UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

FACULTAD DE INGENIERÍA

EVALUACION DEL USO DE MEMORIA DE UN MÉTODO EXACTO PARA EL CONTEO DE CUBIERTAS DE ARISTAS EN GRÁFICAS SIMPLES

TESIS

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA

LUIS ERNESTO SIERRA ALVA

Asesor de Tesis

Dr. José Raymundo Marcial

Coasesora de Tesis

Dra. Rosa María Valdovinos Rosas

TOLUCA, ESTADO DE MÉXICO ENERO DE 2017

Índice de contenidos

Pág.

Índice de tablas

Pág.

Índice de ilustraciones

Pág.

Dedicatorias

A mi familia:

Gracias por apoyarme en todos los sentidos y comprenderme cuando el tiempo no alcanzaba, gracias por estar siempre conmigo, gracias por disfrutar de mis logros, mientras que al mismo tiempo me apoyaron a levantarme de mis fracasos.

A mis amigos:

Por estar conmigo durante todo el transcurso de la carrera, por todos los momentos que vivimos en esos años, por el apoyo mutuo y competencia sana que siempre ha habido entre todos, gracias.

A mis asesores:

Por su apoyo incondicional en la realización de este trabajo, por todo su esfuerzo y compresión, pero sobre todo por brindarme oportunidad que de otro modo no hubiera podido alcanzar y que me han permitido crecer como persona y profesional.

Introducción

La teoría de graficas es un campo de las ciencias computacionales y de las matemáticas, relacionada íntimamente con las ciencias experimentales, sus usos como técnica en diversas disciplinas es cada vez más extendido, incluso en las llamadas ciencias sociales.

Tanto en el campo de las matemáticas como en la teoría de graficas existen problemas que han captado la atención de los investigadores durante años y que aun en nuestros días no se ha podido encontrar una solución algorítmica que pueda resolverlos adecuadamente con un uso de recursos aceptable.

En el caso de la teoría de grafos hay dos grandes grupos con características y dificultades propias, por un lado los problemas NP y por otro los problemas #P, dentro de estos últimos encontramos el problema de conteo de cubiertas de aristas.

El objetivo es encontrar el mínimo número de coberturas de aristas dentro de un grafo simple, es decir, encontrar el mínimo conjunto de aristas que toquen a todos los vértices del grafo. Esto se complica cada vez más mientras aumentamos el número de vértices o el grafo tiene aristas que formen parte de más de un ciclo, aumentando notablemente la complejidad y el tiempo de procesamiento requerido.

Realmente hay muy pocas investigaciones que traten de resolver los problemas #P en general y el problema de conteo de cubiertas de aristas en específico, y de los pocos que lo han intentado se han decidido por un enfoque heurístico o de aproximación.

Una mejor solución es a través de un método exacto, ya que aunque es más complicado de desarrollar con una complejidad algorítmica razonable, tiene muchas ventajas, entre ellas el hecho de que al conocer exactamente como es su funcionamiento es más fácil de estudiar y analizar a fondo.

El algoritmo propuesto en (Hernández Servín, et al., 2014) pretender ser una solución eficaz al problema de conteo de cubiertas de aristas, al mismo tiempo que es un paso hacia adelante en el estudio de los problemas #P al ofrecer un método exacto para la solución de este problema con una complejidad algoritmo exponencial de límite inferior.

Este algoritmo solo está comprobado matemáticamente a falta de una implementación que permita su comprobación práctica. Es por esto que esta Tesis de investigación tiene como fin la implementación del algoritmo y su consecuente comprobación.

Con el fin de poder estudiar a fondo el comportamiento del algoritmo en cuanto a su eficacia para resolver el problema, así como de los recursos utilizados, en cuanto tiempo de ejecución y uso de memoria, de los que hace uso durante su ejecución.

Para de esta forma conocer la fiabilidad el algoritmo como una solución al problema de conteo de cubiertas y definir cuáles podrían ser las siguientes líneas de investigación a seguir para llegar aún más lejos en el intento de solucionar los problemas #P.

Capítulo 1

Planteamiento de la Problemática

1.1 Planteamiento del Problema

Una gráfica se define como una tripleta (V, E, ψ), donde V es un conjunto no vació de vértices y E es un conjunto no vació de aristas, mientras que ψ es una función de incidencia la cual asocia cada arista de G con un par no necesariamente distinto de vértices de G (Bin & Zhongyi, 2010).

Este concepto ha demostrado su utilidad, siendo exitosamente aplicado en varias ciencias y disciplinas, tanto por su representación gráfica, como por sus propiedades. Normalmente se le ha relacionado con las ciencias exactas y computacionales, esto es innegable, al observar trabajos recientes, como la creación de sistemas multiagentes (Jiang, et al., 2014) o en las matemáticas (Núñez, et al., 2012).

El uso de graficas se ha expandido incluso a otras ciencias, como:

- En la Biología se utiliza para representar la estructura de sistema biológicos, como las redes de interacción proteína a proteína (González-Díaz, et al., 2009).

- En la Ecología para estudiar el crecimiento de los arrecifes en la "Gran Barrera de Coral" en Australia representando las redes de conexión interna de los arrecifes como clúster que se comunican entre sí (Thomas, et al., 2013), o utilizando un enfoque orientado a la teoría de gráficas para el diseño de reservas naturales, en el cual cada habitad posible es un nodo que está conectado con otros (Wanga & Önal, 2011).

- En la Geografía se utilizan diferentes técnicas para representar los distintos componentes de los sistemas terrestres (patrones de fracturas en rocas, fenómenos naturales, etc.) como vértices de una red interconectada (Schwanghart, et al., 2015), o utilizar las gráficas para describir los compontes de los sistemas geomorfológicos, lo que permite describir sus diferentes comportamientos (Heckmann, et al., 2014).

- En la Construcción Urbana se pueden representar las distintas calles y avenidas como aristas, para el diseño de todo tipo de construcciones, por ejemplo, pistas ciclistas (Schroeder Barwaldt, et al., 2014).

- En la Medicina se pueden representar las conexiones neuronales del cerebro usando las gráficas para estudiarlo y comprender de mejor manera su funcionamiento (Goldenberg & Galván, 2015) (McGlaughlin, et al., 2015).

Dentro de la teoría de gráficas, que es la ciencia encargada de estudiar las gráficas y sus propiedades, existen problemas que por su carácter combinatorio se clasifican como NP, si son de decisión, o #P si son de conteo, tal como se menciona en Hernández et al. (2014).

En ciencias de la computación un problema de decisión es NP si dada una Máquina de Turing No determinista *M* que representa el problema y una entrada *E* de *M,* se puede determinarse en tiempo polinómico (tiempo acotado por un polinomio) si *E* es verdadera o falsa. Equivalentemente, si se tiene una Máquina de Turing Determinista, bajo la misma entrada *E* tomaría en el peor de los casos un tiempo exponencial determinar si es o no una solución. Los problemas NP se dice que son de decisión ya que únicamente dan como salida SI o NO.

Por otro lado se tienen los problemas de conteo. Un problema es #P si existe una Máquina de Turing No determinista *M* que representa un problema y para cualquier entrada *E* dada a *M* puede contarse en tiempo polinómico (tiempo acotado por n polinomio) el número de soluciones de *E*. Esta mayor complejidad ha hecho que históricamente los problemas #P han sido menos investigados que los NP, el conteo de coberturas de aristas forma parte de los problemas #P,

Formalmente, una cobertura de aristas de una gráfica G es un conjunto de aristas C donde cada vértice es incidente con al menos una arista en C. Encontrar el número de las coberturas de aristas en una gráfica es un problema #P cuando las gráficas tienen topologías cíclicas, cuando se tiene una topología acíclica, se puede resolver en tiempo polinómico (Hernández Servín, et al., 2014).

Actualmente, no existe un método que en tiempo polinómico cuente el número de cubiertas de aristas en una gráfica simple con topología cíclica (Hernández Servín, et al., 2014). El peor caso, en el que el número de aristas es un número exponencial, se presenta en las gráficas completas ya que existe una arista que conecta a cada par de nodos. Trabajos anteriores han intentado darle solución al problema general utilizando algoritmos heurísticos como el presentado en (Lin, et al., 2014), que aproximan la solución, sin embargo, no se reportan algoritmos exactos cuya complejidad sea menor a la trivial 2^n.

Hernández et al. (2014) proponen un algoritmo cuyo tiempo de ejecución está dado por 1.41^n donde *n* es el número de vértices de la gráfica. La estrategia principal consiste en descomponer la gráfica cíclica de entrada en subgráficas que contengan un número menor de ciclos hasta obtener subgráficas acíclicas. La descomposición genera un árbol cuyos nodos interiores representan gráficas cíclicas y cuyas hojas representan gráficas acíclicas.

Los autores demostraron cómo el número de coberturas de las hojas es igual al número de coberturas de la gráfica original. Aunque el tiempo de ejecución disminuye considerablemente con este método, no se ha estudiado el crecimiento

de la memoria utilizada para representar el árbol de subgráficas del método general.

Debido a que la mayoría de las veces, una disminución en el tiempo de ejecución de un algoritmo conlleva el uso de mayor memoria utilizada. En esta Tesis se describe el desarrollo de un método que permita generar los subárboles de tal forma que se pueda establecer, con una muestra aleatoria de gráficas de entrada, el crecimiento del uso de memoria. Además, se plantea verificar si en la práctica, el crecimiento establecido teóricamente en tiempo del método propuesto (Hernández Servín, et al., 2014) se preserva.

1.2 Justificación

La teoría de gráficas y sus propiedades a lo largo de los años han comprobado su utilidad como herramienta o técnica para resolver diferentes cuestiones en áreas como la Biología, donde se utiliza para representar diferentes estructuras biológicas como el ADN, lo cual permite su estudio y comprender su comportamiento, la Ecología, para entender y estudiar el crecimiento de los arrecifes o bien crear reservar naturales que disminuyan el fenómeno de la fragmentación del habitad.

En otras ciencias como la Geografía se ha utilizado para poner a prueba las propiedades de diferentes estructuras y realizar análisis espaciales, de modo que se pueda inferir el comportamiento de diversos fenómenos geológicos, la Construcción Urbana es posible diseñar pistas, caminos o carreteras, asegurándonos de minimizar los costos, distancias o tiempos de transporte utilizando las gráficas.

Incluso en las ciencias de campo humanístico como la Medicina se ha encontrado uso para la teoría de gráficas, donde son utilizadas para estudiar las conexiones cerebrales y encontrar posibles curas a algunas enfermedades así como en el desarrollo de antibióticos.

Por todas las posibles aplicaciones prácticas para la teoría de gráficas, se ha logrado que poco a poco más y más investigadores dediquen sus esfuerzos en resolver los grandes problemas de la misma. Sin embargo los problemas #P al ser más complejos han sido menos investigados, este es el caso del problema de conteo de coberturas de aristas, el cual no ha sido investigado muy a fondo.

Esto no quiere decir que no haya trabajos anteriores que intenten darle solución al problema de conteo de coberturas de aristas, sin embargo estos enfoques han utilizado algoritmos heurísticos para intentar dar una solución, los cuales tienen como desventaja el que su comportamiento sea siempre el mismo, impidiendo que

se pueda estudiar su complejidad computacional para todos los casos posibles. El algoritmo exacto con mejor rendimiento en cuanto a complejidad y tiempo de ejecución para la resolución de este problema es el presentado en (Hernández Servín, et al., 2014).

Es importante mencionar que una característica que permite que los algoritmos que se diseñan sean realmente factibles en la práctica, es el uso de recursos que estos hacen al momento de su ejecución, este es un factor de gran importancia al momento de diseñar algoritmos y merece ser estudiado, centrándose en este caso en particular en el uso de memoria.

1.3 Objetivo General

Analizar, diseñar e implementar un método de uso de memoria para la generación del árbol de subgráficas para el conteo de cubiertas de aristas propuesto en (Hernández Servín, et al., 2014), con el fin de comprobar la relación tiempo de ejecución-uso de memoria.

1.4 Objetivos Específicos

- Implementar el algoritmo exacto para contar cubiertas de aristas en graficas con ciclos intersectados propuesto en (Hernández Servín, et al., 2014).

- Poner a prueba la implementación del algoritmo con distintos tipos de graficas de entrada para corroborar su funcionamiento y estudiar su comportamiento.

- Deducir una ecuación matemática que permita predecir el tiempo de ejecución del algoritmo con base en la gráfica de entrada, de acuerdo a los resultados de las pruebas realizadas.

- Deducir una ecuación matemática que permita predecir el comportamiento sobre el uso de memoria del algoritmo con base en la gráfica de entrada, de acuerdo a los resultados obtenidos.

- Encontrar si existe una relación entre el tiempo de ejecución del algoritmo y el uso de recursos específicamente de memoria del mismo.

1.5 Hipótesis

El uso de memoria del algoritmo propuesto por (Hernández Servín, et al., 2014) es polinómico con respecto al número de nodos de la gráfica de entrada.

1.6 Estructura de la Tesis

La estructura de la Tesis se detalla de la siguiente forma:

El primer capítulo, cómo ya se observó, se explica todo lo relacionado a la definición y delimitación del problema a resolver, además de explicar el por qué es importante la resolución del mismo, e indicar cuáles son los pasos a seguir para poder resolver el mismo, así como definir la conclusión a la que se pretende llegar.

El sustento teórico se incluye en el segundo capítulo, el cual aborda entre otros temas los conceptos y características de las gráficas necesarias para la realización de este trabajo, el diseño de algoritmos y finalmente llegar a la comprensión complejidad algorítmica y la importancia de su medición.

El tercer capítulo es la descripción del algoritmo a implementar y de su complejidad en cuanto a tiempo de ejecución, así mismo se busca comparar este trabajo con otros que intentan resolver el problema de conteo de cubiertas de aristas con el propósito de observar sus ventajas y desventajas con respecto a otros trabajos relacionados.

Entre tanto que la explicación de la metodología seguida se observa en el capítulo 4, donde se describe la implementación del algoritmo para contar el número de coberturas de aristas de una gráfica, mostrando primero un análisis general del algoritmo, para posteriormente especificar cada una de las partes centrales del mismo.

Por último, el capítulo 5 muestra el análisis cuantitativo descriptivo que se le realizó a la implementación a partir de las pruebas realizadas y de los resultados obtenidos, así como las conclusiones a las que se llegaron y posibles líneas abiertas de investigación futura.

Capítulo 2

Marco Teórico

2.1 La Teoría de Gráficas

La teoría de gráficas es la disciplina que se encarga de estudiar las gráficas, sus propiedades y aplicaciones. En este sentido, una gráfica G es una tripleta ordenada $(V(G), E(G), \psi_G)$, donde $V(G)$ es un conjunto no vacío de vértices, $E(G)$ es un conjunto no vacío de aristas disconjunto de $V(G)$ y una función de incidencia ψ_G que asocia cada arista de G y un par no ordenado (no necesariamente distinto) de vértices en G (Bondy & Murty, 2007).

Si e es una arista, de tal modo que u y v son vértices que cumplen con $\psi_G(e) = uv$, entonces se puede decir que e une a u y v, así u y v son llamados los puntos finales de e. De esta forma se dice que u y v son adyacentes, ya que están unidas por e, y e se considera incidente tanto a u como a v (Lipschutz & Lipson, 2007).

Para aclarar de modo visual el concepto de gráfica, se va a utilizar el ejemplo presentado en (Bondy & Murty, 2007). Suponiendo que:

$$G = (V(G), E(G))$$

Donde:

$$V(G) = \{v_0, v_1, v_2, v_3, v_4, v_5\}$$

$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$$

Y ψ_G es definida por:

$$\psi_G(e_1) = v_1v_2 \quad \psi_G(e_2) = v_2v_3 \quad \psi_G(e_3) = v_3v_4 \quad \psi_G(e_4) = v_4v_5 \quad \psi_G(e_5) = v_5v_1$$

$$\psi_G(e_6) = v_0v_1 \quad \psi_G(e_7) = v_0v_2 \quad \psi_G(e_8) = v_0v_3 \quad \psi_G(e_9) = v_0v_4 \quad \psi_G(e_{10}) = v_0v_5$$

La gráfica que es representación de la definición anterior se muestra en la Figura 2-1. Se puede apreciar que v_1 y v_2 son adyacentes entre sí, mientras que e_1 es incidente a los dos, lo cual se repite para cada definición de $\psi_G(e_n)$.

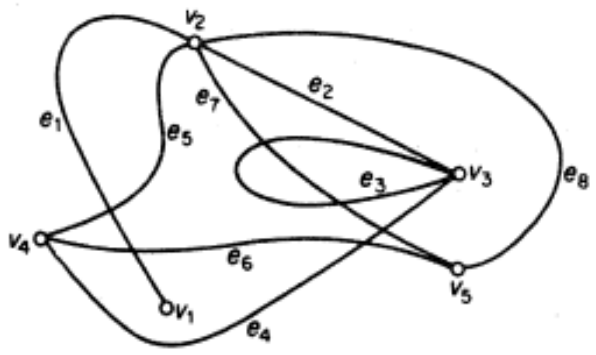

Figura 2-1. *Representación de la gráfica expresada a través de su definición.*
(Bondy & Murty, 2007)

Simplificando la definición anteriormente mencionada se puede obtener un concepto más claro, al definir una gráfica G como un conjunto finito V de elementos llamados vértices y un conjunto E de distintos pares de vértices llamadas aristas. Expresando como $G = (V, E)$ a la gráfica cuyo conjunto de vértices es V y se conjunto de aristas es E (Brualdi, 2009) (Lipschutz & Lipson, 2007).

Utilizando la notación presentada en (Bondy & Murty, 2007) la cual será utilizada a lo largo de esta Tesis donde V se representa como $v(G)$ y E como $e(G)$, los cuales se conocen como orden y tamaño de la gráfica, y no son otra cosa que el número de vértices y aristas que tiene la gráfica respectivamente. Una gráfica bien conocida llamada de Desargues se presenta en la Figura 2-2, para ejemplificar el tamaño y orden de forma gráfica.

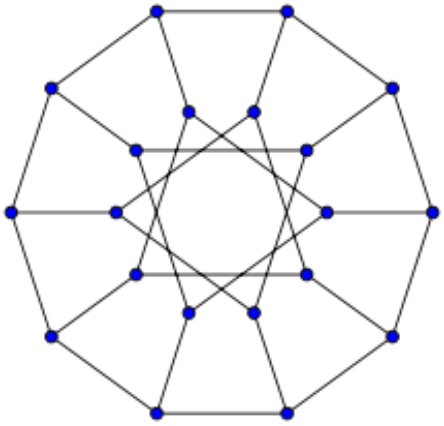

Figura 2-2. *Gráfica de Desargues de orden 20 y tamaño 30.*

(http://es.wikipedia.org/wiki/Gráfica_de_Desargues#/media/File:DesarguesGraph.svg)

Cuando una gráfica tiene orden n, se le denomina completo, esto significa que cada par distinto de vértices forma una arista. Es decir que cada vértice es adyacente a todos los demás, en este caso el número de aristas que tiene la gráfica se obtiene con la formula $n(n-1)/2$, y se denomina K_n (Brualdi, 2009).

Cada vértice v de la gráfica también tiene un grado el cual se representa con el símbolo $d_G(v)$ en (Bondy & Murty, 2007) y representa el número de aristas incidentes a v, si este grado es cero, entonces a ese vértices se le denomina vértice aislado.

La notación de $\delta(G)$ y $\Delta(G)$ se refiere a el mínimo y el máximo grado de un conjunto de vértices de G respectivamente. Recordando que el conjunto de vértices V perteneciente a la gráfica G se denotan como V_G, y si a su vez $v \in V_G$ entonces estamos denotando el conjunto de aristas incidentes a v.

Una gráfica se considera finita si tanto el número de vértices y aristas son finitos, es decir si se cumple la condición de que $|v(G)| < \infty$ y $|e(G)| < \infty$. Es fácil observar que se si una gráfica tiene un numero finito de vértices, automáticamente tiene un numero finito de aristas y por lo tanto es finito. La gráfica finita con un vértice y sin aristas, es decir un punto, es llamado la gráfica trivial, todos las demás gráficas son llamadas no triviales (Lipschutz & Lipson, 2007).

Una gráfica se considera simple si no tiene ciclos y si no hay dos aristas o más, que unan el mismo par de vértices. Las gráficas más estudiadas en la teoría de gráficas son precisamente los que cumplen ambas características, es decir los que son finitos y simples (Bondy & Murty, 2007).

Una comparación entre las gráficas simples y los que no lo son se puede observar en la Figura 2-3.

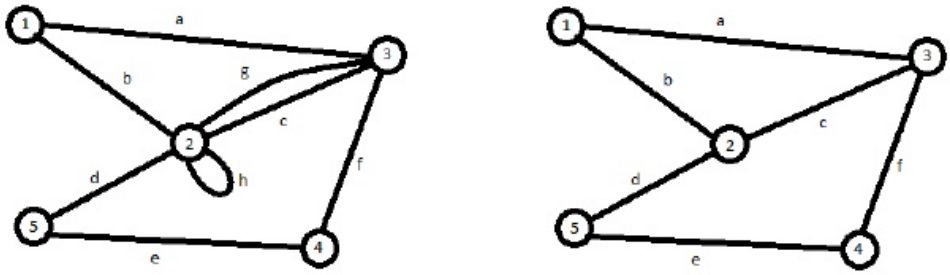

Figura 2-3. *Diferencias entre una gráfica no simple (izquierda) y una gráfica simple (derecha).*

(http://es.slideshare.net/aibaena/glosario-teoria-de-gráficas)

Recordando la definición de una gráfica como $G = (V, E)$, una secuencia de m aristas de la forma $\{X_0, X_1\}, \{X_1, X_2\}$ … $\{X_{m-1}, X_m\}$ es llamada un camino de longitud m, el cual une los vértices X_0 a X_m. El cual también se puede representar como $X_0 - X_1 - X_2 - \ldots - X_{m-1} - X_m$, si se da el caso de que el camino tenga diferentes vértices y $X_0 ¡= X_m$, entonces se le denomina como ruta, la cual se puede observar en la Figura 2-4.

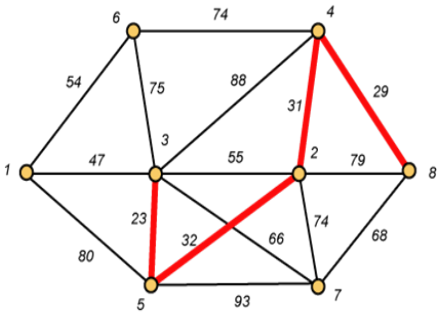

Figura 2-4. *Ruta (líneas rojas) dentro de una gráfica.*

(http://xcodigoinformatico.blogspot.mx/2012/09/algoritmo-de-kruskal-arbol-de-expansion.html)

Si se da el caso de que el camino tenga diferentes vértices y al mismo tiempo $X_0 = X_m$, entonces se le llama ciclo. Como se menciona en (Hernández Servín, et al., 2014) cuando un ciclo es conectado y cada uno de sus vértices tiene grado dos, es decir que cada vértice tiene una entrada y una salida, a este ciclo se le conoce como ciclo básico. En la Figura 2-5 se representa un ciclo básico.

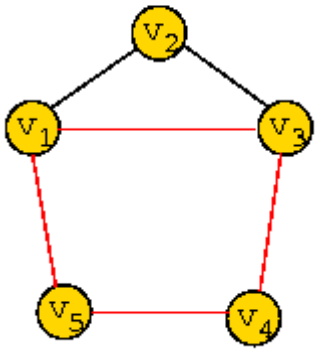

Figura 2-5. *Ciclo (líneas rojas) dentro de una gráfica.*

(http://users.dcc.uchile.cl/~bebustos/apuntes/cc3001/Gráficas/ciclo.gif/)

Un ciclo intersectado ocurre cuando hay una arista m o más que son parte de dos o más ciclos. Esto se puede observar en la Figura 2-6.

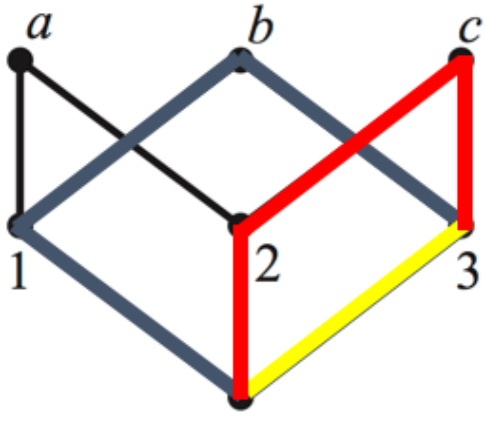

Figura 2-6. *Dos ciclos (rojo y azul) intersectados por una arista (amarilla).*

Considerando la gráfica $G(V, E)$. Entonces $H(V', E')$ es considerado una subgráfica de G si las aristas y vértices de H están contenidas en las aristas y vértices de G, esto ocurre cuando $V' \subseteq V$ y $E' \subseteq E$, o en otras palabras, H es subgráfica cuando sus aristas y vértices son un subconjunto de las aristas y vértices de la gráfica G respectivamente (Lipschutz & Lipson, 2007).

De este modo si e y v son una arista y vértice de la gráfica G, podemos denotar como $G - e$ y $G - v$ a las subgráficas obtenidas de remover una arista y un vértice

de la gráfica G respectivamente. Este proceso se puede realizar n veces, quitando un distinto número de aristas y vértices, para formar distintas subgráficas, un ejemplo de esto se puede observar en la Figura 2-7.

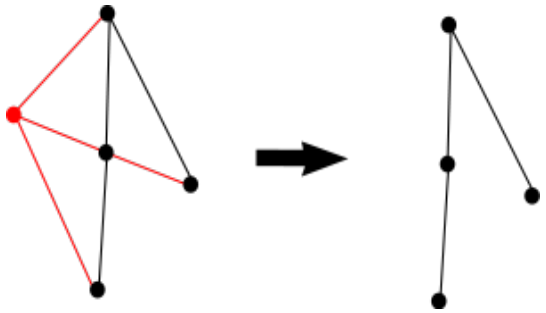

Figura 2-7. *Subgráfica (izquierda) generada a partir de la gráfica original (derecha).*

(http://pt.wikipedia.org/wiki/Subgráfica#/media/File:Subgraph.svg)

En (Hernández Servín, et al., 2014) se denota como $G\backslash e$ y $G\backslash v$ a $G - e$ y $G - v$, al mismo tiempo, se hace referencia a las subgráficas como $G' = (V', E')$ en vez de $H(V', E')$. En (Bondy & Murty, 2007) se define a la subgráfica de expansión, como una subgráfica H de un gráfica G, que cumple con la característica especifica de $G(V) = H(V)$, lo que implica que el conjunto completo de vértices de G son parte de H. Una ruta y un ciclo se pueden considerar como subgráficas.

Una subgráfica de expansión se puede obtener con la eliminación de los ciclos de la gráfica G de tal modo que cada par adyacente de vértices estén unidos por al menos una arista, por lo cual no habrá vértices aislados excepto por aquellos que ya sean aislados en G. En la Figura 2-8 se pueden observar diferentes formas de obtener subgráficas de expansión de una gráfica determinada.

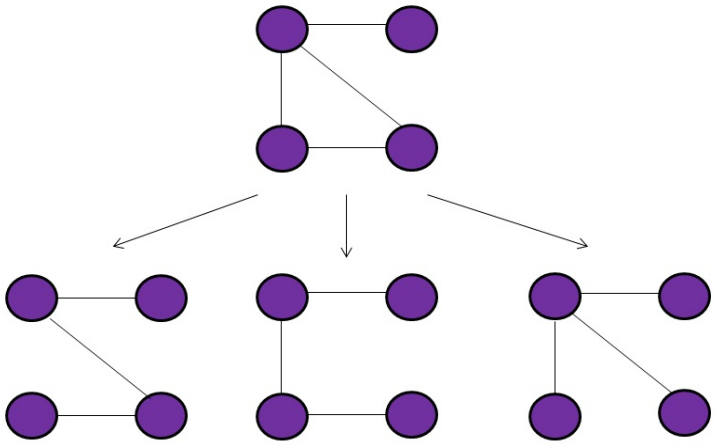

Figura 2-8*. La gráfica principal tiene 3 subgráficas de expansión.*

(https://jariasf.wordpress.com/tag/graph/)

Cuando una gráfica no tiene ciclos se le denomina acíclica, y si además es conectada entonces se le puede denominar árbol. Esto ocurre cuando solo hay una ruta que una a dos vértices cualquiera (Bondy & Murty, 2007). Ejemplos de árboles se pueden observar en la Figura 2-9.

