
*BENEMÉRITA UNIVERSIDAD AUTONOMA*
*DE PUEBLA*
*FACULTAD DE CIENCIAS DE LA COMPUTACIÓN*

# *"COLOREO DE GRAFOS EMBEBIDOS"*

*TESIS PROFESIONAL*
*PARA OBTENER EL TITULO DE*
*INGENIERO EN CIENCIAS DE LA COMPUTACIÓN*

*PRESENTA*

*ISAAC CHANTES QUECHOL*

*ASESOR*

*DR. GUILLERMO DE ITA LUNA*

*JUNIO 2008*

# *AGRADECIMIENTOS*

“A Dios por darme esta satisfacción de terminar mis estudios, por todos aquellos momentos difíciles en que no encontraba una respuesta clara en mi vida y el siempre ha esta ahí guiando mis pasos sin dejarme solo y poner en mi camino a personas tan maravillosas como mi familia, mi novia y mis amigos.”

“A mi mamá Rosy, por todo su amor, su apoyo, su cariño y motivaciones, por todos los principios y valores sembrados en mi para mi formación, y sobre todo por los regaños que hicieron de mi una mejor persona.”

“A mi papá Marce, por el amor, compresión, por enseñarme mis responsabilidades como hijo y por todo ese valiosos esfuerzo por darme todo lo necesario para salir adelante.”

“A mi hermano Noe porque siempre tiene una manera tan especial de expresarme cuanto me quiere y por apoyarme en todo lo que esta a su alcance. “

“A mi novia Moni por todo ese amor y paciencia que me ha tenido, por su apoyo incondicional que me brindo para que llegara hasta este punto tan importante en mi vida, y por compartir conmigo todo este tiempo.”

“A mis amigos Marco, Gerardo, Rodolfo, Palacios, Javier, Gilberto, Miguel, Fermín, Israel, Claudia, Doris, Dulce, Anuar, Betanzos, Luís, Montoya, a todos ellos gracias brindarme su amistad incondicional.”

“A mi tutor de tesis y profesor por el apoyo, paciencia y confianza que me brindo en este tiempo, muchas gracias Dr. Guillermo de Ita.”

# CONTENIDO

# 1. INTRODUCCION

El problema de coloreo de grafos es uno de los problemas clásicos de la teoría de grafos y ha sido estudiado desde el siglo XIX [5]. Más allá de su interés teórico, el problema de coloreo de un grafo tiene diversas aplicaciones prácticas debido a las numerosas situaciones de la vida real en las cuales surgen problemas de calendarización excluyente de tareas.

Los grafos son objetos matemáticos utilizados para modelar situaciones reales diversas. Por ejemplo, un mapa de carreteras, un plano de la red del metro de una ciudad, un plano de algún circuito eléctrico, etc., todos estos escenarios son adecuadamente modelados a través de grafos.

La teoría de grafos ha sido un área fértil para el planteamiento y modelación de problemas en áreas diversas. Aún cuando esta teoría nace en el área de las matemáticas, la teoría de grafos ha jugado un papel relevante en la fundamentación de las Ciencias de la Computación. Más aún, los grafos constituyen una herramienta básica para modelar fenómenos discretos y para la comprensión de las estructuras de datos y el análisis de algoritmos que se proponen en la solución computacional de los problemas planteados.

Uno de los problemas más referenciado en el área de teoría de grafos, es el problema del coloreo de los vértices de un grafo. En este problema se desea asignar a cada vértice del grafo un color único. Con la restricción de que cada par de vértices conectados por una arista deben tener colores diferentes. El número cromático de un grafo $G$ es denotado por $\chi(G)$ y representa el número mínimo de colores necesarios para colorear el grafo $G$.

El problema de coloreo de grafos es una abstracción de cierto tipo de calendarización de tareas. El problema de determinar para cualquier grafo de entrada $G$ su número cromático $\chi(G)$, ha sido, contra su sencillez de planteamiento, uno de los problemas más difíciles de resolver, debido principalmente al tiempo de computación que ha requerido todo algoritmo que se ha diseñado hasta ahora y que encuentra de manera exacta el número mínimo de colores para colorear un grafo.

Nuestro interés en este trabajo de tesis, se centra en el 3-coloreo de un grafo. Se dice que un grafo $G$ es 3-coloreable cuando su número cromático $\chi(G)$ es 3 [1]. Determinar si $\chi(G)$ es 2 se puede realizar computacionalmente de manera *eficiente*. Pero el problema de dado un grafo $G$ de entrada, determinar si es 3- coloreable o no, es un problema, que a la fecha, no se ha podido resolver de manera eficiente, esto es, toda propuesta algorítmica que lo resuelve exactamente requiere de un tiempo exponencial de cómputo sobre el tamaño del grafo (número de vértices y aristas del grafo).

En este trabajo de investigación, nos abocamos a investigar sobre condiciones de suficiencia que nos permitan determinar cuando un grafo es 3-coloreable. Por ejemplo, se encuentra que si un grafo dado es bipartito entonces el grafo es 3-coloreable. Así también, mostramos que si el conjunto de ciclos impares en un grafo de entrada, se pueden expresar como ciclos independientes (sin aristas comunes) con cualquier otro ciclo o bien, que el conjunto de ciclos independientes pueden estar embebidos dentro de otros ciclos entonces el grafo es 3-coloreable.

