conflict \neq check\_conflict\_right(v, v.current\_h)))$ **then**
{

if $v.cycle\_number = v.current\_h$ **then**
$v.\{current\_h\}\_conflict = check\_conflict\_left(v, v.current\_h)$
else
$v.\{current\_h\}\_conflict = check\_conflict\_right(v, v.current\_h)$
}

DV: **if** $v.cycle\_vertex \wedge correct\_numeration(v) \wedge conflict\_correct(v) \wedge$
$has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)) \wedge$
$\neg vertex\_deactivated(v, v.current\_h, vertex\_conflict(v, v.current\_h))$ **then**
{

Desactivar el vértice de manera similar a la función $vertex\_deactivated$
}

CC: **if** $v.cycle\_vertex \wedge \neg is\_cycle\_root(v) \wedge$
$correct\_numeration(v) \wedge conflict\_correct(v) \wedge$
$((has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)) \wedge$
$\neg vertex\_deactivated(v, v.current\_h, vertex\_conflict(v, v.current\_h))) \vee$
$(\neg has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)))) \wedge$
$((v.cycle\_number > v.current\_h + 1 \wedge$
$\neg table\_completed(v.table\_cycle\_\{v.current\_h\}\_D) \wedge$
$table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D)) \vee$
$(v.cycle\_number = v.current\_h + 1 \wedge$
$\neg table\_completed(v.table\_cycle\_\{v.current\_h\}\_D)) \vee$
$(v.cycle\_number = v.current\_h \wedge$
$\neg table\_completed(v.table\_cycle\_\{v.current\_h\}\_I)) \vee$
$(v.cycle\_number < v.current\_h \wedge$
$\neg table\_completed(v.table\_cycle\_\{v.current\_h\}\_I) \wedge$
$table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I)))$ **then**
{

if $v.cycle\_number > v.current\_h + 1$ **then**
Calcular $v.table\_cycle\_\{v.current\_h\}\_I$ igual que primer if de
$M2PT$ sin incrementar $v.current\_tree$
$v.\{current\_h\}\_conflict = v.cycle\_antecessor.\{v.current\_h\}\_conflict$
else if $v.cycle\_number = v.current\_h + 1$ **then**
Calcular $v.table\_cycle\_\{v.current\_h\}\_I$ igual que segundo if de

$M2PT$ sin incrementar $v.current\_tree$
else if $v.cycle\_number = v.current\_h$ **then**
Calcular $v.table\_cycle\_\{v.current\_h\}\_I$ igual que tercer if de
$M2PT$ sin incrementar $v.current\_tree$
else if $v.cycle\_number < v.current\_h$ **then**
Calcular $v.table\_cycle\_\{v.current\_h\}\_I$ igual que cuarto if de
$M2PT$ sin incrementar $v.current\_tree$
$v.\{current\_h\}\_conflict = v.cycle\_successor.\{v.current\_h\}\_conflict$
}

PRC: **if** $is\_cycle\_root(v) \wedge correct\_numeration(v) \wedge conflict\_correct(v) \wedge$
$v.current\_h \leq v.cycle\_antecessor.cycle\_number \wedge$
$correct\_numeration(v.cycle\_antecessor) \wedge$
$(((((v.current\_h \neq 0 \wedge v.current\_h \neq v.cycle\_antecessor.cycle\_number) \wedge$
$v.cycle\_successor.\{v.current\_h\}\_conflict = 0 \wedge$
$table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I) \wedge$
$v.cycle\_antecessor.\{v.current\_h\}\_conflict = 0 \wedge$
$table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D)) \vee$
$v.current\_h = 0 \wedge v.\{v.current\_h\}\_conflict = 0 \wedge$
$v.cycle\_antecessor.\{v.current\_h\}\_conflict = 0 \wedge$
$table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D)) \vee$
$v.current\_h = v.cycle\_antecessor.cycle\_number \wedge v.\{v.current\_h\}\_conflict = 0 \wedge$
$v.cycle\_successor.\{v.current\_h\}\_conflict = 0 \wedge$
$table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I)) \vee$
$(v.current\_h \neq 0 \wedge v.current\_h \neq v.cycle\_antecessor.cycle\_number \wedge$
$v.cycle\_successor.\{v.current\_h\}\_conflict \neq 0 \wedge$
$table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I) \wedge$
$v.cycle\_antecessor.\{v.current\_h\}\_conflict \neq 0 \wedge$
$table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D) \wedge$
$v.cycle\_successor.\{v.current\_h\}\_conflict =$
$v.cycle\_antecessor.\{v.current\_h\}\_conflict \wedge$
$((has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)) \wedge$
$vertex\_deactivated(v, v.current\_h, vertex\_conflict(v, v.current\_h))) \vee$
$(\neg has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h))))) \vee$
$(v.current\_h = 0 \wedge v.cycle\_antecessor.\{v.current\_h\}\_conflict \neq 0 \wedge$
$table\_completed(v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D) \wedge$
$v.\{v.current\_h\}\_conflict = v.cycle\_antecessor.\{v.current\_h\}\_conflict \wedge$
$((has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)t) \wedge$
$vertex\_deactivated(v, v.current\_h, vertex\_conflict(v, v.current\_h))) \vee$
$(\neg has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h))))) \vee$
$(v.current\_h = v.cycle\_antecessor.cycle\_number \wedge$
$v.cycle\_successor.\{v.current\_h\}\_conflict \neq 0 \wedge$
$table\_completed(v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I) \wedge$
$v.\{v.current\_h\}\_conflict = v.cycle\_successor.\{v.current\_h\}\_conflict \wedge$
$((has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)) \wedge$
$vertex\_deactivated(v, v.current\_h, vertex\_conflict(v, v.current\_h))) \vee$
$(\neg has\_to\_deactivate(v, v.current\_h, vertex\_conflict(v, v.current\_h)))))$ **then**
{