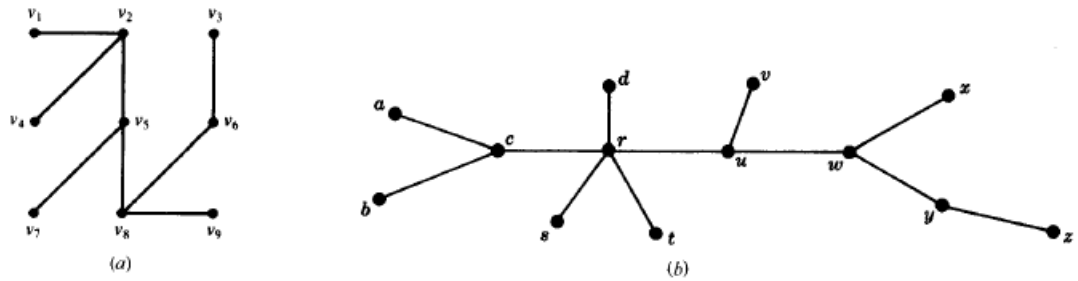

Figura 2-9. *Ejemplos de gráficas árboles.* (Lipschutz & Lipson, 2007)

Una subgráfica T de una gráfica conectada G es llamada un árbol de expansión de G si T es un árbol e incluye todos los vértices de G. En la Figura 2-10 se observa cómo se pueden generar diferentes arboles de expansión.

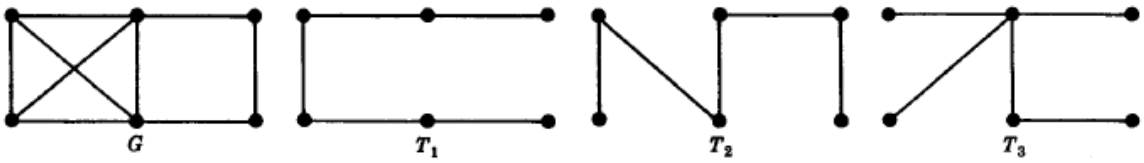

Figura 2-10. *Árboles de expansión Tn generados de una gráfica G.*

(Lipschutz & Lipson, 2007)

Una cobertura de aristas o e-covert de una gráfica G se define en (Hernández Servín, et al., 2014) como un subconjunto de aristas en E_G, en donde se tienen todos los vértices de G. En otras palabras, un conjunto de coberturas de aristas de $G = (V_G, E_G)$ es una familia ε_G tal que $U_{S \in \varepsilon_G} = E_G$ incluye todos los vértices de G, estas se pueden apreciar gráficamente en la Figura 2-11.

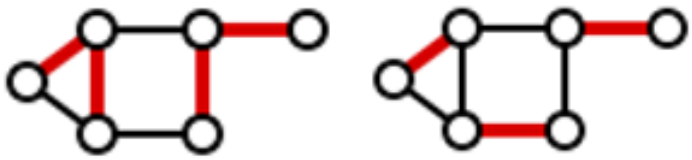

Figura 2-11. *Dos conjuntos de coberturas de aristas de una gráfica (líneas rojas).*

(http://upload.wikimedia.org/wikipedia/commons/thumb/7/72/Edge-cover.svg/200px-Edge-cover.svg.png)

Por lo tanto, el problema de conteo de conjuntos de coberturas de aristas presentado en (Hernández Servín, et al., 2014) puede ser simplemente enunciado como, teniendo una gráfica simple G, sin vértices aislados, entonces para encontrar el total de diferentes conjuntos de coberturas de aristas que se pueden generar para la gráfica G se debe calcular $|\varepsilon_G|$.

Para finalizar los conceptos de la teoría de gráficas, se mencionan las características de un tipo de grafica especial, la cual será de utilidad más adelante, llamada grafica cactus, estas graficas tienen la característica, de que cualquier arista pertenece solo a un ciclo y que cada par de ciclos adyacentes comparten a lo más un vértice (Flores Lamas, 2014), es decir que por más aristas o vértices que se agreguen nunca se genera un ciclo intersectado, la Figura 2-12 muestra un ejemplo de este tipo de gráficas.

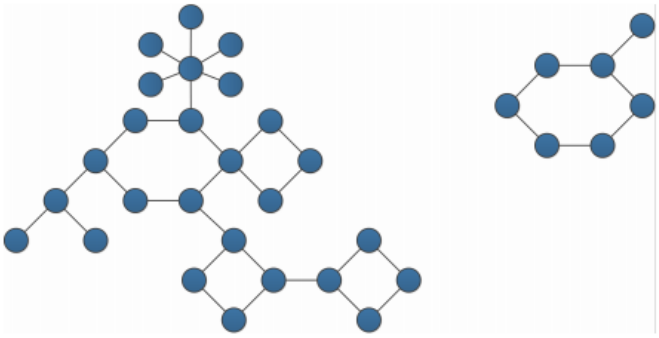

Figura 2-12. *Gráficas cactus.* (Flores Lamas, 2014)

2.2 Diseño De Algoritmos

Un algoritmo se puede definir como un conjunto de pasos finitos para la resolución de un problema o tarea específica. Así mismo todo buen algoritmo debe tener las siguientes propiedades básicas (Corona Nakamura & Ancona Valdez, 2011):

- **Finitud:** Un algoritmo siempre debe constar de un número pasos y tiempo de ejecución finita, es decir que debe tener un principio y un fin.

- **Definitud:** Todo paso de un algoritmo debe evitar las ambigüedades de interpretación definiendo perfectamente las acciones a realizar.

- **Generalidad:** Un algoritmo no debe concentrarse en resolver un problema particular, por el contrario, debe enfocarse en resolver toda una clase de problemas desde su definición en general.

- **Precisión:** Todo algoritmo debe especificar el orden exacto en los que se realizaran los pasos para la resolución del problema.

Los algoritmos deben ser independientes tanto del lenguaje de programación como de la computadora con los que se ejecuta, esto hace que el algoritmo y su diseño sean más importante que el lenguaje o la computadora, ya que son solo un medio para poder expresarlo.

Los algoritmos pueden expresar la solución a los distintos problemas y las partes esenciales que lo componen son los datos de entrada, el proceso a los que se someten esos datos y finalmente los datos de salida (Joyanes Aguilar, 2008).

Durante los años 30, lo que buscaban los investigadores de la época, era encontrar un conjunto de axiomas y sus respectivas reglas de inferencia, que permitieran expresar todas las matemáticas, así verificar la veracidad o falsedad de cualquier enunciado matemático, podían automatizarse y reducirse a un simple cálculo computacional (Jurado Málaga, 2008).

Posteriormente K. Gödel con su Teorema de Incompletitud, demostraron que era imposible automatizar completamente las matemáticas, por los que esas ideas fueron descartadas. En cambio, los investigadores que los precedieron como Church, Kleene, Turing y Post centraron sus esfuerzos en demostrar que problemas tenían o no soluciones (Quiroga Rojas, 2008).

En 1937 Alan Turing, presento la Maquina de Turing (MT), la cual es una entidad abstracta que teóricamente puede resolver cualquier problema que tenga una solución algorítmica, esto permitió formalización del concepto de algoritmo, y al mismo tiempo, demostrar que hay ciertos problemas que no tienen solución, es decir que no son computables (Jurado Málaga, 2008).

Estos avances llevaron a nuevos estudios y clasificación de los problemas en los que eran computables y los que no lo eran, al mismo tiempo se buscó separar cada vez más los algoritmos de su implementación sobre una determinada computadora, y acercarlos cada vez más al problema, es decir buscar una mayor abstracción (Quiroga Rojas, 2008).

Por otra parte, los estudios de Shannon durante los años de 1948, permitieron la formalización de la teoría de autómatas, siendo su principal objetivo el representar foralmente el comportamiento de un dispositivo autónomo, que acepta señales de su ambiente, las procesa y las transmite de regreso (Jurado Málaga, 2008).

El concepto de autómata junto con el de la máquina de Turing, son esenciales para estudiar la complejidad algorítmica, independientemente, de si se estudian los problemas que pueden ser resueltos por una computadora (decidibles) o si se estudian los problemas que se pueden resolverse en un tiempo proporcional de acuerdo a una función que dependa del tamaño de la entrada (tratables) (Hopcroft, et al., 2007).

2.2.1 Autómatas

Un autómata es un concepto que se relaciona con un dispositivo capaz de imitar el funcionamiento de los seres vivos, pero en el campo de la informática un autómata es una maquina abstracta que puede manipular cadenas de símbolos de forma secuencial, que representan la información de entrada, los procesa en su interior y produce nuevas cadenas de símbolos a su salida, las cuales en un momento determinado dependen de toda la secuencia de símbolos, que se ha procesado hasta ese instante (Jurado Málaga, 2008).

Formalizando la definición de autómata (Quiroga Rojas, 2008), se puede decir que es una quíntupla:

$$A = (E, S, Q, f, g)$$

Donde:

E: es un conjunto finito, llamado conjunto de entrada, cuyos elementos se llamaran símbolos de entrada.

S: es un conjunto finito, llamado conjunto de salida, cuyos elementos se llamaran símbolos de salida.

Q: es el conjunto de estados.

f: es una función $f: E\ x\ Q \rightarrow Q$, llamada función de transición.

g: es una función $g: E\ x\ Q \rightarrow S$, llamada función de salida.

La definición de A puede representar un definición formal de algoritmo siempre y cuando sea una sucesión finita. Los autómatas finitos son los más sencillos y fáciles de estudiar, ya que están compuestos por una serie de estados finitos. El propósito de los estados es recordar la parte relevante de la historia del sistema (información necesaria) para a partir del símbolo de entrada deducir el símbolo de salida en un momento determinado (Hopcroft, et al., 2007).

Esta información puede ser la historia de símbolos de entrada, así como el estado inicial, el estado en el que se encontraba el autómata al recibir el símbolo de entrada, el estado final, etc. Y se considera un movimiento del autómata como el cambio entre un estado y otro al procesar un símbolo de entrada, aunque también puede permanecer en el mismo estado (Quiroga Rojas, 2008).

La diferencia fundamental entre los distintos tipos de autómatas finitos, es su tipo de control, si son deterministas, quiere decir que no se puede estar en más de un estado al mismo tiempo, en cambio los no deterministas pueden encontrarse en varios estados a la vez (Hopcroft, et al., 2007).

2.2.1.1 *Autómatas Finitos Deterministas*

Los Autómatas Finitos Deterministas (AFD) son máquinas teóricas que van cambiando de estados dependiendo de la cadena de símbolos que reciban, la salida de estos solo pueden variar entre dos valores, aceptado si la cadena de inicio recibida en la entrada es válida y no aceptado si no es válida (Jurado Málaga, 2008). Formalmente los AFD se pueden definir como una quintupla (Hopcroft, et al., 2007):

$$A = (Q, \Sigma, \delta, q_0, F)$$

Donde:

Q: es un conjunto finito de *estados*.

Σ: es un conjunto finito de *símbolos de entrada.*

δ: es una *función de transición* que toma como argumentos un estado y un símbolo de entrada y devuelve un estado.

q_0: es el estado inicial, que cumple con $q_0 \in Q$.

F: es el conjunto de estados finales o de aceptación, que cumple con $F \in Q$

Intuitivamente un AFD puede ser visto como una caja negra, que lee símbolos de una cadena escrita, el autómata en cada momento está en alguno de los posibles estados de Q, iniciando con q_0. En cada paso, se lee un símbolo y de acuerdo al estado en que se encuentre, hay un cambio de estado, para posteriormente leer el siguiente símbolo de la entrada.

Esta operación se realiza sucesivamente hasta que se terminan de procesar todos los símbolos de la entrada. Si al finalizar el AFD se encuentra en un estado final F, entonces el autómata acepta esa cadena de entrada, en caso contrario la rechaza (Quiroga Rojas, 2008).

Los ADF se pueden representar con tablas de transición o diagramas de transición, un ejemplo de un autómata representado a través de un diagrama de transición se presenta en la Figura 2-13.

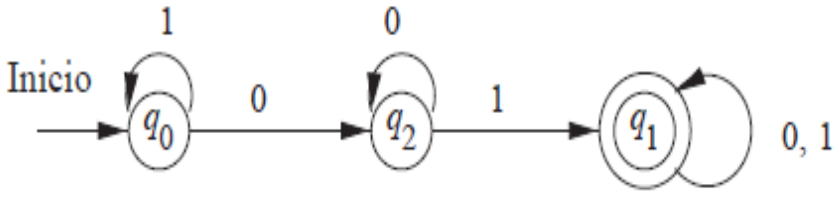

Figura 2-13. *ADF que acepta todas las cadenas que contienen la subcadena 01.*
(Hopcroft, et al., 2007)

2.2.1.2 Autómatas Finitos No Deterministas

La capacidad de los Autómatas Finitos No Deterministas (AFN) de estar en varios estados a la vez, se puede expresar como la capacidad que tiene el autómata de realizar conjeturas respecto a su entrada, por ejemplo al buscar palabras en un texto, si se conjetura que en algún momento se está al inicio de alguna de ellas, basta con comprobar la aparición de esa palabra carácter a carácter desde ese punto (Hopcroft, et al., 2007).

El problema con los AFN es que a diferencia de los AFD no es posible saber con exactitud cuál es la siguiente transición a realizar, ya que un mismo símbolo de entrada en el mismo estado te puede llevar diferentes estados (Jurado Málaga, 2008).

Los AFD son un caso particular de los AFN sin restricciones, en cuanto a las transiciones que puede haber de un estado a otro, esto significa que la definición formal de un AFN tiene los mismos elementos que la de un AFD, la única diferencia es que la función de transición δ devuelve un conjunto de cero o más estados, en vez de devolver un estado exactamente (Hopcroft, et al., 2007).

Los AFN aceptaran las cadenas de entrada siempre que sea posible que a partir del estado inicial, siguiendo alguna de las posibles secuencias de transiciones se pueda consumir toda la entrada y llegar a un estado final (Quiroga Rojas, 2008). Un ejemplo de un AFN a través de un diagrama de transiciones se puede observar en la Figura 2-14.

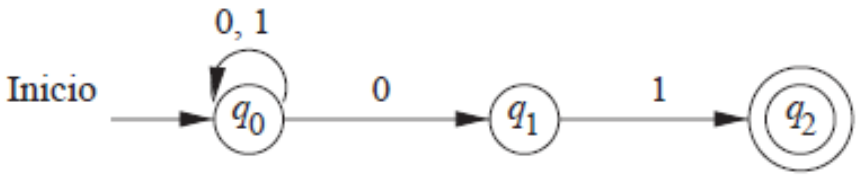

Figura 2-14. *AFN que acepta todas las cadenas que terminan en 01.*
(Hopcroft, et al., 2007)

2.2.2 Máquina de Turing

A finales del siglo XIX, la Teoría de Conjuntos tenía un gran impacto en los matemáticos contemporáneos, sin embargo algunos como Bertrand Russell pensaban que no estaba correctamente formalizada ya que permitía enunciados paradójicos y presentaron un sistema matemático de axiomas para traducir cualquier esquema parecido a un razonamiento lógico matemático (Hopcroft, et al., 2007).

Este sistema de axiomas, posteriormente fue demostrado falso por Kurt Gödel y su teorema de incompletitud, utilizando el teorema de factorización, el cual afirma que todo entero positivo puede representarse de forma única como producto de factores primos, para enumerar objetos que en apariencia no son enumerables. (Jurado Málaga, 2008)

Turing apoyo las teorías de Gödel al demostrar que hay problemas que no pueden resolverse, este es el problema de la indecidibilidad, el procedimiento utilizando por Turing puede observarse de manera muy general en la Figura 2-15. Éste se basa en que H_2 es un algoritmo que imprime "si" o "hola, mundo", dependiendo de la entrada. Si la entrada es el propio algoritmo en sí, la salida será "hola, mundo" si la primera salida del algoritmo de entrada es "si".

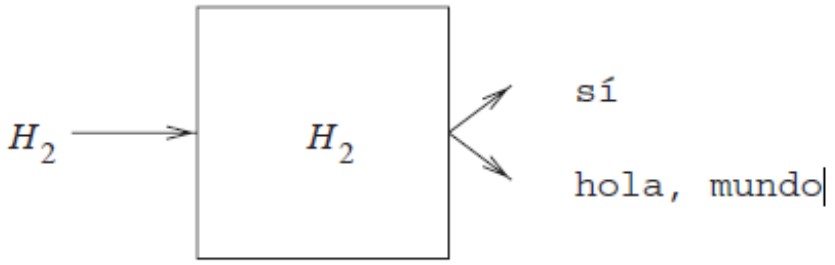

Figura 2-15. *Comprobación del problema de la indecidibilidad de Turing.*
(Hopcroft, et al., 2007)

Si se supone que la salida de es "si", esto significa que su entrada fue "hola, mundo", pero se había supuesto que la entrada era "si" en vez de "hola, mundo", lo mismo pasa cuando la salida de H_2 es "hola, mundo", pero en ese caso la salida tiene que ser "si", cualquier que sea la entrada, se puede deducir que proporciona la otra, esta situación paradójica, concluye que H_2 es indecidible, y por lo tanto no puede ser resuelto por una computadora, este fue un primer acercamiento a la complejidad algorítmica de parte de Turing (Hopcroft, et al., 2007).

El estudio de la indecidibilidad dio origen al estudio de los problemas intratables, es decir aquellos que son decidibles pero que requieren mucho tiempo para ser resueltos, estos últimos presentan una mayor dificultad, ya que a diferencia de los problemas indecidibles que suelen resultar obvios y por lo tanto no buscan resolverse, los problemas intratables se presentan continuamente y se busca solucionarlos (Jurado Málaga, 2008).

Basar la teoría de indecibilidad en un lenguaje en concreto tiene muchos problemas, ya que hay situaciones que no son sencillas de trasladar a un lenguaje en particular, por lo que se optó por un modelo mucho más sencillo, pero mucho más poderoso la Maquina de Turing (M.T) (Hopcroft, et al., 2007).

En la década de los 30, Allan Turing diseño el modelo matemático de una maquina abstracta, como extensión de los autómatas finitos, que tenía un gran poder de computo, pero al mismo tiempo contaba con una gran simplicidad a la que llamo Maquina de Turing (MT) (Quiroga Rojas, 2008). Esta MT es capaz de realizar un algoritmo sin las limitantes de las computadoras, como la velocidad de procesamiento o la capacidad de memoria, volviéndose un símbolo invariante de la informática (Jurado Málaga, 2008).

Los problemas que se puede resolver en un tiempo polinómico por una computadora son exactamente los mismos que pueden resolverse en tiempo polinómico por una M.T, así un problema que necesita tiempo polinómico puede considerarse tolerable, en cambio uno con tiempo exponencial puede considerar insoluble (Hopcroft, et al., 2007).

2.2.2.1 Definición de la Maquina de Turing

Una MT es un autómata finito que consta de un dispositivo de control que se mueve a lo largo de una cinta de longitud finita y que en cualquier momento se encuentra en un determinado estado de un conjunto finito de estados (Figura 2-16). La cinta se divide en casillas donde se escribe un símbolo de un conjunto finito de símbolos o alfabeto (Jurado Málaga, 2008).

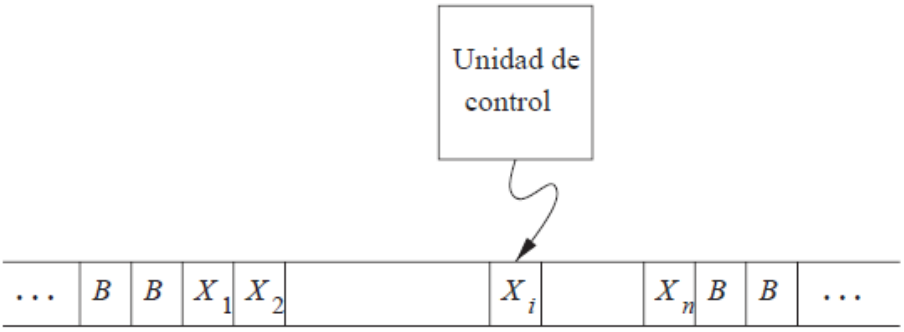

Figura 2-16. *Representación gráfica de una máquina de Turing.*
(Hopcroft, et al., 2007)

La entrada de la MT es una cadena de símbolos de longitud finita que pertenece a un alfabeto de entrada la cual se coloca en la cinta, precedida de un símbolo especial llamado espacio en blanco, este símbolo se conoce como símbolo de cinta, mientras que la cinta se expande indefinidamente hacia la izquierda y la derecha (Hopcroft, et al., 2007).

El cabezal se ubica al inicio en el símbolo más a la izquierda de la entrada, es decir el espacio en blanco. Los movimientos que puede realizar la M.T dependen del estado de la unidad de control y el símbolo de la cinta que se señala en un

determinado momento, pudiendo realizar las siguientes acciones (Jurado Málaga, 2008):

- **Cambiar de estado**. El siguiente estado puede ser opcionalmente el estado actual.

- **Escribirá un símbolo de cinta en la casilla que señala la cabeza.** Este símbolo de cinta reemplaza a cualquier símbolo que estuviera anteriormente en dicha casilla, incluso se puede escribir el mismo símbolo que ya se encontraba en la cinta.

- **Moverá la cabeza de la cinta hacia la izquierda o hacia la derecha.**

Formalmente una MT se puede definir como una septupla (Hopcroft, et al., 2007):

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Donde:

Q: es el conjunto finito de *estados* de la unidad de control.

Σ: es un conjunto finito de *símbolos de entrada.*

Γ*:* es el conjunto completo de símbolos de la cinta.

δ: es una *función de transición* que toma como argumentos $\delta(q, X)$, q es un estado y X un símbolo de la cinta, se valor está definida por (p, Y, D), donde:

- p es el siguiente estado de Q.

- Y es el símbolo de Γ que se escribe en la casilla señalada sustituyendo a cualquier otro símbolo que se encontrara en esa casilla.

- D es una dirección, la cual puede ser izquierda (L) o derecha (R), e indica la dirección hacia la cual se mueve el cabezal.

q_0: es el estado inicial, elemento de Q en el que se ubica inicialmente la unidad de control.

B: es el símbolo de espacio en blanco que cumple con $B \in \Gamma$ pero $B \notin \Sigma$, aparece inicialmente en todas las casillas que no almacenan símbolos de entrada.

F: es el conjunto de estados finales o de aceptación, que cumple con $F \in Q$.

La MT termina su cálculo cuando la unidad de control alcanza un estado de final o de aceptación, partiendo de la situación inicial de que la cadena de entrada este en la cinta, q_0 es el estado inicial de la MT y la cabeza de escritura a punta a la celda de la cinta ocupada por el primer carácter de la cadena de entrada (Jurado Málaga, 2008).

Otra forma de aceptación que se aplica a la M.T es la aceptación por parada, esto ocurre cuando en un determinado momento $\delta(q, X)$ no está definida, por lo que se puede decir que siempre que una M.T se detiene se encuentra en un estado de aceptación. Aunque no siempre es posible una M.T que siempre se detiene, sin importar si acepta o no la entrada, sirven para modelar algoritmos y ayudan al estudio de la teoría de la decibilidad (Hopcroft, et al., 2007).

2.2.2.2 Variaciones de la Maquina de Turing

Algunas variaciones de la MT encontradas en la literatura son las siguientes:

- **Máquina de Turing de varias cintas:** Esta MT tiene k cintas y k cabezas de lectura/escritura, las cuales se mueven sobre una cinta particular y se mueven completamente independientes de las demás. No obstante, un movimiento de la MT realiza las acciones correspondientes sobre todas las cintas (Figura 2-17) (Jurado Málaga, 2008) .

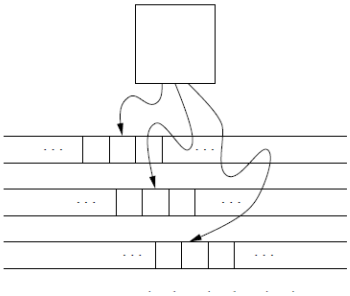

Figura 2-17. *Máquina de Turing de varias cintas.*(Hopcroft, et al., 2007)

- **Máquina de Turing No Determinista:** A diferencia de las MT deterministas, en que su función δ genera un conjunto de tuplas $\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$ y la MTN elige en cada transición cual será el siguiente movimiento de las tuplas posibles. Las MTN suelen utilizar un tiempo menor que las MTD, pero actualmente solo se pueden simular con M.T.D (Figura 2-18) (Hopcroft, et al., 2007).

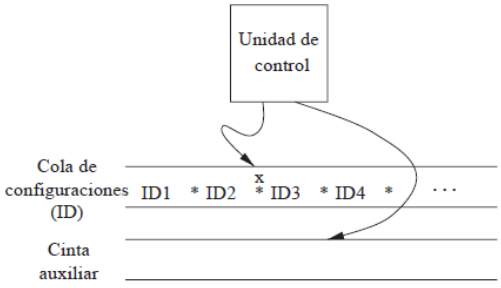

Figura 2-18. *Simulación de una M.T.N mediante una M.T.D.*(Hopcroft, et al., 2007)

- **Máquina de Turing de Múltiples Pistas:** En estas MT la cinta se divide en un número k de pistas, de esta forma los símbolos de la cinta se representan como k-tuplas, cambiando solamente la representación de los símbolos de la MT (Jurado Málaga, 2008).

- **La Máquina de Turing Universal:** La definición de MT es tan potente y general que es posible diseñar una MT que sea capaz de ejecutar cualquier algoritmo, es decir capaz de calcular o simular el comportamiento de cualquier MT (Quiroga Rojas, 2008).

2.2.2.3 La Máquina de Turing y la Complejidad Algorítmica

Una MT es capaz de simular el comportamiento de una computadora, esto se puede observar en la Figura 2-19, la cual si bien es una representación informal y no considera varios aspectos de implementación, sirve para demostrar que una MT puede realizar las mismas acciones que una computadora, difiriendo en un tiempo polinómico, que para el caso de la complejidad, se consideran similares. De esta forma la MT es la representación formal de lo que cualquier dispositivo de computo puede calcular (Hopcroft, et al., 2007).

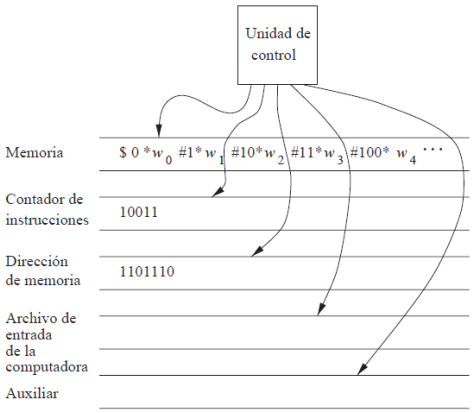

Figura 2-19. *Máquina de Turing que simula una computadora típica.*
(Hopcroft, et al., 2007)

El concepto de procedimiento algorítmico que se ejecuta sobre una secuencia de símbolos es idéntico al de un proceso que puede ser ejecutado por una MT, esta afirmación es conocida como la tesis de Church-Turing, la cual si bien no tiene una afirmación matemática exacta, es aceptada de forma general, definiendo un algoritmo como un procedimiento que puede ejecutarse en una MT, este es un punto de inflexión en el estudio de la complejidad (Jurado Málaga, 2008).

Utilizando la M.T.D se puede expresar lo que puede o no ser resulto por un algoritmo o lo que es lo mismo por un dispositivo de cómputo, así es posible formalizar diversos conceptos que son útiles en el estudio de la complejidad computacional (Jurado Málaga, 2008):

- **Función Parcialmente Calculable:** Es una función $f: D \subset N^r \rightarrow N$ tal que existe una M.T que para cada tupla $t \in D$ la maquina calcula $f(t)$.

- **Función Calculable:** Es una función que es calculable y está definida sobre todo N^r, es decir, $D = N^r$

- **Función Recursiva:** Para que una función se considere recursiva debe tener los siguientes elementos:

    - Definiciones básicas que establezcan de manera axiomática el valor que toma la función para determinados valores del conjunto origen.

    - Una o más reglas recursivas que permitan calcular nuevos valores de la función a partir de otros valores conocidos.

    - Aplicar las definiciones básicas y las recursivas un número finito de veces debe ser suficiente para calcular cualquier valor de la función.

- **Función Recursiva Primitiva:** Es una función que puede ser definida a partir de las funciones básicas mediante cero o más aplicaciones de las reglas de:

    - **Composición:** Teniendo las funciones $a_1, \dots, a_m$ y β definidas como $\alpha_i: N^n \rightarrow N$ y $\beta: N^m \rightarrow N$ se define una función ϕ tal que:

    $$\phi(x_1, \dots, x_n) = \beta(\alpha_1(x_1, \dots, x_n), \dots, \alpha_m(x_1, \dots, x_n))$$

    - **Recursión Mínima:** Teniendo las funciones α y β definidas como $\alpha: N^n \rightarrow N$ y $\beta: N^{n+2} \rightarrow N$ se define una función $\phi: N^{n+1} \rightarrow N$ tal que:

    $$\phi(0, x_1, \dots, x_n) = \alpha(x_1, \dots, x_n)$$

    $$\phi(y+1, x_1, \dots, x_n) = \beta(y, \phi(y, x_1, \dots, x_n), x_1, \dots, x_n)$$

- **Función μ-recursivas:** Es una función que puede ser definida a partir de las funciones básicas mediante cero o más aplicaciones de las reglas de

composición, recursión primitiva y minimización (función ϕ que hace corresponder a cada entrada $\bar{x}$ el menor entero que cumple que $\alpha(\bar{x}, y) = 0$).

