La investigación reportada en esta tesis es parte de los programas de investigación del CICESE (Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California).
La investigación fue financiada por el CONAHCYT (Consejo Nacional de Humanidades, Ciencias y Tecnologías).

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

Maestría en Ciencias en Ciencias de la Computación

Usando la descomposición de un grafo Halin para el diseño de algoritmos autoestabilizantes

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

Daniel Uriel Orozco Lomelí

Ensenada, Baja California, México

2023

Tesis defendida por

Daniel Uriel Orozco Lomelí

y aprobada por el siguiente Comité

Dr. José Alberto Fernández Zepeda

Codirector de tesis

Dr. Joel Antonio Trejo Sánchez

Codirector de tesis

Dr. Carlos Alberto Brizuela Rodríguez

Dra. Carmen Guadalupe Paniagua Chávez

Dr. Pedro Gilberto López Mariscal

Coordinador del Posgrado en Ciencias de la Computación

Dra. Ana Denise Re Araujo

Directora de Estudios de Posgrado

Resumen de la tesis que presenta Daniel Uriel Orozco Lomelí como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Usando la descomposición de un grafo Halin para el diseño de algoritmos autoestabilizantes

Resumen aprobado por:

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

Sea $G = (V, E)$ un grafo no dirigido. El problema de encontrar un conjunto independiente fuerte en G, es identificar un conjunto $S \subseteq V$, tal que dados dos vértices arbitrarios de S, éstos estén separados entre sí por el menos tres aristas. Encontrar un conjunto S de tamaño máximo pertenece a la clase *NP-Difícil*. Por otro lado, el problema de encontrar un conjunto dominante total en G es identificar un conjunto $D \subseteq V$, tal que cualquier vértice en V tenga al menos un vecino que pertenezca a D. Encontrar un conjunto D de tamaño mínimo también pertenece a la clase *NP-Difícil*. En este trabajo de tesis se diseñaron dos algoritmos, uno que resuelve el problema de encontrar un conjunto independiente fuerte maximal y otro que resuelve el problema de encontrar un conjunto dominante total minimal. Estos dos problemas son menos restrictivos que las versiones de optimización descritas al principio de este texto y se sabe que pertenecen a la clase P. Los algoritmos diseñados corren en un sistema distribuido, son autoestabilizantes, son tolerantes a fallas transitorias y funcionan para grafos Halin. Los grafos Halin pertenecen a la clase de grafos 2-outerplanares y tienen la propiedad de que se pueden partir en dos subgrafos muy conocidos, un árbol y un ciclo. Los algoritmos propuestos aprovechan la propiedad anterior para disminuir la complejidad de los mismos. Hasta donde tenemos conocimiento, los algoritmos propuestos, que corren en tiempo lineal en el número de vértices, son los algoritmos más rápidos existentes para los problemas del conjunto independiente fuerte maximal y el conjunto dominante total minimal.

Palabras clave: Grafo Halin, Sistemas Distribuidos, Autoestabilización, Conjunto Independiente Fuerte, Conjunto Dominante Total

Abstract of the thesis presented by Daniel Uriel Orozco Lomelí as a partial requirement to obtain the Master of Science degree in Computer Science.

Using Halin graph decomposition for the design of self-stabilizing algorithm

Abstract approved by:

PhD José Alberto Fernández Zepeda
Thesis Co-Director

PhD Joel Antonio Trejo Sánchez
Thesis Co-Director

Let $G = (V, E)$ be an undirected graph. The problem of finding a strong stable set in G, is to identify a set $S \subseteq V$, such that given two arbitrary vertices of S, they are separated from each other by at least three edges. Finding a set S of maximum size belongs to the class *NP-Hard*. On the other hand, the problem of finding a total dominanting set in G is to identify a set $D \subseteq V$, such that any vertex in V has at least one neighbor belonging to D. Finding a set D of minimum size also belongs to the class *NP-Hard*. In this thesis work, two algorithms were designed, one that solves the problem of finding a maximal strong stable set and one that solves the problem of finding a minimal total dominanting set. These two problems are less restrictive than the optimization versions described at the beginning of this text and are known to belong to the P class. The designed algorithms run on a distributed system, are self-stabilizing, are transient fault tolerant, and work for Halin graphs. Halin graphs belong to the 2-outerplanar class of graphs and have the property that they can be split into two well-known subgraphs, a tree and a cycle. The proposed algorithms take advantage of the above property to decrease the complexity of the algorithms. To the best of our knowledge, the proposed algorithms, which run in linear time in the number of vertices, are the fastest existing algorithms for the maximal strong stable set and minimal total dominating set problems.

Keywords: Halin Graph, Distributed Systems, Self-stabilizing, Strong Stable Set, Total Dominating Set

Dedicatoria

A mis padres.

Agradecimientos

Al Centro de Investigación Científica y de Educación de Ensenada, Baja California por darme la oportunidad de estudiar la maestría y ser parte de mi formación académica.

Al Consejo Nacional de Humanidades, Ciencia y Tecnología (CONAHCYT), por brindarme el apoyo económico para realizar mis estudios.

A mis directores de tesis, Dr. José Alberto Fernández Zepeda y Dr. Joel Antonio Trejo Sánchez, por todo el apoyo, los consejos, el tiempo, la dedicación y la paciencia en todo momento durante esta experiencia.

A mis sinodales, Dr. Carlos Alberto Brizuela Rodríguez y Dra. Carmen Guadalupe Paniagua Chávez, por los buenos comentarios, observaciones y guías durante la presentación de los avances de la tesis.

Al grupo de Pelícanos Viajeros, por toda la buena compañía, las buenas charlas, por llevarme a lugares inimaginables en mi misma ciudad y por todo el apoyo y la enseñanza durante las caminatas, que aun cuando pensaba que no podía dar un paso más, un pelícano siempre se encontraba ahí para apoyarme.

A mis compañeros de la maestría, por los buenos momentos que pasamos durante esta experiencia.

Tabla de contenido

Página

Lista de figuras

Figura Página

Figura Página

Lista de tablas

Capítulo 1. Introducción

Un grafo puede verse como un conjunto de puntos denominados vértices, los cuales están unidos entre sí mediante líneas llamadas aristas. Un vértice no necesariamente está conectado directamente a todos los demás vértices del grafo. El caso donde cada vértice se encuentra conectado a todos los demás se conoce como grafo completo.

Formalmente, un grafo se define mediante la notación $G = (V, E)$, donde V es el conjunto de vértices y E el conjunto de aristas, respectivamente.

Existen diversas clases de grafos. Entre los grafos más simples se encuentra el anillo, el cual es un grafo conectado y cada vértices tiene exactamente dos vecinos. La Figura 1 ilustra un anillo de 6 vértices.

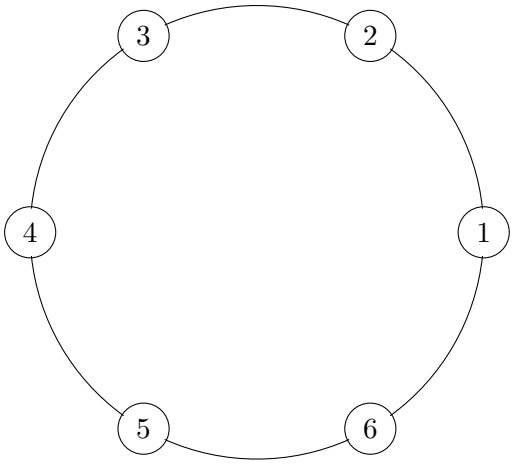

Figura 1. Anillo de seis vértices.

Otro ejemplo simple de grafo es el árbol. Dicho grafo es conectado y sin ciclos, por lo que la cantidad de aristas del grafo es $|E| = |V| - 1$. Normalmente, en muchas aplicaciones en esta clase de grafos, se designa a un vértice especial denominado raíz, al cual se le asignan atributos especiales. Cada vértice restante elige a un vértice vecino como padre, de tal forma que exista una trayectoria desde cualquier vértice a la raíz. Dicha trayectoria se forma únicamente siguiendo a los 'ancestros' de cada vértice fuente. En un árbol, los vértices que sólo cuentan con un vecino se llaman *hojas*. La Figura 2 muestra un ejemplo de un grafo árbol de siete vértices.

Existe una familia de grafos donde una de sus principales características es que estos se componen de una combinación de un árbol y un ciclo que conecta todas las hojas del árbol. Estos grafos se conocen como

grafos Halin. Una descripción más detallada sobre esta clase de grafos y sus propiedades se presenta en el Capítulo 2.

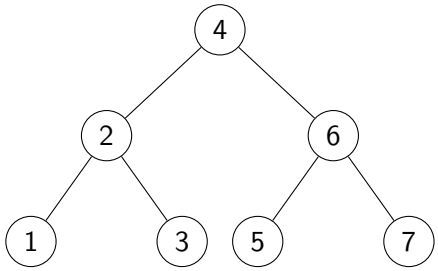

Figura 2. Árbol de siete vértices. El vértice 4 representa la raíz y padre de los vértices 2 y 4. El vértice 2 es padre del 1 y 3. El vértice 6 es padre del 5 y 7. Los vértices $1, 3, 5$ y 7 representan las hojas.

Una de tantas aplicaciones de los grafos es el modelado de mapas. Modelar un mapa como un grafo permite el diseño de algoritmos que pueden resolver problemas de navegación de manera óptima. Por otro lado, sin el uso de grafos, encontrar la respuesta óptima a estos problemas sería más complicado.

Existen varios problemas que pueden modelarse con grafos. Por ejemplo, encontrar la ruta más corta entre dos vértices dados. En este problema, a cada arista del grafo se le asigna un peso (un valor numérico) que representa la distancia o el costo de pasar a través de ella. El problema es determinar la ruta de menor distancia o costo entre los dos vértices.

Las aristas y vértices de un grafo, como se menciona anteriormente, pueden contar con propiedades adicionales, tales como un identificador único asociado a un vértice, un peso asociado a una arista en grafos con pesos, un sentido en una arista para grafos dirigidos, entre otros. Dos grafos con estructura similar en los que se resuelva un mismo problema computacional pueden generar soluciones distintas dependiendo el valor de estas propiedades adicionales.

Sin embargo, a pesar de que resulta sencillo modelar los problemas computacionales usando grafos, no a todos los problemas se le conoce un algoritmo que sea capaz de encontrar su solución de manera 'rápida'. Dentro de los problemas computacionales existen aquellos en donde se conoce al menos un algoritmo lo suficientemente rápido que pueda tomar la entrada del problema y calcular la solución del mismo de manera eficiente. Estos problemas se conocen como problemas de la clase *P*, donde el algoritmo más rápido conocido puede encontrar su solución en tiempo polinomial con respecto al tamaño de la entrada.

Por otro lado, en los problemas de la clase *NP-Difícil* no ha sido posible diseñar un algoritmo que pueda resolverlo en tiempo polinomial para un caso específico arbitrario. Los algoritmos conocidos para este

tipo de problemas por lo general suelen ser de complejidad exponencial con respecto al tamaño de la entrada.

Un ejemplo de esta clase de problemas (también denominados intratables) es el problema del Conjunto Independiente Máximo (también conocido como conjunto 1-Packing). Dado un grafo arbitrario G=(V,E), se desea encontrar un subconjunto *maximal* de vértices $S \subset V$ (i.e., donde al remover un vértice v del conjunto S, dicho conjunto deje de ser un conjunto independiente) de cardinalidad *máxima* (i.e., que además de ser maximal, no exista otro conjunto de cardinalidad más grande que él) de tal manera que para cualquier par de vértices en S no exista una arista en E que los una. Visto de otra manera, la distancia más corta entre cualquier par de vértices en S debe ser de al menos dos aristas. La Figura 3 ilustra un conjunto independiente máximo, representado por los vértices grises. Este problema se ha demostrado que pertenece a la clase *NP-Completo* (una clase que es la intersección de las clases *NP* y *NP-Difícil*) para grafos arbitrarios (Karp, 1972).

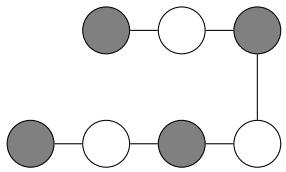

Figura 3. Conjunto independiente máximo en un grafo lineal. Los vértices grises indican aquellos vértices pertenecientes al conjunto independiente.

De manera general, encontrar un conjunto k-packing implica que para cualquier par de vértices en el conjunto S, la distancia más corta entre ellos debe ser de al menos $k+1$ aristas, siendo k una constante entera positiva.

En el problema del Conjunto Independiente Fuerte (CIF), la constante k es 2. Para una $k \geq 2$, encontrar un conjunto k-packing Máximo en grafos arbitrarios se ha demostrado que pertenece a la clase *NP-Difícil* tal como se muestra en el trabajo de Hochbaum & Shmoys (1985).

Sin embargo, cuando el problema del conjunto independiente fuerte máximo (CIF_{MAX}) se restringe a ciertos grafos específicos, la literatura muestra que se han diseñado algoritmos que en tiempo polinomial encuentran la solución óptima del problema. Tales son el caso, por mencionar algunos ejemplos, el trabajo de Mjelde (2004), donde a través de programación dinámica logra encontrar un conjunto k-packing máximo en $O(n^3)$ $u.t.$, en el contexto de algoritmos autoestabilizantes, para grafos árbol. Por otro lado, en el trabajo de Flores-Lamas et al. (2018) se utiliza un algoritmo distribuido para encontrar

el conjunto 2-packing máximo en grafos tipo cactus.

Otro problema computacional fuertemente relacionado al conjunto independiente máximo es el problema del *Conjunto Dominante Mínimo*, en el cual se debe encontrar un conjunto de vértices $D \subset V$ tal que todos los vértices del grafo de entrada deben de ser adyacentes a uno del conjunto D o estar en el conjunto D. Si se encuentra la solución óptima al problema del conjunto independiente máximo, los vértices que están fuera del conjunto conforman la solución óptima del problema del conjunto dominante mínimo. Se conoce que este problema también pertenece a la clase *NP-Difícil* (Garey & Johnson, 1979).

En los trabajos mencionados anteriormente, los algoritmos para encontrar estos conjuntos utilizan ambientes distribuidos. Los sistemas distribuidos consisten de un conjunto de equipos de cómputo o procesadores conectados entre sí mediante enlaces de comunicación. Estos sistemas comúnmente se modelan como un grafo no dirigido $G = (V, E)$ donde V es el conjunto de procesadores y E el conjunto de enlaces de comunicación, respectivamente. Estos sistemas son susceptibles a presentar fallas transitorias, i.e., la información de las variables que un procesador tiene en un determinado es incorrecta, pero no se afecta la operación del procesador.

1.1. Justificación

Se sabe que los problemas del CIF_{MAX} y el Conjunto Dominante Total Mínimo (CDT_{MIN}) pertenecen a la clase *NP-Difícil* para grafos arbitrarios. Sin embargo, para ciertos tipos de grafos restringidos, tales como árboles y ciclos, de acuerdo al trabajo de Mjelde (2004), o cactus por Flores-Lamas et al. (2018), se conocen algoritmos que en tiempo polinomial son capaces de identificar el CIF_{MAX}. Por otro lado, también se sabe que existe una relación muy estrecha y una complementariedad entre los problemas del conjunto independiente y el conjunto dominante.

Entonces, dado que un grafo Halin se puede descomponer en subgrafos simples y bien conocidos, resolver los problemas arriba mencionados en dicho grafo podría favorecer el diseño de técnicas para resolver estos mismos problemas en grafos menos restrictivos. A pesar de que los problemas a resolver no son el CIF_{MAX} y el CDT_{MIN}, respectivamente, resolver el conjunto independiente fuerte maximal (CIF_{max}) y conjunto dominante total minimal CDT_{min} son un primer paso para resolver las versiones de optimización.

1.2. Objetivos

- **Objetivo general**

  El objetivo general de este trabajo de tesis es diseñar e implementar un algoritmo distribuido autoestabilizante síncrono que identifique un Conjunto Independiente Fuerte Maximal y un Conjunto Dominante Total Minimal en grafos Halin. Los algoritmos se diseñarán suponiendo que previamente se han identificado los componentes del grafo *Halin*, i.e., su *árbol* y su *ciclo de hojas*, suposición válida debido al trabajo de Gutiérrez-Medina (2021).

- **Objetivos específicos**

  - Diseñar un algoritmo autoestabilizante que encuentre un CIF_{max} en grafos Halin basado en el algoritmo de Flores-Lamas et al. (2020) y el de Gutiérrez-Medina (2021).
  - Diseñar un algoritmo autoestabilizante que encuentre un CDT_{min} en grafos Halin basado en el algoritmo de Gutiérrez-Medina (2021).
  - Demostrar que el algoritmo que identifica el CIF_{max} es correcto y analizar su tiempo de ejecución.
  - Demostrar que el algoritmo que identifica el CDT_{min} es correcto y analizar su tiempo de ejecución.

Capítulo 2. Marco teórico

2.1. Autoestabilización

Un sistema distribuido consta de un conjunto de computadoras interconectadas por enlaces de comunicación, las cuales buscan resolver un problema en común. Los sistemas distribuidos se pueden modelar como un grafo no dirigido $G = (V, E)$, donde los vértices ("nodos" de ahora en adelante durante la Sección 2.1), representan el conjunto de computadoras y las aristas sus enlaces de comunicación.

Dentro de los sistemas distribuidos, se llama *modelo computacional* a las características que definen las capacidades del sistema. Para un mismo algoritmo, si se emplean modelos distintos, la complejidad del algoritmo diseñado así como su tiempo de ejecución puede variar de modelo a modelo.

Un modelo de *paso de mensajes* es aquel en donde un nodo P_i del sistema se comunica con sus nodos vecinos a través del envió y recepción de mensajes. En este modelo, los mensajes se almacenan en *colas de mensajes*, donde los mensajes nuevos que llegan al nodo se ponen al final de la cola y los mensajes que hayan llegado anteriormente se procesan primero.

Por otro lado, en el modelo de *memoria compartida*, la manera en que los nodos se comunican es mediante la escritura y lectura de *registros de memoria*, donde los registros en el cual un nodo puede leer y escribir no necesariamente son los mismos.

El *estado local* es el conjunto de los valores de las variables que almacena un nodo. El *estado global* es el conjunto de los valores de todas las variables de todos los nodos del sistema.

Los sistemas distribuidos, así como todos los sistemas en general, son susceptibles a fallas. Estas fallas pueden ser de tipo *transitorias*, donde si bien el funcionamiento del equipo se mantiene saludable, el valor de las variables que los nodos almacenan (su estado local) puede verse alterado. También estos sistemas pueden presentar fallas *permanentes*, donde el funcionamiento del equipo sí se ve comprometido.

Por tal razón surgió la necesidad de desarrollar algoritmos con tolerancias a fallas transitorias, y es en el trabajo de Dijkstra (1974) donde se propone por primera vez el concepto de *autoestabilización*.

En la autoestabilización se busca que, iniciando el sistema desde cualquier estado arbitrario (sea o no un estado válido), éste logre *converger* a una configuración deseada en un número finito de pasos y,

una vez que se cuente con dicha configuración, éste se *mantenga* sin alteraciones en ausencia de fallas. Estas propiedades de autoestabilización se conocen como *Propiedad de Convergencia* y *Propiedad de Cerradura*, respectivamente. La Figura 4 ilustra las transiciones entre los estados válidos e inválidos junto a las propiedades de autoestabilización.

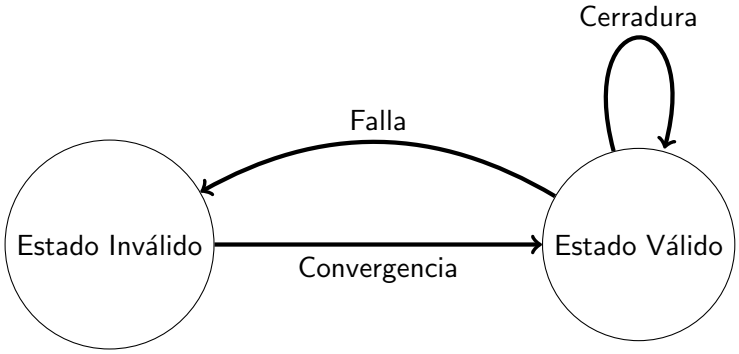

Figura 4. Propiedades de la autoestabilización. Mediante convergencia se logra llegar a un estado válido partiendo desde cualquier estado inválido. Por propiedad de cerradura, el sistema se mantiene en estados válidos en ausencia de fallas.

Para determinar si un nodo puede alterar o no el valor de sus variables para lograr converger al estado deseado, se utiliza el concepto de *Regla*. Las reglas se componen de tres elementos:

- Encabezado: Título o nombre significativo de la regla que se va ejecutar.
- Expresión booleana: Parte lógica de la regla que determina si el nodo es candidato a alterar su estado local.
- Acciones: Operaciones que realiza el algoritmo para actualizar el estado local de un nodo.

Cuando un nodo satisface una o más reglas se dice que el nodo se encuentra *privilegiado*. Cuando un nodo se encuentra privilegiado significa que éste puede alterar los valores de sus variables para así converger al estado deseado. Una vez que en el sistema no se cuente con ningún nodo privilegiado, entonces se dice que el sistema ha convergido, logrado así la autoestabilización.

Partiendo desde un estado c_i, se dice que el estado c_j es alcanzable si al ejecutar un *paso computacional* a (realizar operaciones de lectura y escritura de variables en un modelo de memoria compartida o realizar envío y lectura de mensajes en un modelo de paso mensajes) el estado c_i transciende al estado c_j.

Un calendarizador dentro de los algoritmos autoestabilizantes es un ente ficticio que tiene la habilidad de

seleccionar uno o más nodos privilegiados. Se debe garantizar que el calendarizador seleccione al menos un nodo privilegiado en cada paso del algoritmo si existe alguno en el sistema.

Dentro de los calendarizadores más comunes se encuentra el calendarizador *Asíncrono*, el cual es el que selecciona a uno o más nodos privilegiados. El tiempo que le toma a cada nodo en realizar sus operaciones de lectura/escritura es arbitrario pero finito.

En el calendarizador asíncrono, la manera en que se mide el tiempo de ejecución de un algoritmo es determinando la cantidad de rondas que él requiere. El tiempo que tarda una ronda en completarse en este tipo de calendarizadores puede variar de algoritmo a algoritmo. En Shlomi (2000) se define como *ronda asíncrona* a la cantidad de pasos que le toma al sistema privilegiar al menos una vez a cada uno de los nodos del sistema.

En el calendarizador *síncrono* todos los nodos privilegiados se seleccionan. Trabajos como Peleg (2007) o Dolev & Welch (2004) definen como *ronda síncrona* a un pulso constante y sincronizado para todo nodo en el sistema, en donde todos los nodos son capaces de leer y actualizar sus variables en el mismo pulso.

2.2. Conjunto independiente y sus variantes

Dado un grafo $G = (V, E)$, un Conjunto k-Packing es un subconjunto de vértices $S \subset V$ tal que para cualquier par de vértices del conjunto S, la distancia más corta entre tales vértices es de al menos $k + 1$ aristas. La constante k debe ser entera positiva. La Figura 5 ilustra un conjunto 1-packing o independiente marcado por los vértices sombreados.

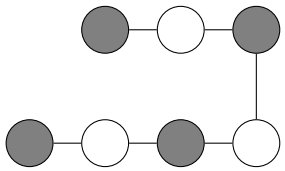

Figura 5. Ejemplo de conjunto 1-packing o independiente.

El *Conjunto Independiente* es un caso específico del Conjunto k-Packing, donde la constante k es 1, mientras que en un *Conjunto Independiente Fuerte* la constante k es 2.