Es relevante mencionar que como resultado de nuestras investigaciones, proponemos un nuevo algoritmo para el 3-coloreo de grafos que cumplen las condiciones determinadas por nosotros para que un grafo sea 3-coloreable. El algoritmo propuesto resulta ser un procedimiento eficiente (que corre en tiempo de computación acotado polinomialmente de acuerdo al número de nodos en el grafo), aunque no es un algoritmo completo, por lo que pueden existir grafos que no cumplan las condiciones establecidas, pero que si son 3-coloreables.

Para verificar si un grafo es 3-coloreable, aplicamos una serie de pasos, entre ellos, si es necesario se aplica una transformación sobre la topología del grafo para obtener un grafo equivalente (en términos del 3-coloreo) y tal que el nuevo grafo cumple las condiciones para que pueda aplicársele nuestro procedimiento de 3-coloreo.

## 1.1. ESTADO DEL ARTE

En 1852 Francis Guthrie [5] planteó el problema de los cuatro colores que cuestiona sobre la posibilidad de que utilizando solamente cuatro colores, se pueda colorear los mapas de cualquier país, de tal forma que dos países vecinos nunca tengan el mismo color. Este problema que no fue investigado sino hasta un siglo después por Kenneth Appel y Wolfgang Haken [5], puede ser considerado como el nacimiento de la teoría de grafos. Al tratar de resolverlo, los matemáticos definieron términos y conceptos teóricos fundamentales para la teoría de grafos.

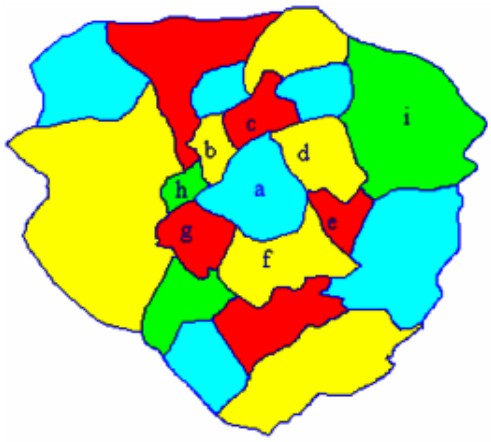



Fig. 1.2 En 1852 Francis Guthrie planteó el problema de los cuatro colores.

El problema de los cuatro colores también fue analizado posteriormente por algunos otros matemáticos, como por ejemplo, en un articulo el cual fue publicado y llamado "*The mathematics of map coloring*", publicado en 1969 por la *Journal of Recreational Mathemathics,* el matemático H. S. M. (Donald) Coxeter [5], mencionó que aproximadamente cuando se colorea un mapa de los Estados Unidos para distinguir a los estados vecinos, son necesarios al menos 5 o 6 colores. Entonces surge la pregunta, cual es el número mínimo de colores que pueden ser usados para colorear los estados de los Estados Unidos si cada dos estados que comparten un borde común deben tener colores diferentes.

Aunque, estados que comparten solo un punto en común, como es el caso de Utah y New México se les permite que tengan el mismo color. Podemos ver que Nevada y Utah son estados vecinos, esto es, ellos comparten un mismo limite, entonces se les debe asignar un color distinto. De hecho, Nevada esta rodeada de 5 estados vecinos, Utah, Idazo, Oregon, California y Arizona. Por consiguiente, cada uno de esos 5 estados se le debe de asignar un color distinto del usado por Nevada. Por otra parte, tres colores son necesarios para colorear los 5 estados que rodean a Nevada. Así que cuatro colores se necesitan para colorear esos 6 estados. Se ha identificado que todos los estados de los estados unidos pueden colorearse con 4 colores.


Fig. 1.3. Oeste de los EUA

Muchos se interesaron en el problema de los cuatro colores el cual fue revivido durante 1878 en una reunión de *Sociedad Matemática de London*. El 13 de junio de 1878 Arthur

Cayley [5] pregunto si el problema había sido resuelto. Una de las personas que asistió a la reunión era un brillante pero joven matemático llamado Alfred Bray Kempe. En el siguiente año, el 17 de junio de 1879 Kempe anuncio en la revista *Nature* que el había resuelto el problema y que cada mapa podría ser realmente coloreado con 4 colores o menos. Posteriormente, Cayley sugirió a Kempe que debería publicar su descubrimiento, en cual el lo hizo en 1879 en el segundo volumen de la *America Journal of Mathematics.*

La conjetura de que cada mapa pueda ser coloreado con 4 colores o menos colores llego a ser conocido como la ***conjetura de los cuatro colores*** [5]. Muchos creyeron que esta conjetura era cierta. Mas sin embargo Percy John Heawood publico en 1890 un articulo llamado "Teorema del coloreo de mapa" en donde el hacia notar un "defecto" en la solución de Kempe del problema de los cuatro colores. Heawood ofreció un contraejemplo en el caso donde una región X es rodeada por cinco regiones. El ejemplo de Heawood era solo un contraejemplo para la técnica de Kempe, no un contraejemplo de la conjetura de los cuatro colores. Aunque el método de Kempe no tuvo éxito, Heawood fue capaz de utilizar este método para probar que cada mapa puede ser coloreado con cinco colores o menos.