El conjunto de funciones μ-recursivas es igual al conjunto de las M.T, así un problema puede ser resulto con una M.T solo se si se puede plantear utilizando funciones recursivas (Jurado Málaga, 2008).

2.3 Complejidad Computacional o Algorítmica

La teoría de la complejidad algorítmica se encarga de definir los criterios básicos para que un problema computable sea factible, es decir, si existe un algoritmo eficiente que pueda resolverse, y en caso de que lo haya su grado de eficiencia (Jurado Málaga, 2008).

Un algoritmo eficiente es aquel que para ser ejecutado utiliza poco tiempo y requiere poco espacio de memoria, aunque tradicionalmente el aspecto más importante para medir la eficiencia es el tiempo (LEE, et al., 2007). Calcular la eficiencia de un algoritmo no es sencillo ya que un problema puede tener varias soluciones con diferentes grados de eficiencia y al mismo tiempo es dependiente del hardware utilizado. La MT es una alternativa libre de estas limitaciones, utilizada para medir la complejidad de un algoritmo (Jurado Málaga, 2008).

Se puede formalizar dos conceptos fundamentales para el estudio de la complejidad de los algoritmos (Jurado Málaga, 2008):

- **Complejidad Espacial:** Es una funcion que se refiere al numero maximo de casillas consultas para cualquier entrada de longitud n y se representa como $S(n)$. De esta complejidad se puede definir dos clases, los que utilizando una MTD o MTN tienen una complejidad espacial *S(n)*.

- **Complejidad Temporal:** Es una funcion que se refiere al numero maximo de movimientos sobre la cadena de longitud n y se representa como $T(n)$. De esta complejidad se puede definir dos clases, los que utilizando una MTD o MTN tienen una complejidad temporal *T(n)*.

Para efectos de estudiar la eficiencia de un algoritmo, la complejidad se mide en términos asintóticos, si este tiene una complejidad temporal (o espacial) de $T(n^3 + n)$, esta se ve reducida o simplificada a $T(n^3)$, ya que n^3 es el termino de mayor orden, por lo cual es el que rige el crecimiento de la complejidad del algoritmo. En la Figura 2-20 se puede observar el crecimiento de distintas funciones de complejidad temporal de acuerdo a la entrada del problema (LEE, et al., 2007).

Las funciones S(n) y T(n) no son funciones exactas, solo son indicadores de la variación del espacio o tiempo ocupado, es decir la tasa de crecimiento en función de la longitud de la entrada. Los algoritmos con complejidad del tipo $O(n)$, $O(n^2)$, $O(n^3)$, en general $O(n^c)$ son llamados algoritmos de complejidad polinómica, los algoritmos con comportamiento 2^n, e general C^n, son llamados algoritmos de complejidad exponencial (Jurado Málaga, 2008).

Función de complejidad temporal	Tamaño del problema (n)			
	10	10^2	10^3	10^4
$\log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n \log_2 n$	0.33×10^2	0.7×10^3	10^4	1.3×10^5
n^2	10^2	10^4	10^6	10^8
2^n	1 024	1.3×10^{30}	$>10^{100}$	$>10^{100}$
$n!$	3×10^6	$>10^{100}$	$>10^{100}$	$>10^{100}$

Figura 2-20. *Crecimiento de la complejidad temporal con respecto a la entrada del problema.* (LEE, et al., 2007)

Los algoritmos con complejidad polinómica son efectivos aun con entradas muy grandes, en cambio, los que tiene complejidad exponencial, solo son efectivos, con entradas muy pequeñas (Baase & Van Gelder, 2002).

Otro factor a considerar es la complejidad del problema, al comparar todos los posibles algoritmos conocidos para resolver un determinado problema, se obtiene una "cota superior de la complejidad" de ese problema, es decir la complejidad del mejor algoritmo que se haya podido encontrar para resolverlo. Por otro lado la cota inferior, se obtiene cuando se prueba que no existe algoritmo que pueda resolver un problema determino si utilizar una mínima de recursos (Jurado Málaga, 2008).

Hay problemas que hasta el momento solo han podido ser resueltos con una complejidad exponencial, por lo que incluso los mejores algoritmos tardar años o incluso siglos, pero tampoco se ha podido demostrar que requieren necesariamente esa cantidad de tiempo o espacio (Baase & Van Gelder, 2002).

Una clase de complejidad es un conjunto de funciones que pueden ser calculadas con cierta cantidad de recursos, en esta categoría entran los problemas P, NP, NP-completos, #P, entre otros, los cuales se usan para clasificar los algoritmos de acuerdo a su complejidad (Arora & Barak, 2007).

Los problemas de la clase P y NP son considerados problemas de decisión, es decir que el problema tiene dos posibles respuestas, si y no, de acuerdo a la entrada que tenga el problema, así se puede definir estos problemas de forma abstracta como la correspondencia entre todas las entradas y el conjunto {si, no}. La relación entre las distintas clases de problemas de decisión se muestra en la Figura 2-21 (Baase & Van Gelder, 2002).

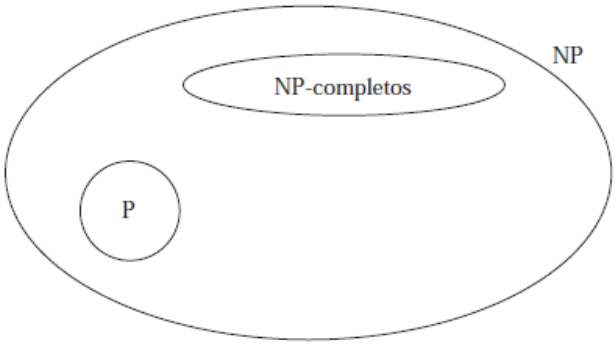

Figura 2-21. *Relación entre las diferentes clases de problemas de decisión.*

(LEE, et al., 2007)

Uno de los problemas más importantes en el estudio de la complejidad es la conjetura $P \neq NP$ la cual supone que los problemas que se pueden resolver con MTD en tiempo polinómico incluyen al menos algún problema que aún no han podido ser resuelto por las MTD en tiempo polinómico, existen muchos problemas que pueden considerarse de este tipo al ser resueltos por MTN en tiempo polinómico pero no por MTD (Hopcroft, et al., 2007).

2.3.1 Problemas P

Un problema pertenece a la clase P si existe un algoritmo que lo resuelve siendo ejecutado en un MTD con una cota temporal $T(n)$ polinómica (Hopcroft, et al., 2007). Un ejemplo es el algoritmo de Kruskal para determinar el árbol de recubrimiento de peso mínimo de un grafo. (LEE, et al., 2007)

2.3.2 Problemas NP

Los problemas de clase NP son aquellos para los que existe una MTN que los puede ejecutar con una cota temporal polinómica. Las MTN no siguen un flujo fijo, sino que actúan en función de una serie de decisiones tomadas en tiempo real, algunos de ellos son eficientes, pero no se puede demostrar que pertenezcan a la clase P por su no determinismo (Jurado Málaga, 2008).

Como todas las M.T.D son M.T.N sin la capacidad de elegir sus movimientos, $P \subseteq NP$, sin embargo hay muchos problemas NP que parecen no pertenecer a P, siendo la comprobación de $P = NP$ (verdadera o falsa) una de las cuestiones más importante en el campo de teoría la complejidad, es decir, demostrar que cualquier cosa hecha por una M.T.N en tiempo polinómico puede ser hecha en el mismo tiempo por una M.T.D, sin importar si este polinomio es de grado mayor (Hopcroft, et al., 2007). Ejemplos de Problemas NP:

- **Problema de Hamilton:** Dado un conjunto de puntos ¿Encontrar una camino que pase una sola vez por cada uno de los puntos? (Jurado Málaga, 2008).

- **Problema del Viajante Comerciante:** Teniendo un grafo con pesos en las aritas y un limite de peso w, ¿Es posible visitar todos los vertices manteniendo el peso como maximo en w? (Baase & Van Gelder, 2002).

El principal objeto de interés es demostrar que el conjunto NP es igual conjunto P, lo cual todavía no se ha podido demostrar, para eso habría que poder transformar una MTN a una MTD con cota temporal polinómica, pero al mismo tampoco se ha demostrado la desigualdad entre los dos al encontrar un algoritmo que pertenezca a NP y no a P (Hopcroft, et al., 2007). Los algoritmos deterministas para los problemas NP son muy costosos aunque generalmente dada una posible solución, comprobar su validez es sencillo (Jurado Málaga, 2008).

Por otro lado, los problemas NP-completos o duros, son un subconjunto de los problemas NP con la características que se si se encontrara un algoritmo que los solucionara en tiempo polinómico para cualquiera de ellos, entonces habría un algoritmo polinómico para todos los problemas NP, demostrando la reducción de todos los problemas NP a problemas P en tiempo polinómico y comprobando que P=NP. NP-difícil no es sinónimo de NP y difícil, sino “al menos tan difícil como cualquier problema en NP” (Baase & Van Gelder, 2002). Algunos ejemplos de este tipo de problemas son:

- **SAT:** Dada una expresión lógica, determinar si es posible satisfacerla, es decir, encontrar los posibles valores para que la expresión sea verdadera (Jurado Málaga, 2008).

- **Cobertura de vértices:** Dado un grafo G encontrar un subconjunto de vértices, que contenga todos las aristas de V (Baase & Van Gelder, 2002).

2.3.3 Problemas #P

Los problemas P y NP como ya se mencionó son considerados problemas de conteo, ya que solo están interesados en verificar o encontrar una posible solución, sin embargo en muchas ocasiones esto no es suficiente, e interesa encontrar todas las soluciones posibles para ese problema, ese tipo de problemas se denomina problemas conteo o #P (Arora & Barak, 2007).

Los problemas #P se pueden definir como aquellos que pueden ser expresados por una MTN, la cual para cualquier entrada puede calcular con una cota de complejidad temporal polinómica todas las posibles soluciones para ese problema

en concreto (Hernández Servín, et al., 2014). Algunos ejemplos de problemas #P (Arora & Barak, 2007): **#SAT** (dada una función booleana encontrar todos los argumentos que la satisfagan), **#CYCLE** (dado un grafo dirigido encontrar todos los ciclos simples del grafo) y el **Conteo de cubiertas de aristas**.

2.3.4 Calculo de la Complejidad

En (Gross, 2009) se estudia la recurrencia lineal con coeficientes constantes, se define una secuencia en un conjunto S como una lista de elementos de ese conjunto S (llamado el rango de la secuencia) de la forma:

$$x_0\ x_1\ x_2\ \dots\ x_n$$

Los cuales como se puede apreciar tienen índices enteros no negativos. Una expresión algébrica en el argumento n para el valor del elemento general x_n de la secuencia es llamada fórmula cerrada para la (elementos de la) secuencia. Ejemplificando, la formula cerrada $x_n = n^3 - 5n$ específica la secuencia:

$$\langle x_n \rangle\colon\ 0\quad -4\quad -2\quad 12\quad 44\quad 100\quad 186\dots\infty$$

Todas las secuencias tienen un comportamiento de crecimiento que es interesante de analizar y estudiar, poniendo como ejemplo la secuencia $\langle x_n = n^2 - 8n + 15 \rangle$, la cual como se puede observar en la Figura 2-22 tiene forma de una parábola con su mínimo en $n = 4$, después de la cual eventualmente aumenta.

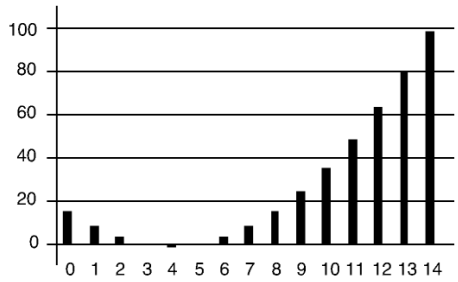

Figura 2-22. *Grafica del Comportamiento de Crecimiento de la Secuencia* (Gross, 2009)

Es importante mencionar que cualquier secuencia polinómica eventualmente crecerá o decrecerá dependiendo del signo de su termino de mayor grado. Una recurrencia estándar para una secuencia se define como un conjunto de valores iniciales:

$$x_0 = b_0 \qquad x_1 = b_1 \qquad \dots \qquad x_k = b_k$$

Y una fórmula de recursión $x_n = \emptyset\ (x_{n-1}, x_{n-2}, \dots, x_0)$ para $n > k$

De la cual se puede calcular el valor de x_n para cualquier valor $n > k$, de los valores de las entradas iniciales.

Resolver una recurrencia implica inferir una formula cerrada de la recurrencia. Los primeros valores especificados por la formula cerrada $x_n = n^2$ son:

$$0^2 = 0 \quad 1^2 = 1 \quad 2^2 = 4 \quad 3^2 = 9 \quad 4^2 = 16$$

Estos coinciden con los especificados por la recurrencia:

$$x_0 = 0 \quad x_1 = 1 \quad x_2 = 4 \quad x_3 = 9 \quad x_3 = 16$$

Mediante inferencia se puede llegar a la conclusión de que $x_n = n^2$ resuelve la recurrencia.

En algunas ocasiones es posible conjeturar la solución a una recurrencia con los siguientes pasos:

1. Examinar pequeños casos sistemáticamente
2. Adivinar un patrón que cubra todos los casos
3. Probar que la conjetura es correcta

Existen varios tipos de recurrencias algunas de las más conocidas son:

1. Recursión de la Torre de Hanói, expresada por: $h_2 = 2h_{n-1} + 1$
2. Recursión de Fibonacci, expresada por: $f_n = f_{n-1} + f_{n-2}$
3. Recursión Catalana, expresada por: $C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-1}C_0$

Esta Tesis se centrara en la recursión de Fibonacci, debido a que como se demostrara posteriormente el algoritmo propuesto (Hernández Servín, et al., 2014) tiene una recursividad prácticamente igual a la Fibonacci, salvo por pequeñas diferencias. En este sentido, la secuencia de Fibonacci $\langle f_n \rangle$ es definida por la recurrencia:

$$f_0 = 0; \quad f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Los primeros valores de esta secuencia se pueden observar en la Tabla 2-1.

Tabla 2-1. *Primeros números de la secuencia de Fibonacci.*

n	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*…*
f_n	0	1	1	2	3	5	8	13	21	34	…

Un número de Fibonacci es cualquier número que ocupa un lugar en la secuencia de Fibonacci. Encontrar una fórmula cerrada para la secuencia de Fibonacci no es fácil de conjeturar partiendo de pequeños casos, pero una vez obtenida, es posible su verificación a través de la inducción.

Con recurrencias simultáneas, se usa la sustitución para reducir el número de diferentes secuencias ocurridas en otras recursiones. El objetivo es reducir la solución del sistema inicial a la solución de una o más recurrencias lineales independientes. Una forma de poder entender la secuencia de Fibonacci de forma gráfica se muestra en la Figura 2-23.

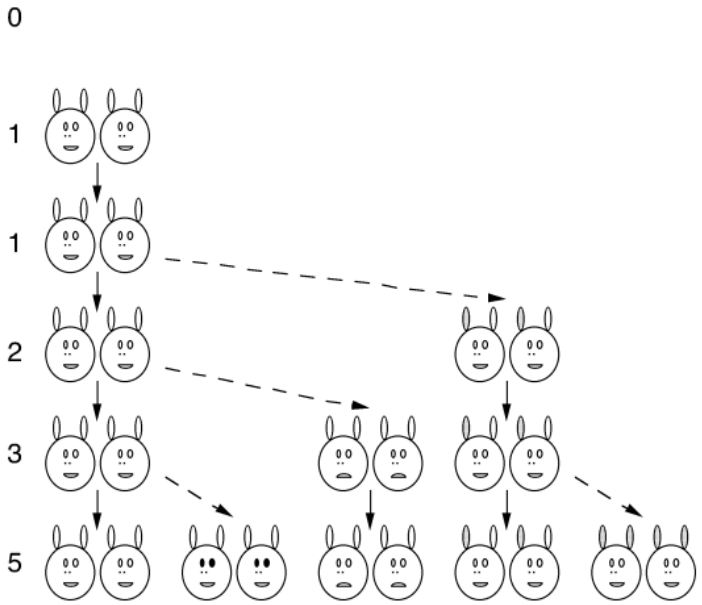

Figura 2-23. *Crecimiento de la secuencia de Fibonacci con conejos.* (Gross, 2009)

En 1202, Fibonacci imagino un tipo de conejo al que el tomara un mes llegar desde su nacimiento a la madurez, con un periodo de gestación de un mes. De cada hembra madura nacen al mes dos conejos, un varón y una hembra. Sea b_n el número de pares de conejos recién nacidos y a_n el número de pares de conejos adultos. Suponiendo que no hay conejos cuando $n = 0$ meses, y que un par de recién nacidos inicia al sistema después del mes 1 (Gross, 2009).

El número total $f_n = a_n + b_n$ de pares de conejos es moldeado como una recursión simultánea con condiciones iniciales:

$$a_0 = 0, \qquad a_1 = 0, \qquad b_0 = 0, \qquad b_1 = 1$$

Y las ecuaciones de relación: $a_n = a_{n-1} + b_{n-1}$

$$b_n = a_{n-1}$$

$$f_n = a_n + b_n$$

El primer paso es usar substituciones para reducirla a una recurrencia con una sola variable, por lo que:

$$a_n = a_{n-1} + b_{n-1} = f_{n-1}$$

$$b_n = a_{n-1} = f_{n-2}$$

$$f_n = a_n + b_n = f_{n-1} + f_{n-2}$$

Y si $f_0 = a_0 + b_0 = 0$ y $f_1 = a_1 + b_1 = 0 + 1 = 1$. El resultado es una recurrencia de una sola variable expresada de la forma:

$$f_0 = 0, \qquad f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

La cual se puede reconocer como la recurrencia de Fibonacci y se puede resolver con el método de generación de funciones que se muestra a continuación:

$$f_n z^n = f_{n-1} z^n + f_{n-2} z^n$$

$$\sum_{n=2}^{\infty} f_n z^n = \sum_{n=2}^{\infty} f_{n-1} z^n + \sum_{n=2}^{\infty} f_{n-2} z^n$$

Usando $F(z)$ como la función de generación para f_n

$$\sum_{n=2}^{\infty} f_n z^n = z \sum_{n=2}^{\infty} f_{n-1} z^{n-1} + z^2 \sum_{n=2}^{\infty} f_{n-2} z^{n-2}$$

$$F(z) - f_1 z - f_0 = z(F(z) - f_0) + z^2 F(z)$$

Resolviendo para $F(z)$.

$$F(z)(1 - z - z^2) = f_1 z + f_0 - f_0 z = 1z + 0 - 0z = z$$

$$F(z) = \frac{z}{1 - z - z^2}$$

Para resolver respecto a f_n se utilizan las ecuaciones cuadráticas

$$1 - z - z^2 = \left(1 - \frac{1 + \sqrt{5}}{2} z\right).\left(1 - \frac{1 - \sqrt{5}}{2} z\right)$$

Sus raíces involucran el “golden mean” y su conjugado $\gamma = \frac{1+\sqrt{5}}{2}$ y $\hat{\gamma} = \frac{1-\sqrt{5}}{2}$

Usando fracciones parciales: $F(z) = \frac{z}{1-z-z^2} = \frac{1}{\sqrt{5}} . (\frac{1}{1-\gamma z} - \frac{1}{1-\hat{\gamma} z})$

Con lo cual se puede concluir que $f_n = \frac{1}{\sqrt{5}} . (\gamma^n - \hat{\gamma}^n)$

La cual es llamada la fórmula de Binet para los números de Fibonacci en honor a Jacquet Binet que la descubrió en 1843.

2.4 Algoritmo Exacto para el Conteo de Cubiertas de Aristas

En (Hernández Servín, et al., 2014) se propone un algoritmo exacto para resolver el problema del conteo de cubiertas de aristas, el cual será implementando en esta Tesis, este proceso se divide en dos casos:

- La gráfica contiene ciclos intersectados.
- La gráfica no contiene ciclos intersectados.

El funcionamiento del algoritmo en el caso general, es decir cuando la gráfica contiene ciclos intersectados, se describe a continuación:

1. Sea G una gráfica conectada, el árbol de expansión construido a partir de G puede denotarse como T_G, cumpliendo con la condición de que $V(T_G) = V(G)$. Al obtener T_G las aristas de la gráfica se dividen en dos grupos:
   - Las aristas pertenecientes a T_G son llamadas aristas del árbol.
   - Las aristas que se obtienen de realizar $E(G) - E(T_G)$ son llamadas aristas de retroceso.
2. Sea e una arista de retroceso, la unión en la ruta en T_G de los puntos finales de e forman un ciclo simple, cada ciclo simple es llamado ciclo fundamental de G con respecto a T_G. Cada arista de retroceso envuelve una única ruta contenida en un ciclo fundamental. En la Figura 2-24, la arista de retroceso se muestra punteada.

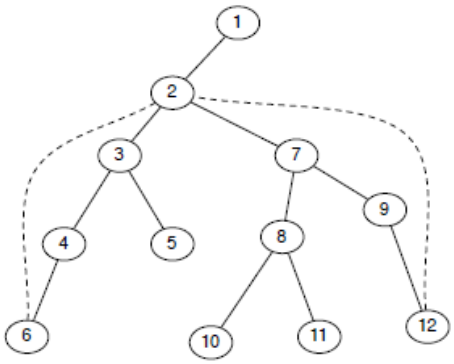

Figura 2-24. *Árbol T_G con dos Aristas de Retroceso (línea punteada) y dos ciclos fundamentales*

(Hernández Servín, et al., 2014)

Considerando a $C = \{C_1, \ldots, C_k\}$ como el conjunto de ciclos fundamentales encontrados mediante un algoritmo de búsqueda en profundidad (DFS) en G.

Tomando cualquier par de ciclos fundamentales $C_i\ C_j$ de C, si C_i y C_j comparten aristas entonces son llamados ciclos intersectados; de otra manera son llamados ciclos independientes.

Sea T_G la gráfica construida a partir de una búsqueda en profundidad sobre una gráfica G. Asumiendo que T_G es conectado y no tiene ciclos intersectados, el método para contar coberturas de aristas en gráficas sin ciclos intersectados se puede resolver en tiempo polinómico (Hernández Servín, et al., 2014) y será descrito posteriormente.

En cambio, si se asume que T_G es conectado y tiene ciclos intersectados, una estrategia para el conteo de cubiertas es reducir la gráfica a una que no contenga ciclos intersectados utilizando reglas que no cambien el conteo de cubiertas. (Hernández Servín, et al., 2014) A continuación se presentan las reglas a utilizar.

Regla de División: Sea $G = (V, E)$ una gráfica que tiene ciclos intersectados. Se escoge $e = \{u, v\} \in E$ de la siguiente manera:

- e está involucrado en un número máximo de ciclos intersectados en G.

- $\delta(u) \geq 3\ o\ \delta(v) \geq 3$

La regla de división consiste en generar dos nuevas subgráficas a partir G de la siguiente manera:

- $H_1 = (V_1, E_1)$ será la gráfica donde $V_1 = V$ y $E_1 = E - e$.

- $H_2 = (V_2, E_2)$ será la gráfica donde V_2 se obtiene de la siguiente manera:

    1) Agregar cada nodo contenido en V excepto u y v.

    2) Por cada arista incidente a $\{u, v\}$ (escrito como $\{x, u\}\ o\ \{v, x\}$) agregar dos nuevos nodos (utilizando u_{x_1} y u_{x_2} para $\{x, u\}$ o v_{x_1} y v_{x_2} para $\{v, x\}$).

    E_2 se obtiene de la siguiente manera:

    1) Por cada arista $\{x, u\}\ o\ \{v, x\}$ en $E(H)$ con $x \neq u$ y $x \neq v$, agregar dos aristas $(\{x, u_{x_1}\}\ y\ \{u_{x_1}, u_{x_2}\})$ o $(\{x, v_{x_1}\}\ y\ \{v_{x_1}, v_{x_2}\})$ respectivamente.

    2) Agregar $E(H) \backslash \{e \cup \{x, u\} \cup \{v, x\}\}$.

Hay que enfatizar dos puntos importantes de esta regla, si se parte de que e es obtenida de al menos un par de ciclos intersectados, entonces H_1 es todavía una gráfica conectada. $|V_1| = n_1 = n, |E_1| = m_1 = m - 1$. El número de ciclos base en

H_1 es $nc_1 = m_1 - n_1 = m - n - 1 = nc - 1$. Entonces, H_1 contiene al menos un par de ciclos intersectados menos que H.

Por otro lado sea $n_2 = |V_2|$ y $m_2 = |E_2|$. Todos los ciclos de H que contienen a e no son ciclos en H_2, por lo que se puede considerar que $m_2 \leq m - 5$ debido a que $\delta(v) \geq 3$, $\delta(u) \geq 2$ y $n_2 = n - 2$ por lo tanto, $nc_2 = m_2 - n_2 \leq m - n - 3 = nc - 3$. El número de ciclos básicos en H_2 y H_1 son representados por nc_2 y nc_1 respectivamente, y se puede decir que el número de conjuntos de coberturas de aristas de la gráfica principal, es la suma de los conjuntos de coberturas de las dos graficas generadas.

Esto se puede expresar matemáticamente como $NE(G) = NE(H_1) + NE(H_2)$, la comprobación y explicación de los teoremas en los cuales se basa este algoritmo se pueden observar en (Hernández Servín, et al., 2014).

El comportamiento en tiempo del algoritmo reside en la recursividad, la cual corresponde con el número de ciclos intersectados en la gráfica asociada con cada nodo de E_G. Tomando como parámetro la complejidad nc, entonces, la complejidad en tiempo del algoritmo puede ser expresado con la recurrencia:

$$T(nc) = T(nc - 1) + T(nc - 3)$$

A esta recurrencia le corresponde el polinomio característico $p(r) = r^3 - r^2 - 1$, la cual al resolverse se obtiene el límite superior de complejidad en el peor de los casos, el cual es de $O\left(r^{(m-n)} * (m + n)\right) \approx O(1.465571^{(m-n)} * (m + n))$.

El proceso completo para el cálculo de la complejidad se puede observar en (Hernández Servín, et al., 2014). A través de la implementación y experimentación de este algoritmo se planea estudiar el comportamiento de uso de memoria del mismo, el cual no ha sido previamente analizado.

2.4.1 Diferencias y Mejoras Respecto a Trabajos Relacionados

Debido a la naturaleza de los problemas $\#P$ y su complejidad, en general, estos son menos estudiados que los problemas NP, esto también ocurre con el problema del conteo de cubiertas de aristas.

Existen algoritmos y aproximaciones para resolver el problema de cubiertas de aristas, sin embargo, prácticamente no hay investigaciones sobre el problema de conteo, uno de los pocos trabajos relacionados con este tema se puede encontrar en (Lin, et al., 2014), en donde los autores desarrollaron un FPTAS (Fully

Polynomial-Time Approximation Scheme) determinista, el cual se va a describir a continuación, sin entrar en detalles, solo con fines de comparación.

El enfoque de los autores es uno de aproximación, para cualquier parámetro $\epsilon > 0$, el algoritmo obtiene el número $\widehat{N}$ tal que $(1-\epsilon)N \leq \widehat{N} \leq (1+\epsilon)N$, donde N es el número preciso de coberturas de aristas para una gráfica de entrada.

El tiempo de ejecución de este algoritmo está ligado a $poly(n, 1/\epsilon)$, donde n es el número de vértices de la gráfica. El enfoque que utiliza el algoritmo, el cual es el enfoque común para los algoritmos de aproximación, es el de relacionar el conteo con una distribución de probabilidad.

Para hacer aleatorio el conteo se utiliza el paradigma de "contar vs muestrear". Este algoritmo se basa en estimar la probabilidad marginal, es decir la probabilidad de que una arista sea escogida cuando se muestre una cobertura de arista uniforme al azar, y con base en ella aproximar el conteo.

En el FPTAS propuesto la probabilidad marginal se calcula directamente utilizando solamente los vecinos locales alrededor de la arista, ya que se demuestra que las aristas lejanas tiene poca influencia en el cálculo de la probabilidad marginal, y finalmente en la aproximación más satisfactoria se utiliza la propiedad de decadencia de la correlación (correlation decay).

La diferencia principal en los trabajos presentado en (Hernández Servín, et al., 2014) y (Lin, et al., 2014), es que mientras que en el primero se presenta una solución o algoritmo exacto, en el segundo se presenta una solución basada en un algoritmo de aproximación (aunque factible), mientras que en los tiempos de ejecución se puede observar que el algoritmo exacto ofrece un límite superior exponencial, mientras que el de aproximación tiene un tiempo polinómico.