En cualquier caso, se dice que un conjunto es *maximal* si al agregar cualquier vértice $v \notin S$ al conjunto S, éste incumple con la condición de que la distancia sea de al menos $k+1$ aristas. En el lado izquierdo de la Figura 6 se muestra un conjunto independiente fuerte maximal (CIF_{max}), representado por los vértices sombreados. Un conjunto S se dice que es *máximo* si además de ser maximal, éste es de cardinalidad máxima (no existe otro subconjunto con más vértices). En el lado derecho de la Figura 6 se muestra el mismo grafo que en el lado izquierdo pero ahora con un CIF_{MAX}, representado por los vértices sombreados.

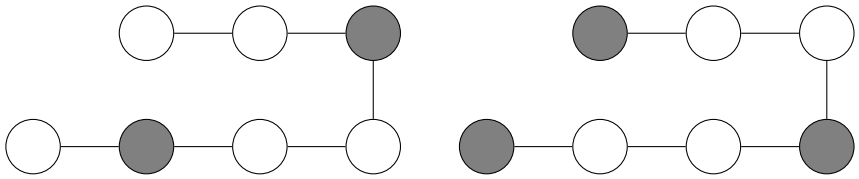

Figura 6. Diferencias entre un CIF_{max} y CIF_{MAX}.

El problema de encontrar un conjunto independiente máximo en grafos arbitrarios pertenece a la clase *NP-Difícil* (Karp, 1972). El problema del CIF_{MAX} también es *NP-Difícil* (Hochbaum & Shmoys, 1985).

2.3. Conjunto dominante total

En el problema del *Conjunto Dominante* se debe encontrar un conjunto de vértices $D \subset V$ del grafo de entrada $G = (V, E)$, en el cual cada vértice de V pertenezca a D o sea adyacente a un vértice $w \in D$ (sea *dominado* por algún vértice w del conjunto D).

Si a un conjunto dominante D se le remueve cualquier vértice, y D deja de ser conjunto dominante, se dice que D es *minimal*. Encontrar un conjunto dominante minimal de menor cardinalidad hace referencia al problema del *Conjunto Dominante Mínimo* (CD_{MIN}). El problema del CD_{MIN} se ha demostrado que pertenece a la clase de problemas *NP-Difícil* (Garey & Johnson, 1979).

La Figura 7 ilustra las diferencias entre un conjunto dominante minimal (CD_{min}) y un CD_{MIN}. El grafo mostrado corresponde a un grafo estrella. En el lado izquierdo de la Figura 7, se muestra un CD_{min} que no es un CD_{MIN}. Al eliminar cualquier hoja del conjunto dominante, dicha hoja ya no es 'dominada' y el conjunto de vértices sombreados resultante ya no es dominante. La cardinalidad de esta solución corresponde a ocho vértices.

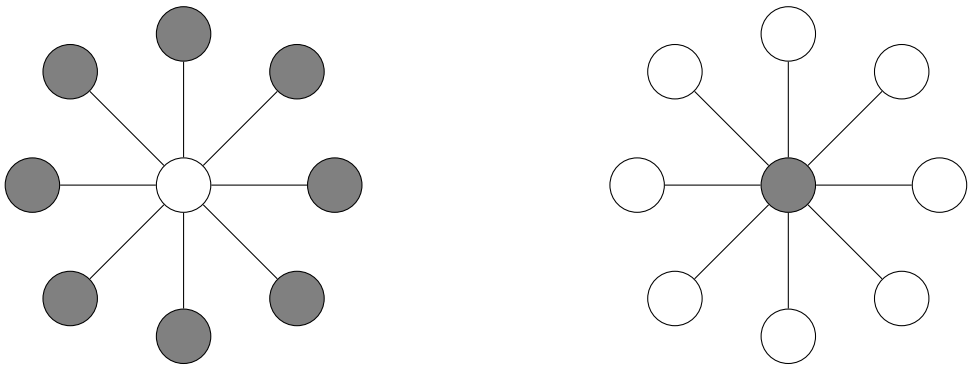

Figura 7. Conjuntos dominantes en un grafo estrella. En el lado izquierdo un CD_{min} que no es CD_{MIN}. En el lado derecho se muestra un CD_{MIN}.

Si el conjunto dominante sólo incluye al vértice central, como se ve a la derecha de la Figura 7, se obtiene un CD_{MIN}. Todos los vértices del grafo los domina el vértice central y el vértice central cumple con la condición de pertenecer a D.

Una variante del problema del conjunto dominante que se propone estudiar en este trabajo es el problema del *Conjunto Dominante Total Minimal* (CDT_{min}). Esta variante del problema añade la restricción que para cualquier vértice v del conjunto dominante, $v \in D$, éste debe ser adyacente a un vértice $w \in D$. En otras palabras, ahora cualquier vértice $v \in D$ debe estar *dominado* por otro vértice distinto w. El problema de encontrar un conjunto dominante total de menor cardinalidad (CDT_{MIN}) también pertenece a la clase de problemas *NP-Difícil*, demostrado en el trabajo de Garey & Johnson (1979).

En la Figura 8 se ilustra el mismo grafo de la Figura 7, donde ahora los vértices grises representan un CDT_{MIN}. Como se muestra en el lado derecho de la Figura 7, previamente para el problema de encontrar un CD_{MIN} se permitía que el vértice central no fuera *dominado* por otro vértice, mientras que para el problema del CDT_{MIN}, alguna hoja del grafo debe dominar al vértice central. De forma complementaria, el vértice central domina a dicha hoja.

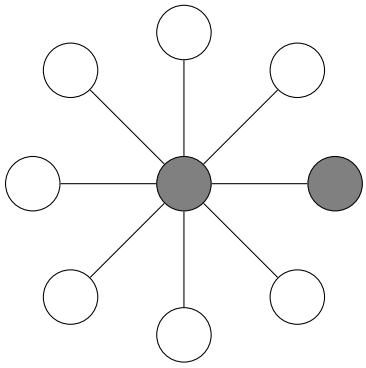

Figura 8. Ejemplo de un CDT_{MIN} en un grafo estrella.

2.4. Grafos Halin

Un grafo se dice que es *planar* si es posible dibujarlo de manera que las aristas no se crucen entre sí. Cuando todos los vértices de un grafo plano tocan la cara exterior (i.e., si al momento de trazar el contorno del grafo es posible alcanzar a todos los vértices), el grafo se denomina *outerplanar*.

Los *grafos Halin* son una familia de grafos pertenecientes a la clase 2-*outerplanar*. En los grafos 2-outerplanar existen dos conjuntos de vértices, unos que tocan la cara exterior y otros que no. Al remover del grafo los vértices del primer conjunto, los vértices del segundo conjunto ahora tocan la cara exterior.

Los grafos Halin se conforman mediante dos estructuras: Un *árbol*, el cual no contiene ningún vértice de grado 2 y es de al menos cuatro vértices, y un *ciclo de hojas*, el cual se forma al conectar todas las hojas del árbol mediante un conjunto de aristas.

La Figura 9 muestran las estructuras del árbol (del lado izquierdo) y el ciclo de hojas (del lado derecho), ambas con lineas continuas, de un grafo Halin, respectivamente.

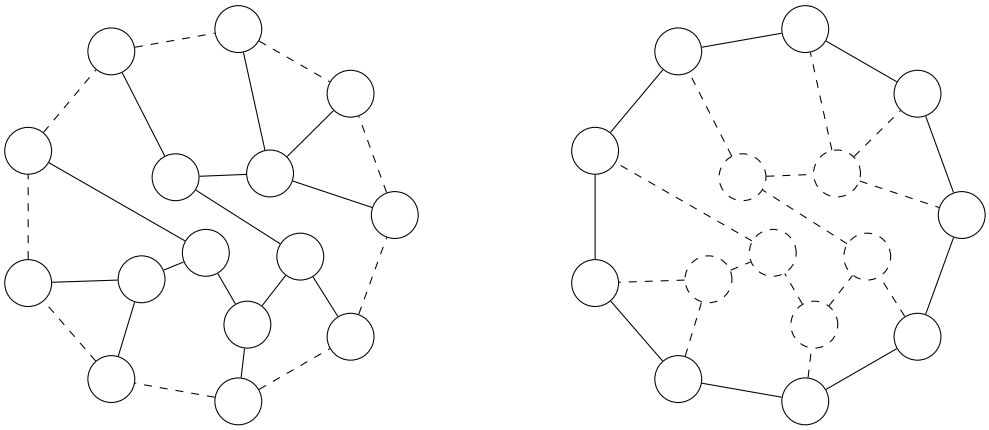

Figura 9. Componentes de un grafo Halin. Del lado izquierdo se incluyen, con lineas continuas, los vértices y aristas que conforman el árbol del Halin. Del lado izquierdo se incluyen, con lineas continuas, los vértices y aristas que conforman el ciclo de hojas.

Si se eliminan los vértices del ciclo de hojas y sus aristas incidentes, al grafo resultante (ilustrado en la Figura 10) se le conoce como *Esqueleto del Halin*.

Algunas características que se presentan en los grafos tipo Halin se listan a continuación:

- Son de la clase 2-outerplanar.
- No tiene vértices de grado 2.

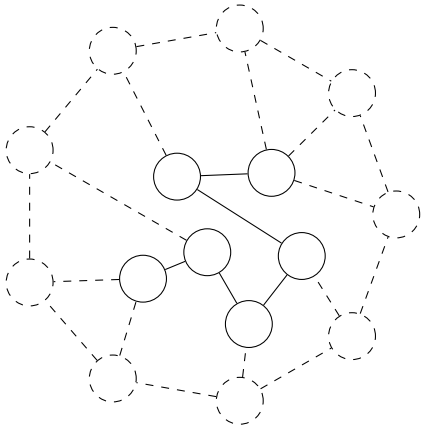

Figura 10. Esqueleto del Halin.

- El tamaño del ciclo de hojas (V_C) equivale a $|V_C| = m - n + 1$, donde n y m representan la cantidad de vértices y aristas, respectivamente.
- Son *triconectados*, por lo que para cualquier par de vértices arbitrarios $u, v \in V$ en el grafo, existen tres caminos independientes que los conectan. En la Figura 11 se muestran tres posibles caminos a seguir, marcados con rojo, azul y verde, para llegar del vértice u al vértice v.

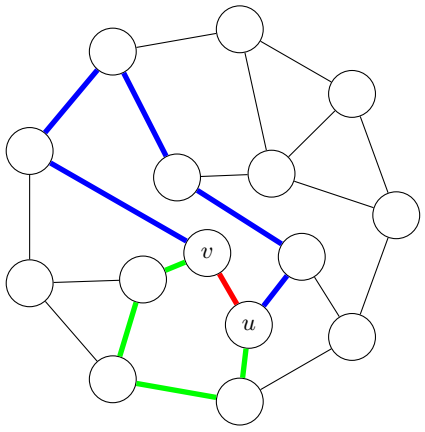

Figura 11. Ejemplos de trayectorias independientes en un grafo Halin. Existen tres trayectorias independientes, marcadas con azul, rojo y verde, para llegar del vértice u al vértice v.

- Son *Hamiltonianos*, por lo que existe una trayectoria cerrada, que comenzando desde cualquier vértice, es posible pasar por todos los vértices exactamente una vez hasta llegar al vértice inicial.

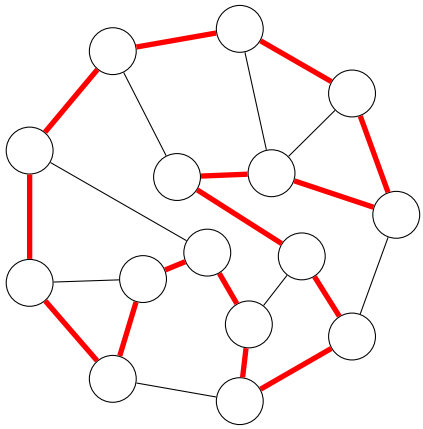

Figura 12. Ejemplo de un ciclo Hamiltoniano en un grafo Halin. La trayectoria del ciclo está resaltada en rojo.

Más información sobre las características de los grafos Halin y sus demostraciones se puede encontrar en el trabajo de Sysło & Proskurowski (1983).

2.5. Antecedentes

2.5.1. Antecedentes sobre el conjunto independiente fuerte

En la tesis escrita por Mjelde (2004), él diseña cuatro algoritmos para el problema del CIF_{MAX} y el problema del Conjunto k-Domination Mínimo (CkD_{MIN}) en grafos tipo árbol. Para cada problema, él diseña un algoritmo secuencial y uno autoestabilizante, respectivamente. En los algoritmos para el CIF_{MAX} y el CkD_{MIN}, Mjelde utiliza una estrategia de programación dinámica. El algoritmo secuencial tiene un tiempo de ejecución de $O(kn)$ unidades de tiempo (*u.t.*) mientras que para el algoritmo autoestabilizante le toma $O(n^3)$ *u.t.*

En Gairing et al. (2004), los autores desarrollaron un algoritmo autoestabilizante para el problema del CIF_{max}. El modelo utilizado es una variante del modelo de memoria compartida, donde se pueden leer las variables del vecindario cerrado de un vértice, pero sólo se puede escribir en sus variables locales. El algoritmo supone que cada vértice tiene un identificador único. Se utilizan dos variables para identificar al

CIF_{max}: $x(v)$, la cual indica si el vértice v pertenece al conjunto, y un apuntador '$\leftarrow$' el cual proporciona información adicional al vecindario del vértice. Su tiempo de ejecución es exponencial.

Shi (2012) propuso un algoritmo autoestabilizante para encontrar un CIF_{max} en grafos arbitrarios. El tiempo de ejecución del algoritmo es de $O(n^2)$ *u.t.*, usando un calendarizador síncrono. El algoritmo se basa en la idea de que cada vértice posee un identificador único que los distingue. A cada vértice se le asigna un valor en su variable c entre $\{0, 1, 2\}$, donde 0 indica que el vértice pertenece al CIF_{max}, 1 indica que el vértice es adyacente a uno que pertenece al conjunto, y 2 significa que es adyacente a vértices con $c = 1$ o $c = 2$, pero nunca a uno con $c = 0$.

En el trabajo de Trejo-Sánchez & Fernández-Zepeda (2014), los autores diseñaron un algoritmo distribuido para encontrar un CIF_{max} en grafos outerplanares geométricos en $O(n)$ ranuras de tiempo con mensajes de tamaño $O(\log n)$. Se hace uso de un vértice líder, el cual mediante el envío de tokens (utilizando la información geométrica del grafo) hace el reconocimiento de las características del grafo para realizar la identificación del CIF_{max}.

En Flores-Lamas et al. (2020), los autores diseñaron un algoritmo distribuido para el problema del CIF_{max} en grafos Halin no geométricos. El algoritmo corre en $O(n)$ ranuras de tiempo. El modelo computacional utilizado para el algoritmo es una versión restringida del modelo de memoria compartida, donde los vértices sólo pueden escribir en sus registros propios, pero sí pueden leer los registros de su vecindario abierto.

El algoritmo identifica el ciclo de hojas del grafo y su árbol utilizando una versión adaptada del algoritmo diseñado por Eppstein (2016). Para identificar CIF_{max}, el algoritmo realiza un coloreado que distingue los vértices del árbol y los vértices del ciclo. Para ambos componentes se designa un líder. Una vez los CIF_{max} en ambos componentes, se realiza un paso de corrección de coloreado.

2.5.2. Antecedentes sobre el conjunto dominante total

Algunos de los trabajos destacables referentes al problema del CDT_{min} en el contexto distribuido y autoestabilizante se describen a continuación:

En Goddard et al. (2003), los autores diseñaron el primer algoritmo autoestabilizante que identifica un CDT_{min}. Este algoritmo tiene un tiempo de ejecución exponencial. Se supone un sistema distribuido

con calendarizador centralizado y funciona para cualquier grafo de entrada.

En el trabajo de Belhoul et al. (2014), los autores desarrollaron el primer algoritmo autoestabilizante que encuentra un CDT_{min} en tiempo polinomial. La complejidad del algoritmo es de $O(nm)$ ranuras de tiempo, donde n denota el número de vértices del sistema y m los enlaces de estos vértices. Este algoritmo también se ejecuta bajo el calendarizador centralizado y funciona para cualquier grafo de entrada.

Recientemente, en el trabajo de Ding et al. (2020), los autores diseñaron un algoritmo autoestabilizante para el problema del CDT_{min} cuya complejidad se redujo a $O(n^2)$ ranuras de tiempo. Este algoritmo, al igual que los algoritmos mencionados previamente, funciona para cualquier grafo de entrada bajo el calendarizador centralizado.

Capítulo 3. Algoritmos de Gutiérrez-Medina

Los algoritmos diseñados para esta tesis toman como base el trabajo previo de Gutiérrez-Medina (2021).

En su trabajo se presenta el diseño de cuatro algoritmos autoestabilizantes sobre un grafo Halin geométrico. Cada algoritmo se estructura como una fase ordenada, donde para que una fase F_{i+1} pueda comenzar, la fase F_i debe converger de manera exitosa. La fase F_{i+1} utiliza la información previamente calculada en $F_i, F_{i-1}, \ldots$ para realizar los cálculos necesarios de la fase actual.

Para el grafo de entrada y el modelo utilizado en su trabajo se hacen las siguientes suposiciones:

- Cada procesador tiene un identificador único almacenado en una variable llamada $v.id$. El identificador de cada vértice es un valor del conjunto $\{0, 1, 2, \ldots, n-1\}$.
- Al ser geométrico, se conoce la posición en el plano (sus coordenadas x, y) de cada vértice y cada arista, por ser ésta una línea recta.
- Se conoce la lista de adyacencias de cada vértice v y se encuentra localmente ordenada. El ordenamiento se realiza tomando como referencia el ángulo que se forma entre la arista $\langle u, v\rangle$ y el eje x. Se hace la suposición que v se encuentra en el origen y que, para cada arista existente de su vecindario, se le asigna una numeración desde $0, 1, ..., \Delta_v - 1$, donde Δ es el grado de v. Esta información se almacena en la variable $v.adj$.
- Cada arista i del grafo puede verse además como un par de aristas dirigidas en sentidos opuestos. La arista de entrada es $e(v)_i^{\leftarrow}$ mientras que la arista de salida es $e(v)_i^{\rightarrow}$.
- Los algoritmos utilizan la convención *First Right Out* (FRO), por lo que si un mensaje entra por la arista $e(v)_i^{\leftarrow}$, éste debe salir por la arista $e(v)_{i+1}^{\rightarrow}$.
- En el sistema sólo se pueden presentar fallas de tipo transitorias.
- El identificador, la lista de adyacencias y la posición en el plano de cada vértice es a prueba de errores, por lo que éstos no se ven afectados por ningún tipo de falla.
- La longitud de cualquier variable es de $O(\log n)$ bits, donde n es el número de vértices.
- El sistema trabaja de manera síncrona, por lo que, de manera sincronizada en una misma ranura de tiempo, todos los vértices pueden leer las variables propias y las de sus vecinos, realizar cálculos y actualizar sus variables.

- El sistema utiliza el modelo de memoria compartida, por lo que un vértice es capaz de leer y escribir en sus propios registros de memoria y puede leer los registros de memoria de sus vértices adyacentes, pero no puede escribir en ellos.

A continuación se describen los cuatro algoritmos.

3.1. Fase 1: Elección del líder

La elección de líder se hace con el propósito de seleccionar a un vértice especial denominado *líder de comunicación*. Dicho vértice se encarga de efectuar acciones especiales de coordinación y de definir un *árbol de comunicación* utilizado como canal seguro para la propagación de mensajes.

El algoritmo va construyendo árboles de expansión enraizados en cada uno de los vértices, de modo que estos árboles al final se fusionan entre sí hasta generar el árbol de comunicación, donde la raíz resultante se toma como líder de comunicación.

Este algoritmo se basa en el trabajo de Trejo-Sánchez & Fernández-Zepeda (2014), tiene un tiempo de ejecución de $O(D)$ rondas (donde D denota el diámetro del grafo), y el tamaño de los mensajes es de $O(\log n)$ bits.

Gutiérrez-Medina adaptó el algoritmo de Trejo-Sánchez & Fernández-Zepeda (2014) para que se considere a un conjunto reducido de vértices como candidatos a ser el líder de comunicación. Los vértices que se encuentran en caras triangulares tales como el $2, 5$ y 8 del ejemplo de la Figura 13a, o los de abanicos, exceptuando al vértice central, tales como $11, 14$ y 15 del ejemplo de la Figura 13b, son los únicos que se consideran candidatos para esta nueva implementación.

La fase de elección de líder utiliza un conjunto de variables, tales como la variable $v.ldr$ que almacena el identificador del líder, $v.dst$ que almacena la distancia entre el vértice v y el líder, o $v.par$ que almacena el padre de v en el árbol enraizado en el líder. También se definen funciones útiles para un mejor entendimiento de la estructura del árbol de comunicación, como la función $children(v)$, que regresa el identificador de todos los vértices vecinos de v que reconocen a v como su padre en el árbol enraizado de comunicación.

En esta fase y en fases posteriores se utilizan variables generales para el manejo de mensajes. Tal es el

caso de las variables $v.wait$ que ayuda a generar un retraso en la ejecución del algoritmo, o $v.done$ para la transmisión del mensaje de finalización de la fase.

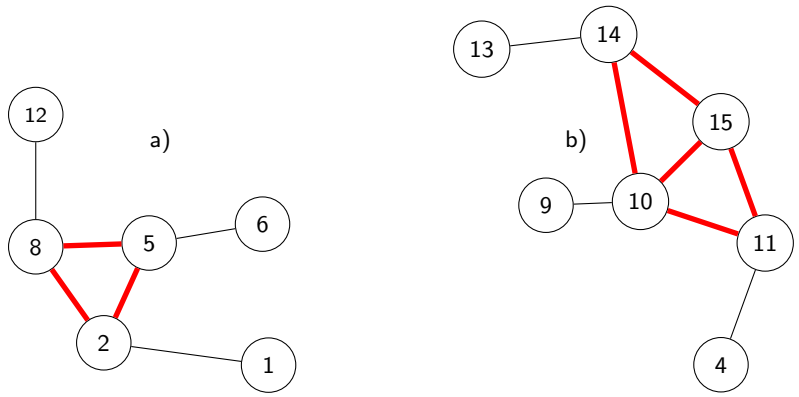

Figura 13. Subgrafos triangulares y abanicos en un grafo Halin. a) Cara triangular de un fragmento de un grafo Halin. b) Grafo abanico en un fragmento de un grafo Halin.

Se utilizan funciones booleanas para la detección de inconsistencias en las variables del sistema, dependiendo de la configuración deseada. Por mencionar algunos ejemplos: se tiene la función $safe\_ldr(v)$ para la validación de las variables del líder, $safe\_par(v)$ para la validación de las variables del padre de v una vez enraizado, o $sane(v)$ que valida que el líder sea el mismo para todo el vecindario, que sea el de menor identificador, entre otras validaciones de las variables.

La estructura de esta fase, así como de las fases posteriores, se basa en la definición de cuatro reglas:

- RESET: Regla donde dado un estado *inválido* (según la definición en cada fase), el vértice privilegiado debe ejecutar la acción de reiniciar sus variables. Esta regla en fases posteriores se usa para el manejo de la función de reinicio una vez se cuente con el canal seguro para la transmisión de mensajes.
- SYNC: Mediante esta regla se genera un paso de espera para que el árbol de expansión crezca más lentamente que la propagación de un mensaje de reinicio. Esta regla, al igual que el caso de la regla RESET, cambia para manejar la sincronización del mensaje de reinicio.
- ALG: Regla donde se ejecuta el algoritmo propio de la elección del líder. En general, esta regla se utiliza para calcular la configuración deseada dado el problema a resolver, i.e., enraizar un árbol, enumerar las hojas, calcular un conjunto independiente, calcular un conjunto dominante, etc.
- DONE: Mediante esta regla se propagan los mensajes de finalización de la fase utilizando la variable

$v.done$. El mensaje se propaga en el árbol de comunicación hasta el líder de comunicación realizando un convergecast hacia éste (i.e., propagación de mensajes desde las hojas hasta la raíz del árbol).