if $v.cycle\_successor.\{v.current\_h\}\_conflict = 0$ **then**
Incrementar $v.current\_h$ al siguiente árbol:
if $v.current\_h$ termina en $r3, r2$ o $r1$, **then**
Restar 1 al dígito de r de $v.current\_h$
else if $v.current\_h$ termina en $\_I\_r0$, **then**
$v.current\_h$ pasa de $\_I\_r0$ a $\_D\_r3$
else if $v.current\_h$ termina en $\_D\_r0$, **then**
Calcular el mejor árbol mezclando subárbol 1 (sin vértice de la izquierda) y subárbol 2 (sin vértice de la derecha)
(línea 28 de Pseudocódigo 9)
Se remueve $\_D\_r0$ y se guarda el árbol.
Se pasa a la siguiente recursión usando esta misma regla.
else if $v.current\_h$ termina en $\_hx\_r0$, **then**
if $x = 0$ **then**
$v.table\_up$ columna 2 = $v.table\_cycle\_0$ columna 1
línea 7 del Pseudocódigo 7
else
for $r = 0 \rightarrow 3$:
$v.table\_up$ columna 2, renglón r =
$max(v.table\_up$ columna 2, renglón r,
$v.table\_cycle\_h$ columna 1, renglón r)
(líneas 8-11 del Pseudocódigo 7)
$\_hx\_r0$ pasa a $\_h(x+1)\_r3$
else if $v.current\_h \neq 0 \wedge$
$v.current\_h \neq v.cycle\_antecessor.cycle\_number$ **then**
Calcular subárbol 1 sin vértice de la izquierda
$(v.table\_cycle\_\{v.current\_h\}\_I)$ y subárbol 2 sin vértice de la derecha $(v.table\_cycle\_\{v.current\_h\}\_D)$
de acuerdo a las líneas 14 y 26 del Pseudocódigo 9:
$v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I$ y
$v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}$ para subárbol 1.
$v.cycle\_successor.table\_cycle\_\{v.current\_h\}$ y
$v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D$ para subárbol 2.
Incrementar $v.current\_h$ a la siguiente recursión:
$v.current\_h = v.current\_h + \_I\_r3$
else if $v.current\_h = 0$ **then**
Calcular árbol 1 sin vértice de la izquierda
$(v.table\_cycle\_\{v.current\_h\}\_I)$ y subárbol 2 sin vértice de la derecha $(v.table\_cycle\_\{v.current\_h\}\_D)$
de acuerdo a las líneas 14 y 26 del Pseudocódigo 9:
$v.table\_cycle\_\{v.current\_h\}\_I$ (columna 0) y
$v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}$ para subárbol 1.
$v.table\_cycle\_\{v.current\_h\}(columna0)$ y
$v.cycle\_antecessor.table\_cycle\_\{v.current\_h\}\_D$ para subárbol 2.
Incrementar $v.current\_h$ a la siguiente recursión:
$v.current\_h = v.current\_h + \_I\_r3$
else if $v.current\_h = v.cycle\_antecessor.cycle\_number$ **then**
Calcular subárbol 1 sin vértice de la izquierda
$(v.table\_cycle\_\{v.current\_h\}\_I)$ y subárbol 2 sin vértice de la derecha $(v.table\_cycle\_\{v.current\_h\}\_D)$
de acuerdo a las líneas 14 y 26 del Pseudocódigo 9:

$v.table\_cycle\_\{v.current\_h\}\_D$ (columna 0) y
$v.cycle\_successor.table\_cycle\_\{v.current\_h\}$ para subárbol 2.
$v.cycle\_successor.table\_cycle\_\{v.current\_h\}\_I$ y
$v.table\_cycle\_\{v.current\_h\}$ (columna 0) para subárbol 1.
Incrementar $v.current\_h$ a la siguiente recursión:
$v.current\_h = v.current\_h + \_I\_r3$
}

S: **if** $\neg v.error \wedge v.completed = false \wedge$
$((v.cycle\_vertex \wedge \neg v.cycle\_error\_\{h\}\forall h) \vee \neg v.cycle\_vertex) \wedge$
$((v.parent = null) \vee (v.parent \neq null \wedge v.parent = completed))$ **then**
{
if $v.parent = null$ **then**
if $v.cycle\_vertex$ **then**
$v.solution = v.table\_up$ (renglón 0, columna 2)
else
$v.solution = v.table\_children\_\{x\}$ (renglón 0, columna 2)
donde x es el último hijo de v
else
$v.solution = v.parent.solution.pointer$
$v.completed = true$
}