Con esto se observaba que solo en las regiones que estaban rodeadas por un anillo de 5 o menos regiones eran las que mostraban dificultades para ser coloreadas con sólo cuatro colores, entonces los matemáticos empezaron a dividir la conjetura. La idea era buscar un conjunto de configuraciones de regiones alrededor del conjunto de regiones dado que cada mapa contiene por lo menos una de estas configuraciones y si estas regiones están fuera del anillo pueden ser coloreadas con cuatro colores. Este conjunto fue referido como un *inevitable conjunto de reducción de configuraciones.*

Este concepto de reducibilidad fue introducido por uno de los matemáticos más conocido del siglo XX, George David Birkhoff.

## 1.2 RELEVANCIA DEL PROBLEMA

Gracias a la teoría de Grafos se pueden resolver diversos problemas como por ejemplo; la síntesis de circuitos secuenciales, contadores o sistemas de apertura.

Los grafos se utilizan también para modelar trayectos como el de una línea de autobús a través de las calles de una ciudad, en el que podemos obtener caminos óptimos para el trayecto aplicando diversos algoritmos como puede ser el algoritmo de Floyd.

Existen múltiples aplicaciones de la teoría de grafos, alguna de las mas importantes dentro de nuestra vida cotidiana es la aplicación para las intersecciones de las calles y avenidas donde existan los comúnmente llamados semáforos, en donde el problema reside en como dejar fluir el tráfico sin que existan colisiones o conflictos para seguir su trayectoria. En este problema, $V(G)$ representa las líneas o rutas y donde son insertados 2 vértices unidos por una arista si los vehículos en esas dos líneas no pueden cruzar de manera segura la intersección al mismo tiempo, ya que existe la posibilidad de un accidente. Este problema

se puede solucionar determinando el número cromático del grafo de intersecciones del tráfico vehicular.

La teoría de los grafos también ha servido de inspiración para las ciencias sociales, en especial para desarrollar un concepto no metafórico de "red social" que substituye los nodos por los actores sociales y verifica la posición, centralidad e importancia de cada actor dentro de la red. Esta medida permite cuantificar y abstraer relaciones complejas, de manera que la estructura social puede representarse gráficamente. Por ejemplo, la "red social" puede representar la estructura de poder dentro de una sociedad al identificar los vínculo (Aristas), su dirección e intensidad y da idea de la manera en que el poder se trasmite y a quienes.

Además la teoría de grafos se utiliza en la planificación de proyectos. Por ejemplo, Lothar Czayka [3], utiliza los grafos para modelar el crecimiento económico social en relación con la política de investigación y sus costos asociados. La aplicación matemática de la teoría de grafos y de la combinatoria para los problemas de planeación de investigación pretende hacer una contribución a la representación de información.

La teoría de grafos se plica en el campo de la planeación de la investigación, de la evaluación y la optimización de proyectos y para la planeación y control de la ejecución de proyectos. Pero además trata sobre la procuración de datos dentro de este campo de la investigación.

# 2. TEORIA DE GRAFOS

Se dice que la “Teoría de Grafos” tuvo sus inicios en 1736 cuando Euler consideró el problema general llamado Los puentes de Königsberg [5]. Esto fue 200 años antes de que el primer libro de Teoría de Grafos fuera escrito en 1936 por Köning. Desde entonces la teoría de grafos ha sido desarrollada dentro de una extensiva y popular rama de las matemáticas, y se ha aplicado a múltiples problemas en matemáticas, ciencias de la computación, y en algunas otras áreas.

El trabajo de Leonhard Euler sobre el problema de los puentes de Königsberg es considerado como uno de los primeros resultados de la teoría de grafos. También se considera uno de los primeros resultados topológicos en geometría (que no depende de ninguna medida). Este ejemplo ilustra la profunda relación entre la teoría de grafos y la topología.

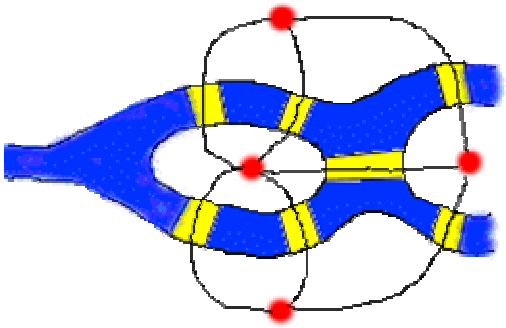


Fig. 1.1 Problema de los 7 puentes de Königsberg, 1736.

La ciudad de Königsberg, hoy Kaliningrado, se encuentra a orillas del Mar Báltico, en territorio Ruso y a unos 50 kilómetros de la frontera con Polonia. En el pasado perteneció a Prusia. Uno de sus habitantes más ilustres fue el filósofo Immanuel Kant. En Königsberg se juntan dos ríos formando una isla en su confluencia. Siete puentes unían (ya no, pues la ciudad fue parcialmente destruida durante la Segunda Guerra Mundial) las diferentes partes de la ciudad, como se aprecia en el mapa de la época. En el siglo XVIII se hizo popular como adivinanza o pasatiempo averiguar si era posible cruzar los siete puentes de la ciudad pasando sólo una vez por cada uno de ellos.