El tiempo de ejecución de esta fase es de $O(D)$ rondas, donde D es el diámetro del grafo.

Al finalizar la fase, todos los vértices cumplen con el predicado A_1, el cual indica que los vértices han finalizado su fase, no se tiene inconsistencias en sus variables, y han coincidido en reconocer a un único líder.

3.2. Fase 2: Exploración

En la segunda fase, la de *Exploración*, se requiere designar a un vértice como el *representante del ciclo de hojas*. Dicho vértice se encuentra en el ciclo de hojas y además en la misma cara triangular que el líder de comunicación. Se utiliza la convención FRO para hacer circular un mensaje (token) por todo el ciclo de hojas.

Para esta fase, las aristas ahora se consideran como dos aristas dirigidas con sentidos opuestos, donde una arista se toma como arista de entrada y la otra como de salida.

Sólo los vértices que se encuentran en la cara triangular, descrita anteriormente, son los que se consideran como candidatos para iniciar el recorrido del token.

Se utiliza la variable de $v.length$ para almacenar el número de aristas que recorrió el token. También se utiliza la variable $v.trace$ para almacenar información adicional sobre el paso del token por un vértice, desde dónde llegó y cuántas aristas ha recorrido hasta el momento en que llegó al vértice inicial.

Aquel vértice candidato cuyo token haya recorrido más aristas, es decir, aquel token que haya recorrido todo el ciclo de hojas, se considera el representante del ciclo de hojas.

Al igual que en la fase de elección de líder, se sigue la misma estructura de utilizar las reglas Reset, Sync, Alg y Done. Se define una quinta regla adicional denominada Error, la cual establece que si un vértice detecta una inconsistencia en los valores de sus variables (i.e., que se encuentra en un estado inválido), o si los vértices descendientes en su canal de comunicación tienen un mensaje de error, dicho vértice propaga el mensaje de error hasta el líder de comunicación para que él coordine el reinicio de la fase. Para esta regla se utiliza una variable booleana adicional denominada $v.error$.

Para definir si un vértice es válido, se utilizan funciones booleanas para verificar la consistencia de las variables con respecto al diseño del algoritmo. Por mencionar algunos ejemplos: todos los vértices deben cumplir con el predicado A_1; aquellos vértices que no sean candidatos deben de tener en todo momento su variable de $v.length = 0$; si existe un vértice que no es candidato pero que tiene las variables de $v.trace$ distinto de Nil y 0, entonces el recorrido del token sobre ese vértice debe cumplir con la convención FRO.

El tiempo de ejecución de este algoritmo es de $O(n)$ rondas, dado que la circulación del token se realiza de forma secuencial y el ciclo de hojas está conformado por $O(n)$ vértices.

Dado un error en el sistema, el mensaje de error tarda en propagarse hasta el líder de comunicación $O(D)$ rondas. Esto se cumple para las fases posteriores.

Al finalizar la fase, todos los vértices cumplen con el predicado A_2, el cual indica que los vértices han finalizado la exploración y no se tienen inconsistencias en sus variables.

3.3. Fase 3: Enumeración

En la tercera fase se realiza el proceso de *Enumeración*, donde a cada vértice del ciclo de hojas se le asigna un número secuencial partiendo del representante del ciclo de hojas. En la Figura 14 se ilustra un ejemplo de un Halin con el ciclo ya enumerado.

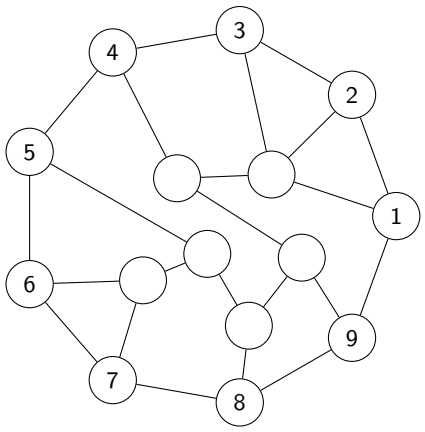

Figura 14. Ciclo de hojas de un Halin ya enumerado. El vértice 1 indica el representante del ciclo.

Si el líder de comunicación resultó estar en un abanico, entonces él mismo se considera como representan-

te del ciclo de hojas y desde él se realiza el proceso de enumeración. Si por el contrario, el representante se encontró en una cara triangular, ahora el líder de comunicación realiza el procedimiento de seleccionar como representante del ciclo a aquel vértice que realizó el recorrido más largo. La función $find\_rep(v)$ realiza dicho procedimiento.

En esta fase se introducen nuevas variables tales como: $v.leaf$, que indica si el vértice v pertenece o no al ciclo de hojas; $v.lf\_rep$, el cual almacena el identificador del representante del ciclo de hojas; y $v.nbr$, que almacena el número de aristas que existen entre el representante del ciclo de hojas y el vértice v al realizar el recorrido secuencial desde el representante del ciclo.

Al igual que en la fase de exploración, se utiliza el mismo conjunto de reglas RESET, SYNC, ERROR, ALG y DONE, donde en ALG se implementa la lógica de realizar la enumeración del ciclo de hojas. De la misma manera que en las fases previas, se hace uso de funciones booleanas para comprobar la consistencia de las variables según el diseño del algoritmo. Algunos ejemplos de las validaciones son: verificar que cualquier vértice cumpla con el predicado A_2; verificar que cualquier vértice marcado como hoja debe tener un vecino hoja; verificar que el líder del ciclo de hojas también debe tener activada su bandera de $v.leaf$; y verificar que cualquier vértice marcado como hoja no puede tener en su variable de enumeración el valor 0.

Debido a que el proceso de enumeración funciona de manera secuencial en el ciclo de hojas, el tiempo de ejecución del algoritmo es de $O(n)$ rondas.

Al finalizar la fase, todos los vértices indican que han finalizado su fase de enumeración y no presentan ninguna inconsistencia en sus variables. Esto comprende el predicado A_3.

3.4. Fase 4: Descomposición de orejas

Finalmente, en la cuarta cuarta fase se realiza una descomposición de orejas en el grafo Halin.

La descomposición de orejas hace referencia a realizar una partición ordenada del conjunto de aristas de modo que éstas queden agrupadas en un ciclo inicial y trayectorias. A dichas agrupaciones o bloques se denominan *orejas*. La Figura 15 ilustra una descomposición de orejas en un grafo Halin.

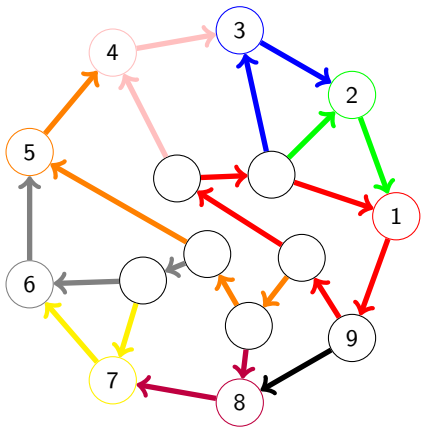

Figura 15. Descomposición de orejas en un grafo Halin.

La idea detrás del funcionamiento de la descomposición de orejas en el grafo Halin es como sigue: cada hoja del ciclo de hojas envía un token con el identificador de la hoja, de modo que el token recorra las aristas de la cara interna que toca la hoja, y asigne el valor del token a cada arista sobre la cual va avanzando. Los recorridos que realizan los tokens siguen la convención FRO.

En la Figura 16 se ilustra cómo la hoja 1 envía un token que almacena su identificador y recorre su cara interna correspondiente. Las aristas $(1,2),(2,6),(6,5),(5,1)$, por las que se transporta el token enviado, adquieren el valor de 1, de modo que ahora estas aristas pertenecen al bloque u oreja 1.

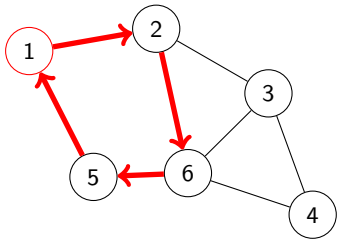

Figura 16. Recorrido y partición de la oreja 1 correspondiente al token enviado por la hoja 1.

Cada arista del ciclo de hojas sólo la recorre un único token, enviado por la hoja adyacente. Por otro lado, cada arista del árbol del Halin la recorren exactamente dos tokens, de modo que esta arista se mantiene en la oreja cuyo token corresponde al del identificador más pequeño.

En el lado izquierdo de la Figura 17 se ilustra el recorrido de los tokens enviados por las hojas $1,2$ y 3. Dado que las aristas $(2,6)$ y $(3,6)$ son aristas del árbol, existe un conflicto en la asignación de las orejas. En el caso de la arista $(2,6)$, dado que el token enviado por la hoja 1 almacena un identificador más pequeño que el token enviado por la hoja 2, esta arista forma parte de la oreja 1. De manera similar

se resuelve el conflicto en la arista $(3, 6)$. La partición de orejas resultante se puede observar en el lado derecho de la Figura 17.

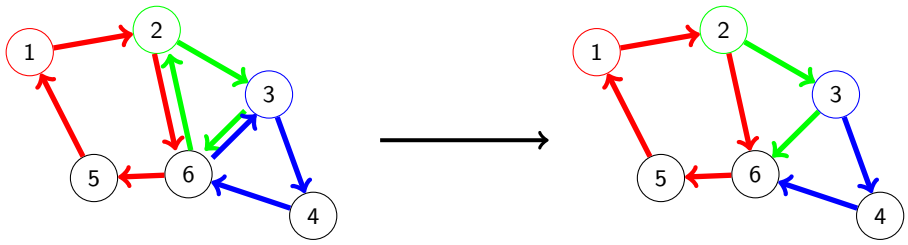

Figura 17. Resolución de la descomposición de orejas en aristas $(2, 6)$ y $(3, 6)$ del árbol.

Para identificar la partición de las aristas en el sistema se hace uso de cinco nuevas variables: tres de ellas siguen la estructura que se ha ido manejando desde la segunda fase, las cuales corresponden a las banderas de $reset, error$ y $done$. En las otras dos variables se almacena la información para identificar las orejas: En $v.ear\_trace$ se guarda los identificadores de los tokens que pasaron por la arista (v, u) para todos los vecinos de u y v. En $v.ears$ se guarda el identificador de la oreja para cada arista del vértice v.

De manera similar que en las fases anteriores, se utilizan funciones booleanas para comprobar la integridad de las variables y así detectar posibles fallas que hagan que el sistema se encuentre en una configuración inválida. Cuando todas las validaciones son correctas para un vértice v, éste se dice que está en un estado válido.

Como el recorrido de los tokens a través de las orejas se hace de manera secuencial, y debido a que en el peor de los casos la oreja más grande del sistema puede ser de $O(n)$ vértices, el tiempo de ejecución de esta fase es de $O(n)$ rondas.

Capítulo 4. Algoritmo Rooting_Tree

Antes de comenzar con el diseño de los algoritmos para encontrar un CIF_{max} o un CDT_{min} en los componentes del Halin, es necesario tener identificados dichos componentes.

Como se mencionó en el capítulo anterior, en el trabajo de Gutiérrez-Medina, ya se conoce cuándo un vértice del sistema pertenece al ciclo de hojas del Halin o cuando éste pertenece al esqueleto del Halin. Aunado a esto, ya se cuenta con un árbol de expansión enraizado en el grafo sobre el cual se ejecutan los procesos de sincronización de mensajes denominado "árbol de comunicación". El problema de este árbol radica en que las hojas del ciclo no necesariamente son hojas en el árbol de comunicación y los vértices del esqueleto no necesariamente son vértices internos en este árbol de comunicación.

Por lo anterior, este capítulo se centra en construir un árbol de expansión enraizado nuevo, denominado "árbol del Halin". El algoritmo que se encarga de realizar este proceso es ROOTING_TREE. Este árbol es exactamente igual al árbol detectado por los algoritmos de Gutiérrez-Medina. La única diferencia es que este nuevo árbol se enraíza en un vértice que no sea hoja.

La clave para construir este nuevo árbol radica en dos puntos importantes: primero, si se remueven las aristas del ciclo de hojas, tal acción determina cuáles aristas sí están presentes en el árbol del Halin. Segundo, si se elige un vértice en particular como raíz, se define la dirección de las aristas del árbol, produciendo al final un árbol enraizado único.

Resulta conveniente que el vértice destinado como raíz para este árbol sea el líder de comunicación, siempre y cuando éste pertenezca al esqueleto del Halin, o el único vértice del esqueleto que es adyacente al líder de comunicación cuando éste pertenece al ciclo de hojas. El vértice que cumple con las condiciones para ser la raíz del árbol se denomina "raíz del Halin".

Una vez definida la raíz del Halin, todos los vértices adyacentes a él, toman a la raíz del Halin como su padre. Posteriormente, los vértices adyacentes a un vértice ya enraizado, identifican como su padre a aquel vértice de su vecindario abierto que ya se encuentre enraizado y que además la arista que los conecte no pertenezca al ciclo de hojas. Este proceso se repite hasta que el árbol esté enraizado.

Un fragmento del funcionamiento del algoritmo se ilustra en las figuras 18 y 19.

Considerando al vértice 1 como la raíz del árbol Halin, a los vértices $2, 3, 4, 7$ y 8 como vértices del ciclo de hojas y a los vértices 5 y 6 como vértices del esqueleto del Halin, el algoritmo haría lo siguiente:

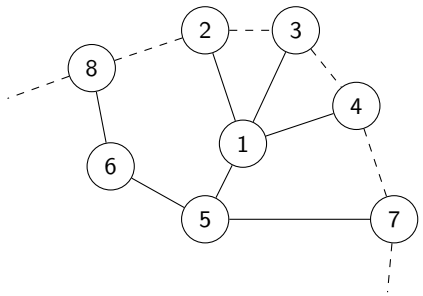

Figura 18. Fragmento del grafo Halin antes del enraizamiento del árbol del Halin.

- Paso 1: Inicialmente los vértices $2, 3, 4$ y 5 toman como padre a la raíz del árbol del Halin.
- Paso 2: El vértice 8 no puede tomar al vértice 2 como su padre, ya que la arista que existe entre ellos es del ciclo de hojas y esta asignación de padre/descendiente es inválida. Este mismo escenario sucede entre los vértices 7 y 4. Sólo el vértice 6 toma al vértice 5 como su padre ya que esta arista no pertenece al ciclo de hojas y el vértice 5 ya se encuentra enraizado.
- Paso 3: En vértice 7 ya puede tomar al vértice 5 como su padre, ya que el vértice 5 está enraizado y la arista que los conecta no pertenece al ciclo de hojas. Esto mismo sucede para el vértice 8, el cual toma al vértice 6 como su padre por los mismos motivos.

La salida producida por el algoritmo de enraizamiento del árbol se ilustra en la Figura 19. Ahora todos los vértices cuentan con un apuntador que identifica al vértice padre. Las aristas del ciclo no se incluyen en el árbol y la raíz del árbol del Halin es forzosamente un vértice del esqueleto del Halin.

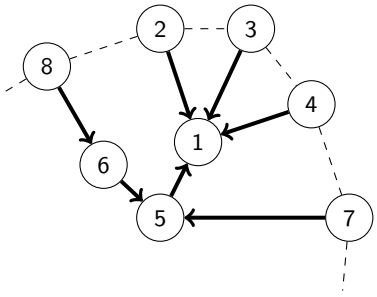

Figura 19. Mismo fragmento de Halin de la Figura 18 después de enraizar al árbol del Halin.

El algoritmo ROOTING_TREE toma como entrada la salida producida por la fase de enumeración de Gutiérrez-Medina descrito en el Capítulo 3. Así que el sistema ya cuenta con los componentes del Halin identificados, además de contar con un canal de comunicación para las funciones de autoestabilización.

La salida del algoritmo es el árbol del Halin enraizado. Las hojas del ciclo son las hojas del árbol, y los vértices del esqueleto son los vértices internos del árbol.

4.1. Variables y funciones del algoritmo Rooting_Tree

Las variables utilizadas en el algoritmo ROOTING_TREE se describen a continuación:

- $v.tree\_root$: Variable booleana que indica quién es la raíz del árbol del Halin. Toma el valor de 1 cuando v es la raíz del Halin; toma el valor de 0 en caso contrario.
- $v.tree\_parent$: Variable apuntadora que almacena el identificador del vértice que v reconoce como padre en el árbol del Halin.
- $v.tree\_distance$: Variable entera que determina la distancia que existe entre el vértice v y la raíz del Halin.
- $v.tree\_reset$: Variable booleana que toma el valor de 1 cuando la función $tree\_reset(v)$ se ha ejecutado en el vértice v en el paso anterior.
- $v.tree\_ready$: Variable booleana que toma el valor de 1 para indicar si el vértice v y sus descendientes ya se reiniciaron correctamente; toma el valor de 0 en caso contrario.
- $v.tree\_error$: Variable booleana que se usa para la transmisión de mensajes de error. Toma el valor de 1 cuando el vértice v o alguno de sus descendientes han detectado un error en sus variables; toma el valor de 0 en caso contrario.
- $v.tree\_done$: Variable booleana que se usa para propagar el mensaje de finalización del algoritmo. Toma el valor de 1 cuando todos los descendientes del vértice v y el propio v han terminado de enraizarse al árbol; toma el valor de 0 en caso contrario.

A continuación se describen las funciones utilizadas en el algoritmo ROOTING_TREE.

- $tree\_root(v)$ es una función que determina si el vértice v es la raíz del Halin o no. Esta función regresa 1 si lo siguiente es verdadero:

$$v.leaf = 0 \wedge (v.ldr = v \vee \exists\, w \in N(v) \mid w.leaf = 1 \wedge w.ldr = w) \tag{1}$$

Esta función se evalúa en verdadero si v es un vértice del esqueleto ($v.leaf = 0$) y además es el líder de comunicación ($v.ldr = v$). Cuando v es un vértice del esqueleto y no es el líder de comunicación, pero existe un vértice en su vecindario abierto tal que sea hoja del ciclo y además esta hoja sea el líder de comunicación ($\exists\, w \in N(v) \mid w.leaf = 1 \wedge w.ldr = w$), esta función también se evalúa en verdadero.

La función $tree\_root(v)$ regresa 0 si no se cumplen estas condiciones.

- La función $tree\_children(v)$ regresa todos los vértices del vecindario abierto de v tal que en su variable $tree\_parent$ almacenen el identificador de v, es decir, el conjunto de los hijos de v en el árbol enraizado del Halin.

$$\{w \in N(v) \mid w.tree\_parent = v\} \tag{2}$$

- La función $tree\_reset(v)$ restablece los valores de las variables del vértice v a su estado por omisión. Los valores reiniciados que toma v se muestran en el Pseudocódigo 1:

  El vértice v inicializa las variables de $v.tree\_distance$, $v.tree\_reset$, $v.tree\_ready$, $v.tree\_error$ y $v.tree\_done$ a los valores de $0, 1, 0, 0$ y 0, respectivamente.

  Se usa la función $tree\_root(v)$ para determinar si el vértice v es la raíz del Halin; en caso afirmativo, v inicializa la variable $v.tree\_parent$ a v. En caso contrario, la variable $v.tree\_parent$ se inicializa en Nil.

Pseudocódigo 1: Función $tree\_reset$

```
1  v.tree_root ← tree_root(v)
2  if tree_root(v) = 1 then
3  |  v.tree_parent ← v
4  else
5  |  v.tree_parent ← Nil
6  end
7  v.tree_distance ← 0
8  v.tree_reset ← 1
9  v.tree_ready ← 0
10 v.tree_error ← 0
11 v.tree_done ← 0
```

- La función $tree\_safe(v)$ verifica la integridad de las variables durante y después de la etapa de convergencia del algoritmo para enraizar al árbol del Halin. Se consideran dos escenarios: Cuando v tiene su variable $v.tree\_parent$ igual a Nil, es decir, cuando v aún no tiene su padre asignado y cuando $v.tree\_parent$ es distinto de Nil (cuando ya tiene a un padre asignado).

  Primero, la función $tree\_safe(v)$ evalúa las siguientes expresiones sin considerar si el vértice v está o no enraizado. La función $tree\_safe(v)$ regresa 0 si se satisface alguna de las siguientes expresiones:

$$v.tree\_root \neq tree\_root(v) \tag{3}$$

  En la expresión 3 se determina si existe una inconsistencia entre la información almacenada en $v.tree\_root$ y la salida de la función $tree\_root(v)$.

$$v.tree\_error = 1 \tag{4}$$

  Tener activada la bandera de $v.tree\_error$ indica que la información del vértice v es errónea.

  Cuando un vértice aún no tiene un padre asignado, la función $tree\_safe(v)$ regresa 0 si se cumple la siguiente expresión:

$$v.tree\_distance \neq 0 \tag{5}$$

  La expresión 5 detecta si un vértice v sin padre tiene almacenada en su variable $v.tree\_distance$ algo distinto a 0. Para los vértices sin padre, como aún no tienen una dirección entre ellos y la raíz del Halin, sólo se permite que almacenen 0 en su variable de distancia.

  Cuando un vértice ya cuenta con un padre identificado en su variable $v.tree\_parent$, la función $tree\_safe(v)$ regresa 0 si se cumple al menos una de las siguientes condiciones:

$$(v.tree\_parent).leaf = 1 \tag{6}$$

  En este caso, como el padre de v es una hoja del ciclo de hojas, implicaría que la arista que une a v y su padre es una arista del ciclo de hojas, o un vértice del esqueleto apunta a una hoja. Ambos casos son inválidos en la definición del árbol del Halin en el algoritmo Rooting_Tree.

$$v.tree\_root = 1 \wedge v.tree\_distance \neq 0 \tag{7}$$

En este escenario, v es la raíz del Halin y su variable $v.tree\_distance$ es distinta de 0. El padre de la raíz del Halin es la propia raíz, por lo que la distancia entre la raíz y su padre debe ser cero.

$$v.tree\_root = 0 \wedge v.tree\_distance \neq (v.tree\_parent).distance + 1 \tag{8}$$

Cuando v no es la raíz del Halin, la diferencia de distancia entre v y su padre debe corresponder a la distancia del padre más una unidad.

Si ninguna de las expresiones se satisface, los valores de las variables almacenadas en v se consideran seguros, por lo que la función $tree\_safe(v)$ regresa 1.

- La función $tree\_parent(v)$ determina el vértice del vecindario abierto de v que se designa como padre en el árbol enraizado del Halin.

$$!\exists\, w \in N(v) \mid w.leaf = 0 \wedge w.tree\_parent \neq Nil \tag{9}$$

Esta función regresa el índice del vértice w, vecino de v, tal que w no sea una hoja del ciclo de hojas, y donde w ya se encuentre enraizado. Si al momento de evaluar la función, v no encuentra ningún vértice que satisfaga dichas condiciones, la función $tree\_parent(v)$ regresa Nil.

- $tree\_ready(v)$ es la función encargada de determinar si un vértice está listo para comenzar a calcular los valores de sus variables. Esta función regresa 1 cuando una de las siguientes expresiones se cumple:

$$children(v) = \emptyset \wedge tree\_safe(v) = 1 \tag{10}$$

$$tree\_safe(v) = 1 \wedge w.tree\_ready = 1,\ \forall\, w \in children(v) \tag{11}$$

En la expresión 10 se contemplan los vértices que no tienen descendientes en el árbol de comunicación. Para dichos vértices, sólo es necesario que la función $tree\_safe(v)$ regrese 1.

En la expresión 11 se evalúan los vértices que tienen descendientes en el árbol de comunicación. Además de tener que cumplir con que $tree\_safe(v)$ regrese 1, todos los descendientes de v en el árbol de comunicación ya deben de tener su bandera $tree\_ready$ en 1.