Resumiendo, las diferencias principales entre estos dos trabajos se muestran en la Tabla 2-2.

Tabla 2-2. *Tabla Comparativa entre los Algoritmos Simple FPTAS y Low Exponential*

Algoritmo	***Autores***	***Complejidad***	***Tipo de Algoritmo***
Simple FPTAS	Chengyu Lin, Jingcheng Liu, Pinyan Lu	$O(m \cdot n^2 \cdot (m \cdot \frac{1}{\epsilon}) \log_2 6)$	Algoritmo de Aproximación
Low Exponential	J. A. Hernández Servín, J. Raymundo Marcial Romero, Guillermo De Ita Luna	$O((1.324718)^m \cdot (m + n))$	Algoritmo Exacto

Capítulo 3

Desarrollo

3.1 Introducción

La metodología de desarrollo utilizada es la metodología incremental, debido a que es la que mejor se ajusta a las necesidades particulares del problema a resolver (Sección 2-4). Esta metodología se basa en dividir el trabajo a realizar en distintas fases secuenciales (Figura 3-1), con el propósito de ir incrementando la funcionalidad del sistema en cada iteración, hasta llegar a la última donde se obtiene el sistema completo (Laboratorio Nacional de Calidad del Software de la INTENCO, 2009).

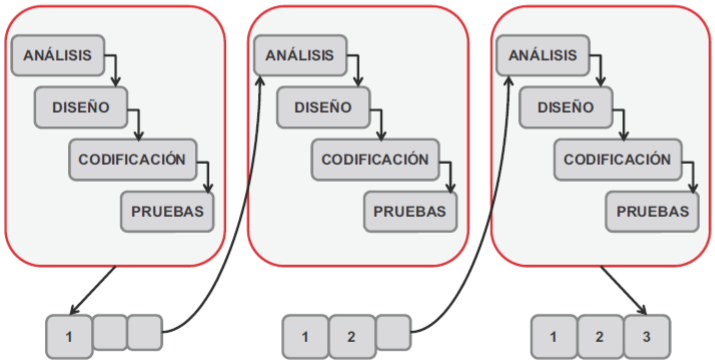

Figura 3-1. *Esquema de la metodología incremental.*

(Laboratorio Nacional de Calidad del Software de la INTENCO, 2009)

Las ventajas de este modelo sobre otros es que permite la creación de sistemas operativos rápidamente, los cuales se pueden poner a prueba constantemente. Las actividades de cada fase son las siguientes (Corona Nakamura & Ancona Valdez, 2011) (Joyanes Aguilar, 2008):

- **Análisis:** En esta etapa el problema se analiza teniendo presente la especificación de los requisitos dados por el cliente o por la persona que encarga el programa.

- **Diseño:** Esta etapa se centra en desarrollar el algoritmo basándonos en las especificaciones de la etapa del análisis

- **Codificación:** En esta etapa se transcribe el algoritmo definido en la etapa de diseño en un código reconocido por la computadora; es decir, en un lenguaje de programación; a éste se le conoce como código fuente.

- **Pruebas:** Esta etapa consiste en capturar datos hasta verificar que el programa funcione correctamente.

Las fases que se consideraron después de analizar el sistema completo son las siguientes:

- **Fase 1:** Creación del Sistema de Archivos.
- **Fase 2:** Conteo de Cubiertas de Aristas en Graficas sin Ciclos Intersectados.
- **Fase 3:** Proceso para Graficas con Ciclos Intersectados.
- **Fase 4:** Conteo Completo de Cubiertas y Creación de Estadísticas.

Estas fases corresponden a los módulos que forman el núcleo central del funcionamiento del algoritmo completo (Figura 3-2).

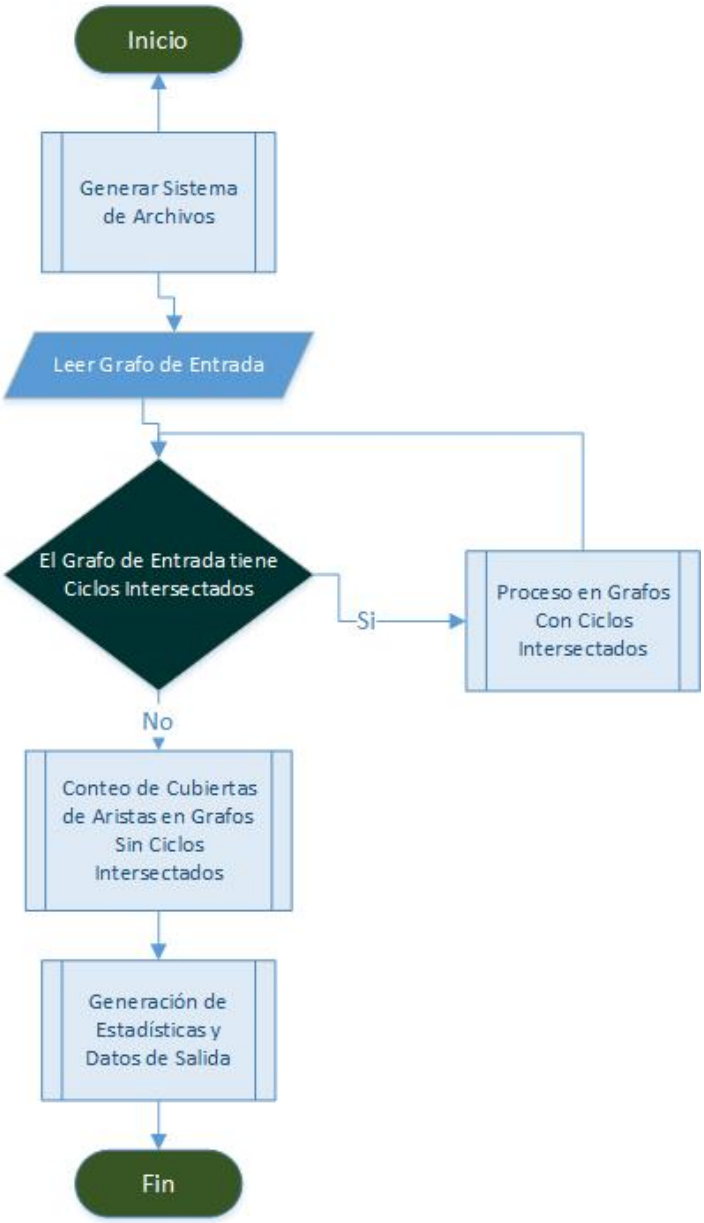

Figura 3-2. *Relación entre los módulos centrales del sistema.*

Antes de explicar cada una de las fases a continuación se enlistaran algunos aspectos técnicos de la implementación:

- Se utilizó el lenguaje de programación C/C++ por las ventajas de rendimiento que ofrece respecto a otros lenguajes de mayor nivel.

- Se utilizó la librería libre gmp (https://gmplib.org) para calcular el conteo completo de cubiertas de aristas, debido a que se superaba rápidamente los límites que tiene la representación de enteros en el lenguaje.

- La utilería Dot del paquete Graphviz se utilizó para la visualización de las gráficas en tiempo de ejecución, debido a su integración con el lenguaje C/C++ y todas las funcionalidades que tiene.

- El sistema recurre a llamadas al sistema para algunas operaciones, por lo que solo se puede ejecutar sobre un sistema Linux.

El código más importante para cada fase se puede consultar en los anexos.

3.2 Fase 1: Creación del Sistema de Archivos

3.2.1 Análisis

En esta primera fase se desarrolló un sistema de archivos que cumpliera los siguientes requerimientos:

- Facilitar el seguimiento de la ejecución del sistema, de forma que se pudieran observar las diferentes subdivisiones que tenía el grafo principal a lo largo de la ejecución del sistema.

- La gráfica de entrada debía manipularse y representarse de forma que su integración con la utilería Dot y el resto del sistema fuera sencilla.

3.2.2 Diseño

La estructura que utiliza el archivo Dot se muestra en la Figura 3-3, en ella se puede observar que la utilería funciona enlistando las aristas que van a ir uniendo una por una, por lo que se guarda la entrada como una lista de aristas, cada una

de ellas se guarda como un vector de dos posiciones, donde cada posición representa un vértice de la arista.

```
graph nombre_del_grafo {
    "idNodo1" -- "idNodo2";
    "idNodo1" -- "idNodo3"; /*estas son relaciones entre nodos*/
    "idNodo3" -- "idNodo1";
}
```

Figura 3-3. *Estructura de un archivo .dot.*

La forma en la que la entrada va a ser ingresada el sistema de archivos se realiza de acuerdo al algoritmo de la Figura 3-4.

Algoritmo 1: Procedimiento para generar el Sistema de Archivos

1: Entrada: Nada
2: Salida: Sistema de archivos sobre el que se ejecutara el algoritmo.
3: Obtener la ruta en el sistema de archivos de la computadora en donde se encuentra el código con una llamada al sistema.
4: Definir el directorio raíz “Cubiertas” en el directorio donde se encuentra el código.
5: Definir el directorio “IMAGENES” sobre el directorio raíz.
6: Definir el directorio “DOT” sobre el directorio raíz.
7: Definir el directorio “ARCHIVOS” sobre el directorio raíz.
8: Crear el archivo “Entrada” sobre el directorio “ARCHIVOS”.
9: Crear el archivo “Estadísticas” sobre el directorio “ARCHIVOS”.
10: Crear el archivo “Log” sobre el directorio “ARCHIVOS”.
11: Eliminar todos los archivos y carpetas del directorio “IMAGENES”.
12: Eliminar todos los archivos y carpetas del directorio “DOT”.
13: Leer archivo de Entrada.
14: **Para** cada línea del archivo
15: **Si** cumple formato de entrada **entonces**
16: **Si** no está repetida **entonces**
17: Agregar arista a la estructura
18: **Fin Si**
19: **Fin Si**
20: **Fin Para**
21: Guardar el archivo .dot del grafo de entrada en el directorio “DOT” como Original.dot
22: Guardar la imagen asociada al grafo de entrada en el directorio “IMAGENES” como Original.jpg

Figura 3-4. *Procedimiento para generar el sistema de archivos.*

El primer paso del algoritmo es obtener el directorio desde donde se ejecuta el código esto con el fin de generar el sistema de archivos a partir de ahí, logrando una independencia de la ejecución del programa respecto a la estructura de directorios de la máquina que lo ejecuta. Así el sistema se puede ejecutar directamente sobre cualquier sistema operativo Linux, que tenga bien configuradas las dependencias del programa.

El siguiente paso es crear un nuevo directorio raíz del sistema de archivos creado al mismo nivel que el código fuente, esto con el fin de lograr una jerarquía y orden al momento de observar la ejecución del algoritmo y analizar los resultados obtenidos. Este directorio raíz es nombrado como “Cubiertas”.

Para obtener los resultados y comprobar el funcionamiento del algoritmo es necesario mostrar el procedimiento interno de manera visual, por lo cual se crea el directorio “IMAGENES”, donde se almacenaran todas las subgráficas generadas por los distintos procesos que ocurren durante la ejecución del algoritmo.

Debido a que con la utilería DOT para cada grafo al que se le genera su imagen es necesario tener un archivo .dot asociado, se decidió que mantener todos los archivos .dot utilizados durante la ejecución era una mejor alternativa que ir borrándolos cada que se fueran utilizando, almacenándolos sobre el directorio “DOT”, ambos directorios comparten la misma estructura de archivos de esta forma los archivos .dot y las imágenes asociados se encuentran en los mismos niveles, lo cual ayuda a controlar la generación de las distintas gráficas y presentarlos de forma coherente.

La forma en cómo se van generando los niveles dentro de los directorios está relacionada directamente con las operaciones de la Fase 3, las cuales serán explicadas posteriormente. El último directorio que se genera dentro del directorio raíz “Coberturas”, es el directorio “Archivos” en el cual se crean los siguientes tres archivos:

- **Entrada:** De este archivo se lee el grafo de entrada, el cual es ingresado como un listado de aristas con un formato simple: (vértice, vértice).

- **Estadísticas:** En este archivo se guardaran todas las estadísticas generadas por el sistema y que se utilizaran para el análisis de resultados.

- **Log:** Este archivo guarda un registro de todas las aristas que se leyeron del archivo “Entrada” y que no cumplen con el formato establecido.

Los vértices de la gráfica solo pueden ser numéricos, encontrar un carácter que no sea digito en cualquiera de los dos vértices, hará que esa arista sea agregada al archivo “Log” y que se despliegue un mensaje de error. Esto se debe a que vértices con caracteres alfanuméricos complicarían la generación de nuevas aristas de las gráficas en tiempo de ejecución.

Finalmente, una vez que se realiza la lectura de la gráfica de entrada éste se convierte en un archivo .dot el cual se guarda en el directorio “DOT” y su imagen asociada en el directorio “IMAGENES” ésta va a ser la gráfica de entrada original y las operaciones subsecuentes se realizaran en ambos directorios para mantener sincronizado el sistema de archivos.

3.2.3 Pruebas

Las pruebas realizadas en esta etapa fueron más de carácter subjetivo. Tal cual se mencionó anteriormente, el objetivo de esta etapa es generar un sistema de archivos que facilite la comprobación paso a paso del algoritmo así como guardar esta entrada de una forma que facilitara la integración de la utilería Dot.

La lectura del archivo y su posterior conversión a un archivo .dot para finalmente generar una imagen es sencilla de realizar al guardar la entrada como una lista de aristas, ya que el proceso básicamente consiste en recorrer la lista y agregar cada elemento al archivo .dot sin pasar por un proceso de transformación complejo. A través de llamadas al sistema las imágenes se generan rápidamente y manteniendo la sincronización del sistema de archivos el resultado es satisfactorio.

La Figura 3-5 muestra que el uso de la utilería Dot genera una imagen para el grafo procesado clara (siempre y cuando el número de vértices no sea excesivo) sin necesidad de profundizar en las distintas opciones de personalización que ofrece la herramienta.

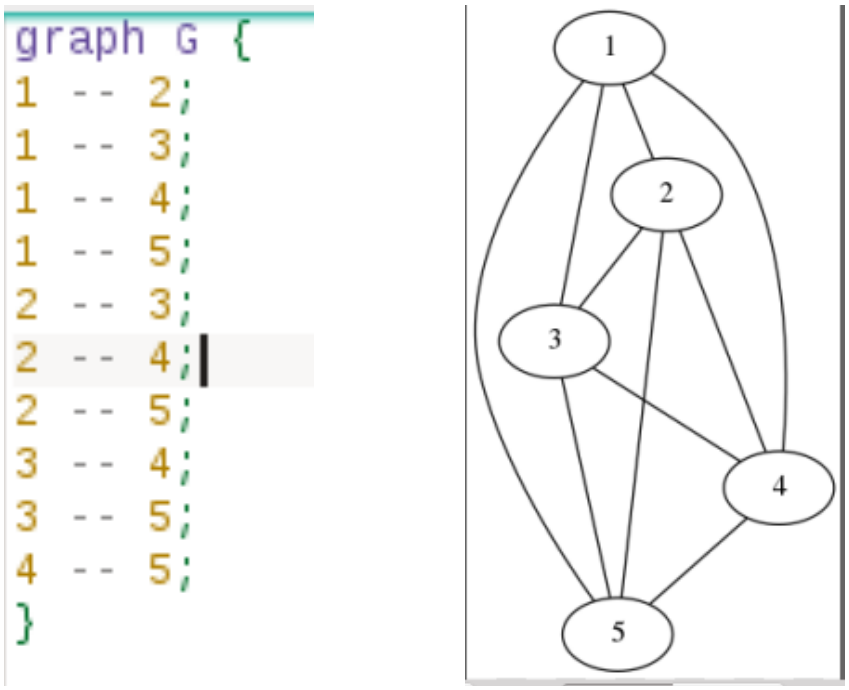

Figura 3-5. *Archivo .dot y su imagen correspondiente.*

La estructura obtenida al finalizar la ejecución del sistema es semejante a la que se muestra en la Figura 3-6, en ella se puede apreciar claramente las operaciones que el sistema ejecuto así como las distintas subgráficas creadas, lo cual permite seguir la ejecución del algoritmo fácilmente.

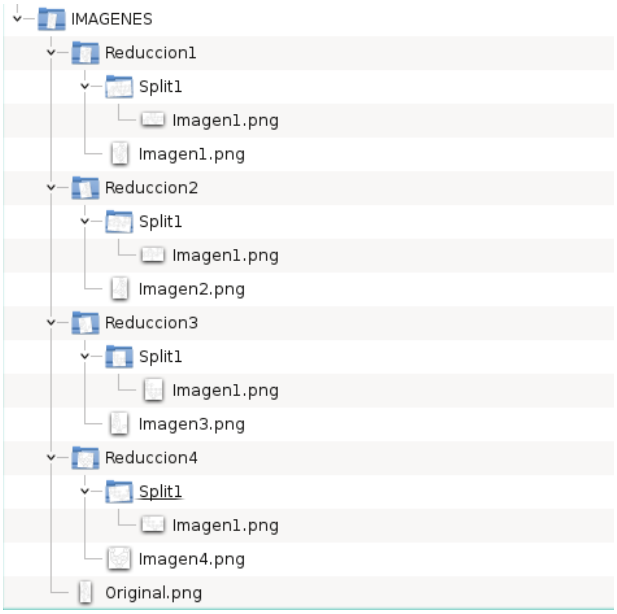

Figura 3-6. *Estructura del sistema de archivos del sistema.*

3.3 Fase 2: Conteo de cubiertas de aristas en graficas sin ciclos intersectados

3.3.1 Análisis

Para realizar el conteo de cubiertas de aristas en graficas sin ciclos intersectados se implementó el algoritmo mencionado en (Hernández Servín, et al., 2014) el cual se describe a continuación:

Sea G una gráfica sin ciclos intersectados, se obtiene a partir de ella una gráfica directa obtenida de la realización de una búsqueda en profundidad T_G la cual se construye de acuerdo a los siguientes pasos:

1. Construir una gráfica como resultado de una búsqueda en profundidad G' de G (recordar que las aristas de T_G son aristas árbol o de retroceso).

2. Para cada arista árbol $e = (u, v) \in G'$, agregar la arista direccionada $e = u \rightarrow v$ a T_G si u es un hijo de v en G'.

3. Para cada arista de retroceso $e = (u, v) \in G'$, agregar la arista direccionada $e = u \rightarrow v$ a T_G si u es un descendiente de v en G'.

Un ejemplo de gráfica dirigida se muestra en la Figura 3-7, en ella las aristas árbol que forman la estructura central de la gráfica están marcadas como líneas

continuas, mientras que las aristas de retroceso se representan con líneas discontinuas.

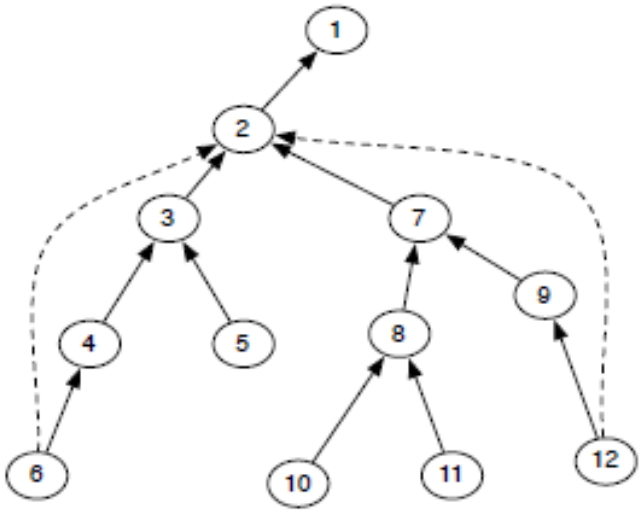

Figura 3-7. *Gráfica dirigida obtenida de realizar una búsqueda en profundidad.* (Hernández Servín, et al., 2014)

Sea T_G la gráfica obtenida de realizar una búsqueda en profundidad y $v \in V(T_G)$ se define como:

$$input(u) = |\{v | v \rightarrow u \in E(T_G)\}|$$

$$output(u) = |\{v | u \rightarrow v \in E(T_G)\}|$$

Sea G la gráfica obtenida de realizar una búsqueda en profundidad, y $e = u \rightarrow v \in E(G)$. Se asocia una tupla (α_e, β_e) a e tal que α_e representa el número de coberturas de aristas de G donde e aparece en las cubiertas por u. El número β_e representa el número de coberturas de aristas de G donde e no aparece en las cubiertas por u.

La Figura 3-8 muestra los pasos del algoritmo para realizar el conteo de cubiertas de aristas en gráficas sin ciclos intersectados mencionado en (Hernández Servín, et al., 2014).

Algoritmo 2: Calculo del número de coberturas de aristas en una gráfica G sin ciclos intersectados.

1: Entrada: T_G: una gráfica generada de la aplicación de una búsqueda en
2: profundidad en G, que no contenga ciclos intersectados.
3: Salida: $NE(G)$: El número de coberturas de aristas de G.
4: Recorrer los nodos de G desde las hojas hasta la raíz.
5: Para cada arista árbol $e_i = u \rightarrow v$ calcular un par $(\alpha_{e_i}, \beta_{e_i})$ {es decir las aristas
6: de retroceso no tiene par}.

7: **switch** $(e_i = u \rightarrow v)$
8: **case** u es un nodo hoja y $output(v)$ == 1:
9: $(\alpha_{e_i}, \beta_{e_i})$ = (1,0); {no hay aristas de retroceso desde r}
10: break;
11: **case** u es un nodo hoja y $output(v)$ == 2:
12: $(\alpha_{e_i}, \beta_{e_i})$ = (1,1); {hay una arista de retroceso desde r}
13: break;
14: **default:**
15: **Si** $e_j = x \rightarrow v$ es una arista de retroceso y $path(e_i, e_j)$ es un ciclo **entonces**
16: $(\alpha_j, \beta_j) = (\alpha_j + \beta_j,\ \alpha_j + 1)$
17: **Fin si**
18: Sea $A = \{e_1, e_2, \ldots, e_j\}$ el conjunto de aristas tales que $e_k = x \rightarrow u$, donde
19: $1 \leq k \leq j$, para cualquier x
20: **Si** $output(v) > 0$ y cada par $(\alpha_{e_j}, \beta_{e_j})$ para cada arista de A ha sido
21: computada **entonces**
22: $(\alpha_{e_i}, \beta_{e_i}) = \left(\prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}), \prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}) - \prod_{r=1}^{j}\beta_{e_r}\right)$; {s no es el nodo
23: raíz}
24: **Fin si**
25: **Si** $output(v) == 0$ y cada par $(\alpha_{e_j}, \beta_{e_j})$ para cada arista de A ha sido
26: computada **entonces**
27: $(\alpha_i, \beta_i) = \left(\prod_{r=1}^{j}(\alpha_r, \beta_r) - \prod_{r=1}^{j}\beta_r, \prod_{r=1}^{j}\beta_r\right)$; {s es el nodo raíz}
28: **regresa** α_i;
29: **Fin si**
30: **Fin switch**

Figura 3-8. *Procedimiento para calcular coberturas de aristas para G sin ciclos intersectados.*

(Hernández Servín, et al., 2014)

El algoritmo funciona de la siguiente manera:

1. Convertir la gráfica no dirigida en dirigida utilizando el algoritmo DFS para generar su árbol de expansión y obtener su complemento.

2. Asigna a cada nodo hoja de la gráfica un par (α, β) igual a (1,0) si el nodo hoja no tiene aristas de retroceso (el nodo hoja aparece una vez en una cobertura y si se quita no hay coberturas), y (1,1) si el nodo hoja sí tiene una arista de retroceso (el nodo hoja aparece una vez en una cobertura y si se quita también se tiene una cobertura) (línea 7 y 10 de la Figura 3-9).

3. A cada nodo interno de la gráfica que se va encontrando en el recorrido, se le asigna su propio par (α, β) con base a los pares de los nodos hijos a él, usando la fórmula $(\alpha_{e_i}, \beta_{e_i}) = \left(\prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}), \prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}) - \prod_{r=1}^{j}\beta_{e_r}\right)$ (línea 18 de la Figura 3-9).

4. Cuando se encuentre un vértice que sea el fin o la unión de una arista de retroceso con el árbol después de calcular su par (α, β) con la fórmula del paso 3, se utiliza la formula $(\alpha_j, \beta_j) = (\alpha_j + \beta_j,\ \alpha_j + 1)$ para actualizar el par (α, β) de ese vértice (línea 13 de la Figura 3-9).

5. Cuando se llega al nodo raíz para asignarle su respectivo par (α, β) se utiliza la fórmula $(\alpha_i, \beta_i) = \left(\prod_{r=1}^{j}(\alpha_r, \beta_r) - \prod_{r=1}^{j}\beta_r\ , \prod_{r=1}^{j}\beta_r\right)$.

6. Finalmente el número de coberturas de aristas totales de la gráfica es representada por el valor α del par asignado a la raíz.

Un ejemplo de la ejecución del algoritmo para un determinado grafo se muestra en la Figura 3-9.

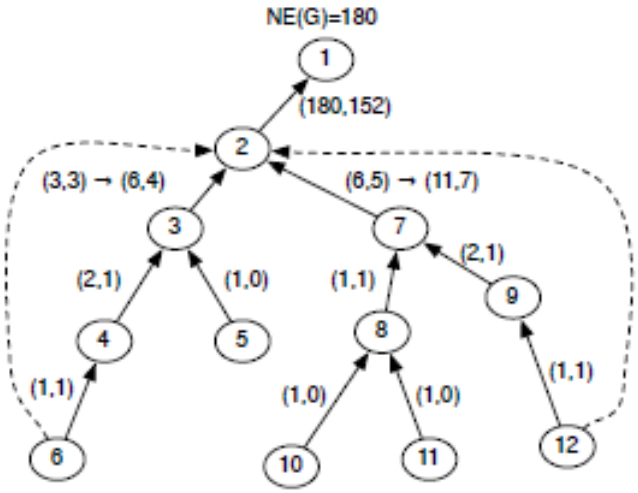

Figura 3-9. *Calculo del número de coberturas de aristas (NG) en graficas sin ciclos intersectados.*

(Hernández Servín, et al., 2014)

3.3.2 Diseño

Una vez definida la forma de manejar los datos de entrada del sistema y el funcionamiento del algoritmo, lo siguiente es obtener el complemento de la gráfica haciendo uso de un árbol de expansión generado por el algoritmo *Depth First Search* (DFS) (Figura 3-10).

Algoritmo 3: Obtención del Complemento de una Gráfica utilizando un árbol de expansión DFS.

1: Entrada: T_G: Una grafica G no dirigida que contiene ciclos intersectados.
2: Salida: El árbol de expansión y el complemento de T_G.
3: **Para** cada arista de la grafica T_G
4: **Para** cada vértice de la arista
5: **Si** vértice está en arreglo_grado **entonces**
6: Aumentar su valor en el arreglo en 1
7: **Otro**
8: Agregar la arista al arreglo con el valor de 1
9: **Fin Si**
10: **Fin Para**
11: **Fin Para**
12: grado_mayor=primer elemento del arreglo_grado
13: **Para** cada elemento del arreglo_grado
14: **Si** elemento del arreglo_grado > grado_mayor **entonces**
15: grado_mayor = elemento del arreglo_grado
16: **Fin Si**
17: **Fin Para**
18: **Para** cada elemento del arreglo_grado
19: Agregar elemento del arreglo_grado a guia
20: **Fin Para**
21: Obtener el vértice con grado igual a grado_mayor
22: Marcar el vértice como visitado en la guía
23: **Si** guía está completamente visitada **entonces**
24: Ir al paso 38
25: **Otro**
26: **Para** cada elemento de la grafica
27: **Si** alguna arista tiene el vértice seleccionado **entonces**
28: Agregar el otro vértice de la arista al vector adyacente
29: **Fin Si**
30: **Fin Para**
31: **Para** cada vértice adyacente
32: **Si** el vértice no se encuentra en la guía **entonces**
33: Agregar la arista al árbol_expansion
34: Repetir desde el paso 22 con la nueva arista
35: **Fin Si**
36: **Fin Para**
37: **Fin Si**
38: **Para** cada arista de la grafica
39: **Si** arista_grafica no está en árbol_expansion **entonces**
40: Agregar arista_grafica al complemento
41: **Fin Si**
42: **Fin Para**

Figura 3-10. *Algoritmo para generar el complemento utilizando un árbol de expansión DFS.*

La generación del árbol de expansión es fundamental para la ejecución del algoritmo ya que sirve de base para realizar las operaciones tanto de esta fase como de la próxima, las cuales juntas forman el “core” o núcleo del funcionamiento del algoritmo.