Este problema, por supuesto, puede resolverse mediante un estudio exhaustivo de todos los posibles itinerarios. Pero las matemáticas se interesan en generalizar el problema y buscar una solución sencilla y válida para todos los posibles mapas de ciudades, e incluso para

objetos más generales. Este problema consiste básicamente en encontrar un trayecto, alrededor de una serie de puentes que cruce solamente una vez por cada uno de ellos.

## 2.1. DEFINICIONES Y PRELIMINARES

En este capítulo presentamos notaciones estándar y definiciones para nuestro tema de estudio. Dado esto, podemos entender que los nombres utilizados para estos objetos reflejan la aplicación. Así, por ejemplo, si consideramos una red de comunicación, serán llamados nodos en lugar de vértices o puntos [2]. Por otra parte otros nombres son utilizados para las estructuras moleculares en química, diagramas de flujo en computación, relaciones humanas en ciencias sociales, etc.

El origen de la palabra grafo es griego y su significado etimológico es "trazar". Los grafos son utilizados con gran frecuencia como un medio para modelar el planteamiento y las propuestas de solución de problemas. Un grafo puede considerarse como un objeto geométrico aunque en realidad sea un objeto combinatorio, es decir, un conjunto de puntos (llamados vértices) y un conjunto de líneas que unen a estos puntos (llamadas aristas).

Debido a su generalidad y a la gran diversidad de formas en que pueden usarse, los grafos se han convertido en los objetos matemáticos ideales para modelar un sin número de problemas, desde problemas de optimización en fluidos hasta problemas de eficiencia en algoritmos de cómputo.

Un grafo es una pareja $G = (V, E)$, donde $V$ es un conjunto de puntos, llamados vértices, y $E$ es un conjunto de pares de vértices, llamadas aristas. Dos vértices $v$ y $w$ son llamados adyacentes si hay una arista $\{v, w\} \in E$ que los conecta [1].

Dado que los grafos son estructuras de datos no lineales que tienes su naturaleza generalmente dinámica. Para su estudio los grafos pueden dividirse en dos grupos:

- Grafos no dirigidos: denotaremos que es un grafo no dirigido cuando una arista entre los vértices $v, w$ como $\{v,w\}$. Con esto tenemos que al ser una arista no dirigida entonces $\{v,w\}=\{w,v\}$.
- Grafos dirigidos: denotamos como un grafo dirigido cuando una arista entre los vértices $v, w$ como un par ordenado $(v,w)$, donde en tal arista dirigida se tiene $(v,w)\neq(w,v)$. Esta se considera como una flecha dirigida que parte de $v$ y llega a $w$, y se denota como $v \rightarrow w$.

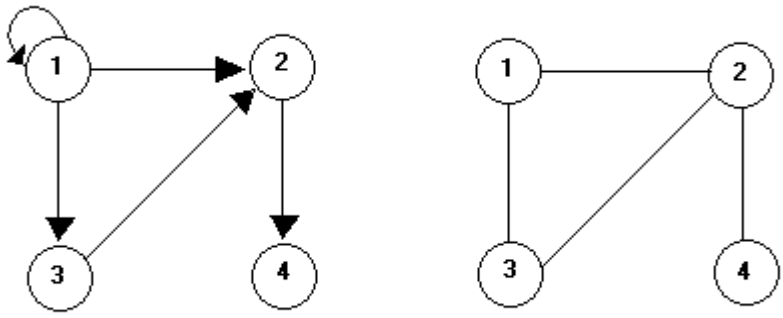



Fig. 2.6. Izquierda: Grafo Dirigido, Derecha: Grafo no dirigido

Los grafos pueden representarse de varias formas, por ejemplo, un grafo no dirigido puede representarse a través de una grafica (Fig. 2.1.) y de forma matricial (Fig. 2.2).

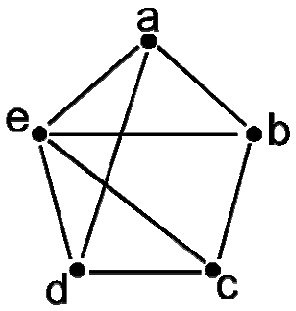



Fig. 2.1. Representación gráfica de un grafo

| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 | 1 |
| b | 1 | 0 | 1 | 0 | 1 |
| c | 0 | 1 | 0 | 1 | 1 |
| d | 1 | 0 | 1 | 0 | 1 |
| e | 1 | 1 | 1 | 1 | 0 |

Fig. 2.2. Representación matricial de un grafo

Sea $G= (V, E)$ un grafo no dirigido con un conjunto de vértices $V$ y un conjunto de aristas no dirigidas $E$. Un vértice $x$ es **incidente** (o **adyacente**) a un vértice $y$ si existe una arista que vaya de $x$ a $y$. Entonces, dos vértices $v$ y $w$ son llamados adyacentes si hay una arista $\{v, w\} \in E$ que los conecta. La vecindad de un vértice $x \in V$ es $N(x) = \{y \in V \mid \{x, y\} \in E\}$, y la vecindad cerrada de $x \in V$ es $N[x] = N(x) \cup \{x\}$. Denotaremos con $|A|$ a la cardinalidad del conjunto $A$. El grado de un vértice $x$, denotado por $\delta(x)$, es $|N(x)|$ y el grado del grafo $G$ es: $\Delta(G) = max\{\delta(x) \mid x \in V\}$ [2].