La función $tree\_ready(v)$ regresa 0 si no se satisface ningún caso de los descritos previamente.

- $tree\_done(v)$ funciona de manera similar a la función $tree\_ready(v)$. La función $tree\_done(v)$ determina si un vértice ya finalizó de calcular los valores de sus variables para enraizar al árbol del Halin.

La función $tree\_done(v)$ regresa 1 si se cumple alguna de las siguientes condiciones:

$$v.leaf = 1 \wedge tree\_safe(v) = 1 \wedge v.tree\_parent \neq Nil \tag{12}$$

$$\begin{aligned} v.leaf = 0 \wedge tree\_safe(v) = 1 \wedge v.tree\_parent \neq Nil \\ \wedge (w.tree\_done = 1,\ \forall\, w \in tree\_children(v)) \end{aligned} \tag{13}$$

La ecuación 12 indica que una hoja v del ciclo de hojas ya finalizó el cálculo de los valores de sus variables cuando el estado de v es válido según la función $tree\_safe(v)$ y cuando v tenga un padre distinto de Nil almacenado en su variable $v.tree\_parent$.

La ecuación 13 cubre el caso cuando v es un vértice del esqueleto. Para este tipo de vértices, además de tener que estar en un estado válido,indicado por $tree\_safe(v)$, y estar enraizado a través de $v.tree\_parent$; también debe cumplir con que todos sus descendientes ya hayan indicado que finalizaron el cálculo de los valores de sus variables.

La función $tree\_done(v)$ regresa 0 cuando no se satisface ninguno de los dos escenarios anteriores.

4.2. Reglas del algoritmo Rooting_Tree

Las reglas presentadas en esta sección siguen la estructura propuesta en el trabajo de Gutiérrez-Medina.

El punto de entrada para el algoritmo ROOTING_TREE se presenta en la regla TREE_RESET mostrada en el Pseudocódigo 2. El líder de comunicación ejecuta la función $tree\_reset(v)$ de modo que los vértices descendientes detecten que la bandera $tree\_reset$ del padre está en 1 para que éstos se activen por la regla TREE_RESET.

Pseudocódigo 2: Regla TREE_RESET

```
1 if A_3 ∧ v.tree_reset = 0 ∧ v.par.tree_reset = 1 then
2 |  tree_reset(v)
3 end
```

La regla TREE_SYNC, mostrada en el Pseudocódigo 3, permite a los vértices descendientes de v que sincronicen el mensaje de reinicio, de modo que éste se propague exitosamente hasta las hojas del árbol de comunicación.

Pseudocódigo 3: Regla TREE_SYNC

```
1 if A_3 ∧ v.tree_reset = 1 then
2 |   v.tree_reset ← 0
3 end
```

La regla TREE_ERROR, mostrada en el Pseudocódigo 4, envía un mensaje de error al líder de comunicación si el vértice v o sus descendientes ya detectaron una inconsistencia en los valores de sus variables. Si el mensaje de error llega hasta el líder de comunicación, éste reinicia sus variables, comenzando así nuevamente el proceso de convergencia del algoritmo ROOTING_TREE.

Pseudocódigo 4: Regla TREE_ERROR

```
1 if A_3 ∧ tree_safe(v) = 0 ∨ ∃ w ∈ children(v) | w.tree_error = 1 then
2 |   v.tree_error ← 1
3 |   if safe_lider(v) = 1 then
4 |   |   tree_reset(v)
5 |   end
6 end
```

La regla TREE_READY se encarga de propagar el mensaje de un vértice v a su padre en el árbol de comunicación para indicar que v ya finalizó su reinicio y está listo para comenzar la creación del árbol enraizado. Esta regla se puede ver en el Pseudocódigo 5

Pseudocódigo 5: Regla TREE_READY

```
1 if A_3 ∧ v.tree_ready = 0 ∧ tree_ready(v) = 1 then
2 |   v.tree_ready ← 1
3 end
```

La regla TREE_ALG, mostrada en el Pseudocódigo 6, permite a los vértices que aún no se han enraizado a su padre, que empiecen el proceso de creación del árbol enraizado por medio de la función $tree\_parent(v)$.

Pseudocódigo 6: Regla TREE_ALG

```
1 if A_3 ∧ v.tree_ready = 1 ∧ v.tree_parent = Nil ∧ v.tree_parent ≠ tree_parent(v) then
2 |   v.tree_parent ← tree_parent(v)
3 end
```

La regla TREE_DONE, mostrada en el Pseudocódigo 7, envía un mensaje al líder de comunicación cuando el vértice v ya finalizó su parte en la construcción del árbol enraizado, i.e., si se satisface la función $tree\_done(v)$. Cuando la raíz del Halin se privilegia por esta regla, dicho vértice ejecuta la función de reinicio de Los siguientes algoritmos: CIF_SKELETON o TREE_DOMINATION, mediante las funciones $skeleton\_reset(v)$ y $domin\_reset(v)$, respectivamente. Dichas funciones se describen en los capítulos 5 y 8.

Pseudocódigo 7: Regla TREE_DONE

```
1 if A_3 ∧ v.tree_done = 0 ∧ tree_done(v) = 1 then
2 |   v.tree_done ← 1
3 |   if tree_root(v) = 1 then
4 |   |   skeleton_reset(v)
5 |   |   domin_reset(v)
6 |   end
7 end
```

4.3. Convergencia del algoritmo Rooting_Tree

A continuación se presentan los lemas que ayudan a demostrar la propiedad de convergencia del algoritmo ROOTING_TREE, así como el análisis de su tiempo de ejecución.

Lema 4.1 *El algoritmo* ROOTING_TREE *reinicia correctamente sus variables en el grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. Se demuestra por inducción sobre la distancia entre el líder de comunicación y cualquier vértice del árbol de comunicación.

Caso base: Cuando la distancia es cero, el líder de comunicación es el único vértice del sistema que se encuentra a dicha distancia con respecto a sí mismo. Debido a la ejecución de función $tree\_reset(v)$ del

algoritmo, este vértice reinicia correctamente sus variables según la definición de $tree\_reset(v)$, donde establece su bandera $v.tree\_reset$ en 1.

Cuando la distancia es uno, sin pérdida de generalidad, considere un vértice v a una distancia de una arista con respecto al líder de comunicación. Suponga dos escenarios en relación a su bandera $v.tree\_reset$: Cuando $v.tree\_reset$ es igual a 1 y cuando $v.tree\_reset$ es igual a 0.

Si $v.tree\_reset = 1$, por la Regla 3 y en la misma ranura de tiempo donde el líder de comunicación reinicia sus variables, v cambia su bandera $v.tree\_reset$ de 1 a 0. En la ranura de tiempo siguiente por la regla $\textsc{Tree\_Reset}$, mientras que el padre de v (el líder de comunicación) está ejecutando su acción de sincronización de reinicio, v ejecuta la función de reinicio definida en $tree\_reset(v)$, reiniciando sus variables correctamente en un número constante de ranuras de tiempo.

Si la variable $v.tree\_reset = 0$ en la ranura de tiempo donde el líder de comunicación reinicia sus variables, v no se privilegia por ninguna regla de reinicio o sincronización. En la ranura de tiempo siguiente, sucede el mismo escenario explicado en el caso anterior cuando el líder de comunicación se encuentra sincronizando su reinicio, reiniciando sus variables correctamente.

Hipótesis inductiva: Suponga que el algoritmo $\textsc{Rooting\_Tree}$ ha reiniciado correctamente los valores de las variables de todos los vértices desde la raíz hasta una distancia $k-1$ en $O(k-1)$ ranuras de tiempo.

Paso inductivo: Ahora se demuestra que en $O(1)$ ranuras de tiempo adicional, el algoritmo reinicia los valores de todas las variables hasta una distancia k de la raíz. Esta demostración es análoga a la demostración del reinicio de variables de distancia 1. Se supone el mismo escenario donde en la ranura de tiempo $k-1$, los padres de los vértices de distancia k reinician sus variables. En esa ranura de tiempo, los vértices de distancia k pueden o no privilegiarse por la regla $\textsc{Tree\_Sync}$, dependiendo del valor de su bandera $tree\_reset$. En la ranura de tiempo siguiente, la ranura de tiempo k, cuando los vértices de distancia $k-1$ se privilegian por la regla $\textsc{Tree\_Sync}$, los vértices de distancia k se privilegian por la regla $\textsc{Tree\_Reset}$, donde ejecutan la función $tree\_reset(v)$ y reinician sus variables correctamente en $O(1)$ ranuras de tiempo.

Como la hoja más lejana h_D se encuentra una distancia de D aristas con respecto al líder de comunicación, se demuestra que en $O(D)$ ranuras de tiempo el sistema se reinicia correctamente. □

Lema 4.2 *El algoritmo* $\textsc{Rooting\_Tree}$ *actualiza la variable* $tree\_ready$ *de* 0 *a* 1 *en todos los vértices*

del grafo en $O(D)$ ranuras de tiempo.

Demostración. Se demuestra por inducción sobre el nivel del árbol. La hoja más lejana h_0 se encuentra en el nivel cero y el líder de comunicación en el nivel D.

Caso base: En el nivel cero del árbol, por el Lema 4.1, la hoja h_D tiene reiniciadas sus variables. En la ranura de tiempo 0, la hoja h_0 se privilegia por la regla $\textsc{Tree\_Ready}$, debido a que cumple con las condiciones de la expresión 11 de la función $tree\_ready(v)$ y cumple con que su variable $h.tree\_ready$ vale 0. A través de la acción de la regla $\textsc{Tree\_Ready}$, la hoja h_0 actualiza correctamente su variable $tree\_ready$ de 0 a 1 en un número constante de ranuras de tiempo.

Hipótesis inductiva: Suponga que el algoritmo $\textsc{Rooting\_Tree}$ ha actualizado correctamente los valores de las variables $tree\_ready$ de 0 a 1 de todos los vértices desde el nivel cero hasta el nivel k en $O(k)$ ranuras de tiempo.

Paso inductivo: A continuación se demuestra que los vértices en el nivel $k+1$ actualizan su variable $tree\_ready$ de 0 a 1 en un número constante de ranuras de tiempo adicionales. Sin pérdida de generalidad, considere un vértice v a distancia $k+1$ con respecto al líder de comunicación. El vértice v en la ronda $k+1$ evalúa la función $tree\_ready(v)$ por la Regla 6. Mediante la Expresión 11 de esta función, el vértice cumple con ser seguro según $tree\_safe(v)$ debido a que éste se reinicio previamente por el Lema 4.1. También por la hipótesis inductiva, se supone que todos sus descendientes ya tienen su variable $tree\_ready$ en 1. Como v cumple satisfactoriamente la función $tree\_ready(v)$ y tiene su variable de $v.tree\_ready$ en 0, v se privilegia por la regla $\textsc{Tree\_Ready}$ y actualiza su variable $v.tree\_ready$ de 0 a 1 satisfactoriamente.

Como la distancia de la hoja h_0 con respecto al líder de comunicación es de a lo más D aristas, el algoritmo $\textsc{Rooting\_Tree}$ actualiza todas las variables de $tree\_ready$ de 0 a 1 en $O(D)$ ranuras de tiempo. □

Lema 4.3 *El algoritmo $\textsc{Rooting\_Tree}$ enraiza al árbol del Halin en $O(D)$ ranuras de tiempo.*

Demostración. Se demuestra por inducción sobre la distancia entre la raíz del Halin y los demás vértices. Suponga que h_D es la hoja más lejana a la raíz.

Caso base: Cuando la distancia desde la raíz del Halin es cero. La raíz del Halin es el único vértice a una distancia cero. Este vértice se enraíza correctamente en $O(1)$ ranuras de tiempo ya que sólo basta con que éste ya cuente con sus variables reiniciadas.

Vértices a distancia 1 de la raíz. Sin pérdida de generalidad, sea v un vértice a una distancia 1 de la raíz, en una ranura de tiempo, el vértice v por los lemas 4.1 y 4.2 cumple con que su variable $v.tree\_ready$ sea igual a 1 y que su variable $v.tree\_parent$ sea igual a Nil. Al ejecutar la función $tree\_parent(v)$ solicitada por la regla TREE_ALG, v detecta a un vértice w tal que no es hoja del ciclo de hojas y que su variable $w.tree\_parent$ es distinta de Nil. Dicho vértice es la raíz del Halin. Los demás vértices de su vecindario abierto no cuentan con un enraizamiento debido a que éstos sólo están reiniciados. Como se satisfacen todas las condiciones de la regla TREE_ALG, v actualiza su variable $v.tree\_parent$ de Nil al identificador del vértice que regresó $tree\_parent(v)$ (la raíz del Halin), enraizándose satisfactoriamente.

Hipótesis inductiva: Suponga que el algoritmo ROOTING_TREE ha enraizado correctamente todos los vértices desde la raíz hasta una distancia $k-1$ en $O(k-1)$ ranuras de tiempo.

Paso inductivo: Ahora se demuestra que en $O(1)$ ranuras de tiempo adicional, el algoritmo enraíza todos los vértices hasta una distancia k. La demostración del enraizamiento de los vértices a distancia k es análoga a la demostración del enraizamiento de los vértices a distancia 1. Sin pérdida de generalidad considere un vértice v de distancia k en la ranura de tiempo k, el cual tiene que elegir a un vértice de su vecindario abierto para tomarlo como padre y enraizarse. Mediante la función $tree\_parent(v)$, el vértice v sólo encuentra un único vértice que ya se encuentra enraizado. v lo toma como su padre a través de la regla TREE_ALG en la ranura de tiempo siguiente.

Como la hoja más lejana h_D se encuentra a una distancia de D aristas con respecto al líder de comunicación, se demuestra que el árbol queda enraizado en $O(D)$ ranuras de tiempo. □

Lema 4.4 *El algoritmo* ROOTING_TREE *actualiza la variable* $tree\_done$ *de* 0 *a* 1 *en todos los vértices del grafo en* $O(D)$ *ranuras de tiempo.*

Demostración. Se hace la suposición que todos los vértices del sistema ya reiniciaron y calcularon los valores de sus variables para el proceso de enraizamiento, consecuencia de los lemas 4.1, 4.2 y 4.3.

La demostración de este lema es análoga a la demostración del Lema 4.2, donde en lugar de cumplir con las condiciones de la regla TREE_READY ,se satisfacen las condiciones de la regla TREE_DONE. En TREE_DONE se debe cumplir que el vértice que verifica esta regla cumpla con que su variable $tree\_done$ sea igual a 0, consecuencia del Lema 4.1 y que su variable de $tree\_parent$ sea distinta de Nil, consecuencia de los lemas 4.2 y 4.3. □

Lema 4.5 *El algoritmo* ROOTING_TREE*, en ausencia de errores, enraíza al árbol del Halin en* $O(D)$

ranuras de tiempo.

Demostración. La demostración de este lema es consecuencia de las demostraciones de los lemas 4.1, 4.2, 4.3 y 4.4. □

Lema 4.6 *En presencia de una inconsistencia en las variables del sistema, el algoritmo* ROOTING_TREE *propaga un mensaje de error hasta el líder de comunicación y enraíza nuevamente al árbol del Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es por inducción sobre el nivel del árbol y es similar a la demostración del Lema 4.2, donde se analiza el caso de la hoja más lejana h_0 la cual satisface la regla TREE_ERROR y propaga el mensaje de error hasta el líder de comunicación mediante el árbol de comunicación.

Caso base: Cuando el nivel del árbol es cero y la hoja h_0 tiene inconsistencias, la Regla 3 detecta una configuración inválida mediante la función $tree\_safe(v)$. Una vez que la regla TREE_ERROR se privilegie, en un número constante de ranuras de tiempo, v altera su bandera $v.tree\_error$ de 0 a 1 iniciando la propagación del mensaje de error hasta el líder de comunicación.

Si el nivel del árbol es uno y suponga que v es padre de la hoja h_0, al igual que el caso base, v puede detectar inconsistencias en sus variables a través de la función $tree\_safe(v)$ de la regla TREE_ERROR, causando que inicie la propagación de su mensaje de error al líder de comunicación en un número constante de ranuras de tiempo. También, mediante la regla TREE_ERROR se detecta si un descendiente de v en el árbol de comunicación tiene activada su bandera $tree\_error$. De este modo, v se privilegia por TREE_ERROR y activa su bandera $v.tree\_error$, propagando el mensaje de error hacia su padre en el árbol de comunicación.

Hipótesis inductiva: Suponga que el algoritmo ROOTING_TREE ha propagado correctamente un mensaje de error desde el nivel cero hasta un vértice en el nivel k en $O(k)$ ranuras de tiempo.

Paso inductivo: A continuación se demuestra que el mensaje de error también se propaga hacia un vértice en el nivel $k+1$. La demostración es análoga a la demostración del caso base de vértices en el nivel 1, donde le toma hasta $O(1)$ ranuras de tiempo en que actualice su bandera $tree\_error$ de 0 a 1 y propagar el mensaje de error.

Como la altura del árbol de comunicación está acotada por el diámetro del grafo, se concluye que la propagación del mensaje de error toma $O(D)$ ranuras de tiempo.

Una vez que el mensaje de error llega al líder de comunicación, mediante la Regla 3, el líder de comunicación ejecuta la función de reinicio, causando que el sistema nuevamente se autoestabilize consecuencia del Lema 4.5. □

4.4. Cerradura del algoritmo Rooting_Tree

A continuación se demuestra que en ausencia de fallas y una vez que el sistema haya convergido a una configuración válida, éste permanece en una configuración válida.

Cuando el sistema converge, en cualquier vértice del grafo se cumplen las siguientes características:

1. $tree\_parent \neq Nil$
2. $tree\_reset = 0$
3. $tree\_error = 0$
4. $tree\_ready = 1$
5. $tree\_done = 1$

Como la bandera $tree\_reset$ está apagada, no satisface las condiciones de las reglas TREE_SYNC o TREE_RESET. Dado que la bandera error $tree\_error$ también está apagada, la regla TREE_ERROR no se activa mediante la expresión $\exists\, w \in children(v) \mid w.tree\_error = 1$. Como la bandera $tree\_ready$ está encendida, la regla TREE_READY no se privilegia. Como ya se tiene un padre identificado y almacenado en la variable $tree\_parent$, la regla TREE_ALG no se privilegia.

Finalmente, dado que la bandera $tree\_done$ se encuentra encendida una vez que el sistema se autoestabiliza, la regla TREE_DONE no se privilegia. Esto también implica que la función $tree\_safe(v)$ se haya cumplido, de modo que TREE_ERROR no se puede privilegiar por la expresión $tree\_safe(v) = 0$, requerida por la regla antes mencionada.

Como ninguna de las seis reglas para la autoestabilización se satisfacen, el estado global del sistema se mantiene intacto, demostrando así la propiedad de cerradura.

Las propiedades que cumplen los vértices, mostradas en el listado anterior, y junto con las características previas que cumple el sistema agrupadas en el predicado A_3, constituye el nuevo predicado A_5 (especificado en la Ecuación 14), indicando que el sistema ya cumple con el enraizamiento del árbol del Halin.

$$A_5 = A_3 \wedge tree\_parent \neq Nil \wedge tree\_reset = 0 \tag{14}$$
$$\wedge\, tree\_error = 0 \wedge tree\_ready = 1 \wedge tree\_done = 1$$

Capítulo 5. Algoritmo CIF_Skeleton

El objetivo del algoritmo CIF_Skeleton es encontrar un CIF_{max} en el esqueleto del Halin. La entrada a este algoritmo es un grafo Halin que satisface el predicado A_5, descrito en el Capítulo 4. Su salida es un grafo Halin que cumple el predicado A_5 y además cada vértice del esqueleto almacena un color para indicar si pertenece al CIF_{max} o no.

La estrategia general de pintado se basa en el trabajo de Flores-Lamas et al. (2020). En dicho trabajo se realiza un pintado en el esqueleto del Halin desde la raíz hasta las hojas.

El esqueleto sobre el cual se va a realizar el pintado corresponde a los vértices del Halin que no sean hojas del ciclo. El coloreado comienza en la raíz del Halin (introducido en el capítulo anterior), el cual toma el color Red, indicado que pertenece al CIF_{max}.

Los demás vértices del esqueleto leen el color y apuntador de su padre en el enraizamiento del árbol del Halin y calculan su coloreado según sea el caso.

A partir de este algoritmo y en adelante, la propagación de mensajes durante el proceso de autoestabilización se lleva a cabo en el árbol del Halin y no en el de comunicación. Similarmente, las relaciones ancestro/descendiente se refieren al árbol de Halin.

5.1. Variables y funciones del algoritmo CIF_Skeleton

Las nuevas variables utilizadas en el algoritmo CIF_Skeleton para identificar un CIF_{max} sobre el esqueleto del Halin son las siguientes:

- $v.halin\_color$: Variable que toma un valor del conjunto $\{Nil, Red, Blue_1, Blue_2\}$. Un vértice con color Nil indica ausencia de color. Los vértices color Red son aquellos que pertenecen al CIF_{max}. Los vértices $Blue_1$ indican que se encuentran a una distancia de una arista de un vértice Red. Los vértices $Blue_2$ indican que se encuentran a una distancia de 2 aristas con respecto a un vértice Red.
- $v.halin\_pointer$: Variable apuntadora que almacena el identificador de un vértice w del vecindario cerrado de v. Esta variable se utiliza para brindar información adicional a los vértices vecinos de v con el propósito de que ellos puedan calcular el valor de sus variables.

- $v.skeleton\_reset$: Variable booleana que toma el valor de 1 para indicar si el vértice v ya ejecutó la función de reinicio $skeleton\_reset(v)$; toma el valor de 0 en caso contrario.
- $v.skeleton\_error$: Variable booleana que toma el valor de 1 si el vértice v o alguno de sus descendientes en el árbol del Halin han detectado una inconsistencia en sus variables durante o después del proceso de autoestabilización en el algoritmo actual; toma el valor de 0 en caso contrario.
- $v.skeleton\_done$: Variable booleana que toma el valor de 1 si el vértice v y sus descendientes ya terminaron el cálculo del CIF_{max}; toma el valor de 0 en caso contrario.

Una descripción de las funciones necesarias para el algoritmo CIF_SKELETON se muestra a continuación:

- La función $skeleton\_reset(v)$, mostrada en el Pseudocódigo 8, reinicia las variables de los vértices de este algoritmo a su estado por defecto. En el caso de la raíz del Halin, inicializa sus variables a su color final.

  Para cualquier vértice del grafo, establece las variables de $skeleton\_reset$, $skeleton\_error$ y $skeleton\_done$ a 1, 0 y 0, respectivamente.

  Para la raíz del Halin, inicializa sus variables de $halin\_color$ y $halin\_pointer$ a Red y v, respectivamente. Para cualquier otro vértice del esqueleto, inicializa las variables de $halin\_color$ y $halin\_pointer$ a Nil y Nil, respectivamente. Las hojas del ciclo de hojas no reinician el valor de sus variables $halin\_color$ y $halin\_pointer$ en este algoritmo.

Pseudocódigo 8: Función $skeleton\_reset$

```
1  if v.leaf = 0 then
2      if v.tree_root = 1 then
3          v.halin_color ← Red
4          v.halin_pointer ← v
5      else
6          v.halin_color ← Nil
7          v.halin_pointer ← Nil
8      end
9  end
10 v.skeleton_done ← 0
11 v.skeleton_error ← 0
12 v.skeleton_reset ← 1
```

- La función $skeleton\_color(v)$ es la función que calcula el color del vértice v para el CIF_{max} en el esqueleto del Halin. Para cualquier vértice del esqueleto distinto a la raíz del Halin, el vértice v identifica a su padre en el árbol del Halin, lee su color, y calcula su propio color según los siguientes cuatro casos:

  1. Si el padre de v es Red: v asigna $Blue_1$ a su variable $halin\_color$ y el identificador del padre de v a su variable $halin\_pointer$. En la Figura 20 se ilustra este caso.