El primer paso del algoritmo consiste en obtener el grado de todos los vértices de la gráfica, esto es, el número de aristas diferentes en las que aparece el vértice, esto se hace simplemente recorriendo las aristas de la gráfica en orden e ir contando cuantas veces aparece cada vértice en ese recorrido.

Después se selecciona el vértice cuyo grado sea mayor, no hay mucho que agregar en cuanto al cómo se realiza este y el paso anterior, ya que es algo trivial, lo importante que hay que mencionar es el por qué se realizan estas operaciones, esto es debido a que después de realizar un análisis, se llegó a la conclusión de que aunque no es mencionado en el artículo de referencia, si se selecciona como raíz del árbol de expansión el vértice con mayor grado, entonces, la posibilidad de romper más ciclos intersectados al iniciar la ejecución del sistema era mayor que si solo se seleccionaba una arista al azar.

Para obtener el árbol de expansión una vez que se obtiene el vértice raíz se obtienen todos los vértices adyacentes o vecinos al mismo y se van recorriendo, cada nuevo vértice por el que se pasa se marca como visitado utilizando la guía, una vez que todos los vértices han sido recorridos, esto se comprueba con la guía, las aristas por las que se pasó en el momento forman el árbol de expansión de la gráfica.

Pese a que mantener un registro de todos los vértices adyacentes implica comprobar caminos dentro de la gráfica que ya han sido visitados previamente, es necesario de realizar esta comprobación, debido a que en ocasiones para poder alcanzar todos los vértices de la gráfica será necesario regresar varios de los niveles del árbol de expansión para probar otras aristas y al fin de cuentas, se sigue manteniendo el tiempo de ejecución polinómico.

Finalmente, se obtiene el complemento seleccionado las aristas que aparezcan en la gráfica pero que no estén en el árbol de expansión, estas aristas formaran lo que se conoce como aristas de retroceso las cuales son indispensables de encontrar para poder realizar el conteo de cubiertas (Figura 3-11).

Algoritmo 4: Conteo de cubiertas de aristas en una gráfica sin ciclos intersectados

1: Entrada: Árbol de expansión y complemento de la gráfica G.
2: Salida: $NE(G)$: El número de coberturas de aristas de G.
3: **Para** cada arista del árbol de expansión
4: Agregar una arista vacía al arreglo CUBIERTAS
5: **Para** cada vértice del arreglo Subnivel
6: **Si** vértice = primer vértice de la arista **entonces**
7: Aumentar el número de subniveles del vértice en 1
8: **Otro**
9: Agregar al arreglo Subnivel el vértice con el valor de 1
10: **Fin Si**
11: **Fin Para**
12: **Fin Para**
13**: Para** cada vértice del arreglo Subnivel

Si número de subniveles <= 1 **entonces**
  Eliminar ese vértice del arreglo Subnivel
Fin Si
Fin Para
Para cada vértice del Árbol de Expansión {Recorrido de atrás hacia adelante}
Si vértice = nodo hoja **entonces**
  Si vértice = inicio arista_retroceso **entonces**
    Asignar el par $(\alpha, \beta) = (1,1)$ al vértice
    Indicar inicio de arista de retroceso en el complemento
  Otro
    Asignar el par $(\alpha, \beta) = (1,0)$ al vértice
  Fin Si
Otro
  Si vértice es un subnivel **entonces**
    Si subnivel está completo **entonces**
      Calcular el par (α, β) del vértice usando $(\alpha_{e_i}, \beta_{e_i}) = (\prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}), \prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}) - \prod_{r=1}^{j}\beta_{e_r})$ sobre todos los hijos del vértice
      Si vértice es parte de una arista de retroceso **entonces**
        Indicar que es arista de retroceso
      Fin Si
      Si vértice = inicio arista_retroceso **entonces**
        Indicar que es arista de retroceso
        Indicar inicio de arista de retroceso en el complemento
      Fin Si
      Si vértice = final arista_retroceso **entonces**
        Ajustar el par (α, β) del vértice usando $(\alpha_j, \beta_j) = (\alpha_j + \beta_j,\ \alpha_j + 1)$
        Indicar en el complemento que se recorrió la arista de retroceso
      Fin Si
      Eliminar los subniveles que ya se complementaron
      Eliminar los vértices ya visitados
      Eliminar los valores (α, β) de los vértices visitados
    Otro
      Calcular el par (α, β) del vértice usando $(\alpha_{e_i}, \beta_{e_i}) = (\prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}), \prod_{r=1}^{j}(\alpha_{e_r}, \beta_{e_r}) - \prod_{r=1}^{j}\beta_{e_r})$ sobre el vértice anterior
      Si vértice es parte de una arista de retroceso **entonces**
        Indicar que es arista de retroceso
      Fin Si
      Si vértice = inicio arista_retroceso **entonces**
        Indicar que es arista de retroceso
        Indicar inicio de arista de retroceso en el complemento
      Fin Si
      Si vértice = final arista_retroceso **entonces**
        Ajustar el par (α, β) del vértice usando $(\alpha_j, \beta_j) = (\alpha_j + \beta_j,\ \alpha_j + 1)$
        Indicar en el complemento que se recorrió la arista de retroceso
      Fin Si
      Eliminar los vértices ya visitados
      Eliminar los valores (α, β) de los vértices visitados
  Fin Si
Fin Si
Fin Para

49: Ajustar el par (α, β) de la raíz usando $(\alpha_i, \beta_i) = \left(\prod_{r=1}^{j}(\alpha_r, \beta_r) - \prod_{r=1}^{j}\beta_r, \prod_{r=1}^{j}\beta_r\right)$
50: Regresar el valor α de la raíz

Figura 3-11. *Algoritmo para obtener el conteo de cubiertas de aristas.*

Este procedimiento se debe realizar en todos las gráficas que se vayan generando y recibe el árbol de expansión y su correspondiente complemento obtenidos con el Algoritmo 3.

Un punto a resaltar antes de profundizar la explicación de la ejecución del procedimiento, es la técnica utilizada y el modo de recorrer los vértices del árbol, originalmente se utilizó un *backtracking* recorriendo el nodo de la raíz hasta las hojas y de ahí moverse hacia atrás para ir calculando los valores α y β de cada vértice para que al momento de procesar todos los vértices se obtuviera el resultado esperado.

Sin embargo, este método demostró ser ineficiente debido a que se necesitaba ir suponiendo cuáles vértices tenían más de un hijo y cuáles no, otro problema era que no se podían procesar los nodos hasta alcanzar los nodos hojas, al mismo tiempo que el número de casos particulares que iban apareciendo a medida que se realizaban las pruebas, originó un cambio de estrategia.

En la ejecución del Algoritmo 4 lo primero es encontrar los subniveles (vértices que tengan más de un nodo hijo) que tiene el árbol de expansión, por lo tanto el arreglo denominado “Subnivel” se utiliza para guardar la relación entre estos vértices y el número de nodos hijos que les corresponde, aunque solo importan aquellos vértices que tengan más de un vértice hijo.

Después es necesario asignarle un par (α, β) a cada vértice del árbol, para ello el arreglo “CUBIERTAS” es el encargado de asignar y relacionar a cada vértice del árbol con su respectivo par. Con estos dos arreglos auxiliares se puede proceder a la realización del algoritmo, eliminando uno de sus problemas: la falta de información de que vértices tenían más de un hijo. Juntando esta información con un recorrido de las hojas a la raíz, se obtuvo una mejor implementación de lo que en algoritmo se le llama hacer dirigida la gráfica no dirigida.

Esto se debe a que en un solo recorrido del árbol se puede obtener el número de coberturas, siempre y cuando el árbol se lea de la raíz a las hojas, y esto es justamente lo que se obtiene por la aplicación del DFS, por lo que no es necesario ningún procedimiento extra.

Durante el recorrido de los vértices del árbol (de atrás hacia delante), lo primero que hay que comprobar es si el vértice es un nodo hoja. Verificar esto en el primer nodo hoja encontrado es redundante ya que al ser precisamente el último valor en ingresar al árbol de expansión claramente es un nodo hoja, para hacer la verificación en los demás casos se utiliza el arreglo de subniveles.

La forma en que funciona esta identificación de los nodos hoja es bastante sencilla cuando se va recorriendo una rama del árbol desde la hoja hasta la raíz, cada vértice que se encuentra en el camino es procesado antes de pasar al siguiente, pero estos no se eliminan del arreglo completamente hasta que todas las ramas de un subnivel son computadas. De esta forma, si un determinado nodo no es subnivel pero el que lo sigue si lo es, significa que ese vértice es un nodo hoja de otra rama.

Si el vértice es un nodo hoja, se le asigna el par (α, β) igual a (1,1) o (1,0). Para esto, hay que comprobar si ese nodo hoja pertenece o no a una arista de retroceso recorriendo el complemento del árbol de expansión, y si el vértice se encuentra en él entonces significa que es el inicio de una arista de retroceso y se le asigna el valor de (1,1), en caso contrario se le asigna (1,0).

En cambio, si el vértice no es un nodo hoja se calcula su par (α, β) aplicando la fórmula de la línea 29 del algoritmo 4 sobre todos los nodos hijos que tenga ese vértice. Si es parte de la rama de algún subnivel solo se toma su vértice inmediato superior, en cambio sí es un subnivel como tal si no se han procesado todos sus nodos hijo se pasa a la siguiente rama empezando por un nuevo nodo hoja, si ya han sido procesadas todos sus vértices hijo, entonces se procesa el subnivel aplicando la formula sobre los valores de todos sus hijos.

Una vez procesado los vértices se comprueba si es el inicio, fin o parte de una arista de retroceso para hacer los ajustes correspondientes. Si es el inicio o parte de la aristas de retroceso solo se debe indicar en el complemento que esa rama del árbol es parte de una arista de retroceso, del mismo modo cuando se llega al final de la arista de retroceso, esto se debe reflejar en el complemento y al mismo tiempo se corrige el valor del par asignado al vértice usando la fórmula del paso 16 del Algoritmo 3.

Al final de cada iteración del ciclo las cubiertas y aristas que ya han sido procesadas son eliminadas de sus respectivos arreglos dejando solamente aquellas que siguen siendo necesarias para las siguientes iteraciones. De este modo al finalizar el ciclo solo quedaran los valores que corresponden a la raíz, así para obtener el valor de su par (α, β) se aplica la fórmula del paso 27 del Algoritmo 3 sobre los elementos que queden en el arreglo CUBIERTAS.

Finalmente, el número de coberturas de aristas de la gráfica de entrada corresponden al valor α de su par asignado y ese valor es el que se regresa.

3.3.3 Pruebas

Para comprobar el correcto funcionamiento de la implementación de este algoritmo se muestran diferentes ejecuciones con graficas de entrada diferentes, empezando por el ejemplo que se muestra en (Hernández Servín, et al., 2014). La Figura 3-12 muestra el resultado de la creación del árbol de expansión y del complemento sobre el ejemplo de referencia.

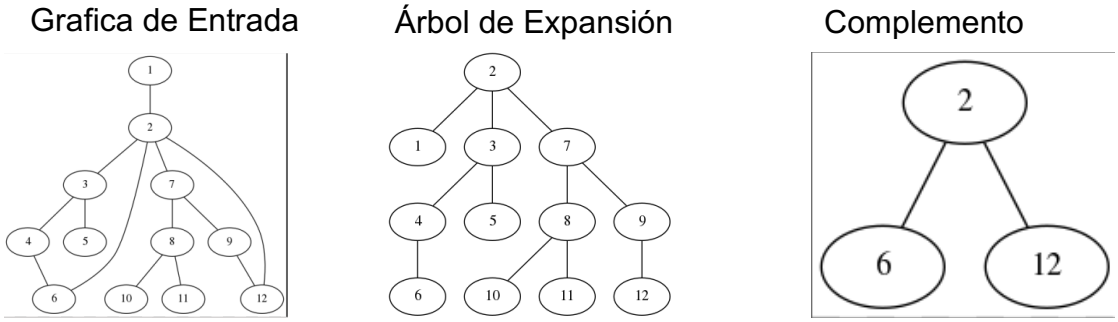

Figura 3-12. *Aplicación del algoritmo para generar el árbol de expansión sobre el ejemplo de referencia.*

El árbol de expansión creado por el sistema comienza por el vértice que tiene el mayor número de coberturas de aristas (en este ejemplo el vértice 2) y las aristas de retroceso están presentes en el complemento, es importante hacer la aclaración de que esta implementación tiene una diferencia respecto al algoritmo presentado en (Hernández Servín, et al., 2014), esto se debe a que en la referencia el vértice que se selecciona como raíz del árbol se escoge aleatoriamente en cambio, en cambio este sistema toma como raíz el vértice que pertenezca al mayor número de ciclos con el fin de romper más ciclos intersectados desde el inicio, esta diferencia no afecta el procesamiento del algoritmo, debido a que diferentes ejecuciones del algoritmo con el mismo ejemplo de entrada genera siempre el mismo resultado, aunque se tomen diferentes raíces del árbol, para el ejemplo presentando, 180 coberturas de aristas.

Sin embargo, esto no es suficiente para corroborar que el algoritmo funcione en todos los casos. Ejecutando el algoritmo con una gráfica no tenga ciclos intersectados se obtiene el árbol de expansión de la Figura 3-13.

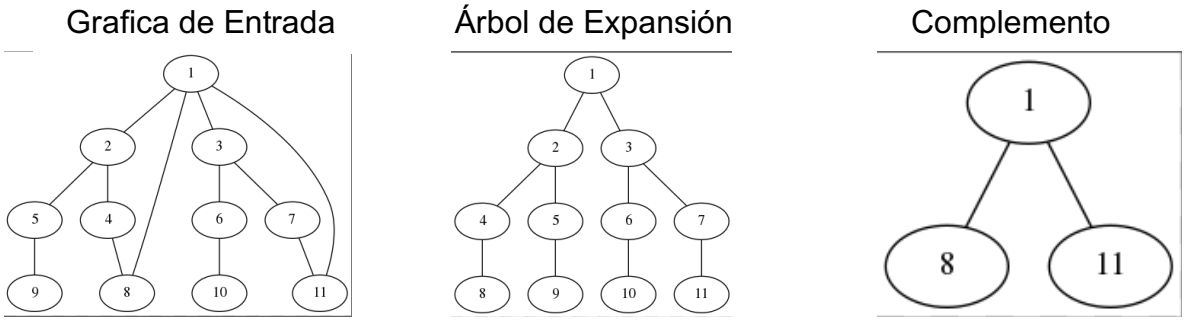

Figura 3-13. *Ejemplo No.2 para generar el árbol de expansión.*

Así mismo la ejecución del algoritmo sobre este ejemplo obtuvo un total de 275 coberturas de aristas, este coincide con el valor esperado, con lo cual se puede decir que el algoritmo funciona adecuadamente y se puede proceder con la siguiente fase del sistema.

3.4 Fase 3: Proceso para Graficas con Ciclos Intersectados

3.4.1 Análisis

Esta fase es el proceso central de todo el sistema ya que une todas las demás fases, involucra las diferentes reglas y principios del algoritmo propuesto en (Hernández Servín, et al., 2014), el cual se explica a continuación (Figura 3-14):

Algoritmo 5: $NE\_General(T_G)$. Procedimiento para descomponer una gráfica G que tenga ciclos intersectados.

1: Entrada: $H = (V, E)$: la gráfica producida por el algoritmo de búsqueda en profundidad, $n = |V(H)|$, $m = |E(H)|$ y $nc = m - n - 1$.
2: Salida: $NE(G)$: El número de coberturas de aristas de G.
3: Seleccionar una arista $e = \{u, v\} \in E(H)$ de acuerdo a los criterios establecidos en (Hernández Servín, et al., 2014) {la arista e puede ser encontrada en $O(nm \log m)$}
4: Aplicar la regla de división sobre e generando H_1 y H_2.
5: **Si** H_1 tiene ciclos intersectados **entonces**
6: $NE\_General(H_1)$
7: **Fin si**
8: **Si** H_2 tiene ciclos intersectados **entonces**
9: $NE\_General(H_2)$
10: **Fin si**
11: $NE(G) = \sum_{G' \in H(A_G)} NE(G')$ donde $H(A_G) = \{ G_h : G_h$ es el grafo asociado al nodo hoja $A(T_G)\}$.

Figura 3-14. *Procedimiento para realizar* $NE\_General(T_G)$.(Hernández Servín, et al., 2014)

El algoritmo funciona calculando $NE(G)$ a partir de la construcción de un árbol de enumeración sobre T_G, denotado como $A(T_G)$. Cada nodo de $A(T_G)$ tiene una gráfica asociada. El nodo raíz de $A(T_G)$ es la gráfica inicial. Los nodos internos de $A(T_G)$ corresponden a graficas no básicas (con ciclos intersectados), mientras que los nodos hoja de $A(T_G)$ solo corresponden a graficas básicas (sin ciclos intersectados).

Por cada nodo interno de $A(T_G)$ se escoge una arista $e \in E(H)$ para dividir el problema de calcular $NE(H)$. La arista e debe ser parte de un ciclo intersectado entonces se aplica la regla de división sobre e, contando separadamente el número de coberturas de aristas que contienen a e y aquellas que no.

El resultado de aplicar recursivamente la regla de división en la gráfica G, siguiendo los pasos previamente mencionados, realizará el cálculo de $NE(H)$. Un ejemplo de esta primera parte se muestra en la Figura 3-15.

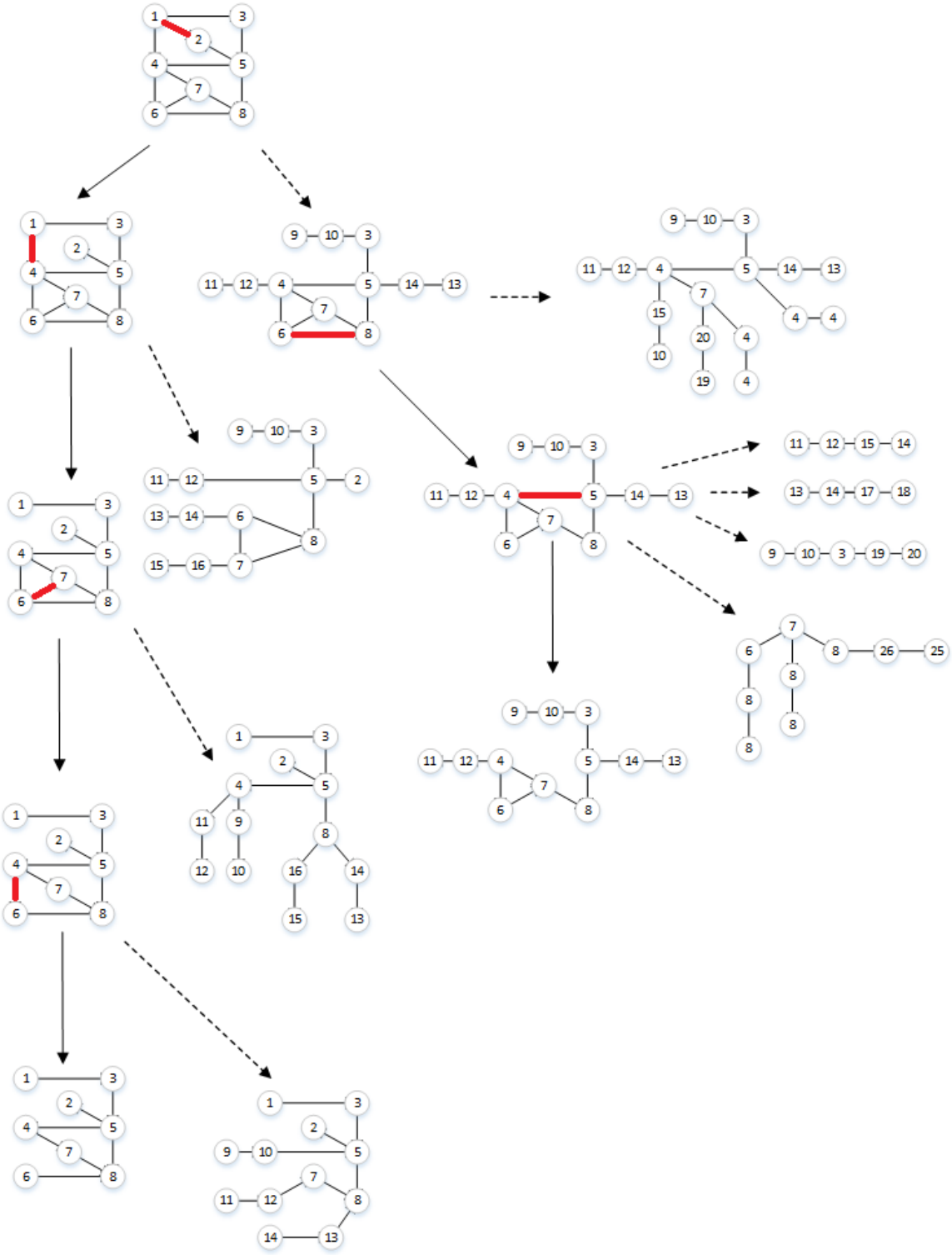

Figura 3-15. *Ejemplo de aplicación del algoritmo en graficas con ciclos.*

Sea $G = (V, E)$ la gráfica con ciclos intersectados mostrada en la parte superior de la Figura 3-12. Aplicando el procedimiento $NE\_General(G)$, se obtiene un árbol enumerativo $A(G)$. La regla de división es aplicada seis veces sobre los nodos interiores del árbol, la arista escogida se muestra de color rojo.

Las líneas negras continuas, indican las gráficas H_1 y las líneas negras discontinuas, indican las gráficas H_2, que fueron resultado de la aplicación de la regla de división.

Los nodos hojas del árbol son gráficas con ciclos no interceptados, entonces su número de coberturas de aristas puede ser calculado directamente. La suma de las coberturas de aristas de cada hoja de los árboles es igual al número de coberturas para la gráfica G en general.

Si una hoja de $A(G)$ consiste de un grupo de gráficas cíclicas desconectadas no intersectadas, este conjunto se conoce como bosque (hijo derecho de $A(G)$ en la Figura 3-3). De este modo, el número de coberturas de aristas para ese nodo hoja en particular es calculado por el producto de las coberturas de aristas individuales de cada grafica que pertenece al bosque.

Una vez que se ha reducido la gráfica de entrada G con ciclos intersectados, en un conjunto de gráficas de salida sin ciclos intersectados, se necesita calcular el conjunto de coberturas de aristas para cada uno de ellos, para después sumar o multiplicar de acuerdo a su posición en el árbol, el proceso para realizarlo se describe a continuación

3.4.2 Diseño

Esta parte del diseño consiste en unir los diferentes módulos o subsistemas que hasta el momento se han creado con la regla de división propuesta en (Hernández Servín, et al., 2014), la cual fue explicada en la sección 2.4:

- Generar una nueva grafica quitando una arista de un ciclo intersectado.

- Generar una nueva grafica quitando la misma arista del ciclo intersectado y agregar dos nuevos vértices a cada vértice adyacente a algún extremo de la arista eliminada.

Estas reglas son denominadas como regla de reducción y de expansión respectivamente, la implementación de cada una de ellas así como su relación con el resto de las fases implementadas también se verá reflejada en la descripción de cada una.

La regla de reducción se muestra en el Algoritmo 6 de la Figura 3-16. El contador que se observa como entrada del algoritmo tiene como objetivo enumerar las diferentes graficas generadas, con el fin de poder graficarlas y agregarlas al sistema de archivos generado.

Algoritmo 6: Procedimiento para aplicar la regla de reducción.

1: Entrada: Gráfica G, Árbol de Expansión, Complemento, Contador
2: Salida: El Árbol de Expansión después de aplicar la regla de reducción.
3: **Para** cada arista del complemento
4: **Para** cada arista del árbol de expansión
5: **Si** vértice es parte de la arista de retroceso **entonces**
6: Agregar al vector de recorrido de la arista del complemento
7: **Fin Si**
8: Agregar el vector de recorrido al vector de ciclos
9: **Fin Para**
10: **Fin Para**
11: **Si** Tamaño_Complemento > 1 **y** Hay al menos una arista que se repita en el vector de ciclos más de una vez **entonces**
12: Recuperar la arista que aparezca más veces en el vector de ciclos
13**:** Eliminar la arista de la grafica
14: Graficar la nueva grafica generada
15: Obtener el árbol de expansión y complemento de la nueva grafica
16: Repetir desde el paso 3 sobre la nueva grafica
17: **Otro**
18: Aplicar algoritmo 5 sobre la grafica
19: **Fin Si**
20: Sumar el número de coberturas de aristas de la gráfica al conteo total

Figura 3-16. *Algoritmo de la regla de división.*

Una vez obtenido el árbol de expansión y el complemento de la gráfica de entrada, se procede a generar un vector de ciclos, es decir un vector que contenga un contador de todas las aristas que pertenecen a un ciclo. Para esto se itera sobre el complemento, porque ahí están los extremos de los ciclos y se recorre el árbol de expansión agregando cada arista que se encuentre en ese recorrido.

Para saber si una arista pertenece o no a un ciclo hay que llegar desde la hoja hasta el extremo final de las aristas que se encuentran en el complemento. Esto se realiza de manera sencilla ya que el árbol está ordenado de manera secuencial y simplemente es seguir las aristas que se van encontrando en el camino y comprobar si alguno de sus extremos coincide con el extremo final de la arista que se va recorriendo.

Una vez que se tiene el vector de ciclos hay que verificar si la gráfica de entrada tiene ciclos intersectados, para aplicar el procedimiento correspondiente o bien si no los tiene aplicar el algoritmo de conteo directamente, esto se hace a través del complemento y el vector de ciclos. Para que haya un ciclo intersectado debe haber

al menos más de una arista en el complemento y al mismo tiempo alguna arista dentro del vector de ciclos debe aparecer más de una vez en el mismo, eso indicaría que esa arista pertenece a más de dos ciclos y por lo tanto indica la presencia de ciclos intersectados.

Comprobada la presencia de al menos un ciclo intersectado en la gráfica usando el vector de ciclos se obtiene la arista que aparece en más ciclos con el objeto de eliminar la mayor cantidad de ciclos intersectados posibles en cada paso y, en consecuencia, reducir el tiempo de ejecución y la cantidad de memoria utilizada por el algoritmo.

Una vez encontrada la arista que participa en más ciclos intersectados de la gráfica de entrada (si se da el caso de que haya más de una arista que aparezca el mismo número de ciclos se escoge al azar entre ellas), ésta es eliminada de la gráfica generando una nueva grafica igual a la original, solo que con una arista menos. Esta nueva grafica se agrega al sistema de archivos generando su respectivo archivo .dot y su grafica correspondiente, al mismo tiempo que pasa de nuevo por todo el proceso generando del árbol de expansión y aplicando la regla de reducción, hasta que la nueva grafica generada no contenga ciclos intersectados.

Una vez que se llega a este punto, la última grafica generada por la regla de división, es decir la que ya no presenta ciclos intersectados, es enviada al algoritmo de conteo para contar las cubiertas de esa gráfica y es sumada al conteo total de cubiertas de aristas de la gráfica de entrada.

Se menciona que es sumada al conteo total de cubiertas de aristas, debido a que la siguiente regla de división, también genera graficas que se mandan al algoritmo de conteo, así que al final el número toral de cubiertas de aristas de la gráfica de entrada será igual a la suma de las coberturas obtenidas por ambas reglas.

La regla de división se debe aplicar al mismo tiempo que la regla de reducción sobre la gráfica de entrada, es por eso que las entradas y primeros pasos del Algoritmo 7 de la Figura 3-17 corresponden a las del Algoritmo 6 incluida la variable contador, es decir solo sirve de ayuda para agregar las gráficas generadas por la regla de división al sistema de archivos y no repercute directamente sobre el funcionamiento del mismo.