Tanto a las aristas como a los vértices se les puede asociar información. A esta información se le llama etiqueta. Si la etiqueta que se asocia es un número se le llama peso, costo o longitud. Un grafo cuyas aristas o vértices tienen pesos asociados recibe el nombre de

**grafo etiquetado o ponderado**. El número de elementos de *V* se denomina **orden** del grafo. Un **grafo nulo** es un grafo de orden cero.

En un grafo dirigido, si $(x, y) \in E$, a *x* se le denomina origen del arco y a *y* extremo del mismo. Se dice que dos arcos son adyacentes entre sí cuando tienen un vértice común que es a la vez origen de uno y extremo del otro. En el caso de que el grafo sea no dirigido si *x* es incidente a *y* entonces *y* también es incidente a *x*.

En un grafo dirigido, se denomina **grado de entrada** de un vértice *x* al número de arcos que arriban a él y se denota como: $\delta_{in}(x)$. Se denomina **grado de salida** de un vértice *x* al número de arcos que salen de él y se denota como: $\delta_{out}(x)$.

El *camino* de un vértice *v* a un vértice *w* en un grafo es una secuencia de aristas: $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$ donde $v=v_0$ y $v_n=w$ y $v_k$ es adyacente a $v_{k+1}$ para $0 \leq k \leq n$ y la longitud del camino es *n* [12]. Un camino simple es un camino tal que cada uno de los vértices del camino es diferente uno del otro. Cualquier camino *x*-*y* donde $x = y$ (y con $n > 1$) es un *camino cerrado,* al que denominaremos *ciclo*. Un ciclo simple es un camino cerrado en donde ningún vértice se repite, con excepción del primero y último vértice del ciclo.

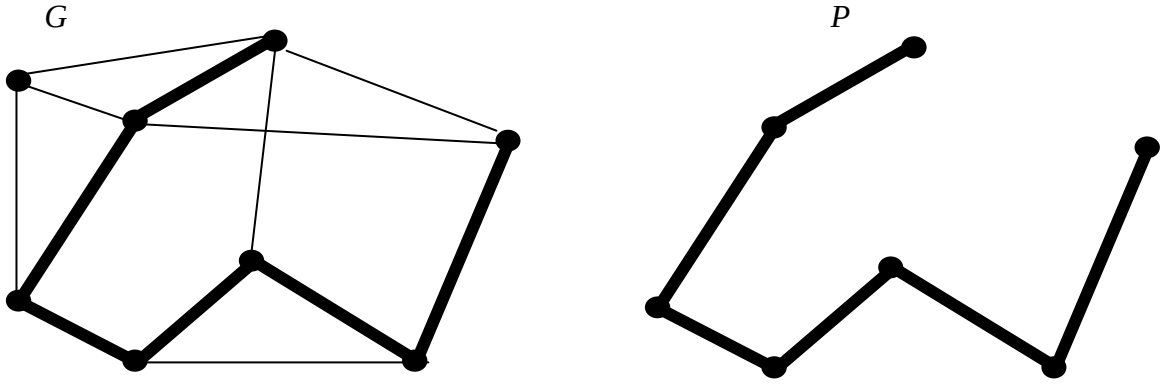



Fig. 2.4. *P* es un camino del grafo *G*

Sea $G = (V, E)$ un grafo $|v| = n$, $|E| = m$, $S \subseteq V$ es un *conjunto independiente* en *G* si es que para cualesquiera dos elementos (vértices) que hay en *S* no son adyacentes. Se define como *I*(*G*) al conjunto que contiene todos los conjuntos independientes que hay en *G*. Se dice que $S \in I(G)$ es maximal si *S* no es subconjunto de cualquier otro conjunto independiente y S es máximo si S tiene la longitud mas grande entre todos los conjuntos independientes en *I*(*G*). Entonces si *S* es un conjunto independiente máximo, si $S_{max} = |\max |S_i|| : S_i \in I(G)$, a $|S_{max}|$ se le denota como $\beta(G)$ y es llamado el numero de vértices independientes o el numero independiente.

## 2.2. CLASES DE GRAFOS

Existe una clase particular de grafos, a los que se les llama árboles, que juegan un papel importante en el estudio tanto de algoritmos como de estructura de datos, ya que han resultado ser extremadamente útiles para guardar y manipular información. Un grafo *G* se dice que es un árbol si y solo si se verifica que *G* es conexo y que *G* no posee ciclos. Es inmediato probar que si un grafo *G* no tiene ciclos entonces puede verse como la unión disjunta de árboles.