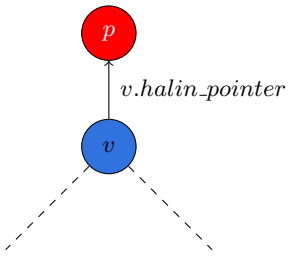

Figura 20. Caso 1 de coloreo. El vértice p representa al padre de v que es Red. el vértice v se vuelve $Blue_1$.

  2. Si el padre de v es $Blue_1$ y la variable apuntadora $halin\_pointer$ almacena a v: El vértice v asigna Red a su variable $halin\_color$ y asigna su propio identificador a su variable $halin\_pointer$. La Figura 21 ilustra este caso, donde el vértice p es el padre de v, es $Blue_1$ y apunta hacia v.

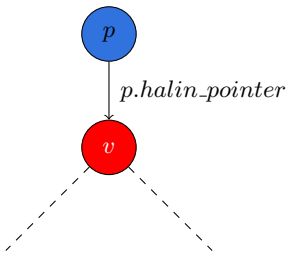

Figura 21. Caso 2 del coloreo. El padre de v es $Blue_1$ y apunta a v.

  3. Si el padre de v es $Blue_1$ y la variable apuntadora $halin\_pointer$ no apunta a v: Este caso se divide en dos escenarios:

     3.1 Cuando existe al menos un vértice hijo de v tal que no sea hoja del ciclo de hojas: El vértice v selecciona entre todos sus los posibles candidatos (hijos que no son hojas del ciclo de hojas) a aquel vértice cuyo identificador es más pequeño. La variable $halin\_pointer$ apunta a dicho hijo de v y v se vuelve $Blue_1$. La Figura 22 ilustra cómo v seleccionó al vértice 1, según el caso 3.1. En este ejemplo, los vértices $1, 2$ y 3 son hijos de v en el árbol del Halin y los tres son vértices del esqueleto.

3.2 Cuando todos los hijos de v son hojas del ciclo: El vértice v asigna $Blue_2$ a su variable $halin\_color$ y su propio identificador a su variable $halin\_pointer$. La Figura 23 muestra cómo todos los descendientes del vértice v, identificados por lf $(leaf)$, son hojas del ciclo, por lo que v cumple con el caso 3.2.

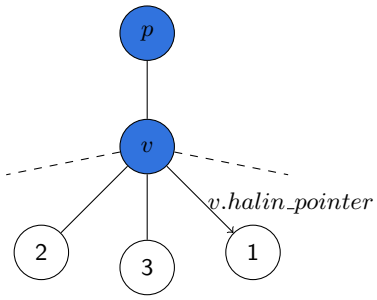

Figura 22. Caso 3.1 del coloreo.

- La función $skeleton\_safe(v)$ verifica que un vértice v del esqueleto y su color sean consistentes con sus variables y con las variables de su padre. Para los vértices del ciclo de hojas, esta función regresa siempre 1, independientemente del valor de sus variables.

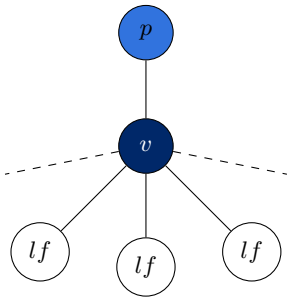

Figura 23. Caso 3.2 del coloreo.

Los vértices del esqueleto validan un conjunto de condiciones de acuerdo a su color.

Independientemente del color de v, la función $skeleton\_safe(v)$ regresa 0 si se cumple la siguiente expresión:

$$v.skeleton\_error = 1 \tag{15}$$

Cualquier vértice del esqueleto que tenga su bandera de $skeleton\_error$ en 1 se considera erróneo en el presente algoritmo.

Cuando el color de v es Nil, la función $skeleton\_safe(v)$ regresa 0 si se cumple alguna de las siguientes

expresiones:

$$v.tree\_root = 1 \tag{16}$$

La raíz del Halin sólo puede ser Red. Su color no puede ser Nil.

$$v.halin\_pointer \neq Nil \tag{17}$$

Los vértices Nil sólo pueden almacenar en su variable apuntadora el valor Nil, correspondiente a lo establecido por la función $skeleton\_reset(v)$.

$$v.skeleton\_done = 1 \tag{18}$$

Un vértice Nil no puede indicar que ha finalizado el cálculo de sus variables debido a que éste aún no ha determinado si pertenece o no al CIF_{max}.

Para vértices Red, la función $skeleton\_safe(v)$ regresa 0 si algo de lo siguiente se cumple:

$$v.tree\_root = 0 \wedge v.skeleton\_reset = 1 \tag{19}$$

Los vértices que no sean Nil no pueden tener su bandera de $skeleton\_reset$ en 1. Sólo la raíz del Halin puede tener su bandera $skeleton\_reset$ en 1 y ser Red debido a que éste es su estado por defecto, según la función $skeleton\_reset(v)$.

$$v.halin\_pointer \neq v \tag{20}$$

Debido a lo establecido por la función $skeleton\_color(v)$, los vértices Red deben almacenar su propio identificador en su variable $halin\_pointer$.

$$v.tree\_root = 0 \wedge (v.tree\_parent).halin\_color \neq Blue_1 \tag{21}$$

Cuando v es Red, su padre sólo puede almacenar $Blue_1$ en su variable $halin\_color$. Sólo el padre de la raíz del Halin puede ser Red (debido a que el padre de la raíz del Halin es la propia raíz).

$$(v.tree\_parent).halin\_pointer \neq v \tag{22}$$

Como v es Red, su padre debe apuntar a v, consecuencia del caso 2 de la función $skeleton\_color(v)$.

Cuando v es $Blue_1$, la función $skeleton\_safe(v)$ regresa 0 si se cumple alguna de las siguientes condiciones:

$$v.tree\_root = 1 \tag{23}$$

Al igual que la expresión 16, la raíz del Halin no puede ser distinta de Red.

$$v.skeleton\_reset = 1 \tag{24}$$

De manera similar a la expresión 19, un vértice distinto de Nil, no puede tener su bandera de $skeleton\_reset$ en 1.

$$v.halin\_pointer = Nil \vee v.halin\_pointer = v \tag{25}$$

Un vértice $Blue_1$ no puede apuntar a Nil o a sí mismo. Sólo es correcto que apunte a otro vértice Red de su vecindario abierto.

$$\begin{aligned}(v.halin\_pointer).halin\_color = Nil \vee (v.halin\_pointer).halin\_color = Blue_1 \\ \vee (v.halin\_pointer).halin\_color = Blue_2\end{aligned} \tag{26}$$

El vértice al que apunta v sólo puede ser Red.

$$(v.tree\_parent).skeleton\_pointer = v \tag{27}$$

El padre de v no puede apuntar a v debido a que v es $Blue_1$.

Finalmente, para los vértices cuyo color es $Blue_2$, si se cumple alguna de las siguientes condiciones, la función $skeleton\_safe(v)$ regresa 0:

$$v.tree\_root = 1 \tag{28}$$

Al igual que las expresiones 16 y 23, la raíz del Halin no puede ser distinta de Red.

$$v.skeleton\_reset = 1 \tag{29}$$

Al igual que la expresión 24, un vértice distinto de Nil, no puede tener su bandera de $skeleton\_reset$ en 1.

$$v.halin\_pointer \neq v \tag{30}$$

Al igual que la expresión 20, los vértices $Blue_2$ deben de almacenar su propio identificador en su variable $halin\_pointer$.

$$(v.tree\_parent).halin\_color = Nil \vee (v.tree\_parent).halin\_color = Red \tag{31}$$

Un vértice Red o Nil no puede estar junto a uno $Blue_2$ si ambos pertenecen al esqueleto.

$$(v.tree\_parent).halin\_pointer = v \tag{32}$$

El padre de v no puede apuntar a v por ser $Blue_2$. Sólo podría apuntar a v si v es Red.

Cuando no se cumple ninguna de las condiciones anteriores, mencionadas en la función $skeleton\_safe(v)$, dicha función regresa 1, indicando que el vértice es seguro y es válido en la configuración actual.

- La función booleana $skeleton\_done(v)$ determina si el vértice v ya finalizó el algoritmo CIF_Skeleton. Regresa 1 si se cumple alguna condición de las siguientes:

$$v.leaf = 1 \tag{33}$$

$$\begin{aligned} v.leaf = 0 \wedge skeleton\_safe(v) = 1 \wedge v.halin\_color \neq Nil \\ \wedge (w.skeleton\_done = 1, \forall\, w \in tree\_children(v)) \end{aligned} \tag{34}$$

La expresión 33 considera a los vértices del ciclo de hojas. Estos vértices por no pertenecer al esqueleto del Halin se consideran que ya finalizaron el algoritmo CIF_Skeleton sin importar los valores de sus variables.

La segunda expresión, descrita en 34, evalúa a los vértices del esqueleto. Esta expresión se convierte en verdadera cuando el vértice es seguro según la definición de $skeleton\_safe(v)$, es distinto de Nil, y cuando todos sus descendientes han indicado que ya finalizaron el algoritmo CIF_Skeleton.

La función $skeleton\_done(v)$ regresa 0 cuando ninguna de las dos expresiones anteriores se cumple.

5.2. Reglas del algoritmo CIF_Skeleton

Las reglas dedicadas a manejar la autoestabilización de este algoritmo son análogas a las reglas descritas en el Capítulo 4. De manera similar que en el capítulo anterior, el proceso de autoestabilización comienza en la raíz del Halin que ejecuta la función de reinicio $skeleton\_reset(v)$. Posteriormente, sus vértices descendientes se privilegian por la regla Skeleton_Reset continuando con la propagación del reinicio. Esta regla se puede observar en el Pseudocódigo 9

Pseudocódigo 9: Regla Skeleton_Reset

```
1 if A_5 ∧ v.skeleton_reset = 0 ∧ (v.tree_parent).skeleton_reset = 1 then
2 |   skeleton_reset(v)
3 end
```

La regla Skeleton_Sync, mostrada en el Pseudocódigo 10, al igual que la regla Tree_Sync genera un retardo de una unidad de tiempo para que los vértices hijos de v puedan reiniciarse.

Pseudocódigo 10: Regla Skeleton_Sync

```
1 if A_5 ∧ v.skeleton_reset = 1 then
2 |   v.skeleton_reset ← 0
3 end
```

La regla Skeleton_Error, mostrada en el Pseudocódigo 11, es análoga a la regla Tree_Error. Mediante esta regla se maneja el envío del mensaje de error cuando v o alguno de sus descendientes en el árbol del Halin han detectado una inconsistencia en sus variables.

La regla Skeleton_Alg, mostrada en el Pseudocódigo 12, permite a cada vértice Nil del esqueleto

calcular su color siempre y cuando sea seguro, no se encuentre sincronizando su mensaje de reinicio, y cuando no sea Nil.

Pseudocódigo 11: Regla SKELETON_ERROR

```
1 if A_5 ∧ skeleton_safe(v) = 0 ∨ (∃ w ∈ tree_children(v) | w.skeleton_error = 1) then
2 |   v.skeleton_error ← 1
3 |   if v.tree_root = 1 then
4 |   |   skeleton_reset(v)
5 |   end
6 end
```

Pseudocódigo 12: Regla SKELETON_ALG

```
1 if A_5 ∧ v.leaf = 0 ∧ skeleton_safe(v) = 1 ∧ v.halin_color = Nil ∧ v.skeleton_reset =
    0 ∧ (v.tree_parent).halin_color ≠ Nil then
2 |   skeleton_color(v)
3 end
```

La regla SKELETON_DONE, mostrada en el Pseudocódigo 13, es análoga a la regla TREE_DONE. Esta regla envía un mensaje al líder de comunicación cuando el vértice v ya finalizó el algoritmo CIF_SKELETON. Cuando la raíz del Halin se privilegia por esta regla, ésta ejecuta la función $cycle\_reset(v)$, comenzando así el algoritmo CIF_CYCLE.

Pseudocódigo 13: Regla SKELETON_DONE

```
1 if A_5 ∧ v.skeleton_done = 0 ∧ skeleton_done(v) = 1 then
2 |   v.skeleton_done ← 1
3 |   if v.tree_root = 1 then
4 |   |   cycle_reset(v)
5 |   end
6 end
```

5.3. Convergencia del algoritmo CIF_Skeleton

A continuación se describen los lemas que demuestran la propiedad de convergencia del algoritmo CIF_Skeleton y su análisis de tiempo de ejecución.

Lema 5.1 *El algoritmo* CIF_Skeleton *reinicia correctamente sus variables en grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del reinicio de variables para el algoritmo Rooting_Tree descrito en el Lema 4.1. □

Lema 5.2 *El algoritmo* CIF_Skeleton *calcula correctamente un* CIF_{max} *en el esqueleto del Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración formal de este lema se encuentra descrita en el trabajo de Flores-Lamas et al. (2020), en el cual se calcula y demuestra la validez del CIF_{max} para un grafo Halin en el contexto distribuido no autoestabilizante. □

Lema 5.3 *El Algoritmo* CIF_Skeleton *actualiza la variable* $tree\_done$ *de* 0 *a* 1 *en todos los vértices del grafo en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del Lema 4.4. □

Lema 5.4 *El Algoritmo* CIF_Skeleton*, en ausencia de errores, calcula un* CIF_{max} *en el esqueleto del Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es consecuencia de las demostraciones de los lemas 5.1, 5.2 y 5.3. □

Lema 5.5 *En presencia de una inconsistencia en las variables del grafo, el algoritmo* CIF_Skeleton *propaga un mensaje de error hasta la raíz del Halin y se reinicia el algoritmo* CIF_Skeleton *en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del Lema 4.6, donde para este algoritmo se utiliza la función $skeleton\_safe(v)$ para detectar las inconsistencias en lugar de la función $tree\_safe(v)$. □

5.4. Cerradura del algoritmo CIF_Skeleton

Después de que el sistema converja, lo siguiente se cumple para todos los vértices del sistema:

- $v.skeleton\_reset = 0$
- $v.skeleton\_error = 0$
- $v.skeleton\_done = 1$

Adicionalmente, para cualquier vértice en el esqueleto, lo siguiente también se cumple:

- $v.halin\_color \neq Nil$
- $v.halin\_pointer \neq Nil$

A continuación se demuestra que, en ausencia de fallas y dadas las características mencionadas en los listados anteriores, el sistema permanece sin alterar su estado:

Dado que todos los vértices tienen su variable $skeleton\_reset$ en 0, entonces las reglas SKELETON_RESET y SKELETON_SYNC no se ejecutan. Como todos los vértices cumplen con ser seguros, según la función $safe\_skeleton(v)$, entonces la regla SKELETON_ERROR no se activa. Dado que todos los vértices tienen su bandera de $skeleton\_done$ en 1, esto implica que la regla SKELETON_DONE no se cumple.

Como los vértices del esqueleto tienen un color válido distinto de Nil, la regla SKELETON_ALG no se satisface. Debido a que la segunda condición de la regla SKELETON_ALG solicita que el vértice sea del esqueleto, los vértices del ciclo de hojas no pueden satisfacer dicha regla.

Dado que ninguna regla se activa al llegar a este punto del algoritmo, el estado global del sistema permanece sin cambios, por lo que se demuestra que la propiedad de cerradura se ha cumplido satisfactoriamente.

El conjunto de condiciones mencionadas previamente junto con el predicado A_5 conforma la definición del predicado A_6, mostrada en la expresión 35, indicando que el algoritmo CIF_SKELETON ha finalizado.

$$A_6 = A_5 \wedge v.skeleton\_reset = 0 \wedge v.skeleton\_error = 0 \wedge v.skeleton\_done = 1 \tag{35}$$
$$\wedge ((v.leaf = 0 \wedge v.halin\_color \neq Nil \wedge v.halin\_pointer \neq Nil) \vee (v.leaf = 1))$$

Capítulo 6. Algoritmo CIF_Cycle

Ya se ha hablado de que es posible descomponer el grafo Halin en un árbol y un ciclo de hojas. Para calcular un CIF_{max} en un ciclo cualquiera, basta con incluir a un vértice del CIF_{max} a cada tercer vértice del ciclo partiendo desde un vértice arbitrario. En los grafos Halin, sin embargo, cualquier vértice del ciclo de hojas es vecino de un vértice del esqueleto, por lo que seguir la estrategia planteada anteriormente no se garantiza que se cumplan las condiciones del CIF_{max}. Lo anterior se debe a que el CIF_{max} en el ciclo puede tener conflictos con el CIF_{max} del esqueleto, y por lo tanto no se pueden calcular en forma independiente.

Dado que ya existe un coloreado asignado a los vértices del esqueleto, generado por el algoritmo CIF_Skeleton, esta información la utiliza el algoritmo CIF_Cycle para asignar un coloreado válido al ciclo de hojas en esta clase de grafos.

La idea detrás del funcionamiento del algoritmo CIF_Cycle se tomó del trabajo propuesto por Flores-Lamas et al. (2020).

La entrada al algoritmo CIF_Cycle es un grafo Halin que ya pasó por el algoritmo CIF_Skeleton (descrito en el Capítulo 5), de modo que los vértices del esqueleto ya cuentan con un coloreado correcto. La salida producida por el algoritmo CIF_Cycle es un grafo Halin, el cual ya incluye el coloreado de un CIF_{max} en el ciclo, respetando las condiciones del CIF_{max}, sin alterar la configuración actual de los vértices del esqueleto.

6.1. Variables y funciones del algoritmo CIF_Cycle

Las nuevas variables introducidas en el algoritmo CIF_Cycle son las siguientes:

- $v.cycle\_sktcolor$: Variable que almacena el color del vecino del esqueleto cuando v es un vértice del ciclo de hojas. Los posibles valores que puede tomar esta variable corresponden a los posibles valores que puede tomar la variable $halin\_color$, descritos en el Capítulo 5.
- $v.cycle\_reset$: Variable booleana indicadora que toma el valor de 1 si el vértice v ya ejecutó la función de reinicio $cycle\_reset(v)$.
- $v.cycle\_ready$: Variable booleana que indica si el vértice v y todos sus descendientes ya reiniciaron correctamente sus variables y se encuentran listos para calcular su coloreado.

- $v.cycle\_error$: Variable booleana que indica cuando v o sus descendientes han detectado un error en el algoritmo CIF_CYCLE.
- $v.cycle\_done$: Variable booleana que toma el valor de 1 si el vértice v y todos sus descendientes ya finalizaron el cálculo del CIF_{max}. Toma el valor de 0 en otro caso.

Las variables de $v.halin\_color$ y $v.halin\_pointer$, descritas en el Capítulo 5, se utilizan en este algoritmo para identificar un CIF_{max} en el ciclo de hojas.

A continuación se describe las funciones utilizadas en el algoritmo CIF_CYCLE:

- La función $cycle\_sktneighbor(v)$ regresa aquel vértice único w en el vecindario abierto de v tal que w pertenezca al esqueleto del Halin. Esta función requiere que v sea un vértice del ciclo.

$$!\exists\, w \in N(v) \mid w.leaf = 0 \tag{36}$$

- $cycle\_pred(v)$ es una función que regresa el vértice p del vecindario abierto de v tal que p sea del ciclo de hojas y además su variable $p.nbr$ sea igual a la variable $(v.nbr - 1) \mod |V_C|$, es decir, aquel vértice que se encuentre inmediatamente antes que v en el ciclo de hojas según su enumeración. Esta función al igual que $cycle\_sktneighbor(v)$ requiere que v sea del ciclo de hojas.

$$!\exists\ p \in N(v) \mid p.nbr = (v.nbr - 1) \mod |V_C| \tag{37}$$

- $cycle\_succ(v)$ es una función análoga a $cycle\_pred(v)$ con la diferencia de que ésta regresa al vértice s que se encuentra inmediatamente después de v en el ciclo de hojas.

$$!\exists\ s \in N(v) \mid s.nbr = (v.nbr + 1) \mod |V_C| \tag{38}$$

- La función $cycle\_reset(v)$, ilustrada en el Pseudocódigo 14, reinicia las variables del vértice v. El valor por defecto de cualquier vértice del grafo Halin sobre las variables $cycle\_reset, cycle\_ready, cycle\_error$ y $cycle\_done$ corresponde a $1, 0, 0$ y 0, respectivamente.

  Para el caso del representante del ciclo, éste inicializa sus variables $halin\_color$ y $halin\_pointer$ a los valores $Blue_1$ y $cycle\_sktneighbor(v)$, respectivamente. Para los demás vértices del ciclo, sus variables $halin\_color$ y $halin\_pointer$ se inicializan en Nil y Nil.

Esta función también reinicia el valor de la variable $cycle\_sktcolor$ al valor del color almacenado en el vértice vecino del esqueleto, calculado por $cycle\_sktneighbor(v)$.

Pseudocódigo 14: Función $cycle\_reset$

```
1  if v.leaf = 1 then
2      v.cycle_sktcolor ← cycle_sktneighbor(v).halin_color
3      if v.lf_rep = v then
4          v.halin_color ← Blue_1
5          v.halin_pointer ← cycle_sktneighbor(v)
6      else
7          v.halin_color ← Nil
8          v.halin_pointer ← Nil
9      end
10 end
11 v.cycle_reset ← 1
12 v.cycle_ready ← 0
13 v.cycle_error ← 0
14 v.cycle_done ← 0
```

- La función $cycle\_color(v, p, q, s)$ determina el color y la variable apuntadora de un vértice v en ciclo de hojas. Esta función recibe como parámetros cuatro vértices: El vértice v, al cual se va calcular su color, el predecesor p, definido en $cycle\_pred(v)$; el vértice vecino del esqueleto q, definido en $cycle\_sktneighbor(v)$; y el vértice sucesor s, definido en $cycle\_succ(v)$. La Figura 24 ilustra los vértices a los que un vértice del ciclo de hojas v es adyacente, sobre los cuales requiere conocer su estado para el cálculo de su color.

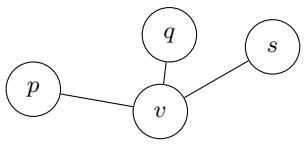

Figura 24. Información requerida para calcular el color de v en el ciclo de hojas. El vértice v calcula su color, p es el vértice predecesor de v en el ciclo, q es el vecino de v en el esqueleto, y s es el sucesor de v en el ciclo.

La información detallada del cálculo del color y apuntador de v dado todos los posibles colores del

vértice predecesor p, el vecino del esqueleto q y el sucesor s, se puede consultar en el trabajo de Flores-Lamas et al. (2020). Un fragmento de las posibles combinaciones consideradas por la función $cycle\_color(v)$ se puede ver en la Tabla 1.

Tabla 1. Fragmento de tabla de todas las posibles combinaciones contempladas en $cycle\_color(v)$ del vértice predecesor p, vecino del esqueleto q y sucesor s para determinar las variables de v en el pintado del ciclo de hojas.

$(p.halin\_clr, ptr)$	$(q.halin\_clr, ptr)$	$s.cycle\_sktclr$	$(v.halin\_clr, v.halin\_ptr)$
(Red, p)	(Red, p)	*cualquiera*	*combinación imposible*
(Red, p)	$(Blue_1, x \neq v \in N(q))$	*cualquiera*	$(Blue_1, p)$
(Red, p)	$(Blue_2, q)$	*cualquiera*	$(Blue_1, p)$
...	...	...	...

- La función $cycle\_safe(v)$ es la función booleana que determina si un vértice del ciclo es consistente con sus variables y de que éstas cumplan las condiciones del CIF_{max}.

  De manera análoga a la función $skeleton\_safe(v)$, descrita en el Capítulo 5, los vértices del esqueleto siempre generan 1 como salida de esta función, independientemente de sus variables. Dependiendo del color del vértice v del ciclo, se aplica un conjunto de validaciones para v, su vecino predecesor p del ciclo y su vecino q en el esqueleto.