Algoritmo 7: Procedimiento para aplicar la regla de expansión.

```
1: Entrada: Gráfica G, Árbol de Expansión, Complemento, Contador
2: Salida: El Árbol de Expansión después de aplicar la regla de reducción.
3: Para cada arista del complemento
4:     Para cada arista del árbol de expansión
5:         Si vértice es parte de la arista de retroceso entonces
6:         Agregar al vector de recorrido de la arista del complemento
7:         Fin Si
```

```
8:        Agregar el vector de recorrido al vector de ciclos
9:     Fin Para
10: Fin Para
11:   Si Tamaño_Complemento > 1 y Hay al menos una arista que se repita en el vector de ciclos más de una vez entonces
12:       Arista_Mayor la arista que aparezca más veces en el vector de ciclos
13:       Valor_actual=vértice con valor más alto + 1
14:     Para cada arista de la grafica
15:          Si arista tiene algunos de sus extremos igual a los extremos de la Arista_Mayor entonces
16:             Remover la arista del grafo
17:             Agregar dos vértices más al vértice que conectaba con la arista eliminada
18:          Fin Si
19:       Fin Para
20:       Guia_Bosque = Todos los nodos de la nueva grafica
21:       Mientras haya aristas en la Guia_Bosque Hacer
22:          Hacer
23:             nodos=Nodos visitados en Guia_Bosque
24:             Para todas las aristas de la nueva grafica
25:                Si Guia_Bosque no tiene ningún nodo visitado entonces
26:                      Marcar como visitado el primer vértice de los extremos en Guia_Bosque
27:                      Marcar como visitado el segundo vértice de los extremos en Guia_Bosque
28:                Si Guia_Bosque tiene ingresado el primer vértice pero no el segundo entonces
29:                      Marcar como visitado el segundo vértice de los extremos en Guia_Bosque
30:                Si Guia_Bosque tiene ingresado el segundo vértice pero no el primero entonces
31:                      Marcar como visitado el primer vértice de los extremos en Guia_Bosque
32:                Fin Si
33:             Fin Para
34:          Repetir Mientras nodos < Nodos visitados en Guía_Bosque
35:          Para todas las aristas de la nueva grafica
36:             Si los extremos de la arista están en Guia_Bosque entonces
37:                Agregar la arista al arreglo árbol
38:             Fin si
39:          Fin Para
40:        Agregar el arreglo árbol al arreglo bosque
41:          Eliminar todas las aristas eliminada de Guia_Bosque
42:       Fin Mientras
43:       Para cada elemento del arreglo Bosque
44:          Aplicar el algoritmo 6 sobre el elemento
45:       Fin Para
46:    Otro
47:       Aplicar algoritmo 5 sobre la grafica
48:    Fin Si
49:    Sumar el número de coberturas de aristas de la gráfica al conteo total
```

Figura 3-17. *Algoritmo para realizar la Regla de expansión.*

Las diferencias entre la implementación de esta regla y la anterior se pueden observar hasta después de que se han encontrado ciclos intersectados en la gráfica de entrada, una vez obtenido el vector de ciclos e identificado la arista que

pertenece a más ciclos, se debe obtener el número de vértices que hay en la gráfica.

Una vez obtenido el número total de vértices de la gráfica, se recorre la gráfica de entrada y se elimina la arista que aparece en más ciclos intersectados. A todos los vértices que sean adyacentes a algunos de sus extremos se les agregan dos aristas más, esto se logra recorriendo la gráfica, cuando se encuentra la arista a eliminar esta es ignorada y todas las demás son agregadas a un nuevo arreglo. Cuando se encuentran vértices adyacentes a cualquiera de los dos extremos de la arista eliminada, además de ser agregada como tal, se le agregan dos vértices generados a partir del número mayor de nodos encontrado anteriormente, de esta forma se puede conservar el orden de la gráfica.

Una vez realizada esta expansión (agregar nuevas aristas a la gráfica original) se debe analizar la nueva grafica generada, ya que existe la posibilidad de que al aplicar la regla se hayan generado aristas aisladas dentro de la misma grafica o lo que es lo mismo a partir de una gráfica se puede generar un bosque de graficas no conectadas entre sí. Para esto, se utiliza el arreglo Guia_Bosque, este arreglo es un registro de todos los vértices de la gráfica estén o no conectados entre sí y si estos ya han sido visitados o no. La forma en la que se utiliza es la siguiente: mientras que Guia_Bosque tenga elementos, se recupera el número de vértices que ya han sido visitados, para después recorrer los elementos gráfica.

Por cada elemento de la gráfica (la cual almacena aristas) si en Guia_Bosque no hay vértices visitados se agregan los dos extremos de la arista y se indica en Guia_Bosque que estos dos vértices ya han sido visitados, en caso de que Guia_Bosque ya tenga vértices visitados, se comprueba que la arista tenga uno de sus vértices visitados y el otro no y si cumple esta condición se agrega el vértice que todavía no está ingresado y se hace el ajuste adecuado en Guia_Bosque.

La grafica se recorre una y otra vez hasta que no sea posible visitar más vértices desde el vértice con el que se inició. Esto se verifica con la variable nodos, la cual al asignarle el número de nodos visitados en Guia_Bosque (antes de recorrer la gráfica), su valor será menor al número de nodos visitados después de recorrer la gráfica, siempre y cuando se haya alcanzado al menos un nuevo vértices, si el valor de nodos es igual al valor de vértices visitados en Guia_Bosque al finalizar el proceso entonces no es posible agregar más vértices y el ciclo termina.

Una vez realizado lo anterior se vuelve a recorrer los elementos de la gráfica para agregar todas las aristas que tengan sus dos extremos visitados en Guia_Bosque a un nuevo arreglo el cual se puede denominar como árbol. Una vez que se han agregado todas las aristas que le corresponden al arreglo árbol éste es agregado a un arreglo llamado Bosque que contiene todos los arboles generados y en Guia_Bosque se eliminan todos los vértices que ya han sido visitados.

Así, en una nueva iteración, si existen todavía elementos en Guia_Bosque indica la presencia de graficas separadas y al empezar con un nuevo vértice (ya que todos los que fueron visitados fueron eliminados previamente) se generará un árbol distinto al anterior. Este procedimiento se ejecutará hasta que todos los vértices de Guia_Bosque hayan sido visitados y eliminados, guardando en el arreglo Bosque todas las gráficas no conectadas generadas por la regla de expansión.

Al igual que en la regla de división, todas las nuevas graficas generadas pasan por el proceso de obtener su árbol de expansión y comprobar la existencia de ciclos intersectados, hasta que estas ya no tengan ciclos intersectados y entonces pasan por el algoritmo de conteo de cubiertas sumando su número de cubiertas al total de la gráfica de entrada original.

Es importante resaltar que cuando la regla de división genera sólo una gráfica conectada, su número de cubiertas se suma normalmente, en tanto que cuando ésta genera un bosque de graficas no conectadas entre sí, antes de sumarse al número de coberturas de aristas totales, primero se deben multiplicar entre sí y entonces es el resultado el que se suma al total, esto se explicara más a fondo en la Fase 4 de la metodología.

3.4.3 Pruebas

Como ya se mencionó previamente esta fase es la que une todas las fases anteriormente desarrolladas, así que la mejor forma de realizar las pruebas correspondientes es la ejecución de distintos ejemplos con características diferentes que permitan observar el comportamiento de la implementación ante tales circunstancias.

Primero hay que observar el caso base donde la gráfica de entrada no contenga ciclos intersectados, en esta situación, el sistema debería aplicar directamente el algoritmo de conteo de cubiertas sin generar ninguna grafica extra en el sistema de archivos (Figura 3-18).

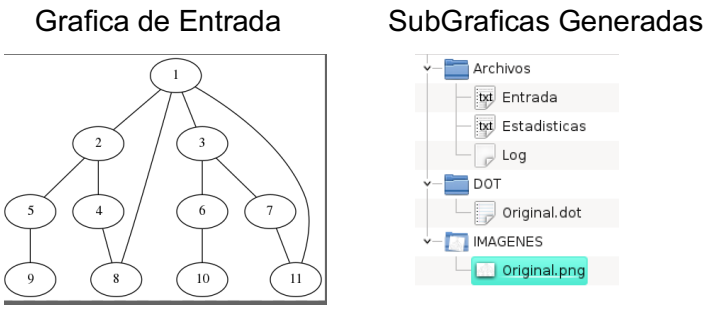

Figura 3-18. *Aplicación del algoritmo completo sobre una gráfica sin ciclos intersectados.*

La Figura 3-19 muestra un ejemplo con una gráfica que no contiene ciclos intersectados desde el inicio, el sistema solo ejecuta el algoritmo de conteo.

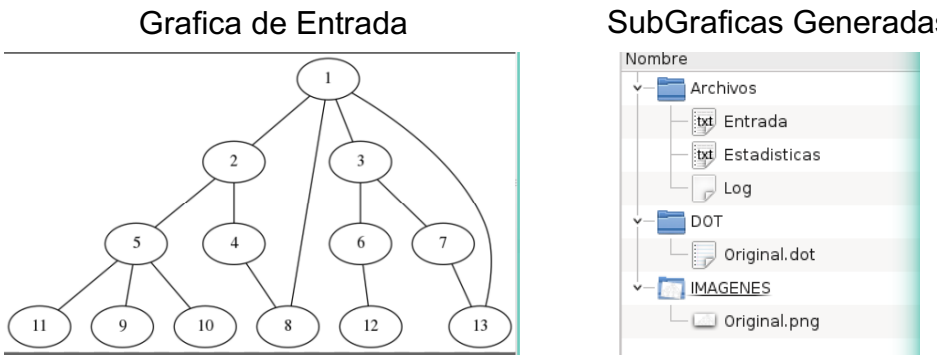

Figura 3-19. *Aplicación del algoritmo completo sobre una gráfica sin ciclos intersectados No. 2.*

Si la entrada es una gráfica con ciclos intersectados, dependiendo de la entrada se deben aplicar las reglas de reducción y expansión. Un primer ejemplo de este caso se puede observar en la Figura 3-20, donde las líneas negras continuas indican la regla de reducción, líneas negras discontinuas, indican la regla de expansión y la línea roja la arista eliminada en cada fase.

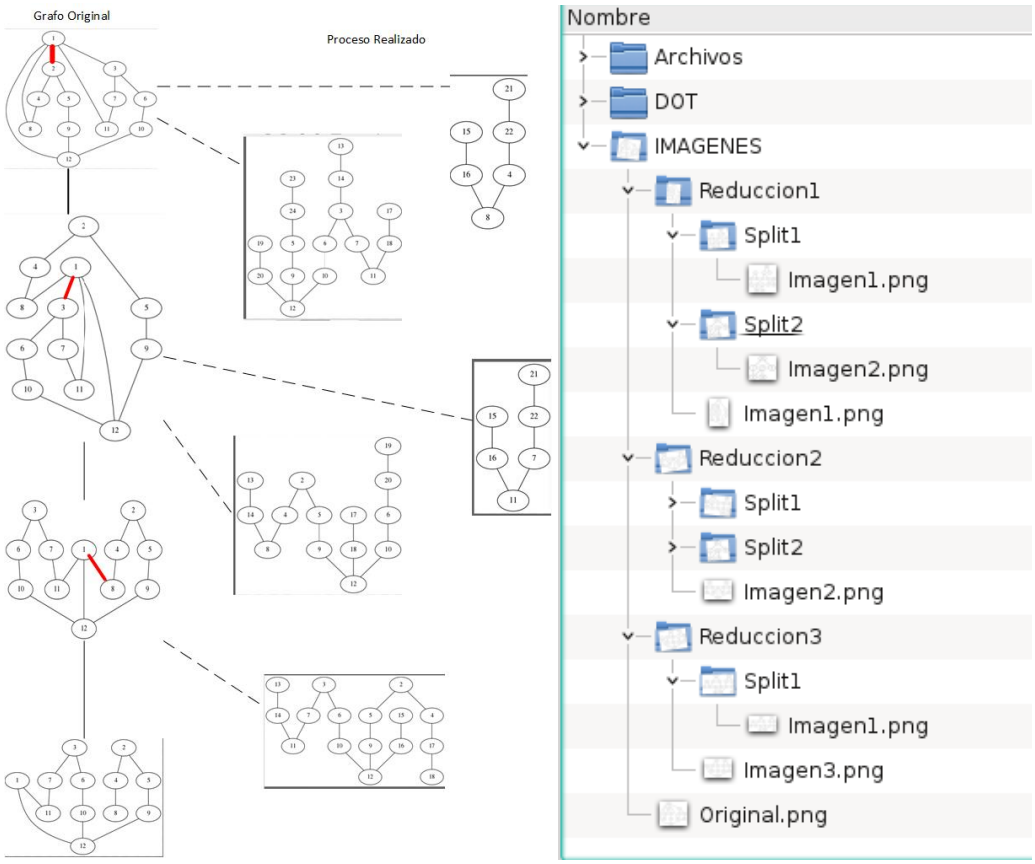

Figura 3-20. *Aplicación del algoritmo en una gráfica con ciclos intersectados.*

El sistema de archivos apenas y se puede mostrar completo en este ejemplo sin embargo comprueba que la relación entre las imágenes generadas durante la ejecución del algoritmo y su incorporación al sistema de archivos esta sincronizada y es comprensible cumpliendo con su objetivo.

Para finalizar la etapa de pruebas de esta etapa se muestra la ejecución de un ejemplo más en este caso en un grafo completo 5x5, los cuales son los más complicados de analizar para este algoritmo ya que cada una de sus aristas está unida a todas las demás generando muchos ciclos intersectados, sin embargo como se puede observar en la Figura 3-21, la implementación cumple adecuadamente con su objetivo.

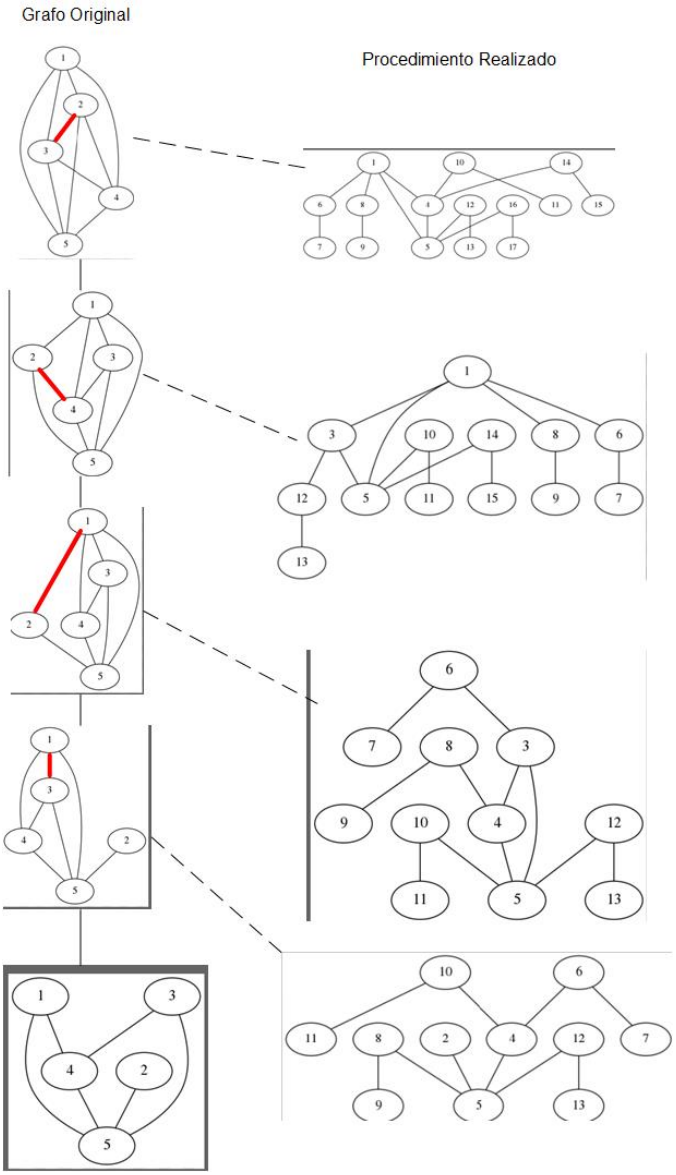

Figura 3-21. *Aplicación del algoritmo en una gráfica completa de 5x5 con ciclos intersectados.*

3.5 Fase 4: Conteo Completo de Cubiertas y Creación de Estadísticas

3.5.1 Análisis

Esta última fase de implementación es realmente corta ya que solo se ajustan los detalles finales para la correcta interrelación de las fases anteriores. En específico se centró en dos puntos principales:

- Obtener el conteo total de cubiertas de aristas, sumando el número de cubiertas individuales de cada grafica generada y solo en el caso de que se genere un bosque, primero realizar la multiplicación antes de sumar.

- Generar estadísticas de la ejecución que permitan obtener información y obtener conclusiones.

De acuerdo con los objetivos que se plantearon al inicio de esta Tesis, la información importante que se necesita para cumplirlos es la siguiente:

- El tiempo de ejecución del algoritmo

- El espacio en memoria que se ocupa durante la ejecución del algoritmo

- El número de nodos hojas que se van generando

3.5.2 Diseño

El procedimiento para realizar el conteo de cubiertas de aristas total se observa en el Algoritmo 8 de la Figura 3-22, el cual debe ser ejecutado siempre que se genere una gráfica sin ciclos intersectados.

Algoritmo 8: Procedimiento para aplicar la regla de reducción.

1: Entrada: Gráfica G
2: Salida: El Número de Coberturas de Aristas de la Grafica G
3: Inicializar variable TotalCubiertas
4: Iniciar proceso para realizar conteo de Coberturas
5: …
6: Después de aplicar la regla de división y obtener el arreglo árbol
7: **Para** cada elemento del arreglo árbol
8: Variable EsunForest = verdadero
9: Mandar el elemento del arreglo al proceso general
10: **Fin Para**

```
11: Si EsunForest=verdadero entonces
12:       Sumar TotalSubCubiertas a TotalCubiertas
13:       InicializarTotal SubCubiertas en 1
14: Fin Si
15: EsunForest=falso
16: …
17: Aplicar Algoritmo de Conteo sobre una Grafica Hoja sin Ciclos Intersectados
18:   Si EsunForest = falso entonces
19:       Sumar el resultado a TotalCubiertas
20: Otro
21:       Multiplicar el resultado a TotalSubCubiertas
22: Fin Si
23: …
24: Finalizar ejecución del algoritmo
25: Agregar TotalCubiertas al archivo Estadísticas
```

Figura 3-22. *Algoritmo para obtener el conteo total de cubiertas de aristas.*

Analizando la implementación del algoritmo hay que recordar que para la suma total de cubiertas de aristas se hizo uso de la librería gmp para manejar y representar números muy grandes que el lenguaje por sí mismo no es capaz de manejar, es por esto que la variable TotalCubiertas debe inicializarse antes de iniciar el procedimiento completo, debido a que así es como funciona la librería gmp.

Para realizar el cálculo adecuadamente es necesario hacer ajustes en dos partes distintas del algoritmo. Los puntos suspensivos en el algoritmo indican precisamente estos saltos a las partes del algoritmo donde se realizan estos ajustes. Lo primero que hay que hacer es ir a la ejecución de la regla del *split* y verificar si se ha generado un Bosque.

Al momento de recorrer el arreglo “Bosque” se pone la bandera EsunForest en verdadera para indicar que se está recorriendo un Bosque y que todas las gráficas que se han enviadas durante su recorrido sean multiplicadas entre si antes de sumarse al número total de cubiertas.

Lo anterior queda reflejado en la segunda sección del algoritmo donde una vez que se han contado las cubiertas de aristas en graficas sin ciclos intersectados, se utiliza la bandera para decidir si se suman directamente al total de cubiertas de la gráfica de entrada o si se multiplica por el valor que se lleva hasta el momento en TotalSubCubiertas.

Finalmente, regresando a la primera sección del algoritmo una vez que se ha terminado de procesar el arreglo Bosque, si la bandera EsunBosque sigue en verdadera al final de su ejecución entonces, el valor de TotalSubCubiertas se suma al de ToltalCubiertas, agregando todas las cubiertas de aristas del Bosque al valor final de cubiertas de la gráfica de entrada.

El último paso de este algoritmo es reiniciar el valor de TotalSuCubiertas en 1 para que esté listo para procesar el siguiente bosque. Al término de la ejecución del algoritmo, el TotalCubiertas se agrega al archivo Estadísticas junto a todas las medidas estadísticas (Figura 3-23).

Algoritmo 9: Procedimiento para Generar las Estadísticas del Sistema

1: Entrada: Gráfica G
2: Salida: Estadísticas de la Ejecución del Algoritmo
3: Inicializar el número de hojas
4: Guardar fecha de inicio del algoritmo
5: Ejecutar el algoritmo completo
6: Guardar fecha de terminación del algoritmo
7: Calcular la diferencia entre la fecha de inicio y terminación del algoritmo
8: Moverse a la dirección del archivo “Estadísticas”
9: Calcular uso de memoria
10: Abrirlo para solo escritura
11: Guardar uso de memoria del algoritmo
12: Guardar en archivo tiempo de ejecución
13: Guardar en archivo número de hojas
14: Guardar en archivo total de cubiertas de aristas
15: Cerrar archivo “Estadisticas”

Figura 3-23. *Procedimiento para generar las estadísticas del sistema.*

La ejecución de este procedimiento es simple, antes de ejecutar el algoritmo se inicializa la variable Num_Hojas, que es la encargada de indicar cuantas graficas hojas genero la implementación del algoritmo, la cual como se vió en el Algoritmo 8 se va aumentando en uno cada vez que se encuentra una gráfica sin ciclos intersectados.

Para guardar la fecha de inicio y terminación de la ejecución del algoritmo se utilizaron librerías propias del lenguaje c por lo que no es importante profundizar más allá del hecho de que debido a la naturaleza del algoritmo, es necesario que el tiempo de ejecución se exprese en milisegundos para obtener una medida representativa tanto para los ejemplos que se tarden mucho, como para los que se tarden poco.

Para guardar las estadísticas de número de hojas y total de cubiertas, esto se realiza directamente utilizando en el caso de las cubiertas de aristas las herramientas que la propia librería gmp ofrece para escribir el resultado en el archivo, así mismo el número de hojas se guarda igualmente en el archivo.

Para finalizar, el uso de memoria se calcula utilizando el comando “du –sh” sobre la carpeta “DOT”, “IMÁGENES” y la carpeta raíz de todo el sistema “CUBIERTAS” para poder analizar el crecimiento en el uso de memoria del algoritmo, éste al

igual que las anteriores se guardan en el archivo “Estadísticas” para su posterior análisis.

3.5.3 Pruebas

Las pruebas para esta última fase de la metodología y con la que se concluye formalmente el desarrollo del sistema, consiste en observar los resultados que el sistema genera cuando se realizan distintos ejemplos, verificando que la suma de las coberturas sea correcta y que las estadísticas generadas sean entendibles y útiles para el análisis posterior. La Figura 3-24 muestra el procedimiento por el que pasa el ejemplo de la Figura 3-18 para calcular sus coberturas de aristas.

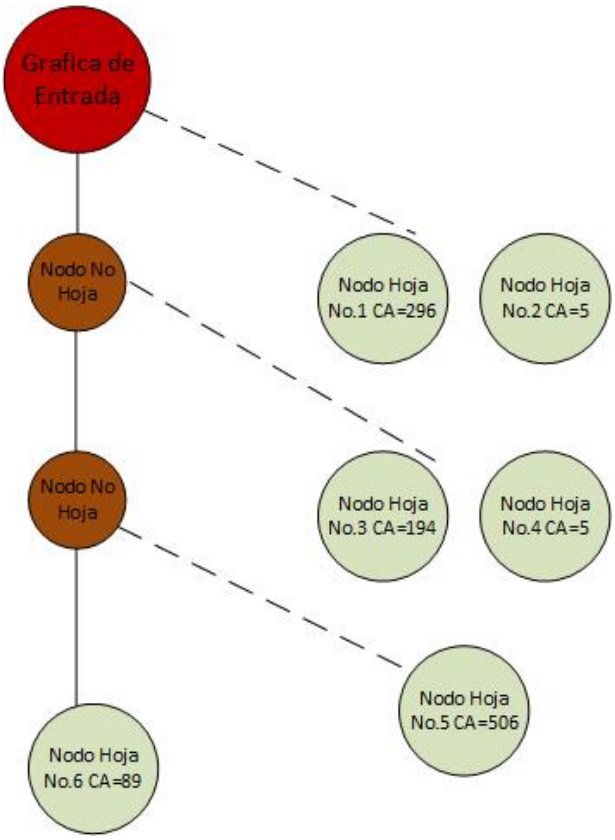

Figura 3-24. *Procedimiento completo para el ejemplo de la figura 3-18.*

En el procedimiento se resalta el valor de las coberturas de aristas de los nodos hojas de la estructura, ya que la ejecución del algoritmo hasta ese punto ya se comprobó en las fases anteriores.

Como se puede observar, para calcular el total de coberturas de aristas se deben multiplicar el valor de los nodos hoja 1 y 2 por un lado y por otro los nodos hojas 3 y 4, ya que pertenecen a su respectivo árbol generado por la regla de expansión.

Realizando el procedimiento paso a paso se obtiene:

$$(296 * 5) + (194 * 5) + 506 + 89$$

$$1480 + 970 + 506 + 89$$

$$3045$$

Así 3045 es el valor esperado del total de cubiertas de aristas de la gráfica de entrada, y es este mismo el que se debe observar en el archivo estadísticas, al terminar la ejecución del algoritmo, esto se puede observar en la Figura 3-25 donde se muestra el contenido del archivo estadísticas para el mismo ejemplo.

```
36K     /home/luis/Documentos/CubiertasAristas/Coberturas/DOT/
252K    /home/luis/Documentos/CubiertasAristas/Coberturas/IMAGENES/
304K    /home/luis/Documentos/CubiertasAristas/Coberturas/
Tiempo de ejecucion: 830.9569358825684 millisegundos
Numero de Nodos Hojas: 6
Total de CUbiertas de Aristas: 3045
```

Figura 3-25. *Estadísticas generadas para el ejemplo de la Figura 3-18.*

Lo primero a señalar del archivo estadísticas es que el número de coberturas de aristas es igual al esperado, comprobando que el algoritmo y su correspondiente implementación funcionan de acuerdo a lo planeado. Así mismo el archivo estadísticas contiene información adicional sobre el tamaño que ocupan los archivos generados por el sistema en algoritmos, el número de hojas generadas y su tiempo de ejecución, estos datos estadísticos se utilizaran más adelante en análisis posteriores.

Es claro que comprobar el funcionamiento del sistema con solo un ejemplo y más si este es tan sencillo que no genero más de 6 nodos hojas, no tiene más de un subnivel y no genera bosques, no es una muestra contundente e irrefutable de que el algoritmo funciona como debe ser, por lo tanto es necesario presentar un ejemplo más complejo el cual se puede observar en la Figura 3-26.

Este segundo ejemplo muestra la aplicación del procedimiento completo para una gráfica completa de 6x6, la cual presenta una complejidad mucho mayor que la del ejemplo anterior, generando un total de 43 nodos hojas y alcanzando a tener dos subniveles, por lo cual representa una muestra mucho más representativa.

Para corroborar la funcionalidad del algoritmo en este caso más complejo se siguen los mismos pasos que en el ejemplo anterior, es decir primero multiplicar el número de coberturas de aristas de las gráficas que formen arboles entre sí, antes de sumar ese valor al conteo total de cubiertas de aristas de la gráfica de entrada original.