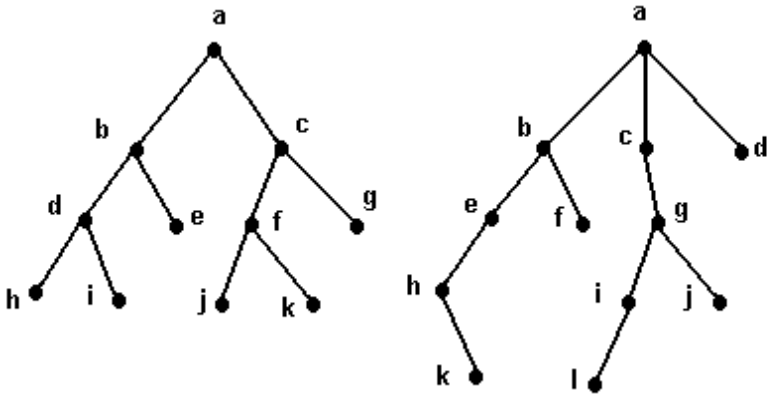



Fig. 2.3. Ejemplo de árboles

Se denomina **camino lineal** (algunos autores le llaman **cadena** si se trata de un grafo no dirigido), y lo denotaremos con $P_n$, =($V$,$E$), al grafo dirigido formado el conjunto de $n$+1 nodos $V = \{v_i : i=1,\ldots,n+1\}$ y la sucesión de $n$ arcos adyacentes: $P_n = \{(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},v_n), (v_n,v_{n+1})\}$.

Un ciclo es un camino no vacío donde el primer y último vértice son los mismos, y un ciclo simple es un ciclo en el cual ningún vértice se repite, con excepción del primero y último nodo. Un camino se dice **Euleriano** si es simple y además contiene a todos los arcos del grafo.

Un $k$-ciclo, denotado por $C_k$, es un ciclo de longitud igual a $k$, es decir, un $k$-ciclo tiene $k$ aristas. Un ciclo de longitud impar es llamado ciclo impar, mientras que un ciclo de longitud par es llamado ciclo par. Un grafo es acíclico si no tiene ningún ciclo.

Un grafo se denomina **simple** si no tiene bucles y no existe más que un camino para unir dos nodos. Un circuito elemental que incluye a todos los vértices de un grafo lo llamaremos circuito **Hamiltoniano**.

Un subgrafo de $G$=($V$, $A$) es a su vez un grafo $G'$= ($V'$, $E'$), con $V' \subseteq V$, y tal que $E'$ contiene aristas $\{v,w\} \in A$ tal que $v \in V'$ y $w \in V'$. Si $E'$ contiene a toda arista $\{v, w\} \in A$ donde $v \in V'$ y $w \in V'$ entonces a $G'$ se le conoce como un grafo inducido de $G$. Dado un

grafo *G*, diremos que dos vértices están **conectados** si entre ambos existe un camino que los une.

Una componente conectada *G* es un subgrafo inducido maximal de *G*, esto es, un subgrafo conectado, el cual no es un subgrafo propio de ningún otro subgrafo conectado de *G*. Nótese que en una componente conectada, para todo par: *x*,*y* de sus vértices hay un camino de *x* a *y*.

Se llama **componente conexa** a un conjunto de vértices de un grafo tal que entre cada par de vértices hay al menos un camino. Matemáticamente se puede ver que la conexidad es una relación de equivalencia que descompone al conjunto de vértices *V* en clases de equivalencia, cada una de estas clases de equivalencia da lugar a un subgrafo al que se le llama una componente conexa. Un grafo es conexo si sólo existe una componente conexa que coincide con todo el grafo.

Denotemos con $K_n$ =(*V*,*E*) al grafo completo de *n* nodos, esto es, |*V*| = *n* y |*E*| = $\binom{n}{2}$ [2].

Dado un grafo *G= (V, E)*, *G* es un grafo bipartito si y sólo si *V* puede ser particionado en 2 subconjuntos $U_1$ y $U_2$, llamados conjuntos partitos, donde cada arista de *G* une un vértice de $U_1$ con un vértice de $U_2$.

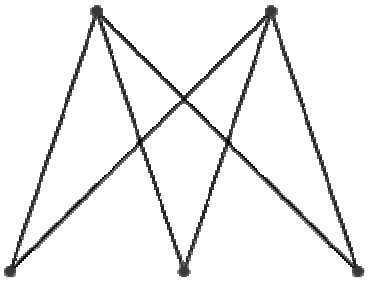


Fig.2.7. Grafo bipartito.