  Independientemente del color de v, la función $cycle\_safe(v)$ regresa 0 si alguna de las siguientes condiciones se cumple:

$$v.cycle\_sktcolor \neq (cycle\_sktneighbor(v)).halin\_color \tag{39}$$

$$v.cycle\_error = 1 \tag{40}$$

  La expresión 39 verifica que el color almacenado en la variable $cycle\_sktneighbor$ corresponda al color real de su vecino del esqueleto.

  La expresión 40, por otro lado, verifica la variable $cycle\_error$. Si esta bandera se encuentra activada, v se considera con información errónea en el presente algoritmo.

  Cuando el color de v es Nil, las siguientes condiciones hacen que la función regrese 0:

$$v.halin\_pointer \neq Nil \tag{41}$$

  La expresión 41 verifica la variable apuntadora $halin\_pointer$ de v. Esta variable sólo puede ser Nil

cuando el vértice v es Nil, por ser la salida de la función $cycle\_reset(v)$ cuando v se encuentra reiniciado.

$$v.cycle\_done = 1 \tag{42}$$

Un vértice Nil no puede tener encendida su bandera $cycle\_done$ debido a que aún no ha finalizado el cálculo de sus variables.

Para los vértices Red del ciclo de hojas, la función $cycle\_safe(v)$ regresa 0 si se cumple alguna de las siguientes condiciones:

$$v.cycle\_reset = 1 \tag{43}$$

La bandera $cycle\_reset$ no puede estar encendida en este caso debido a que para que v sea Red, v tuvo que haber reiniciado y sincronizado su mensaje de reinicio en ranuras de tiempo anteriores.

$$v.halin\_pointer \neq v \tag{44}$$

Por la definición del color de la función $cycle\_color(v)$, cada vértice Red siempre debe apuntarse a sí mismo a través de su variable $halin\_pointer$.

$$(cycle\_pred(v)).halin\_color = Nil \wedge (cycle\_pred(v)).halin\_color = Red \tag{45}$$

El predecesor de un vértice Red en el ciclo sólo puede ser $Blue_1$ o $Blue_2$.

$$(cycle\_pred(v)).halin\_color = Blue_1 \vee (cycle\_pred(v)).halin\_pointer \neq v \tag{46}$$

La expresión 46 verifica si el predecesor de v es $Blue_1$ y su apuntador no va dirigido a v.

$$(cycle\_sktneighbor(v)).halin\_color = Red \tag{47}$$

La expresión 47 verifica cuando dos vértices Red (v y el vecino de v en el esqueleto) son adyacentes entre si.

Cuando v es $Blue_1$, la función $cycle\_safe(v)$ devuelve 0 si algo de lo siguiente se cumple:

$$(cycle\_pred(v)).halin\_color = Red \wedge v.halin\_pointer \neq cycle\_pred(v) \tag{48}$$

Si el predecesor de v en el ciclo es Red, v debe de apuntar a su predecesor.

$$(cycle\_sktneighbor(v)).halin\_color = Red \wedge v.halin\_pointer \neq cycle\_sktneighbor(v) \tag{49}$$

Al igual que la expresión 48, si el vecino de v en el esqueleto es Red, v debe apuntar a su vecino en el esqueleto.

$$(cycle\_pred(v)).halin\_color = Red \wedge (cycle\_sktneighbor(v)).halin\_color = Red \tag{50}$$

La expresión 50 verifica que dos vértices Red (el vecino predecesor de v y el vecino de v en el esqueleto) se encuentran a una distancia de dos aristas.

Finalmente, cuando v es $Blue_2$, la función $cycle\_safe(v)$ regresa 0 cuando al menos una de las siguientes expresiones se cumple:

$$v.halin\_pointer \neq v \tag{51}$$

Al igual que en la expresión 44, v debe validar que su apuntador esté dirigido a sí mismo.

$$(cycle\_pred(v)).halin\_color = Red \vee (cycle\_pred(v)).halin\_color = Nil \tag{52}$$

La expresión 52 verifica que un vértice $Blue_2$ no puede tener como predecesor en el ciclo a un vértice Red o Nil (situaciones imposibles de generarse mediante $cycle\_color(v)$).

$$(cycle\_sktneighbor(v)).halin\_color = Red \tag{53}$$

La expresión 53 verifica el caso en que el vecino de v en el esqueleto es Red, cuya combinación de colores resultan imposible de generarse con la función $cycle\_color(v)$.

Si ninguna de las condiciones anteriores se satisface, la función $cycle\_safe(v)$ regresa 1, indicando que la configuración de las variables de v con respecto a su entorno son válidas y seguras.

- La función $cycle\_ready(v)$ verifica que el vértice v se encuentra listo para comenzar a calcular sus variables. Esta función regresa 1 si algo de lo siguiente se cumple:

$$v.leaf = 1 \wedge cycle\_safe(v) = 1 \tag{54}$$

$$v.leaf = 0 \wedge cycle\_safe(v) = 1 \wedge w.cycle\_ready = 1, \forall\, w \in tree\_children(v) \tag{55}$$

La expresión 54 verifica que los vértices hojas sólo deben de ser seguros según la definición de $cycle\_safe(v)$.

Los vértices del esqueleto los verifica la función $cycle\_ready(v)$ de acuerdo a la expresión 55. Estos vértices deben ser seguros según $cycle\_safe(v)$ y todos sus vértices descendientes deben haber indicado que se encuentran listos para calcular su color, mediante la bandera $cycle\_ready$.

Si ninguna condición se cumple, $cycle\_ready(v)$ regresa 0.

- La función $cycle\_done(v)$ evalúa si un vértice del grafo ya concluyó con el algoritmo $\mathrm{CIF\_Cycle}$. Esta función regresa 1 si alguna de las siguientes condiciones se cumple:

$$v.leaf = 1 \wedge v.halin\_color \neq Nil \wedge cycle\_safe(v) = 1 \tag{56}$$

$$\begin{aligned} v.leaf = 0 \wedge v.halin\_color \neq Nil \wedge cycle\_safe(v) = 1 \\ \wedge\, w.cycle\_done = 1, \forall w \in tree\_children(v) \end{aligned} \tag{57}$$

La primera condición, mostrada en la expresión 56, cubre el caso cuando v es una hoja del ciclo. En este caso, v debe ser distinto de Nil y debe ser seguro según las condiciones contempladas en $cycle\_safe(v)$.

La segunda expresión, mostrada en 57, cubre el caso para los vértices del esqueleto. La función regresa 1 cuando v es distinto de Nil, es seguro según $cycle\_safe(v)$, y cuando todos los descendientes de v ya han indicado en su variable $cycle\_done$ que finalizaron el algoritmo $\mathrm{CIF\_Cycle}$.

Esta función regresa 0 si no se cumple ninguno de los dos escenarios descritos.

6.2. Reglas del algoritmo CIF_Cycle

A continuación se describen las reglas utilizadas en el algoritmo CIF_Cycle.

La regla Cycle_Reset, mostrada en el Pseudocódigo 15, es análoga a la regla Skeleton_Reset del Capítulo 5. Inicialmente, la raíz del Halin ejecuta la función de reinicio $cycle\_reset(v)$, donde posteriormente los descendientes de la raíz se privilegian por la regla Cycle_Reset, comenzando así el reinicio de variables del sistema.

Pseudocódigo 15: Regla Cycle_Reset

```
1 if A_6 ∧ (v.cycle_reset = 0) ∧ (v.tree_parent).cycle_reset = 1) then
2 |   cycle_reset(v)
3 end
```

La regla Cycle_Sync, mostrada en el Pseudocódigo 16, es análoga a la regla Skeleton_Sync. Esta regla permite a los vértices descendientes de v que puedan ejecutar correctamente la regla de reinicio Cycle_Reset.

Pseudocódigo 16: Regla Cycle_Sync

```
1 if A_6 ∧ (v.cycle_reset = 1) then
2 |   v.cycle_reset ← 0
3 end
```

La regla Cycle_Error, mostrada en el Pseudocódigo 17, es análoga a la regla Skeleton_Error del Capítulo 5. En esta regla, cualquier vértice que no sea seguro según la definición de $cycle\_safe(v)$ o donde cualquier vértice hijo que tenga en sus variables activada la bandera de error, hace que v encienda su bandera $cycle\_error$, propagando el mensaje de error hasta la raíz del Halin. Cuando el mensaje llegue a la raíz, ésta comienza nuevamente el algoritmo CIF_Cycle.

La regla Cycle_Ready, descrita en el Pseudocódigo 18, es análoga a la regla Tree_Ready del Capítulo 4. En esta regla se propagan los mensajes de que los vértices ya se encuentran listos para comenzar el cálculo de su color en el ciclo.

Pseudocódigo 17: Regla CYCLE_ERROR

```
1 if A_6 ∧ ((cycle_safe(v) = 0) ∨ (∃ w ∈ tree_children(v) | w.cycle_error = 1)) then
2 |   v.cycle_error ← 1
3 |   if v.tree_root = 1 then
4 |   |   cycle_reset(v)
5 |   end
6 end
```

Pseudocódigo 18: Regla CYCLE_READY

```
1 if A_6 ∧ v.cycle_ready = 0 ∧ cycle_ready(v) = 1 then
2 |   v.cycle_ready ← 1
3 end
```

La regla CYCLE_ALG, mostrada en el Pseudocódigo 19, permite a los vértices Nil del ciclo a calcular su color mediante la función $cycle\_color(v, p, q, s)$.

Pseudocódigo 19: Regla CYCLE_ALG

```
1 if A_6 ∧ v.leaf = 1 ∧ cycle_safe(v) = 1 ∧ v.halin_color = Nil ∧ v.cycle_reset =
    0 ∧ (w.ready = 1 ∀ w ∈ N[v]) then
2 |   cycle_color(v, cycle_pred(v), cycle_sktneighbor(v), cycle_succ(v))
3 end
```

La regla CYCLE_DONE, mostrada en el Pseudocódigo 20, es análoga a la regla SKELETON_DONE. Se envía un mensaje al padre de v a través del árbol del Halin cuando v y todos sus descendientes ya finalizaron el algoritmo CIF_CYCLE. Cuando la raíz del Halin reciba todos los mensajes de que sus hijos ya finalizaron el algoritmo CIF_CYCLE, la raíz comienza el algoritmo CIF_ADJUST (ver Capítulo 7).

Pseudocódigo 20: Regla CYCLE_DONE

```
1 if A_6 ∧ (v.cycle_done = 0) ∧ (cycle_done(v) = 1) then
2 |   v.cycle_done ← 1
3 |   if v.tree_root = 1 then
4 |   |   adjust_reset(v)
5 |   end
6 end
```

6.3. Convergencia del algoritmo CIF_Cycle

A continuación se muestran los lemas necesarios para demostrar la propiedad de convergencia del algoritmo CIF_Cycle y su análisis de tiempo de ejecución.

Lema 6.1 *El algoritmo* CIF_Cycle *reinicia correctamente sus variables en el grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del reinicio de variables para el algoritmo Rooting_Tree, descrito en el Lema 4.1. Primero se demuestra que la raíz del Halin se reinicia correctamente mediante la función $cycle\_reset(v)$ en una ranura de tiempo, y este proceso de reinicio se va propagando hasta la hoja más lejana h_D a una distancia de D aristas. □

Lema 6.2 *El algoritmo* CIF_Cycle *actualiza la variable* $cycle\_ready$ *de* 0 *a* 1 *en todos los vértices del grafo en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración de la actualización de la variable $tree\_ready$, descrito en el Lema 4.2 para el algoritmo Rooting_Tree. □

Lema 6.3 *El algoritmo* CIF_Cycle *calcula un* CIF_{max} *en el ciclo de hojas del grafo Halin en* $O(n)$ *ranuras de tiempo.*

Demostración. La demostración formal de este lema se encuentra descrita en el trabajo de Flores-Lamas et al. (2020). La demostración muestra que la función $cycle\_color(v, p, q, s)$ contempla todos las posibles combinaciones de color del vértice predecesor de v en el ciclo, vecino de v en el esqueleto, sucesor de v en el ciclo y en el vecino del esqueleto del sucesor de v. □

Lema 6.4 *El algoritmo* CIF_Cycle *actualiza la variable* $cycle\_done$ *de* 0 *a* 1 *en todos los vértices del grafo en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del Lema 4.4, donde se demuestra la actualización de la variable $tree\_done$ para el Rooting_Tree. □

Lema 6.5 *El Algoritmo* CIF_Cycle*, en ausencia de errores, calcula un* CIF_{max} *en el ciclo hojas del Halin en* $O(n)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es consecuencia de las demostraciones de los lemas 6.1, 6.2, 6.3 y 6.4. □

Lema 6.6 *En presencia de una inconsistencia en las variables del sistema, el algoritmo* CIF_CYCLE *propaga un mensaje de error hasta la raíz del Halin y calcula nuevamente el* CIF_{max} *en el ciclo de hojas en* $O(n)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del Lema 4.6. □

6.4. Cerradura del algoritmo CIF_Cycle

La demostración de la propiedad de cerradura del algoritmo CIF_CYCLE es análoga a la demostración de cerradura del algoritmo CIF_SKELETON.

Una vez que el sistema converge a una configuración válida, las siguientes condiciones para cualquier vértice son verdaderas:

- $v.halin\_color \neq Nil$
- $v.halin\_pointer \neq Nil$
- $v.cycle\_reset = 0$
- $v.cycle\_error = 0$
- $v.cycle\_ready = 1$
- $v.cycle\_done = 1$

Ya que todos los vértices tienen en su variable $cycle\_reset$ el valor 0, implica que ningún vértice puede privilegiarse por las reglas CYCLE_RESET o CYCLE_SYNC.

Dado que todos los vértices cumplen con ser seguros según $cycle\_safe(v)$, entonces ningún vértice se privilegia por la regla CYCLE_ERROR. Adicionalmente, ya que todos los vértices tienen su bandera $cycle\_error$ apagada, ningún vértice puede propagar el mensaje de error a través de la regla CYCLE_ERROR.

Como todos los vértices tienen activada su bandera de $cycle\_ready$, entonces la regla CYCLE_READY no se activa. Dado que todos los vértices del ciclo son distintos de Nil, ningún vértice se privilegia por la regla CYCLE_ALG. Finalmente, ya que todos los vértices cuentan con su bandera $v.cycle\_done$ encendida, entonces ningún vértice puede privilegiarse por la regla CYCLE_DONE.

Ya que en ningún momento ningún vértice se privilegia por las seis reglas de este algoritmo, entonces la configuración global se mantiene sin cambios, por lo que se demuestra que la propiedad de cerradura se cumple.

Las nuevas condiciones que cumplen los vértices y junto con las condiciones definidas en el predicado A_6, se define el predicado A_7 (ilustrada en la expresión 58), el cual garantiza la identificación completa de un CIF_{max} en el grafo Halin.

$$A_7 = A_6 \wedge v.cycle\_reset = 0 \wedge v.cycle\_error = 0 \wedge v.cycle\_ready = 1 \wedge v.cycle\_done = 1 \quad (58)$$
$$\wedge\, v.halin\_color \neq Nil \wedge v.halin\_pointer \neq Nil$$

Capítulo 7. Algoritmo CIF_Adjust

Dado que se supone que se han ejecutado los algoritmos CIF_Skeleton y CIF_Cycle, existe un CIF_{max} identificado en el grafo Halin, donde los vértices Red son los que se encuentran dentro del conjunto y los vértices $Blue_1$ y $Blue_2$ los que no.

Si bien, este conjunto es válido, el grafo queda con una ligera inconsistencia con respecto a la definición de los vértices $Blue_2$: Después de ejecutar los algoritmos CIF_Skeleton y CIF_Cycle, algunos vértices $Blue_2$ (los cuales representan aquellos vértices a distancia 2 de un vértice Red) se encuentran adyacentes a un vértice Red. Un ejemplo de esta situación se puede observar en la Figura 25. En el ejemplo, los vértices p y q (vértices $Blue_2$) son adyacentes al vértice v, el cual pertenece al CIF_{max}.

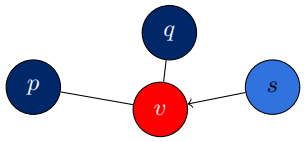

Figura 25. Inconsistencia en vértices $Blue_2$. Después de ejecutar los algoritmos CIF_Skeleton y CIF_Cycle, los vértices $Blue\_2$ p y q son adyacentes al vértice Red v, lo cual contradice su definición.

El objetivo del algoritmo CIF_Adjust es modificar el color de aquellos vértices $Blue_2$ sin alterar la cardinalidad del CIF_{max}.

La Figura 26 ilustra la salida del algoritmo CIF_Adjust en el grafo mostrado previamente. Los vértices p y q han actualizado sus colores de $Blue_2$ a $Blue_1$.

Figura 26. Inconsistencias de color resueltas por el algoritmo CIF_Adjust. a) Fragmento de grafo con inconsistencias en los colores de p y q. b) Colores corregidos a $Blue_1$ en los vértices p y q.

La entrada a este algoritmo es un grafo Halin con un CIF_{max} que cumple con el predicado A_7. La salida de este algoritmo es el grafo Halin con la corrección de colores necesaria.

7.1. Variables y funciones del algoritmo CIF_Adjust

Las nuevas variables requeridas por el algoritmo CIF_Adjust se listan a continuación:

- $v.adjust\_color$: Esta variable es análoga a la variable $halin\_color$. En esta variable se almacena el color actualizado del vértice v en caso de ser necesario.
- $v.adjust\_pointer$: Variable apuntadora análoga a la variable $halin\_pointer$. De manera similar a $v.adjust\_color$, en esta variable se almacena la información actualizada del apuntador.
- $v.adjust\_reset$: Variable booleana que indica si un vértice ya ejecutó la función $adjust\_reset(v)$.
- $v.adjust\_error$: Variable booleana encargada de identificar errores en el sistema. Toma el valor de 1 si se han detectado inconsistencias a través de la función $adjust\_safe(v)$.
- $v.adjust\_done$: Variable booleana que toma el valor de 1 si v y todos sus descendientes ya indicaron que terminaron la ejecución del algoritmo CIF_Adjust.

Una descripción de las funciones utilizadas en el algoritmo CIF_Adjust se presenta a continuación:

- La función $adjust\_reset(v)$, mostrada en el Pseudocódigo 21, inicializa las variables del vértice v a su estado por omisión. Para cualquier vértice v del grafo, esta función asigna a las variables $v.adjust\_reset$, $v.adjust\_error$, $v.adjust\_done$ los valores 1, 0 y 0, respectivamente. Las variables $v.adjust\_color$ y $v.adjust\_pointer$ se inicializan con los valores almacenados en $v.halin\_color$ y $v.halin\_pointer$.

Pseudocódigo 21: Función $adjust\_reset$

```
1 v.adjust_color ← v.halin_color
2 v.adjust_pointer ← v.halin_pointer
3 v.adjust_done ← 0
4 v.adjust_error ← 0
5 v.adjust_reset ← 1
```

- La función $adjust\_color(v)$, mostrada en el Pseudocódigo 22, es la función que determina el color y el apuntador final que el vértice v debe almacenar en sus variables $v.adjust\_color$ y $v.adjust\_pointer$, respectivamente. El Pseudocódigo 22 ilustra su funcionamiento.

Cuando el valor almacenado en $v.halin\_color$ es Red o $Blue_1$, entonces la función regresa lo almacenado en $v.halin\_color$ y $v.halin\_pointer$ ya que el vértice no es $Blue_2$.

Cuando el valor almacenado en la variable $v.halin\_color$ es $Blue_2$, se verifica si existe un vértice Red en su vecindario. En caso de que esto sea cumpla, el color de v se debe actualizar, por lo que la función regresa la tupla $(Blue_1, w)$ con los valores actualizados.

Si ninguna de las dos condiciones se cumple, implica que el vértice v de color $Blue_2$ se encuentra rodeado de vértices $Blue_1$ y $Blue_2$, por lo que no es necesario actualizar sus variables.

Pseudocódigo 22: Función $adjust\_color$

```
1 if v.halin_color = Red ∨ Blue_1 then
2 |   return (v.halin_color, v.halin_pointer)
3 end
4 if v.halin_color = Blue_2 ∧ (!∃ w ∈ N(v) | w.halin_color = Red) then
5 |   return (Blue_1, w)
6 else
7 |   return (v.halin_color, v.halin_pointer)
8 end
```

- La función $adjust\_safe(v)$ verifica la integridad de las variables de este algoritmo. Regresa 0 si alguna de las siguientes condiciones se cumple:

$$v.adjust\_error = 1 \tag{59}$$

$$\begin{aligned}(v.adjust\_color, v.adjust\_pointer) &\neq (v.halin\_color, v.halin\_pointer) \\ \wedge\, (v.adjust\_color, v.adjust\_pointer) &\neq adjust\_color(v)\end{aligned} \tag{60}$$

La expresión 59, al igual que en los algoritmos anteriores, produce que la función regrese 0 cuando se tiene encendida la bandera de error.

La expresión 60 detecta si el nuevo color y la variable apuntadora (almacenados en $v.halin\_color$ y $v.halin\_pointer$, respectivamente) no corresponden a las producidas por el coloreado original.

La función $adjust\_safe(v)$ regresa 1 si ninguna de las expresiones se cumplió, indicando así que el v es válido en la configuración actual.

- $adjust\_done(v)$ es una función que verifica si v ya finalizó la ejecución del algoritmo CIF_Adjust. Se debe cumplir con una de las siguientes condiciones:

$$v.leaf = 1 \wedge adjust\_safe(v) = 1 \wedge (v.adjust\_color, v.adjust\_pointer) = adjust\_color(v) \quad (61)$$

$$v.leaf = 0 \wedge adjust\_safe(v) = 1 \wedge (v.adjust\_color, v.adjust\_pointer) = adjust\_color(v) \quad (62)$$
$$\wedge\, w.adjust\_done = 1, \forall w \in tree\_children(v)$$

La expresiones 61 y 62 verifican a los vértices del ciclo de hojas y del esqueleto, respectivamente.

Para cualquier caso, v ser seguro según la definición en $adjust\_safe(v)$ y la tupla de variables $(v.adjust\_color, v.adjust\_pointer)$ debe corresponder con la salida de la función $adjust\_color(v)$.

Los vértices del esqueleto deben cumplir además con que todos sus descendientes hayan indicado que finalizaron el algoritmo CIF_Adjust.

La función $adjust\_done(v)$ regresa 1 si una de las dos condiciones mencionadas se cumple; egresa 0 en caso contrario.

7.2. Reglas del algoritmo CIF_Adjust

La regla Adjust_Reset, mostrada en el Pseudocódigo 23, es la encargada de propagar el mensaje y funcionalidad de reinicio en el algoritmo. Esta regla es análoga a las reglas de reinicio de fase Tree_Reset, Skeleton_Reset y Cycle_Reset.

Pseudocódigo 23: Regla Adjust_Reset

```
1 if A_7 ∧ (v.adjust_reset = 0) ∧ ((v.tree_parent).adjust_reset = 1) then
2 |   adjust_reset(v)
3 end
```

ADJUST_SYNC, mostrada en el Pseudocódigo 24, es una regla análoga a las reglas TREE_SYNC, SKELETON_SYNC y CYCLE_SYNC. Esta regla permite a los vértices descendientes de v que tengan tiempo suficiente para ejecutar la regla ADJUST_RESET.

Pseudocódigo 24: Regla ADJUST_SYNC

```
1 if A_7 ∧ (v.adjust_reset = 1) then
2 |   v.adjust_reset ← 0
3 end
```

La regla ADJUST_ERROR, mostrada en el Pseudocódigo 25, es la encargada de propagar un mensaje de error desde el vértice que ha detectado el error hasta la raíz del Halin. Esta regla es análoga a las reglas TREE_ERROR, SKELETON_ERROR y CYCLE_ERROR.