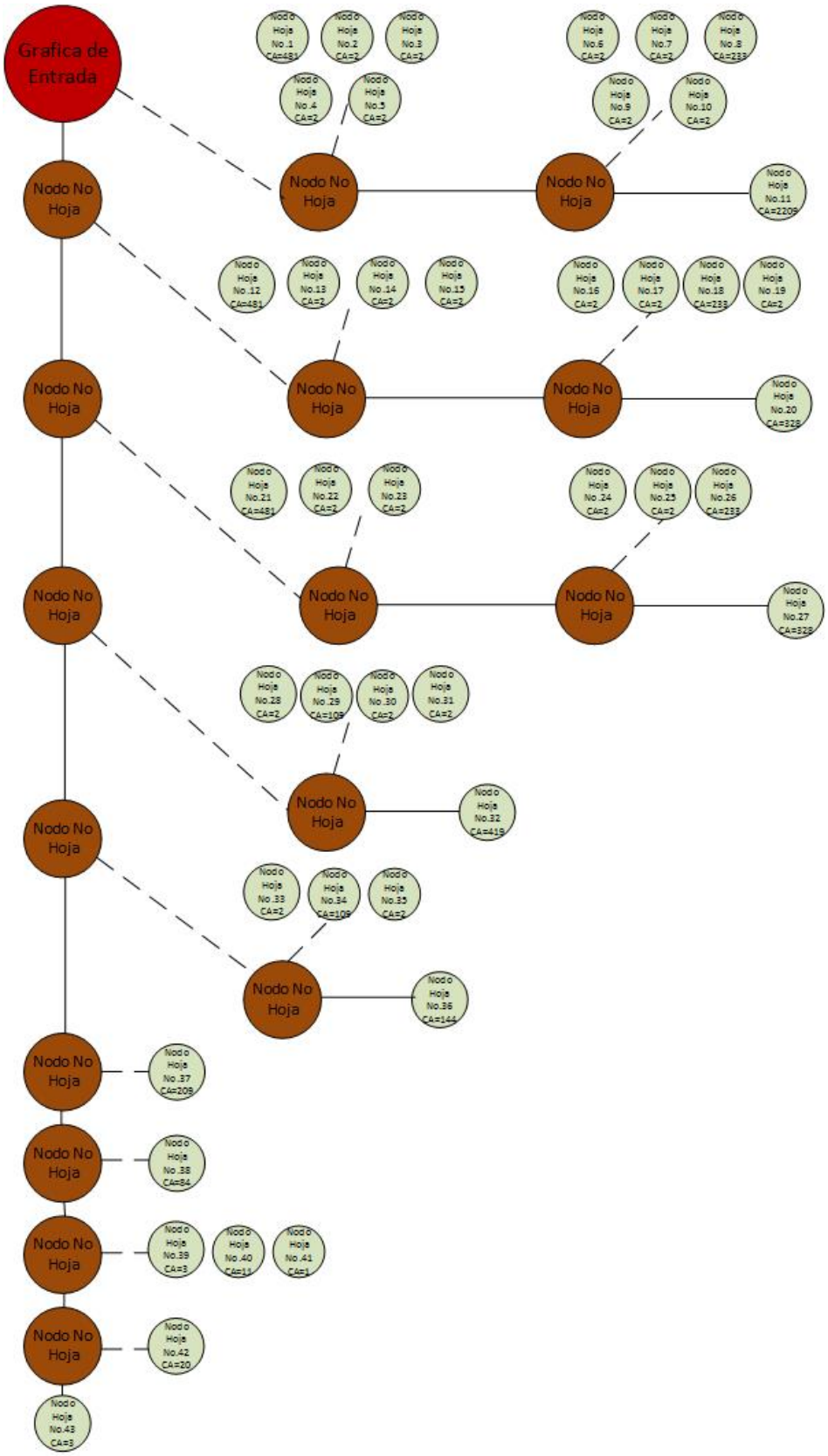

Figura 3-26. *Procedimiento completo para una gráfica completa de 6x6.*

Las operaciones que se deben realizar para esta comprobación se deben realizar paso a paso para analizar si se obtuvo el resultado esperado, estas se pueden observar a continuación:

$$(481 * (2)^4) + (233 * (2)^4) + 2209 + (481 * (2)^3) + (233 * (2)^3) + 328 + (481 * (2)^2) + (233 * (2)^2) + 328 + (109 * (2)^2) + 419 + (109 * (2)^2) + 144 + 209 + 84 + (3 * 11 * 1) + 20 + 3$$

$$7{,}696 + 3728 + 2209 + 3848 + 1864 + 955 + 328 + 932 + 1924 + 419 + 872 + 436 + 144 + 209 + 84 + 33 + 20 + 3$$

$$25704$$

El valor esperado que debe devolver el sistema debe ser 25704, el cual como se puede observar en la Figura 3-27 es justamente el valor regresado por el sistema, comprobando el funcionamiento del algoritmo en casos más complejos y por lo subsecuente verificando que la implementación que se realizó del algoritmo funciona correctamente.

```
240K»   /home/luis/Documentos/CubiertasAristas/Coberturas/DOT/
1,4M»   /home/luis/Documentos/CubiertasAristas/Coberturas/IMAGENES/
1,7M»   /home/luis/Documentos/CubiertasAristas/Coberturas/
Tiempo de ejecucion: 12939.66388702393 millisegundos
Numero de Nodos Hojas: 43
Total de CUbiertas de Aristas: 25704
```

Figura 3-27. *Estadísticas generadas para una gráfica de 6x6.*

Capítulo 4

Resultados

4.1 Mejor de los Casos (Gráficas Cactus)

El primer caso que se estudia y analiza es el correspondiente con las gráficas cactus, este tipo de gráficas tiene la particularidad de que no tiene ciclos intersectados (Sección 2-1). Para este tipo de gráficas el crecimiento que se espera en el uso de recursos es lineal (Sección 2-4).

Para comprobar que el algoritmo cumpla con el comportamiento esperado, se generaron aleatoriamente gráficas cactus incrementando en cada ejecución el tamaño de la gráfica de entrada.

Hay dos parámetros a considerar al momento de trabajar con gráficas cactus, el primero de ellos, es el número de ciclos simples que tiene la gráfica y el segundo es el número de vértices que contiene cada ciclo. Con el fin de obtener una medida estadística representativa se analizan los resultados manteniendo constante uno de los dos parámetros.

Primero se muestran los resultados de dejar constante el número de vértices de cada ciclo simple contenido en la gráfica, en este caso cinco, mientras se aumenta el número de ciclos simples. Los resultados obtenidos de la ejecución del algoritmo se muestran en la Tabla 4.1.

Tabla 4-1. *Resultados obtenidos sobre gráficas cactus dejando constante el número de vértices en cada ciclo simple.*

TIPO DE GRAFICA	TIEMPO DE EJECUCIÓN	NODOS HOJA	TAMAÑO DE LOS ARCHIVOS CREADOS	NUMERO DE CUBIERTAS DE ARISTAS
2 CICLOS SIMPLES	45.24 ms	1	52 K	94
3 CICLOS SIMPLES	55.947 ms	1	68 K	1,691
4 CICLOS SIMPLES	60.4870 ms	1	84 K	30,348
5 CICLOS SIMPLES	68.4769 ms	1	100 K	544,577
6 CICLOS SIMPLES	75.4461 ms	1	120 K	9,772,042
7 CICLOS SIMPLES	83.3659 ms	1	136 K	175,352,183
8 CICLOS SIMPLES	90.5311 ms	1	152 K	361,151,576

9 CICLOS SIMPLES	96.5819 ms	1	168 K	628,283,581
10 CICLOS SIMPLES	105.887 ms	1	184 K	882,154,230
11 CICLOS SIMPLES	113.2328 ms	1	200 K	1,268,498,067

El incremento en el tiempo de ejecución del algoritmo, para las gráficas cactus, se puede observar en la Figura 4-1.

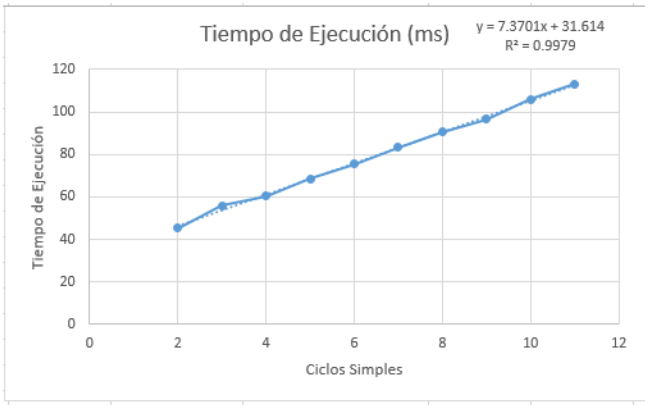

Figura 4-1. *Incremento en tiempo de ejecución del algoritmo para gráficas cactus manteniendo constante el número de vértices en cada ciclo simple.*

En la Figura 4-1 se puede apreciar claramente como el tiempo crece de manera constante a medida que aumentan el número de vértices y aristas presentes en la gráfica de entrada, lo cual representa un crecimiento lineal representado por la gráfica $y = 7.3701(x) + 31.614$ con un desfase posible de 0.9979, este es el comportamiento esperado para este tipo de gráficas.

Por su parte el incremento en el uso de memoria del algoritmo para las gráficas cactus se puede observar en la Figura 4-2.

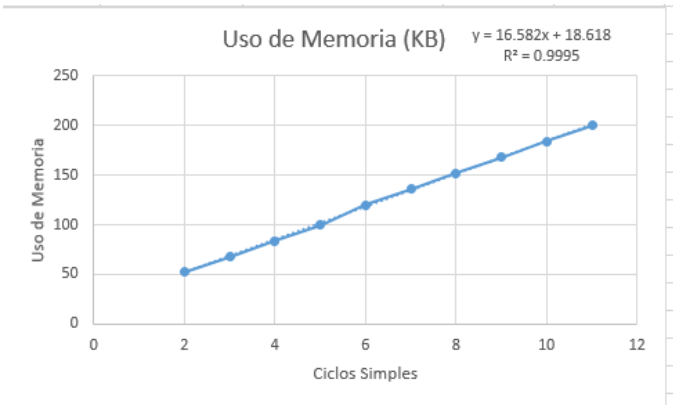

Figura 4-2. *Incremento en uso de memoria del algoritmo para graficas cactus manteniendo constante el número de vértices en cada ciclo simple.*

El uso de memoria al igual que el tiempo de ejecución muestra el mismo comportamiento de crecimiento lineal, representado en este caso por la ecuación $y = 16.582x + 18.618$ con una posible desviación de 0.9995, esto es lo esperado y deseado para el funcionamiento del algoritmo ante estas gráficas a las cuales solo se les debe aplicar el algoritmo de conteo directamente, no se realizará un análisis del crecimiento en el número de hojas, debido a que la Tabla 4-1 muestra claramente que esta es siempre 1, como estaba previsto.

Una vez analizado el primer caso de crecimiento para las gráficas cactus y comprobado que cumple con el comportamiento esperado, se realizó el segundo tipo de pruebas en donde el número de ciclos simples se mantiene constante mientras aumenta el número de vértices en cada uno de ellos.

La Tabla 4-2 muestra los resultados obtenidos sobre la ejecución del algoritmo en gráficas cactus con 10 ciclos simples constantes, mientras se aumenta paulatinamente el número de vértices en cada uno.

Tabla 4-2. *Resultados obtenidos sobre graficas cactus dejando constante el número de ciclos simples.*

TIPO DE GRAFICA	TIEMPO DE EJECUCIÓN	NODOS HOJA	TAMAÑO DE LOS ARCHIVOS CREADOS	NUMERO DE CUBIERTAS DE ARISTAS
3 VERTICES	78.389 ms	1	120 K	74,049,690
4 VERTICES	87.156 ms	1	152 K	282,154,230
5 VERTICES	110.2969 ms	1	184 K	669,281,871
6 VERTICES	122.8277 ms	1	216 K	930,818,044
7 VERTICES	139.1739 ms	1	244 K	1,345,816,216
8 VERTICES	160.7110 ms	1	272 K	1,700,915,434
9 VERTICES	170.9468 ms	1	300 K	2,264,166,582
10 VERTICES	184.4570 ms	1	328 K	2,296,041,959
11 VERTICES	191.8769 ms	1	356 K	3,057,590,578
12 VERTICES	213.4101 ms	1	380 K	3,368,905,968
13 VERTICES	225.5809 ms	1	408 K	3,874,592,557
14 VERTICES	239.4680 ms	1	436 K	4,101,900,342

Los resultados obtenidos tanto para el tiempo de ejecución como para el uso de memoria se observan en la Figura 4-3 donde se puede apreciar que ambos

conservan un crecimiento lineal esperado, el tiempo de ejecución está representando por la ecuación $y = 14.716x + 35.104$ con una desviación posible de 0.9975, mientras que el uso de memoria está representado por la ecuación $y = 28.434x + 41.315$ con una desviación posible de 0.9988.

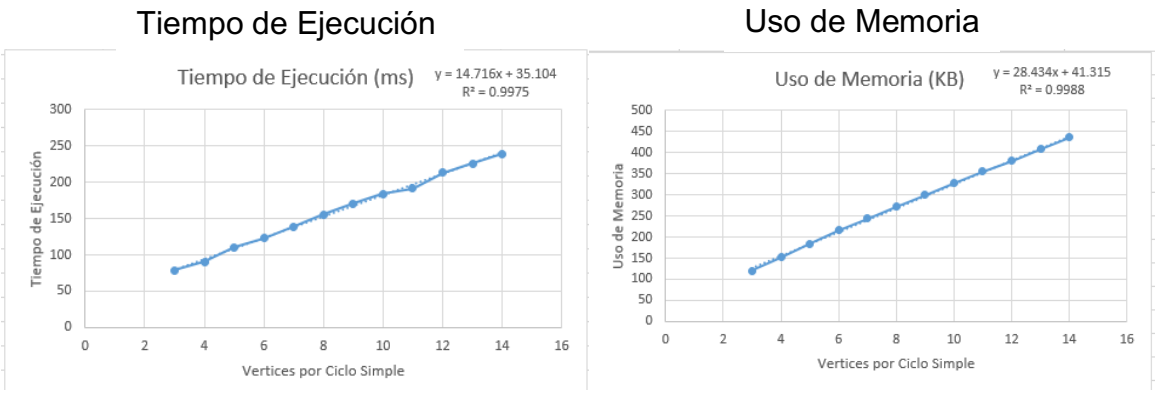

Figura 4-3.*Resultados para las gráficas cactus manteniendo constante el número de ciclos simples.*

Esto permite verificar que el comportamiento del algoritmo ante las gráficas cactus es justamente el esperado, tener un crecimiento lineal tanto en el tiempo de ejecución como en el uso de memoria.

4.2 Caso General (Gráficas Simples Aleatorias)

El siguiente caso a considerar es el general, para esto las gráficas se generaron de forma aleatoria, en el sentido de que el número de ciclos intersectados que pueden contener son independientes de su número de vértices y aristas, a diferencia de las gráficas cactus que no contienen ciclos intersectados, por más que se agreguen aristas o vértices.

Para esto se analizaron diferentes gráficas a las cuales se les mantuvo constante su número de vértices, mientras se les aumento el número de aristas paulatinamente, para definir el rango permitido en el cual se hicieron estos aumentos, se partió del hecho de que considerando n como el número de vértices $n - 1$ indica el número de aristas de un árbol (límite inferior), mientras que $\frac{n(n-1)}{2}$ indica el número de aristas de una gráfica completa (límite superior).

El primer caso que se analizo fue la gráfica con 19 vértices, la cual tiene como límite inferior 18 aristas y como límite superior 171 aristas, los resultados obtenidos ante la ejecución de estas gráficas aleatorias se puede observar en la Tabla 4-3.

Tabla 4-3.*Resultados obtenidos en gráficas aleatorias de 19 vértices.*

Numero de Aristas	Ciclos Intersectados Posibles	Tiempo de Ejecución	Nodos Hoja	Tamaño de los Archivos Creados	Numero de Cubiertas de Aristas
23	5	2.083 s	23	1.1 M	34,137
28	10	4.302 s	49	2.7 M	3,263,443
33	15	2.452 s	27	1.6 M	429,640
38	20	2.80 min	1595	71 M	8,899,020,683
43	25	8.67 min	4771	205 M	221,360,928,994,608,263,628
48	30	25.65 min	12697	639 M	41,874,109,052,583,540,684,482
53	35	4.5 hrs	76602	3.2 G	11,031,152,958,191,971,576,533

La Figura 4-4 muestra el crecimiento del tiempo de ejecución conforme se aumentan las aristas de la gráfica, este crecimiento como se puede apreciar ya es exponencial y le corresponde la ecuación de crecimiento $y = 0.5797e^{0.3146x}$ con un rango de error de 0.9228.

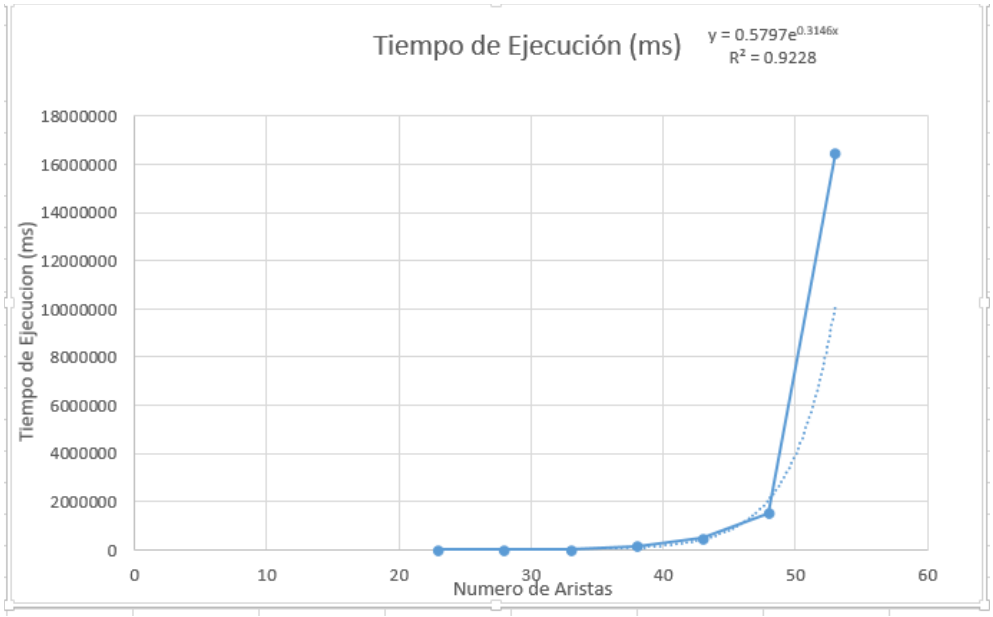

Figura 4-4. *Incremento en el tiempo de ejecución en graficas aleatorias de 19 vértices.*

La Figura 4-5 muestra el crecimiento del uso de memoria del algoritmo conforme se aumenta aristas a la gráfica, al igual que el tiempo de ejecución, este también tiene un comportamiento exponencial, al cual le corresponde la ecuación $y = 0.0008e^{0.2842x}$ con un rango de error de 0.9324.

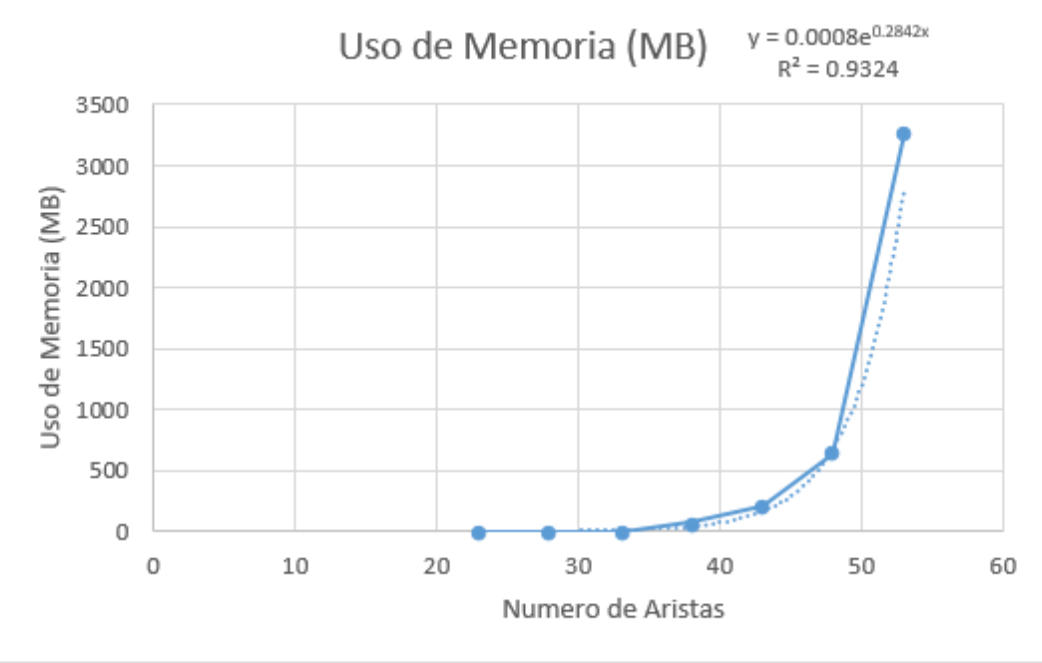

Figura 4-5. *Incremento del uso de memoria en gráficas aleatorias de 19 vértices.*

El comportamiento corresponde con lo esperado, debido a que conforme más aristas se agregan a la gráfica, esta se convierte poco a poco en una gráfica completa, sin embargo la Tabla 4-3 presenta un dato atípico que rompe con el patrón de todos los demás, cuando la gráfica tiene 33 aristas, pero esto tiene una explicación debido a que esa gráfica generada aleatoriamente a pesar de tener más aristas género muy pocos ciclos intersectados, este comportamiento del algoritmo muestra características interesantes.

Pero para poder corroborar que el patrón se repite se tiene que realizar diferentes ejemplos, es por esto que la Tabla 4-4 muestra los resultados obtenidos en graficas con 28 vértices con los mismos incrementos que el caso anterior, solo que ahora sus límites son 27 como límite inferior y 378 como límite superior.

Tabla 4-4. *Resultados obtenidos en gráficas aleatorias de 28 vértices.*

Numero de Aristas	Ciclos Intersectados Posibles	Tiempo de Ejecución	Nodos Hoja	Tamaño de los Archivos Creados	Numero de Cubiertas de Aristas
32	5	4 s	42	2.4 M	2957615
37	10	22 s	142	11 M	288713807
42	15	1.26 min	956	52 M	1065467499
47	20	5.139 min	4456	218 M	73786976355208298492
52	25	16.92 min	12896	676 M	3984496720587548370509

57	30	24.76 min	20394	976 M	20088504296950244769309
62	35	1.9 hrs	78949	3.6 G	66039343786487898464405

Las gráficas que representan el crecimiento en tiempo de ejecución y uso de memoria se pueden observar en las Figuras 4-6 y 4-7, respectivamente. Ambos tienen un crecimiento exponencial, la ecuación $y = 2.9205e^{0.2385x}$ con un rango de error de 0.9834, mientras que el uso de memoria está representado por la ecuación $y = 0.0017e^{0.2396x}$ con un rango de error de 0.9777.

Figura 4-6. *Incremento en Tiempo de Ejecución en gráficas de 28 vértices.*

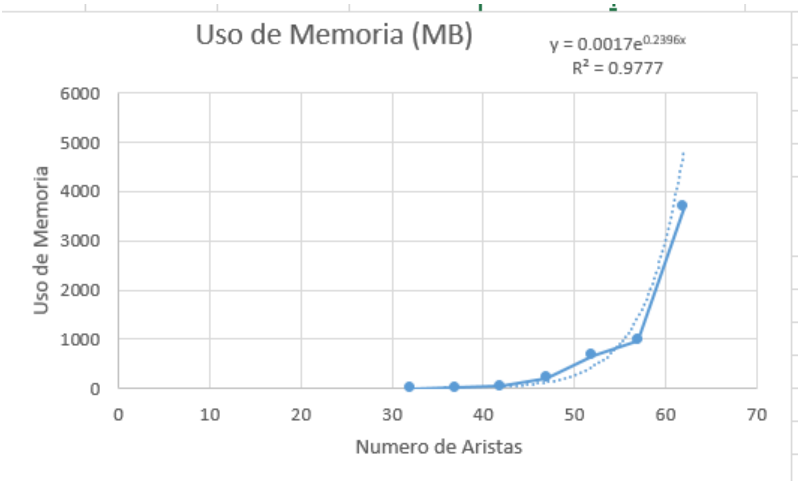

Figura 4-7. *Incremento en uso de memoria en graficas de 28 vértices.*

Algo que se puede notar en ambas graficas es el hecho de que ciertos valores no cumplen correctamente con el crecimiento exponencial esperado, mostrando desvíos ya sea menores o mayores respecto a la curva que se espera obtener esto se debe principalmente a los ciclos intersectados que no crecen de manera uniforme con el crecimiento de aristas de la gráfica.

Otro ejemplo se pueden observar en las Tablas 4-5 con graficas de 45 vértices, donde los valores de las aristas se mueven entre los límites del 44 al 990. La Figura 4-8 muestra el crecimiento tanto en tiempo de ejecución como uso de memoria.

Tabla 4-5. *Resultados obtenidos en gráficas aleatorias de 45 vértices.*

Numero de Aristas	Ciclos Intersectados Posibles	Tiempo de Ejecución	Nodos Hoja	Tamaño de los Archivos Creados	Numero de Cubiertas de Aristas
49	5	12.21 s	90	7.3 M	1657353710
54	10	1.38 min	759	58 M	553402322295590354125
59	15	1.9 min	1028	81 M	940783947838045234233
64	20	20.584 min	11158	872 M	24718637059494271436203
69	25	1.93 hrs	74180	4.2 G	113041647685556789003183

Tanto el tiempo de ejecución como el uso de memoria tiene el mismo patrón, un comportamiento exponencial, representado por las ecuaciones $y = 0.0033e^{0.3078x}$ con un rango de error de 0.9645 y $y = 2E-06e^{0.3094x}$ con un rango de error de 0.9677, respectivamente, en ambos casos el comportamiento exponencial se presenta con muy pocas desviaciones, por lo cual en este caso las gráficas aleatorias tuvieron un crecimiento en ciclos intersectados cercano a lo que se esperaría encontrar en gráficas completas (analizadas a continuación).

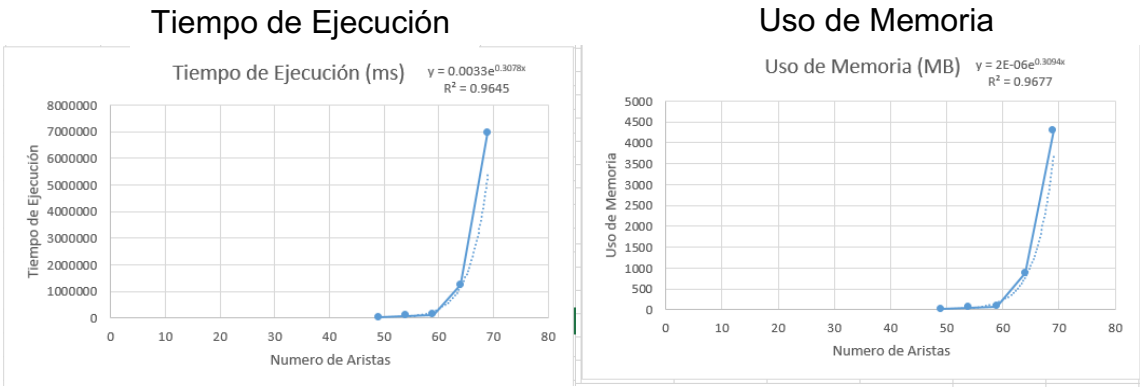

Figura 4-8. *Resultados para las gráficas de 45 vértices.*

El ejemplo final para este tipo de graficas se observa en la Tabla 4-6 donde se observan los resultados obtenidos en graficas de 50 vértices, con un rango de incremento de aristas posible entre 49 y 1225, mientras que la Figura 4-9 muestra el crecimiento en tiempo de ejecución y uso de memoria del algoritmo.

Tabla 4-6. *Resultados obtenidos en gráficas aleatorias de 50 vértices.*

Numero de Aristas	Ciclos Intersectados Posibles	Tiempo de Ejecución	Nodos Hoja	Tamaño de los Archivos Creados	Numero de Cubiertas de Aristas
54	5	17.9 s	122	8.1 M	6989745533
59	10	1.62 min	923	63 M	1051464412272793389788
64	15	7.046 min	3983	301 M	9149585060762956211137
69	20	11.34 min	6153	486 M	15144776884694511656060
74	25	3.5 hrs	85347	7.2 G	249031044996292047690102

Tanto el tiempo de ejecución como el uso de memoria tienen un comportamiento exponencial, representado por las ecuaciones $y = 0.0015e^{0.3014x}$ con un rango de error de 0.9529 y $y = 4E-07e^{0.3134x}$ con un rango de error de 0.9621, respectivamente, en ambos casos el comportamiento exponencial vuelve a presentar desviaciones en la curva que debe de seguir.

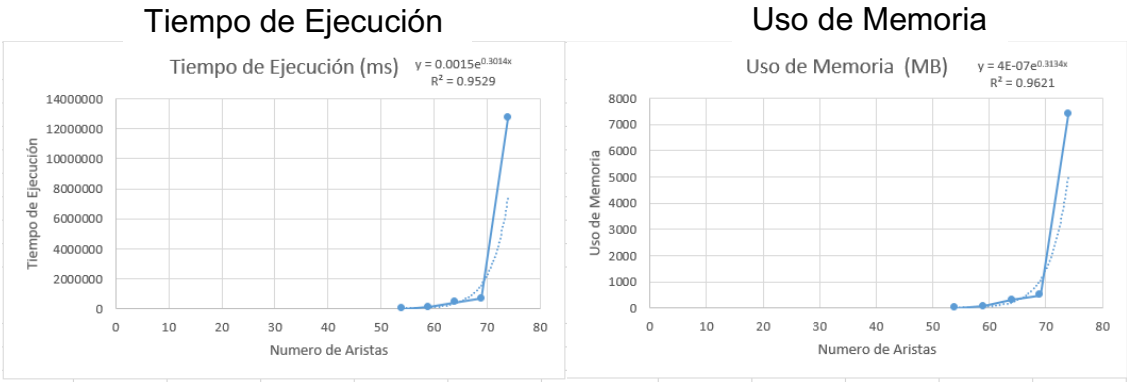

Figura 4-9. *Resultados para las gráficas de 45 vértices.*

De los ejemplos estudiados en esta sección se puede observar que el incremento de aristas y vértices afecta al algoritmo al necesitar más tiempo de ejecución y uso de memoria para ejemplos con un número parecido de ciclos intersectados pero con un número de aristas y vértices diferentes.

Pero al mismo tiempo se puede observar que debido a la aleatoriedad con la que fueron creadas estas gráficas el número de ciclos intersectados no aumenta en proporción al aumento de vértices y aristas, sino más bien depende de cómo estas se acomodan dentro de la gráfica, esto ocasiona los desvíos, ya se aumentó o disminución, respecto al crecimiento que se esperaría en ambos caso estudiados y el cual se puede observar en las Figuras correspondientes.