## 2.3 ALGORITMOS

Por conveniencia, la teoría de la complejidad está diseñada especialmente para aplicarse a problemas de decisión. Un *problema de decisión PD* [9] se plantea como una pregunta general la cual acepta como respuesta sólo una de dos posibilidades, la respuesta 'SI' o la respuesta 'NO'. En forma abstracta, un problema de decisión, puede describirse como:

*PD* : < *D*, *S* >, donde: *D* - Dominio de valores posibles y $S \subseteq D$ - son las instancias que resuelven el problema. El planteamiento del *PD* es:

$$\forall x \in D\text{: PD}(x) = \begin{cases} \text{Sí} & \text{si } x \in S \\ \text{No} & \text{en otro caso} \end{cases}$$

En estos términos, un *PD* puede usarse como medio para reconocer un *lenguaje S* que es un subconjunto de palabras de un alfabeto *D*. En la práctica, el *PD* intenta reconocer algún conjunto específico de objetos matemáticos, tales como: fórmulas lógicas verdaderas, gráficas que tienen una cierta propiedad, etc.,.... Pero esto requiere el codificar adecuadamente las estructuras subyacentes de los objetos para cualquier instancia del problema, de tal forma que finalmente, se pueda construir un predicado que aún sólo con respuestas 'SI' o 'NO' obtenga cualquier otra información que se desee.

El formato que usaremos para especificar problemas de decisión, consistirá de dos partes; en la primera parte se especifica una instancia genérica del problema en términos de sus componentes o parámetros. Puede verse esta parte como la presentación de una lista de variables libres indicando los valores tipo que cada variable podrá tomar, sean estos conjuntos, gráficas, funciones, etc.,..... La segunda parte establece una pregunta en términos de la instancia genérica que espera sólo una respuesta de 'SI' o de 'NO'. Una instancia particular o muchas veces sólo diremos una *instancia*, se obtiene de la instancia genérica por asignar valores particulares del tipo especificado a cada uno de los parámetros del problema.

Por ejemplo, el problema de decisión relacionado con el problema del agente viajero, puede plantearse de la manera siguiente:

> **Instancia:** Un conjunto finito *C* de ciudades, $C = \{c_1, c_2, ..., c_m\}$, se define una distancia $d(c_i, c_j) \in \mathbf{Z}^+$ para todo par de ciudades $c_i$, $c_j$ en *C*, y una cota $B \in \mathbf{Z}^+$ ($\mathbf{Z}^+$ denota los enteros positivos).
> **Pregunta:** ¿Existirá un *recorrido* que visita a todas las ciudades en *C* sin visitar dos veces una misma ciudad y cuya longitud total del recorrido no sea mayor que *B*? En otras palabras, se busca encontrar una permutación Π de {1,...,m} , tal que:
> $$\sum_{i=1}^{m-1} d(c_{\Pi(i)}, c_{\Pi(i+1)}) + d(c_{\Pi(m)}, c_{\Pi(1)}) \leq B$$

Un ejemplo de una instancia específica es: $C=\{c_1, c_2, c_3, c_4\}$; $d(c_1, c_2) = 5$; $d(c_1, c_3) = 5$; $d(c_1, c_4) = 8$ ; $d(c_2, c_3) = 6$ ; $d(c_2, c_4) = 12$ ; $d(c_3, c_4) = 9$ ; con cota: $B = 29$ y la pregunta a resolver sería:
¿Existirá una permutación Π de {1,...,m} , tal que:

$$\sum_{i=1}^{3} d(c_{\Pi(i)}, c_{\Pi(i+1)}) + d(c_{\Pi(4)}, c_{\Pi(1)}) \leq 29 \text{ ?[10]}$$

El problema de decisión que abordamos en este trabajo de tesis, es el de coloración de un grafo.

> **Instancia:** Una gráfica $G = (V,E)$, donde $V$ es el conjunto de nodos o vértices y $E$ es el conjunto de aristas entre los vértices. Una cota $K \in \mathbf{Z}^+$, con $K \leq |V|$.
>
> **Pregunta:** ¿ Será $G$ $K$-coloreable ?, es decir, existirá una función $f$: $V \rightarrow \{1,2,...,K\}$ tal que $f(u) \neq f(v)$ siempre que $(u,v) \in V$ ?

Un *algoritmo* es un procedimiento general que trabaja paso a paso y en un número finito de estos, resuelve un problema [9]. Un algoritmo resuelve un problema de decisión *PD* si puede aplicarse a cualquier instancia *I* del *PD* y garantiza que siempre produce una solución para esa instancia. Podemos pensar en un *algoritmo* como un programa de computadora o como la descripción de la función de transición de una máquina de Turing.

En general, nos interesa encontrar el *algoritmo más eficiente* que resuelve un problema dado. En el sentido más amplio, la noción de eficiencia tiene que ver con los varios recursos computacionales necesarios para ejecutar el algoritmo. Aunque el recurso dominante es el del tiempo, por tal, normalmente el *algoritmo más eficiente* que resuelve un problema *PD* es aquel que se ejecuta más rápidamente de entre todos los algoritmos que resuelven el mismo problema *PD* [8].

Los requerimientos de tiempo en la ejecución de un algoritmo se expresan en términos de una sola variable: la longitud o tamaño de las instancias del problema, que refleja la cantidad de datos de entrada y la magnitud de cada uno de estos datos.

Para definir la longitud de una instancia *I* de un problema de decisión *PD* , denotado como *long*(*I*) necesitamos primero definir la longitud de cada uno de los posibles parámetros que pueden aparecer en la instancia.
Así, para todo $n \in \mathbf{Z}$, se define su *longitud* (*magnitud*) como:

(*) $$long(n) = 1 + \lceil log(|n| + 1) \rceil .$$

Donde $\lceil x \rceil$ es la función que da como resultado al entero: $z \in \mathbf{Z}$ inmediatamente superior a *x*, y *log* denota la función logaritmo en base 2. En la expresión (*) el primer 1 indica que se requiere un bit para almacenar el signo de *n*, por tanto (*) expresa el número de bits necesarios para codificar al número *n* en binario [12].