Pseudocódigo 25: Regla ADJUST_ERROR

```
1 if A_7 ∧ ((adjust_safe(v) = 0) ∨ (∃w ∈ tree_children(v) | w.adjust_error = 1)) then
2 |   v.adjust_error ← 1
3 |   if v.tree_root = 1 then
4 |   |   adjust_reset(v)
5 |   end
6 end
```

La regla ADJUST_ALG, mostrada en el Pseudocódigo 26, es la encargada de efectuar la actualización de color en caso de ser requerida. Esta regla es análoga a las reglas TREE_ALG, SKELETON_ALG y CYCLE_ALG.

Pseudocódigo 26: Regla ADJUST_ALG

```
1 if A_7 ∧ (adjust_safe = 1) ∧ (v.adjust_color, v.adjust_pointer) ≠ adjust_color(v) then
2 |   (v.adjust_color, v.adjust_pointer) ← adjust_color(v)
3 end
```

La regla ADJUST_DONE, mostrada en el Pseudocódigo 27, es la que propaga los mensajes de finalización del algoritmo desde las hojas del árbol de comunicación hasta su líder. Esta regla es análoga a las reglas TREE_DONE, SKELETON_DONE y CYCLE_DONE.

Pseudocódigo 27: Regla ADJUST_DONE

```
1 if A_7 ∧ (v.adjust_done = 0) ∧ (adjust_done(v) = 1) then
2 |   v.adjust_done ← 1
3 end
```

7.3. Convergencia del algoritmo CIF_Adjust

Se utilizan los siguientes lemas para demostrar la propiedad de convergencia y el análisis del tiempo de ejecución.

Lema 7.1 *El algoritmo* CIF_ADJUST *reinicia correctamente sus variables en el grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a las demostraciones de reinicio de variables para el algoritmo ROOTING_TREE, CIF_SKELETON y CIF_CYCLE, descritos en los lemas 4.1, 5.1 y 6.1, respectivamente. □

Lema 7.2 *El algoritmo* CIF_ADJUST *actualiza correctamente los vértices* $Blue_2$ *a* $Blue_1$*, en caso de ser necesario, sin alterar el* CIF_{max} *original, en* $O(D)$ *ranuras de tiempo.*

Demostración. Se demuestra por inducción sobre la distancia desde la raíz del Halin hasta la hoja más lejana h_D.

Para esta demostración, se supone que el grafo tiene reiniciadas sus variables correctamente, consecuencia del Lema 7.1.

Caso base: Vértices a una distancia de una arista de la raíz del Halin. Sin pérdida de generalidad, suponga que existe un vértice v a una distancia de una arista con respecto a la raíz del Halin cuyo color es $Blue_2$ y éste lo debe actualizar la función $adjust\_color(v)$. En una ranura de tiempo, v se privilegia por la regla ADJUST_ALG, debido a que cumple con las condiciones necesarias para actualizar su color. Debido a que se privilegió por ADJUST_ALG, v cambia de $Blue_2$ a $Blue_1$ junto a su variable apuntadora en las nuevas variables $adjust\_color$ y $adjust\_pointer$, sin alterar las variables originales de color y apuntador en un número constante de ranuras de tiempo.

Hipótesis inductiva: Se supone que el sistema actualiza todos los vértices $Blue_2$ a $Blue_1$, en caso de ser necesario, desde una distancia 1 hasta a una distancia de $k-1$ aristas con respecto a la raíz, en $O(k-1)$ ranuras de tiempo y sin alterar la configuración original.

Paso inductivo: La demostración de la actualización de color para los vértices de distancia k es análoga a la demostración de la actualización de color de los vértices a una distancia 1. En la ranura de tiempo k, un vértice cualquiera a una distancia k ya se encuentra reiniciado y listo para verificar la regla ADJUST_ALG. Si dicho vértice se privilegia por esta regla, el vértice actualiza correctamente sus variables $adjust\_color$ y $adjust\_pointer$ según la salida de la función $adjust\_color(v)$, sin alterar las variables originales.

Como la hoja más lejana h_D se encuentra una distancia de D aristas con respecto a la raíz del Halin, se concluye que el sistema actualiza correctamente los vértices $Blue_2$ a $Blue_1$, sin alterar la configuración original, en $O(D)$ ranuras de tiempo. □

Lema 7.3 *El algoritmo* CIF_ADJUST *actualiza la variable de* $adjust\_done$ *de* 0 *a* 1 *en todos los vértices del grafo en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a las demostraciones de los lemas 4.4, 5.3 y 6.4. □

Lema 7.4 *El Algoritmo* CIF_ADJUST*, en ausencia de errores, actualiza las vértices* $Blue_2$ *a* $Blue_1$*, en caso de ser necesario, y sin afectar la configuración original del* CIF_{max} *en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es consecuencia de los lemas 7.1, 7.2 y 7.3. □

Lema 7.5 *En presencia de una inconsistencia en las variables del sistema, el algoritmo* CIF_ADJUST *propaga un mensaje de error hasta la raíz del Halin y calcula nuevamente la actualización de color en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a la demostración del Lema 4.6. □

7.4. Cerradura del algoritmo CIF_Adjust

Al finalizar el algoritmo CIF_ADJUST, las siguientes propiedades se cumplen para cualquier vértice en el grafo:

- $adjust\_reset = 0$
- $adjust\_error = 0$
- $adjust\_done = 1$
- $(adjust\_color, adjust\_pointer) = adjust\_color(v)$

A continuación se presenta la demostración de que el estado no cambia, en ausencia de errores, una vez cumplidas las propiedades anteriores:

Los vértices no se pueden privilegiar por las reglas ADJUST_RESET y ADJUST_SYNC debido a que ningún vértice en el grafo tiene activada su bandera de $adjust\_reset$. Como la variable $adjust\_error$ se encuentra desactivada, entonces ningún vértice se privilegia por la regla ADJUST_ERROR. Dado que $(adjust\_color, adjust\_pointer) = updated\_color(v)$, entonces la regla ADJUST_ALG no puede activarse.

Finalmente, como todos los vértices del grafo tienen activada su bandera de $adjust\_done$, entonces la regla ADJUST_DONE no puede activarse. Como ningún vértice del sistema se puede privilegiar por ninguna regla del algoritmo, el sistema no modifica su estado y por lo tanto la propiedad de cerradura se cumple.

El predicado A_8, como lo ilustra la expresión 63, garantiza la identificación completa del CIF_{max} en el grafo Halin, cumpliendo la definición del coloreado descrita en el Capítulo 5.

$$A_8 = A_7 \wedge v.adjust\_reset = 0 \wedge v.adjust\_error = 0 \wedge v.adjust\_done = 1 \quad (63)$$
$$\wedge (adjust\_color, adjust\_pointer) = adjust\_color(v)$$

Capítulo 8. Algoritmo TREE_DOMINATION

Este capítulo se centra en el diseño del algoritmo TREE_DOMINATION, para encontrar un conjunto dominante total minimal (CDT_{min}) en el grafo Halin. Al igual que en los capítulos anteriores, se utiliza la descomposición del grafo Halin para el diseño de este algoritmo. Primero, se describe un algoritmo para encontrar un CDT_{min} en un árbol. Posteriormente, se muestra cómo adaptar dicho algoritmo al grafo Halin:

La entrada al algoritmo TREE_DOMINATION es un árbol enraizado. Su salida es un CDT_{min}. Las variables mencionadas durante la explicación general de este algoritmo se describirán a detalle en la sección 8.1.

Mediante la variable booleana $v.domin\_dominating$ (que indica cuándo un vértice está dentro o fuera del conjunto), las hojas del árbol indican que quedan fuera del conjunto. Mediante la variable booleana $v.domin\_worked$, las hojas comunican a su padre que ellos han realizado sus cálculos en sus variables, de modo que los padres tengan oportunidad para determinar si pertenecen o no al conjunto. Los demás vértices del sistema comienzan a calcular sus variables cuando todos sus descendientes tienen activada la bandera $domin\_worked$.

Un vértice interno v que sabe que todos sus hijos ya calcularon los valores sus variables ($w.domin\_worked = 1 \mid \forall\, w \in tree\_children(v)$), busca entre sus hijos si existe al menos uno que no se encuentre dominado.

Para saber si un vértice v es dominado por algún otro, se usa la variable entera $v.domin\_qtydominants$. En esta variable se almacena la cantidad de vértices dominantes que existen en el vecindario abierto de v. Esta variable se mantiene actualizada en cada ranura de tiempo al ejecutar la función $domin\_qtydomin-ants(v)$.

Si un vértice v que ya puede comenzar a calcular los valores de sus variables detecta que al menos uno de sus hijos no es dominado, v asigna a su variable $v.domin\_dominating$ el valor de 1 para indicar que v ya es dominante y así dominar a todos sus hijos no dominados. Este procedimiento continúa hasta que la raíz calcule sus variables.

En la figuras 27, 28, 29, 30 y 31 se ilustra un ejemplo del funcionamiento del algoritmo sobre un fragmento de un árbol durante las primeras ranuras de tiempo. La entrada al algoritmo se ilustra en la Figura 27.

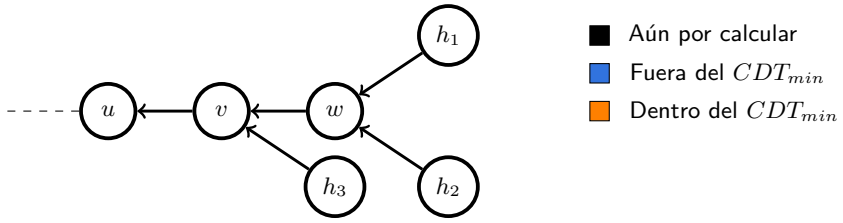

Figura 27. Entrada al algoritmo para calcular un CDT_{min}. Ranura de tiempo $t = 0$.

En la ranura de tiempo $t = 0$, los vértices h_1,h_2 y h_3 asignan a su variable $domin\_dominating$ el valor de 0, ya que cualquier hoja del árbol queda fuera del conjunto. En su variable $domin\_worked$ asignan el valor de 1 indicando que han realizado sus cálculos en sus variables y así sus respectivos padres puedan comenzar el cálculo de los suyos.

Los vértices u, v y w, en esta misma ranura de tiempo, no pueden comenzar a realizar cálculos en sus variables debido a que en este instante las hojas aún almacenan 0 en sus variables $domin\_worked$.

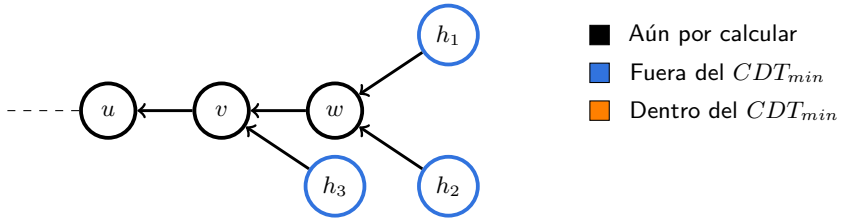

Figura 28. Ejemplo de cálculo de un CDT_{min}. Ranura de tiempo $t = 1$. Las hojas determinaron que no debían entrar al conjunto

En la ranura de tiempo $t = 1$, ilustrado en la Figura 28, el vértice w cumple con que todos sus descendientes tienen encendida su bandera $domin\_woked$, por lo que w calcula el valor sus variables. Como w encuentra que tanto su hijo h_1 y su hijo h_2 no son dominados por nadie, w activa su bandera $w.domin\_dominating$, entrando así al conjunto dominante y dominar a todos sus hijos.

El vértice w enciende además su bandera $w.domin\_worked$ para indicar que ya ha realizado los cálculos en sus variables. En esta misma ranura de tiempo, u y v aún no pueden comenzar a calcular los valores de sus variables ya que u aún está a la espera de que v encienda su bandera y v está a la espera de que w haga lo mismo.

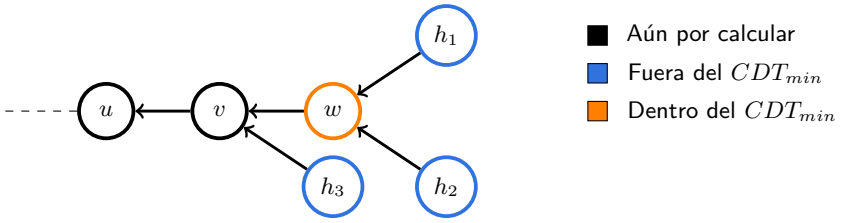

Figura 29. Ejemplo de cálculo de un CDT_{min}. Ranura de tiempo $t = 2$.

En la ranura de tiempo $t = 2$, ilustrado en la Figura 29, el vértice v comienza a calcular los valores de sus variables. Como encuentra que ni h_3 ni w están dominados, v entra al CDT_{min}. Como v ya determinó si debe pertenecer o no al CDT_{min}, v enciende su bandera $v.domin\_worked$.

En esta misma ranura de tiempo, las hojas h_1, h_2 y el mismo vértice v descubren que son adyacentes a exactamente un vértice dominante (w), mediante la función $domin\_qtydominants(v)$, por lo que ellos incrementan su variable $domin\_qtydominats$ de 0 a 1.

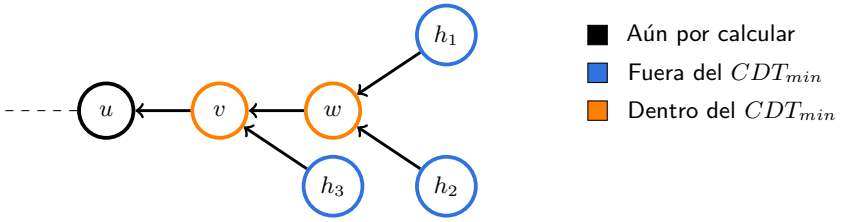

Figura 30. Ejemplo de cálculo de un CDT_{min}. Ranura de tiempo $t = 3$.

Cuando la ranura de tiempo es $t = 3$, como se observa en la Figura 30, u descubre que v tiene activada su bandera $v.domin\_worked$, por lo que u comienza a calcular los valores de sus variables. Como u nota que v ya es dominado por alguien (debido a que en su variable $v.domin\_qtydominants$ almacena el valor de 1), u decide que no debe de entrar al conjunto dominante ya que no necesita dominar a ningún hijo. El vértice u cambia su bandera $u.domin\_worked$ de 0 a 1.

En esta misma ranura de tiempo, tanto u como h_3 descubren que v es dominante, por lo que incrementan sus variables de $domin\_qtydominants$ de 0 a 1, indicando que son dominados por un vértice.

La salida del algoritmo aplicado sobre el ejemplo de la Figura 27 se puede observar en la Figura 31. El estado del árbol corresponde al comienzo de la ranura $t = 4$.

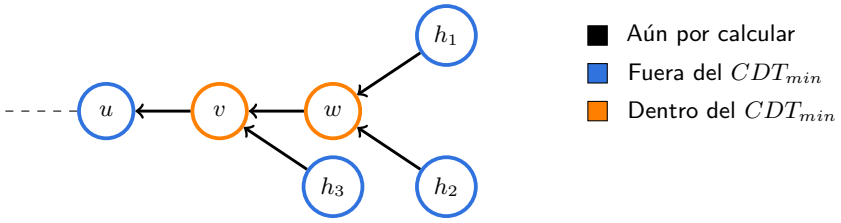

Figura 31. Ejemplo de cálculo de un CDT_{min}. Ranura de tiempo $t = 4$.

8.1. Variables y funciones del algoritmo TREE_DOMINATION

El listado de variables introducidas para el algoritmo TREE_DOMINATION se describe a continuación:

- $v.domin\_dominating$: Variable booleana que distingue a los vértices que pertenecen al CDT_{min}. Toma el valor de 1 si v pertenece al CDT_{min}; toma el valor de 0, en caso contrario.
- $v.domin\_qtydominants$: Variable entera que almacena la cantidad de vértices dominantes que se encuentra en el vecindario abierto de v. Tener un 0 en esta variable indica que v no tiene vecinos dominantes y por lo tanto v no está dominado.
- $v.domin\_worked$: Variable booleana que indica si un vértice v ya realizó los cálculos en sus variables para permitir al padre de v realizar el cálculo de sus variables cuando todos los hermanos de v tengan encendida esta bandera. Toma el valor de 0 cuando v aún no ha realizado los cálculos en sus variables; toma el valor de 1 en caso contrario.
- $v.domin\_pointer$: Variable apuntadora encargada de manejar casos especiales cuando la raíz del árbol no está dominada en la etapa final de la ejecución del algoritmo. Toma el valor del identificador de algún hijo.
- $v.domin\_reset$: Variable booleana encargada de manejar los mensajes de reinicio del algoritmo. Toma el valor de 1 si v ya ejecutó la función $domin\_reset(v)$.
- $v.domin\_error$: Variable booleana encargada de manejar los mensajes de error del algoritmo. Toma el valor de 1 si v detecta una configuración inválida al ejecutar la función $domin\_safe(v)$.
- $v.domin\_done$: Variable booleana encargada de manejar los mensajes de finalización del algoritmo. Toma el valor de 1 si v y todos sus descendientes ya terminaron el cálculo de sus variables determinado por la función $domin\_done(v)$.

Las funciones introducidas para el algoritmo TREE_DOMINATION se describen a continuación:

- La función $domin\_qtydominants(v)$, como se muestra en expresión 64, regresa el total de vértices dominantes en el vecindario abierto de v.

$$|\{w \in N(v) \mid w.domin\_dominating = 1\}| \tag{64}$$

- La función $domin\_reset(v)$ inicializa las variables a su estado por defecto. Estos valores varían dependiendo de si v pertenece al ciclo de hojas o no. En el Pseudocódigo 28 se ilustra los valores que toma el vértice v al reiniciarse.

  Los vértices del ciclo de hojas asignan a su variable $domin\_worked$ el valor de 1, ya que no calculan nada, permitiendo así a sus padres a que comiencen a calcular los valores de sus propias de variables. Los demás vértices del árbol asignan a su variable $domin\_worked$ el valor 0, debido a que necesitan esperar a que todos sus hijos hayan terminado de calcular los valores de sus variables.

  Todos los vértices quedan fuera del CDT_{min} y asignan en su variable $domin\_qtydominants$ el valor de 0, para indicar que aún no son dominados por nadie.

Pseudocódigo 28: Función $domin\_reset$

```
1  if v.leaf = 1 then
2  |  v.domin_worked ← 1
3  else
4  |  v.domin_worked ← 0
5  end
6  v.domin_dominating ← 0
7  v.domin_qtydominants ← 0
8  v.domin_pointer ← Nil
9  v.domin_reset ← 1
10 v.domin_error ← 0
11 v.domin_done ← 0
```

- La función $domin\_safe(v)$ valida el estado y la integridad de las variables de v y las de sus vecinos. A través de esta función, se identifican las configuraciones que incumplen las condiciones del CDT_{min},

generando así un mensaje de error. La función $domin\_safe(v)$ regresa el valor de 0 si se presenta alguno de los siguientes escenarios:

$$v.domin\_dominating = 1 \wedge (w.domin\_qtydominants \geq 2, \forall\, w \in N(v)) \tag{65}$$

La expresión 65 valida la propiedad de minimalidad. Si v pertenece al CDT_{min} y dado que todos sus vecinos son dominados por al menos 2 vértices dominantes, v descubre que es un vértice redundante.

$$v.halintree\_raiz = 0 \wedge v.domin\_pointer \neq Nil \tag{66}$$

La expresión 66 valida el uso de la variable $domin\_pointer$. Solo la raíz tiene la capacidad de utilizar esta variable para seleccionar a un hijo y que éste lo domine a ella en caso de ser necesario.

$$\begin{aligned} &v.domin\_done = 1 \wedge (v.tree\_parent).domin\_done = 1 \\ &\wedge\, v.domin\_qtydominants \neq domin\_qtydominants(v) \end{aligned} \tag{67}$$

A través de la expresión 67, se identifica si la cantidad de vértices dominantes que tiene almacenado v en su variable $domin\_qtydominants$ no coincide con lo calculado a través de la función $domin\_qtydominants(v)$. El vértice v y su padre ya deben haber finalizado el cálculo de sus variables.

$$v.domin\_done = 1 \wedge v.domin\_worked = 0 \tag{68}$$

La expresión 68 indica que para tener encendida la bandera de finalización del algoritmo, v debe haber calculado los valores de sus variables.

$$v.domin\_error = 1 \tag{69}$$

La expresión 69 indica que la bandera de error debe encontrarse apagada en todo momento siempre y cuando el sistema no haya experimentado fallas transitorias.

Cuando un vértice cumple con alguna condición de la función $domin\_safe(v)$, significa que sus variables no cumplen con una configuración válida, por lo que es necesario que el sistema se autoestabilize

nuevamente. La función $domin\_safe(v)$ regresa el valor de 1 si no se cumplió ninguna de las condiciones listadas, indicando así que el vértice es seguro para la configuración actual.

- La función $domin\_dominate(v)$ es la encargada de determinar si un vértice puede entrar o no al CDT_{min}. En el Pseudocódigo 29 se muestra las condiciones de esta función:

  Esta función se encarga de efectuar distintas tareas para encontrar un CDT_{min} en un árbol:

  - Con la expresión $\exists\, w \in tree\_children(v) \mid w.domin\_qtydominants = 0$, se identifica si un vértice hijo no es dominado por nadie. Cuando esto sucede, v actualiza su variable $v.domin\_dominating$ y le asigna el valor de 1, indicando así que v entra al CDT_{min} para dominar a todos sus hijos no dominados.
  - Con la instrucción $v.domin\_qtydominants \leftarrow domin\_qtydominants(v)$ se mantiene actualizada la variable $domin\_qtydominants$.
  - Al actualizar la bandera $v.domin\_worked$ a 1, el vértice v le comunica a su padre que v ya ha realizado los cálculos en sus variables.

  Esta función también contempla el caso cuando la raíz del árbol no es dominada durante el procedimiento de las líneas 1 a la 5. Un ejemplo de este caso se muestra en la Figura 32.

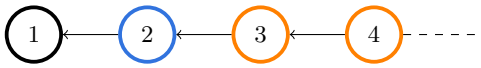

Figura 32. Caso especial del algoritmo TREE_DOMINATION cuando no se domina a la raíz.

Teniendo como raíz al vértice 1 y dada una ranura de tiempo k, lo siguiente sucedería en las próximas ranuras de tiempo:

En la ranura de tiempo k, a través de la función $domin\_dominate(v)$, la raíz 1 verifica que al menos un hijo no sea dominado. Dado que su único hijo, el vértice 2, es dominado por el vértice 3, la raíz no satisface la condición de la línea 1 de esta función, así que la raíz no cambia a dominante.

Como la raíz es el único vértice del árbol que puede satisfacer la condición de la línea 6 del Pseudocódigo 29, la raíz, a través de las líneas 7 a 11, selecciona entre todos sus hijos a aquel con el identificador más pequeño, dándole preferencia a los hijos que no sean hoja. Como resultado de esta operación, la raíz selecciona al vértice 2 y lo apunta mediante la variable $domin\_pointer$.

En la ranura de tiempo siguiente, $t = k + 1$, el vértice 2 cumple la condición de la línea 17, ya que su padre lo apunta. El vértice 2 actualiza su variable $domin\_dominating$ de 0 a 1 y entra así al conjunto

dominante.

Finalmente, en la ranura de tiempo $t = k + 2$, la raíz nuevamente al seleccionar y apuntar al vértice 2, descubre que éste es dominante. La raíz incremente su variable $domin\_qtydominants$ de 0 a 1 al ser dominada por su hijo y deja de apuntarlo en su variable $domin\_pointer$.

De este modo, la raíz ya es dominada y a su vez cumple con las condiciones del CDT_{min}.