4.3 Peor de los Casos (Gráficas Completas)

El último caso que se analizó son las gráficas completas ya que representan el mayor grado de dificultad para el algoritmo al tener todos sus vértices unidos entre sí, esto significa que el número de ciclos intersectados es grande en comparación a número de vértices y aristas de la gráfica.

Este último análisis se dividió en dos etapas, en la primera de ellas se analizaron los resultados del algoritmo generando todos los archivos intermedios y finales, y la segunda solo obtiene el número de cubiertas al final del procedimiento con el fin de observar el impacto en el uso de memoria y en el tiempo de ejecución de agregar al sistema de archivos cada una de las gráficas generadas. Los resultados del algoritmo generando la salida se puede observar en la Tabla 4-7.

Tabla 4-7. *Resultados en gráficas completas generando las gráficas intermedias.*

TIPO DE GRAFICA	TIEMPO DE EJECUCIÓN	NODOS HOJA	TAMAÑO DE LOS ARCHIVOS CREADOS	NUMERO DE CUBIERTAS DE ARISTAS
4X4 COMPLETO	215.6021 ms	3	104 K	38
5X5 COMPLETO	402.0230 ms	5	268 K	543
6X6 COMPLETO	8.9 s	43	1.6 M	25,704
7X7 COMPLETO	36.89 s	173	5.7 M	1,447,320
8X8 COMPLETO	59. 345 s	755	24 M	31,098,876
9X9 COMPLETO	3.054 min	3112	95 M	5,179,245,574
10X10 COMPLETO	13.974 min	13002	394 M	36,181,185,762
11X11 COMPLETO	50.872 min	55578	1.6 G	102,296,041,959
12X12 COMPLETO	3.921 horas	237757	6.9 G	433,057,590,578

Tanto en las gráficas completas en particular como en el proceso general del algoritmo, el comportamiento esperado en su tiempo de ejecución y uso de memoria debería ser de comportamiento exponencial. Si se observa la Figura 4-10

se puede observar la gráfica de crecimiento del tiempo de ejecución del algoritmo cuando se genera la salida, la cual está representada por la ecuación $y = 1.1046e^{1.3643x}$ con un rango de error de 0.9789, comprobando el comportamiento exponencial esperado.

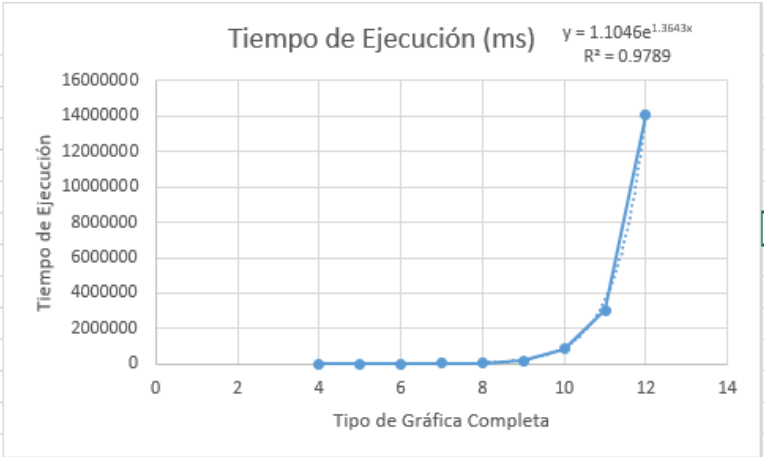

Figura 4-10. *Crecimiento del Tiempo de Ejecución en Graficas Generando la Salida.*

Este crecimiento al ser de tipo exponencial rápidamente alcanza números difíciles de procesar por su uso de recursos, lo mismo ocurre con el uso de memoria, el cual tiene un comportamiento similar como se puede apreciar en la Figura 4-11 y cuya ecuación corresponde a $y = 0.3065e^{1.4109x}$ con un rango de error de 0.9989.

Figura 4-11. *Crecimiento del Uso de Memoria en Graficas Generando la Salida.*

Para medir el impacto que tiene el graficar todas las subgraficas generadas por el procedimiento, se realizó una prueba sin generar la entrada, buscando obtener solo el número de coberturas de aristas, estos resultados se pueden observar en la Tabla 4-8, donde puede apreciarse una reducción en el tiempo de ejecución, sin embargo, aunque el crecimiento es un poco más lento, sigue conservando el comportamiento exponencial esperado.

Tabla 4-8. *Resultados en gráficas completas generando solo el total de cubiertas.*

TIPO DE GRAFICA	TIEMPO DE EJECUCIÓN	NODOS HOJA	TAMAÑO DE LOS ARCHIVOS CREADOS	NUMERO DE CUBIERTAS DE ARISTAS
4X4 COMPLETO	37.51 ms	3	-	38
5X5 COMPLETO	65.8659 ms	5	-	543
6X6 COMPLETO	83.9619 ms	43	-	25,704
7X7 COMPLETO	163.748 ms	173	-	1,447,320
8X8 COMPLETO	2.534 s	755	-	31,098,876
9X9 COMPLETO	8.147 s	3112	-	5,179,245,574
10X10 COMPLETO	40.11 s	13002	-	36,181,185,762
11X11 COMPLETO	2.48 min	55578	-	102,296,041,959
12X12 COMPLETO	5.382 min	237757	-	433,057,590,578
13X13 COMPLETO	23.16 min	1062532	-	1,233,368,905,968
14X14 COMPLETO	1.893 horas	4685143	-	2,123,3,874,592,557

La Figura 4-12 muestra el crecimiento en tiempo de ejecución, comprobando el comportamiento exponencial con la ecuación $y = 0.077e^{1.2866x}$ con un rango de error de 0.9793.

Figura 4-12. *Crecimiento en tiempo de ejecución sin generar la salida.*

4.4 Conclusiones

De acuerdo a los resultados obtenidos, hay varias cosas que se pueden mencionar del funcionamiento del algoritmo, una de las más evidentes desde las primeras fases es que el tiempo de ejecución y el uso de memoria comparten el mismo comportamiento, esto quiere decir que si uno crece exponencialmente el otro también lo hará.

El hecho de que el algoritmo funcione de forma lineal en las gráficas cactus indica que el número de vértices o aristas de la gráfica no es determinante, sino, que lo que realmente incrementa su complejidad de análisis y por lo tanto el uso de recursos del algoritmo de forma significativa es el número de ciclos intersectados.

Un claro ejemplo de esto se puede observar cuando una gráfica cactus de más de 500 vértices tiene un tiempo de ejecución mucho menor a la una gráfica completa de 9x9, y esta idea se refuerza al no encontrar un patrón de crecimiento si solo nos fijamos en el número de vértices o aristas, el factor que define que tan grande o pequeño es el tiempo de ejecución sobre una gráfica es la cantidad de ciclos intersectados.

Obviamente esto no quiere decir que no afecte en nada el número de vértices y aristas, una entrada grande tardara más en procesarse que una pequeña, pero este factor no es tan determinante, en otras palabras un aumento en el número de ciclos intersectados origina un incremento exponencial, mientras un incremento en el número de vértices y aristas origina un incremento lineal, siempre y cuando ese aumento no origine más ciclos intersectados.

Esto es producto de que el algoritmo de conteo tenga un tiempo polinómico de ejecución y el algoritmo para procesar los algoritmos con ciclos intersectados tenga un tiempo de tiempo de un límite exponencial bajo, por lo tanto un incremento en los procesos que debe hacer uno u otro refleja su respectivo incremento.

El hecho de que el algoritmo mantenga su funcionamiento constante y cumpla con lo esperado permite decir que se pueda generar una ecuación característica de crecimiento que permite deducir los valores que se obtendrán con ejemplos que sería muy tardado de procesar, estas se encuentran junto a las gráficas que representan los diferentes crecimientos tanto en tiempo como uso de memoria.

Todas estas características de la implementación hacen que su aplicación para resolver el problema de conteo de cubiertas sea eficaz al ofrecer un algoritmo exacto con una complejidad de límite exponencial inferior, la cual sin embargo no llega a ser una solución completamente optima, ya que no reduce la complejidad al ideal polinómico.

El mérito de este acercamiento es sentar las bases y dar los primeros pasos para alcanzar el nivel de complejidad idóneo, esta es solo una pequeña pero que puede llegar a ser muy significativa si las líneas de investigación siguen abiertas y expandiéndose.

4.5 Trabajos Futuros

El tema del conteo de coberturas de aristas así como la mayoría de los problemas #P son muy poco investigados en pos de sus contrapartes en los problemas de decisión, es por esto que las líneas de investigación y trabajos futuros para este tema son muy extensos debido al poco interés e investigaciones realizadas.

Siguiendo esta línea de investigación encontrar un modo de mejorar el algoritmo para cuando se tengan que procesar graficas que tengan un gran cumulo de ciclos intersectados en alguna parte, como las gráficas completas, sería una gran mejora, ya que el algoritmo aunque las procesa adecuadamente su uso de recursos es alto, de ahí que la complejidad sea exponencial, y si este proceso sea realiza más eficientemente se podría reducir esta complejidad aún más.

Otro camino podría ser desarrollar un algoritmo exacto completamente desde cero, para acercarse a este problema desde otra perspectiva, y ya que no hay una sola solución correcta, a lo mejor este nuevo enfoque puede ofrecer una mejor solución con algo de investigación.

Sin embargo un algoritmo exacto no tiene por qué ser el único camino, un enfoque o algoritmo de aproximación también puede ser viable como alternativa, aunque siempre existirá el debate de a que enfoque deben destinarse más recursos y tiempo, lo ideal es que sea cual sea el enfoque avanzar siempre hacia adelante en la resolución de este complicado problema.

Al final lo importante es que los investigadores se adentren a estos temas tan poco estudiados aun dentro de la comunidad científica, y que poco a poco usando el enfoque de su preferencia se vayan acercando cada vez más a su resolución en una complejidad polinómica ideal.

Anexos

a) Código para la creación del sistema de archivos

/ Descripción del Método: Obtiene el directorio donde se encuentra el código fuente y genera el sistema de archivos raíz del sistema**

Parámetros de Entrada: Ninguno

Retorno: Ninguno*/

```
void DefinirRutas(){
  stringstream rutaprincipal;
  char buffer[1000];
  getcwd(buffer, 1000); /*Obtiene el directorio donde se encuentra el código fuente*/
  string ruta(buffer); /*Asigna el directorio raíz a la variable buffer */
  rutaprincipal<<ruta<<"/Coberturas";  /*Crea la ruta del directorio Coberturas sobre el directorio raíz*/
  rutaR=rutaprincipal.str() ; /*Asigna el directorio Coberturas a la variable rutaR*/
  if(ExisteArchivo(rutaR)==false){ /*Si el directorio Coberturas no existe se crea*/
    CrearCarpeta(rutaR);
  }
  stringstream rutaimagenes;
  rutaimagenes<<rutaR<<"/IMAGENES";  /*Crea la ruta del directorio IMAGENES sobre el directorio raíz*/
  rutaI=rutaimagenes.str(); /*Asigna el directorio IMAGENES a la variable rutaI*/
  if(ExisteArchivo(rutaI)==false){ /*Si el directorio IMAGENES no existe se crea*/
    CrearCarpeta(rutaI);
  }
  stringstream rutadot;
  rutadot<<rutaR<<"/DOT"; /*Crea la ruta del directorio DOT sobre el directorio raíz*/
  rutaD=rutadot.str(); /*Asigna el directorio DOT a la variable rutaD*/
  if(ExisteArchivo(rutaD)==false){ /*Si el directorio DOT no existe se crea*/
    CrearCarpeta(rutaD);
  }
  stringstream rutarchivos;
  rutarchivos<<rutaR<<"/Archivos"; /*Crea la ruta del directorio Archivos sobre el directorio raíz*/
  rutaA=rutarchivos.str(); /*Asigna el directorio Archivos a la variable rutaA*/
  if(ExisteArchivo(rutaA)==false){ /*Si el directorio Archivos no existe se crea*/
```

```
    CrearCarpeta(rutaA);
  }
}
```

b) Código para el conteo de cubiertas en gráficas sin ciclos intersectados

/ Descripción del Método: Obtiene el número de coberturas de aristas en gráficas sin ciclos intersectados**

Parámetros de Entrada: El árbol de expansión de la gráfica de entrada y su complemento

Retorno: Numero de Coberturas de Aristas de la Gráfica de Entrada*/

```
int ConteoCubiertas2(vector<pair<int,ii> > grafo, vector<pair<int,ii> > complemento){
  vector< pair<int,int> > Subniveles;
  vector<pair<int,int> > CUBIERTAS;
  pair<int,int> Cinicial;
  int raiz =grafo.at(0).second.first; /*Se selecciona el primer elemento del árbol como raíz*/
  Cinicial.first=0; /*Se inicializa el valor alfa en cero*/
  Cinicial.second=0; /*Se inicializa el valor beta en cero*/
  complemento=LimpiarComplemento(complemento); /*Se inicializan los valores auxiliares del complemento*/
  for(int i=0;i<grafo.size();i++){
    CUBIERTAS.push_back(Cinicial); /*Asignar a cada elemento del árbol un par alfa - beta*/
    grafo.at(i).first=0; /*Inicializar el valor auxiliar de cada elemento del árbol*/
    Subniveles=LlenarSubnivel(Subniveles, grafo.at(i)); /*Agregar al arreglo cada vértice que sea un Subnivel del árbol*/
  }
  Subniveles=AjustarSubniveles(Subniveles); /*Eliminar los vértices que no sean un subnivel del árbol*/
  for(int i=grafo.size()-1;i>=0;i--){ /*Recorrer todos los elementos del árbol*/
    if(EsHoja(grafo,i,Subniveles,CUBIERTAS)==true){ /*Si el elemento es una hoja del árbol*/
      if(EsBackEdge(grafo.at(i),complemento)==true){ /*Si el elemento es una arista de retroceso*/
          CUBIERTAS.at(i).first=1;  /*Asignar el valor alfa en 1*/
          CUBIERTAS.at(i).second=1; /*Asignar el valor beta en 1*/
          grafo.at(i).first=1; /*Indicar que el elemento ya es parte de la arista de retroceso*/
          complemento=InicioBackEdge(complemento,grafo.at(i).second.second); /*Indicar en el auxiliar del complemento que ese elemento inicia una arista de retroceso*/
      }else{ /*Si no es arista de retroceso*/
```

```
        CUBIERTAS.at(i).first=1; /*Asignar el valor alfa en 1*/
        CUBIERTAS.at(i).second=0; /*Asignar el valor beta en 1*/
    }
  }else{ /*El elemento no es un nodo hoja del arbol*/
        if(EsSubNivel(grafo.at(i).second.second,Subniveles)==true){  /*Si el nodo es un Subnivel del árbol*/
        if(SubnivelCompleto(grafo,i,Subniveles)==true){ /*Si el Subnivel ya se completó*/
         /*Calcular el valor alfa del nodo usando el valor de todos sus hijos*/
          CUBIERTAS.at(i).first=CalcularV1Sb(CUBIERTAS,i,Subniveles,grafo,grafo.at(i).second.second);
        /*Calcular el valor beta del nodo usando el valor de todos sus hijos*/
          CUBIERTAS.at(i).second=CalcularV2Sb(CUBIERTAS,i,Subniveles,grafo,grafo.at(i).second.second);
          if(ExisteBackEdgeSb(grafo.at(i).second.second,grafo)==true){    /*Si es parte de una arista de retroceso*/
            grafo.at(i).first=1; /*Marcar el elemento como parte una arista de retroceso*/
          }
          if(EsInicioBackEdgeSb(grafo.at(i),complemento)==true){ /*Si es inicio de una arista de retroceso*/
            grafo.at(i).first=1; /*Marcar el elemento como parte una arista de retroceso*/
           complemento=InicioBackEdgeSb(complemento,grafo.at(i).second.second); /*Indicar en el auxiliar del complemento que ese elemento inicia una arista de retroceso*/
          }
          if(EsFinBackEdgeSb(grafo.at(i), complemento)==true){ /*Si es fin de una arista de retroceso*/
            int a=CUBIERTAS.at(i).first;
            int b=CUBIERTAS.at(i).second;
            CUBIERTAS.at(i).first=a+b; /*Ajustar el valor alfa del elemento*/
            CUBIERTAS.at(i).second=a+1; /*Ajustar el valor beta del elemento*/
           complemento=FinBackEdgeSb(complemento,grafo.at(i).second.first); /*Corregir los auxiliares del complemento*/
            grafo.at(i).first=0; /*Marcar que el elemento ya no es parte de una arista de retroceso*/
          }
          /*Eliminar del arreglo Cubiertas las aristas de los Subniveles que ya se visitaron*/
          CUBIERTAS=CorregirCUSb(CUBIERTAS,i,Subniveles,grafo,grafo.at(i).second.second);
          /*Eliminar del arreglo las aristas de los Subniveles que ya se visitaron*/
          grafo=CorregirARISb(grafo,i,Subniveles,grafo.at(i).second.second);
         /*Eliminar los subniveles que ya se procesaron*/
          Subniveles=EliminarSubnivel(Subniveles,grafo.at(i));
        }
```

```
    }else{          /*Si no es ninguno de los casos anteriores*/
          /*Calcular el valor alfa del elemento*/
          CUBIERTAS.at(i).first=CalcularV1(CUBIERTAS,i,Subniveles,grafo);
          /*Calcular el valor beta del elemento*/
          CUBIERTAS.at(i).second=CalcularV2(CUBIERTAS,i,Subniveles,grafo);
          if(grafo.at(i+1).first==1){ /*Si el elemento ya era parte de una arista de retroceso*/
            grafo.at(i).first=1; /*Marcar el elemento como parte la arista de retroceso*/
          }
          if(EsInicioBackEdge(grafo.at(i), complemento)==true){ /*Si es inicio de una arista de retroceso*/
            grafo.at(i).first=1; /*Marcar el elemento como parte la arista de retroceso*/
            complemento=InicioBackEdge(complemento,grafo.at(i).second.second); /*Marcar el inicio de la
arista de retroceso en el complement*/
          }
          if(EsFinBackEdge(grafo.at(i), complemento)==true){ /*Si es fin de una arista de retroceso*/
            int a=CUBIERTAS.at(i).first;
            int b=CUBIERTAS.at(i).second;
            CUBIERTAS.at(i).first=a+b; /*Ajustar el valor alfa del elemento*/
            CUBIERTAS.at(i).second=a+1; /*Ajustar el valor beta del elemento*/
            complemento=FinBackEdge(complemento,grafo.at(i).second.first); /*Marcar el fin de la arista de
retroceso en el complemento*/
            grafo.at(i).first=0; /*Marcar que el elemento ya no es parte de la arista de retroceso*/
          }
          CUBIERTAS=CorregirCU(CUBIERTAS,i,Subniveles,grafo); /*Eliminar las cubiertas que ya han
sido procesadas*/
          grafo=CorregirARI(grafo,i,Subniveles); /*Eliminar las aristas que ya han sido recorridas*/
    }
   }
  }
   int V1R=CalcularV1R(CUBIERTAS); /*Corregir el valor alfa de la raíz*/
   int V2R=CalcularV2R(CUBIERTAS); /*Corregir el valor beta de la raíz*/
   return V1R; /*Regresar el valor alfa de la raíz*/
}
```

c) Código para gráficas con ciclos intersectados

/ Descripción del Método: Procedimiento que se aplica en las gráficas con ciclos intersectados para eliminar los ciclos.**

Parámetros de Entrada: El árbol de expansión de la gráfica de entrada, su complemento y el contador de gráficas.

Retorno: Ninguno*/

```
void DoReducir(Conjunto grafo ,vector<pair<int,ii> > complemento, int cont){

    vector<pair<int,ii> > comp;

    vector<pair<int,ii> > VCiclo=ObtenerVCiclos(grafo.spt,complemento); /*Obtener el vector de ciclos*/

    if(complemento.size()>1 && ComVCiclos(VCiclo)){ /*Si hay más de una arista en el complemento y la gráfica tiene ciclos intersectados*/

        pair<int, ii> nodo = RecuVMAXVCiclo(VCiclo); /*Obtener arista que pertenece  a mas ciclos*/

        vector<pair<int,ii> > Bosque = Split(grafo.grafo,nodo); /*Aplicar regla de división sobre la arista escogida*/

        grafo.grafo=EliminarArista(grafo.grafo,nodo); /*Aplicar regla de reducción sobre la arista escogida*/

        grafo.spt.clear(); /*Eliminar el árbol de expansión*/

        grafo.grafo=CorregirEntrada(grafo.grafo); /*Ordenar las aristas de la gráfica de entrada*/

        grafo=DFS(grafo); /*Obtener un nuevo árbol de expansión*/

        comp=RecComp(grafo.grafo,grafo.spt); /*Obtener un nuevo complemento*/

        if(Salida==true){ /*Si el algoritmo genera las gráficas intermedias*/

            CrearImagenReduccion(grafo.grafo,cont);  /*Crear el archivo .dot de la gráfica*/

            GraficarImagenReduccion(cont);   /*Generar la imagen de la gráfica*/

        }

        AnalizarBosque(Bosque); /*Comprobar si la nueva gráfica genero un Bosque*/

        cont++; /*Aumentar el contador de gráficas intermedias generadas*/

        rutaD = ReducirRuta(rutaD); /*Ajustar la ruta de los archivos .dot*/

        rutaI = ReducirRuta(rutaI); /*Ajustar la ruta de las imágenes generadas*/

        DoReducir(grafo,comp,cont); /*Aplicar la regla de reducción sobre la nueva grafica generada*/

    }else{ /*Si la gráfica no tiene ciclos intersectados*/

            NumHojas++; /*Aumentar en 1 el número de hojas*/

            int total=ConteoCubiertas2(grafo.spt,complemento); /*Aplicar el algoritmo de conteo */

            if(EsunForest==false){ /*Si la gráfica no es parte de un Bosque*/

              mpz_add_ui(TotalCubiertas,TotalCubiertas,total); /*Sumar el número de coberturas al total*/

            }else{ /*Si la gráfica es parte de un Bosque*/

              TotalSubCubiertas=TotalSubCubiertas*total; /*Multiplicar el número de coberturas al subtotal*/
```

```
            }
        }
}
```

d) Código para generar las estadísticas.

/*Descripción del Método: Generar las estadísticas obtenidas de la ejecución del algoritmo

Parámetros de Entrada: El tiempo de ejecución del algoritmo en segundos

Retorno: Ninguno*/

```
void GenerarEstadisticas(double s){
    FILE *estadistica;
    stringstream rutaarch;
    stringstream pesosarch;
    rutaarch<<rutaA<<"/Estadisticas"; /*Generar la ruta del archivo Estadísticas*/
    string rutaa=rutaarch.str(); /*Asignar la ruta a la variable ruta*/
    char *ruta = new char[rutaa.length() + 1];
    strcpy(ruta, rutaa.c_str());
    estadistica=fopen(ruta,"w");  /**Crear el archivo Estadísticas*/
    fclose(estadistica);
    estadistica=fopen(ruta,"a"); /**Abrir el archivo Estadísticas para escritura*/
    if(estadistica!=NULL){ /*Si el archivo se abrió exitosamente*/
      if(Salida==true){ /*Si las gráficas intermedias se crearon*/
        CalcularTamano(rutaa); /*Calcular el uso de memoria del algoritmo*/
      }
      fprintf(estadistica,"Tiempo de ejecucion: %.16g millisegundos\n", s * 1000.0); /*Agregar tiempo de ejecución al archivo*/
      fprintf(estadistica,"Numero de Nodos Hojas: %d\n",NumHojas); /**Agregar el número de hojas al archivo*/
      fprintf(estadistica,"Total de Cubiertas de Aristas: "); /*Agregar el total de cubiertas al archivo*/
      mpz_out_str(estadistica,10,TotalCubiertas);
    }else{
      printf("NO SE PUDO CREAR EL ARCHIVO DE ESTADISTICAS\n");
    }
    fclose(estadistica); /*Cerrar el archivo Estadísticas*/
}
```

Referencias Bibliográficas

Arora, S. & Barak, B., 2007. *Computational Complexity: A Modern Approach.* Primera ed. Princeton: Princeton University.

Baase, S. & Van Gelder, A., 2002. *Algoritmos Computacionales Introducción al Análisis y Diseño.* Tercera ed. México: PEARSON Educación.

Bin, X. & Zhongyi, Z., 2010. *Graph Theory.* Tercera ed. Shanghai: World Scientific Pub Co Inc.

Bondy, J. A. & Murty, U. S. R., 2007. *Graph Theory.* San Franciso: Springer-Verlag.

Brualdi, R. A., 2009. *Introductory Combinatorics.* Quinta ed. s.l.:Pearson Education Asia Limited and China Machine Press.

Corona Nakamura, M. A. & Ancona Valdez, M. d. l. A., 2011. *Diseño de Algoritmos y su Codificación en Lenguaje C.* Primera ed. Ciudad de México: McGRAW-HILL.

Flores Lamas, A. (2014). *Algoritmo para encontrar el conjunto independiente fuerte máximo sobre un grafo tipo cactus*. Maestría. CENTRO DE INVESTIGACION CIENTIFICA Y DE EDUCACION SUPERIOR DE ENSENADA.

Goldenberg, D. & Galván, A., 2015. The use of functional and effective connectivity techniques to understand the developing brain. *Elsevier,* p. 10.

González-Díaz, H., Pérez-Montoto, L. & Paniagua, E., 2009. Generalize dlattice graphs for 2D-visualization of biological information. *Elsevier,* p. 12.

Gross, J. L., 2009. *Universidad de Columbia.* [En línea]
Available at: http://www.cs.columbia.edu/~cs4205/files/CM2.pdf
[Último acceso: 21 Septiembre 2016].

Heckmann, T., Schwanghart, W. & D. Phillips, J., 2014. Graph theory—Recent developments of its application in geomorphology. *Elsevier,* p. 17.

Hernández Servín, J., Marcial Romero, J. & De Ita, G., 2014. *Low exponential algorithm for counting the number of edge cover on simple graphs.* s.l., s.n., pp. 1-8.

Hopcroft, J. E., Motwani, R. & Ullman, J. D., 2007. *Teoría de autómatas, lenguajes y computación.* Tercera ed. Madrid: PEARSON EDUCACIÓN S.A..

Jiang, Y., Xia, Z. & Zhong, Y., 2014. Autonomous trust construction in multi-agent systems—a graph theory methodology. *Elsevier,* p. 8.

Joyanes Aguilar, L., 2008. *Fundamentos de Programación Algoritmos, estructuras de datos y objetos.* Cuarta ed. Madrid: Mac Graw Hill.

Jurado Málaga, E., 2008. *Teorías de Autómatas y Lenguajes Formales.* Primera ed. Cáceres: Universidad de Extremadura.

Laboratorio Nacional de Calidad del Software de la INTENCO, 2009. *INGENIERÍA DEL SOFTWARE: METODOLOGÍAS Y CICLOS DE VIDA.* Primera ed. Madrid: INTENCO.

LEE, R., TSAI, Y., CHANG, R. C. & TSENG, S., 2007. *Introducción al diseño y análisis de algoritmos Un enfoque estratégico.* Primera ed. Taiwan: McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V..

Lin, C., Liu, J. & Lu, P., 2014. A Simple FPTAS for Counting Edge Covers. p. 13.

Lipschutz, S. & Lipson, M., 2007. *Discrete Mathematics.* Tercera ed. s.l.:MacGraw-Hill.

McGlaughlin, A., Soldan, A. & Phillips, J. D., 2015. Graph theoretic analysis of structural connectivity across the spectrum of Alzheimers disease: The importance of graph creation methods. *Elsevier,* p. 14.

Núñez, J., Silvero, M. & Villar, M. T., 2012. Mathematical tools for the future: Graph Theory and graphicable algebras. *Elsevier,* p. 13.

Quiroga Rojas, E. A., 2008. *AUTÓMATAS Y LENGUAJES FORMALES.* Primera ed. Bogotá: UNIVERSIDAD NACIONAL ABIERTA Y A DISTANCIA – UNAD.

Schroeder Barwaldt, M., de Fátima Franzin, R. & Vanderlei dos Santos, A., 2014. Using the theory of graphs on the implementation of bike lane in small towns. *Elsevier,* p. 9.

Schwanghart, W., Heckmann, T. & Phillips, J. D., 2015. Graph theory in the geosciences. *Elsevier,* p. 14.

Thomas, C. J., Lambrechts, J., Wolanskic, E. & Deleersnijdera, E., 2013. Numerical modelling and graph theory tools to study ecologicalconnectivity in the Great Barrier Reef. *Elsevier,* p. 15.

Wanga, Y.-c. & Önal, H., 2011. Designing connected nature reserve networks using a graph theory approach. *Elsevier,* p. 6.