Pseudocódigo 29: Función $domin\_dominate$

```
1  if ∃ w ∈ tree_children(v) | w.domin_qtydominants = 0 then
2  |  v.domin_dominating ← 1
3  end
4  v.domin_worked ← 1
5  v.domin_qtydominants ← domin_qtydominants(v)
6  if v.tree_root = 1 ∧ v.domin_qtydominants = 0 then
7  |  min_child ← mín({w ∈ tree_children(v) | w no se hoja})
8  |  if min_child = Nil then
9  |  |  min_child ← mín({w ∈ tree_children(v)})
10 |  end
11 |  v.domin_pointer ← min_child
12 |  if (min_child).domin_dominating = 1 then
13 |  |  v.domin_qtydominants ← 1
14 |  |  v.domin_pointer ← Nil
15 |  end
16 end
17 if (v.tree_parent).domin_pointer = v then
18 |  v.domin_dominating ← 1
19 end
```

- La función $domin\_done(v)$ determina si un vértice ya terminó de calcular los valores de sus variables en el algoritmo actual. Esta función regresa 1 si se cumple alguna de las siguientes condiciones:

$$v.leaf = 1 \wedge domin\_safe(v) = 1 \tag{70}$$
$$\wedge\, v.domin\_qtydominants \geq 1$$
$$\wedge\, (v.domin\_dominating, v.domin\_qtydominants,$$
$$v.domin\_pointer, v.domin\_worked) = domin\_dominate(v)$$

$$v.leaf = 0 \wedge domin\_safe(v) = 1 \wedge v.domin\_qtydominants \geq 1 \tag{71}$$
$$\wedge\, (v.domin\_dominating, v.domin\_qtydominants,$$
$$v.domin\_pointer, v.domin\_worked) = domin\_dominate(v)$$
$$\wedge\, w.domin\_done = 1, \forall\, w \in tree\_children(v)$$

En la expresión 70 se verifican las hojas del árbol. Éstas deben ser seguras mediante la función $domin\_safe(v)$, ser dominadas por al menos un vecino, y que su estado corresponda a la salida de la función $domin\_dominate(v)$.

En la expresión 71 se verifican los vértices internos del árbol. Éstos, además de ser seguros por $domin\_safe(v)$, ser dominados, y que su estado corresponda con la salida de la función $domin\_dominate(v)$, deben cumplir con que todos sus descendientes hayan indicado que finalizaron el cálculo de sus variables.

8.2. Reglas del algoritmo Tree_Domination

A continuación de describen las reglas utilizadas para la correcta convergencia del algoritmo Tree_Domination:

La regla Domination_Reset identifica si el padre de un vértice ya se reinició, permitiéndole a v ejecutar su función de reinicio. Esta regla es análoga a las reglas Tree_Reset, Skeleton_Reset, Cycle_Reset y Adjust_Reset.

Pseudocódigo 30: Regla DOMINATION_RESET

```
1 if A_5 ∧ (v.domin_reset = 0) ∧ ((v.tree_parent).domin_reset = 1) then
2 |  domin_reset(v)
3 end
```

La regla DOMINATION_SYNC genera un retardo de una unidad de tiempo para que se propague correctamente el mensaje de reinicio de variables. Esta regla es análoga a las reglas TREE_SYNC, SKELETON_SYNC, CYCLE_SYNC y ADJUST_SYNC.

Pseudocódigo 31: Regla DOMINATION_SYNC

```
1 if A_5 ∧ (v.domin_reset = 1) then
2 |  v.domin_reset ← 0
3 end
```

La regla DOMINATION_ERROR se encarga de propagar un mensaje de error a la raíz del árbol cuando v o uno de sus descendientes detectaron una inconsistencia en sus variables. Cuando el mensaje de error llega a la raíz del árbol, éste reinicia sus variables y comienza de nuevo el proceso de convergencia del sistema. Esta regla es análoga a las reglas TREE_ERROR, SKELETON_ERROR, CYCLE_ERROR y ADJUST_ERROR.

Pseudocódigo 32: Regla DOMINATION_ERROR

```
1 if A_5 ∧ ((domin_safe(v) = 0) ∨ (∃ w ∈ tree_children(v) | w.domin_error = 1)) then
2 |  v.domin_error ← 1
3 |  if v.tree_root = 1 then
4 |  |  domin_reset(v)
5 |  end
6 end
```

La regla DOMINATION_ALG se encarga de actualizar los valores de las variables de los vértices para calcular un CDT_{min} en el sistema. El vértice debe cumplir con ser seguro por $domin\_safe(v)$ y que todos sus hijos ya hayan calculado los valores de sus variables. Esta regla es análoga a las reglas TREE_ALG, SKELETON_ALG, CYCLE_ALG y ADJUST_ALG.

Pseudocódigo 33: Regla DOMINATION_ALG

```
1 if A_5 ∧ (domin_safe = 1) ∧ (w.domin_worked = 1, ∀ w ∈ tree_children(v)) ∧
    (v.domin_dominating, v.domin_qtydominants, v.domin_pointer, v.domin_worked) ≠
    domin_dominate(v) then
2 |   (v.domin_dominating, v.domin_qtydominants, v.domin_pointer, v.domin_worked) ←
  |     domin_dominate(v)
3 end
```

La regla DOMINATION_DONE es análoga a las reglas TREE_DONE, SKELETON_DONE, CYCLE_DONE y ADJUST_DONE. Esta regla propaga mensajes de finalización del algoritmo desde las hojas hasta la raíz del árbol mediante un proceso de convergecast.

Pseudocódigo 34: Regla DOMINATION_DONE

```
1 if A_5 ∧ (v.domin_done = 0) ∧ (domin_done(v) = 1) then
2 |   v.domin_done ← 1
3 end
```

8.3. Convergencia del algoritmo TREE_DOMINATION

Se emplean los siguiente lemas para demostrar la propiedad de convergencia y el análisis de tiempo de ejecución del algoritmo TREE_DOMINATION.

Lema 8.1 *El algoritmo* TREE_DOMINATION *reinicia correctamente sus variables en el grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a las demostraciones de reinicio de variables para los algoritmos ROOTING_TREE, CIF_SKELETON, CIF_CYCLE y CIF_ADJUST, descritos en los lemas 4.1, 5.1, 6.1 y 7.1. □

Lema 8.2 *El algoritmo* TREE_DOMINATION, *actualiza la variable* $domin\_worked$ *de* 0 *a* 1 *en todos los vértices del grafo, en* $O(D)$ *ranuras de tiempo.*

Demostración. Se demuestra por inducción en el nivel del árbol, desde el nivel 0 (la hoja más lejana de la raíz) hasta el nivel D (la raíz del árbol).

La demostración de este lema es análoga a la demostración del Lema 4.2. □

Lema 8.3 *Cuando el algoritmo* TREE_DOMINATION *converge, el sistema identifica un* CDT.

Demostración. La demostración de este lema se basa en las demostraciones realizadas en el trabajo de Goddard et al. (2003).

Se demuestra por contradicción. Suponga que una vez que converge el algoritmo, el conjunto dominante encontrado no es total, por lo que existe al menos un vértice v, que no es dominado por nadie (ninguno de sus vecinos pertenece al conjunto dominante).

Por la hipótesis de que el algoritmo ya convergió, implica que todos los vértices no satisfacen ninguna regla que altere su estado.

Se analizan dos casos: cuando v es la raíz del árbol y cuando v no lo es.

Si v no está dominado y no es la raíz del árbol, por la condición

$$\exists\, w \in tree\_children(v) \mid w.domin\_qtydominants = 0$$

de la función $domin\_dominate(v)$, el padre de v identifica a un vértice entre sus descendientes que está sin dominar, de modo que éste altera su estado para dominarlo llegando a una contradicción.

Cuando v no está y éste es la raíz del grafo, por las líneas 6 a 16 de la función $domin\_dominate(v)$, v identifica entre sus descendientes al vértice con identificador más pequeño y lo apunta, de modo que tanto v como el descendiente que apuntó alteran su estado, implicando que el sistema aún no había convergido llegando a otra contradicción. □

Lema 8.4 *Una vez que el sistema calcula un* CDT*, éste es minimal.*

Demostración. La demostración de este lema se basa en las demostraciones realizadas en el trabajo de Goddard et al. (2003).

Se demuestra por contradicción. Suponga que el CDT calculado no es minimal, por lo que implica que existe al menos un vértice v dentro del conjunto D, tal que al remover a v de D, $(D - \{v\})$, D sigue siendo un CDT.

Como D almacena un CDT, implica que todos los vértices son dominados por algún vecino.

El vértice v sólo puede pertenecer al conjunto D de dos maneras:

- Caso 1: Para cualquier vértice: cuando alguno de los descendientes de v no estaba dominado en el momento que v calcula el valor de sus variables (condición $\exists\, w \in tree\_children(v) \mid w.domin\_qtydominants = 0$ de la función $domin\_dominate(v)$).
- Caso 2: Para los hijos de la raíz, si la raíz no fue dominada (por lo considerado en las líneas 6 a 19 de la función $domin\_dominate(v)$).

Para la demostración se analizan ambos casos:

Si v está en el conjunto porque la raíz no fue dominada (por el Caso 2) y se remueve, la raíz no tiene a otro vértice que lo domine, de modo que el conjunto D deja de ser un CDT llegando a una contradicción.

Cuando v está en el conjunto dominante por el Caso 1, implica que al menos existe un vértice w que no fue dominado por ninguno de sus descendientes, de modo que al remover a v de D hace que w quede sin ser dominado, incumpliendo la condición del CDT, llegando a otra contradicción. □

Observación 8.1 *Aplicar el algoritmo* Tree_Domination *sobre algún árbol de expansión arbitrario en el grafo Halin distinto al árbol del Halin, no garantiza que la propiedad de minimalidad se cumpla una vez se devuelvan las aristas originales al grafo Halin.*

Demostración. Observe el ejemplo de las figuras 33 y 34. En la Figura 33, dado ese enraizamiento, al aplicar el algoritmo Tree_Domination se obtuvo un CDT_{min} correcto y minimal sobre ese árbol.

Al regresar todas las aristas del grafo Halin, ilustrado en la Figura 34, note como el vértice rojo puede eliminarse del CDT sin afectar las condiciones del mismo, por lo que ese vértice resulto ser redundante, incumpliendo la propiedad de minimalidad.

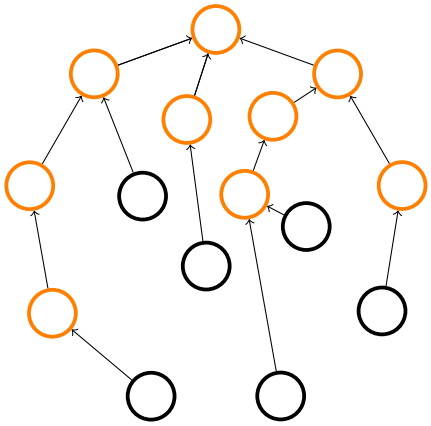

Figura 33. CDT_{min} producido por el algoritmo TREE_DOMINATION sobre el enraizamiento dado.

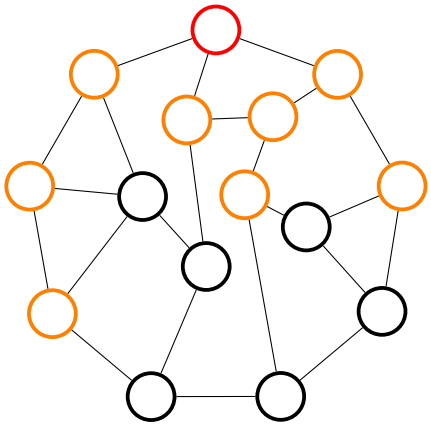

Figura 34. Incumplimiento de la propiedad de minimalidad, al regresar todas las aristas al grafo original.

□

Lema 8.5 *Una vez encontrado un CDT_{min} en el árbol del Halin y al agregar las aristas del ciclo de hojas al grafo, el conjunto sigue siendo minimal.*

Demostración. Debido a que todos los vértices del ciclo de hojas quedaron fuera del conjunto (a excepción de los Halin tipo rueda como el K_4), al devolver las aristas del ciclo de hojas, éstas no aportan un incremento a la cantidad de vecinos dominantes que contempla el vecino predecesor, vecino del esqueleto y sucesor de una hoja. Así que la cantidad de vértices dominantes almacenados en estos vértices se mantiene, preservando así la minimalidad.

En los casos tipo rueda, es necesario agregar una hoja al conjunto para dominar al vértice central o raíz, por lo que no es posible remover a la hoja del conjunto sin incumplir las condiciones del CDT_{min}. □

Lema 8.6 *El algoritmo* TREE_DOMINATION *actualiza la variable* $domin\_done$ *de* 0 *a* 1 *en todos los vértices del grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a las demostraciones de los lemas 4.4, 5.3, 6.4 y 7.3. □

Lema 8.7 *El Algoritmo* TREE_DOMINATION*, en ausencia de errores, calcula un* CDT_{min} *en un grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es consecuencia de los lemas 8.1, 8.2, 8.3, 8.4, 8.5 y 8.6. □

Lema 8.8 *En presencia de una inconsistencia en las variables del sistema, el algoritmo* TREE_DOMINATION *propaga un mensaje de error hasta la raíz del Halin y calcula nuevamente un* CDT_{min} *en el grafo Halin en* $O(D)$ *ranuras de tiempo.*

Demostración. La demostración de este lema es análoga a las demostraciones de los lemas 4.6, 5.5, 6.6 y 7.5 □

8.4. Cerradura del algoritmo TREE_DOMINATION

Al finalizar el algoritmo TREE_DOMINATION en el grafo Halin, las siguientes propiedades para cualquier vértice del grafo se cumplen:

- $domin\_qtydominants \geq 1$
- $domin\_reset = 0$
- $domin\_error = 0$
- $domin\_done = 1$

A continuación se demuestra que una vez el sistema converja, cumpliendo las condiciones listadas anteriormente, el sistema no altera su estado, cumpliendo así la propiedad de cerradura:

Dado que ningún vértice en el grafo tiene activada su bandera de $domin\_reset$, las reglas DOMINATION_RESET y DOMINATION_SYNC no se pueden ejecutar. Como la variable $domin\_error$ se encuentra desactivada, ningún vértice se privilegia por la regla DOMINATION_ERROR. Como todos los vértices del grafo tienen activada su bandera de $domin\_done$, implica que:

$$(v.domin\_dominating, v.domin\_qtydominants,$$
$$v.domin\_pointer, v.domin\_worked) = domin\_dominate(v)$$

por lo que la regla DOMINATION_ALG no se satisface. La regla DOMINATION_DONE tampoco se satisface, consecuencia de tener encendida la bandera $domin\_done$.

Como ningún vértice del sistema se puede privilegiar por ninguna regla del algoritmo, el sistema no modifica su estado por lo que la propiedad de cerradura se cumple. El predicado A_9, ilustrado en la expresión 72, garantiza que el sistema identifica un CDT_{min} en el grafo Halin, identificado a través de la variable $domin\_dominating$.

$$A_9 = A_5 \wedge v.domin\_qtydominants \geq 1 \tag{72}$$
$$\wedge\, v.domin\_reset = 0 \wedge v.domin\_error = 0 \wedge v.domin\_done = 1$$

Capítulo 9. Conclusiones

Como se puede ver en los capítulos introductorios, el diseño de algoritmos para los problemas computacionales estudiados en este trabajo, en sus versiones de optimización, es una tarea no trivial en grafos arbitrarios. Por esta razón, la mayor parte de los trabajos encontrados en la literatura para esta clase de problemas se centra en el diseño de algoritmos en topologías acotadas. Aun siendo este el caso, el diseño de algoritmos en grafos restringidos no resulta tan trivial. Es por eso que esta tesis se enfoca en el diseño de algoritmos autoestabilizantes en los grafos Halin y del aprovechamiento de sus componentes para el diseño de algoritmos.

Como se explica en el Capítulo 2, los grafo Halin cumplen con la particularidad de que se pueden descomponer en subgrafos conocidos tales como un árbol y un ciclo. Para el caso del problema del CIF_{max}, se utilizan los componentes del esqueleto y ciclo de hojas del Halin. Como se explicó en los capítulos referentes a este problema, la estrategia para encontrar dicho conjunto se basa en los algoritmos diseñados por Flores-Lamas et al. (2020). En ese trabajo, el algoritmo cumple con ser distribuido más no autoestabilizante.

Uno de los objetivos de este trabajo es que los algoritmos diseñados fueran tolerantes a fallas, consecuencia de que los sistemas distribuidos, en general, son susceptibles a fallas transitorias. Antes del diseño de los algoritmos autoestabilizantes de este trabajo, y en el mejor esfuerzo del estudio de la literatura, se sabía de la existencia de un algoritmo autoestabilizante para encontrar un CIF_{MAX} en grafos arbitrarios cuyo tiempo de ejecución es $O(n^2)\ u.t.$, propuesto por Shi (2012).

Una aportación del presente trabajo de tesis es el diseño de un algoritmo autoestabilizante para encontrar un CIF_{max} en grafos Halin. El tiempo de ejecución de este algoritmo es $O(n)\ u.t.$, mejorando así al mejor algoritmo reportado en la literatura para este tipo de problema. El algoritmo propuesto, por ser autoestabilizante, también es tolerante a fallas transitorias.

En lo que respecta al problema del CDT_{min}, el componente utilizado para el diseño del algoritmo corresponde al árbol del Halin. Al usar dicho componente, el algoritmo genera siempre una solución válida y minimal para cualquier caso específico del Halin. Como se explica en la Observación 8.1, la aplicación de este algoritmo en algún árbol de expansión arbitrario sobre el grafo Halin no garantizaba la propiedad de minimalidad.

La literatura muestra que los algoritmos diseñados para este problema se enfocan principalmente en grafos arbitrarios. Como se puede ver en el apartado de la literatura estudiada, el algoritmo con mejor

desempeño corresponde al algoritmo propuesto por Ding et al. (2020).

Otra aportación de este trabajo de tesis se relaciona con el problema de encontrar un CDT_{min}. Dicho algoritmo funciona para árboles y grafos Halin. El tiempo de ejecución de este algoritmo es $O(D)$ $u.t.$. Este algoritmo también mejora el tiempo de ejecución del mejor algoritmo reportado en la literatura para este problema.

Una comparativa de los mejores algoritmos para los problemas estudiados y los algoritmos diseñados en este trabajo para grafos Halin, se muestra en la Tabla 2.

Tabla 2. Comparativa de algoritmos entre lo mejor de la literatura y el trabajo presente.

Problema	Mejor algoritmo anterior	Tiempo de ejecución anterior	Tiempo de ejecución de los algoritmos propuestos
CIF_{MAX}	Shi (2012)	$O(n^2)u.t.$	$O(n)u.t.$
CDT_{min}	Ding et al. (2020)	$O(n^2)u.t.$	$O(D)u.t.$

9.1. Trabajo futuro

El diseño de estos algoritmos parte del hecho de que los componentes del Halin ya se encuentran identificados y que se cuenta con un canal de comunicación para la propagación de mensajes durante la autoestabilización. Para la identificación de dichos componentes en el trabajo de Gutiérrez-Medina (2021), se tiene la suposición de que el sistema es geométrico, de modo que se utilizan las posiciones de los vértices y aristas del sistema para completar esta tarea.

Como primera propuesta, se pretende diseñar de un algoritmo autoestabilizante que identifique los componentes del Halin, i.e., las hojas del ciclo de hojas y los vértices del esqueleto en un sistema que no sea geométrico.

Una vez identificados dichos componentes y como segunda propuesta, se puede diseñar un algoritmo autoestabilizante que enraíce el árbol del Halin, de modo que se tenga el canal de comunicación para que en futuros algoritmos, se puedan manejar la transmisión de mensajes durante el proceso de estabilización.

Finalmente, como tercera propuesta, y siguiendo una estrategia similar al trabajo de Mjelde (2004), se puede diseñar un algoritmo secuencial y posteriormente autoestabilizante, que encuentre un CDT_{MIN} en grafos árbol, utilizando programación dinámica.

Literatura citada

Belhoul, Y., Yahiaoui, S., & Kheddouci, H. (2014). Efficient self-stabilizing algorithms for minimal total k-dominating sets in graphs. *Information Processing Letters*, 114(7), 339–343. https://doi.org/10.1016/j.ipl.2014.02.002.

Dijkstra, E. W. (1974). Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 643–644. https://doi.org/10.1145/361179.361202.

Ding, Y., Wang, J. Z., & Srimani, P. K. (2020). An o(n2) self-stabilizing algorithm for minimal total dominating set in arbitrary graphs. In *2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, 49–54. https://doi.org/10.1109/WIIAT50758.2020.00012.

Dolev, S. & Welch, J. L. (2004). Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5), 780–799. https://doi.org/10.1145/1017460.1017463.

Eppstein, D. (2016). Simple recognition of halin graphs and their generalizations. *Journal of Graph Algorithms and Applications*, 20(2), 323–346. https://doi.org/10.7155/jgaa.00395.

Flores-Lamas, A., Fernández-Zepeda, J. A., & Trejo-Sánchez, J. A. (2018). Algorithm to find a maximum 2-packing set in a cactus. *Theoretical Computer Science*, 725, 31–51. https://doi.org/10.1016/j.tcs.2017.11.030.

Flores-Lamas, A., Fernández-Zepeda, J. A., & Trejo-Sánchez, J. A. (2020). A distributed algorithm for a maximal 2-packing set in halin graphs. *Journal of Parallel and Distributed Computing*, 142, 62–76. https://doi.org/10.1016/j.jpdc.2020.03.016.

Gairing, M., Goddard, W., Hedetniemi, S. T., Kristiansen, P., & McRae, A. A. (2004). Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(3-4), 387–398. https://doi.org/10.1142/S0129626404001970.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman & Co.

Goddard, W., Hedetniemi, S. T., Jacobs, D. P., & Srimani, P. K. (2003). A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph. In *Proceedings International Parallel and Distributed Processing Symposium*. https://doi.org/10.1109/IPDPS.2003.1213437.

Gutiérrez-Medina, A. T. (2021). Diseño de algoritmos auto-estabilizantes para grafos halin. [Tesis de Maestría en Ciencias, Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California]. http://cicese.repositorioinstitucional.mx/jspui/handle/1007/3557.

Hochbaum, D. S. & Shmoys, D. B. (1985). A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2), 180–184. https://doi.org/10.1287/moor.10.2.180.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, 85–103, Boston, MA. Springer US. https://doi.org/10.1007/978-1-4684-2001-2_9.

Mjelde, M. (2004). K-packing and k-domination on tree graph. [Tesis de Maestría en Ciencias, University of Bergen]. https://hdl.handle.net/1956/1467.

Peleg, D. (2007). Time-efficient broadcasting in radio networks: A review. In Janowski, T. & Mohanty, H., editors, *Distributed Computing and Internet Technology*, 1–18, Berlin, Heidelberg. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-77115-9_1.

Shi, Z. (2012). A self-stabilizing algorithm to maximal 2-packing with improved complexity. *Information Processing Letters*, 112(13), 525–531. https://doi.org/10.1016/j.ipl.2012.03.018.

Shlomi, D. (2000). *Self-Stabilization*. MIT Press.

Sysło, M. M. & Proskurowski, A. (1983). On halin graphs. In Borowiecki, M., Kennedy, J. W., & Sysło, M. M., editors, *Graph Theory*, 248–256, Berlin, Heidelberg. Springer Berlin Heidelberg. https://www.researchgate.net/publication/230596186_On_Halin_graphs.

Trejo-Sánchez, J. A. & Fernández-Zepeda, J. A. (2014). Distributed algorithm for the maximal 2-packing in geometric outerplanar graphs. *Journal of Parallel and Distributed Computing*, 74(3), 2193–2202. https://doi.org/10.1016/j.jpdc.2013.12.002.