En general, para un número *x*, *long*(*x*) será el número de bits necesarios para codificar a tal número. En caso de que *x* sea una cadena de símbolos, si cada símbolo puede representarse en la memoria de una computadora por un byte (8 bits) se considera la *long*(*x*)= |*x*| = no. de símbolos que componen a *x* (incluyendo al caracter blanco como un símbolo).

Se extiende el dominio de aplicación de la función *long*, para considerar diferentes tipos de datos. Por ejemplo, la longitud de un arreglo *v* con *m* elementos, donde *v*[*i*] se usa para hacer referencia al elemento *i* del arreglo, es: $long(v) = \sum_{i=1,...,m} long(v[i])$

Si *M* es un tipo de dato matriz en $\mathbf{Z}^{nxm}$, entonces: $long(M) = \sum_{i=1}^{n}\sum_{j=1}^{m} long(m[i,j])$.

Es decir, de acuerdo al tipo de dato a considerar, la longitud del dato será la sumatoria de las longitudes de cada uno de sus componentes de manera recursiva, hasta llegar a los componentes escalares.

Dada una instancia *I* de un problema de decisión *PD* con *m* datos de entrada; ($d_1$, ..., $d_m$) se define la longitud de la instancia como: $long(I) = \sum_{i=1}^{m} long(d_i)$, donde $long(d_i)$ es la longitud de cada uno de los datos de entrada.

Por ejemplo, si se tuviera una instancia *I* del problema del agente viajero, se requiere codificar las ciudades y las distancias entre estas. Si una instancia específica tiene *m* ciudades, se necesitan *m* etiquetas para diferenciar las ciudades $c_i$ y *m(m-1)/2* números $d_j$, que podemos suponer enteros positivos, que definen las distancias entre cualquier par de ciudades, y un número entero para indicar la cota *B*, en total se tendría que:

$$long(I) = \sum_{i=1}^{m} long(c_i) + \sum_{j=1}^{\frac{1}{2}m(m-1)} long(d_j) + long(B)$$

La función de complejidad de tiempo de un algoritmo, expresa los requerimientos de tiempo que un determinado modelo de computación necesita al ejecutar el algoritmo para resolver cada posible instancia del problema. El modelo de computación básico, para el análisis de las complejidades de tiempo y espacio de los algoritmos, son las llamadas máquinas de Turing.

## 2.4 MAQUINAS DE TURING

### 2.4.1 Máquina de Turing determinista

Existen varios modelos formales de computación, uno de los más comúnmente usado y que es muy simple de enunciar, es la llamada *máquina de Turing* (que abreviaremos con mT) [10]. Se definen diferentes mT's de acuerdo a la forma en que trabajan. Un modelo básico de una mT, es la llamada determinista (que abreviaremos con mTd), que consiste de un control finito, una cabeza de lectura-escritura, y una cinta dividida en un número infinito de celdas, estas celdas son etiquetadas usando números enteros, tal y como se muestra en la figura 2.8.

De manera formal una mTd *M* se representa por: $M = <Q,\Sigma,t,q_0,F>$, donde:

*Q*: es un conjunto finito de estados que indica los diferentes estados en los que puede estar el control finito, este conjunto de estados incluye dos estados distinguidos de parada: $q_y$ y $q_n$ y el estado de inicio $q_0$.

$q_0$: es el estado distinguido en el que arranca la mT

Σ: alfabeto de símbolos, son los símbolos usados en la codificación de la cadena de entrada así como los símbolos que pueden colocarse en la cinta de la mT.
Un símbolo distinguido es el espacio en blanco denotado por *b*.

*F*: subconjunto de *Q*, $F = \{q_y , q_n\}$ que denota a los estados finales o de paro de la mT.

*t*: es la función de transición, $t:(Q-F)\times\Sigma \rightarrow Q\times\Sigma\times\{-1,0,1\}$. La función de transición codifica al programa que va a ser ejecutado por la máquina de Turing.

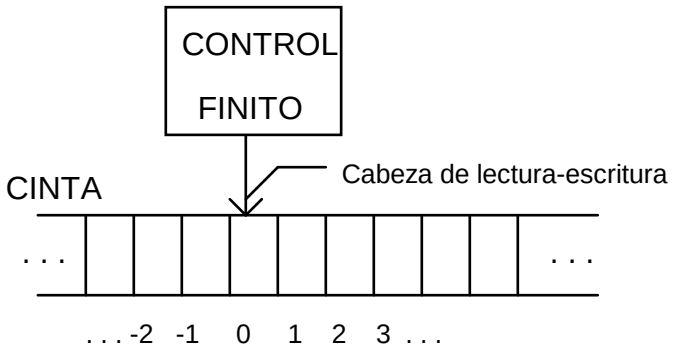



Figura 2.8. Representación esquemática de una máquina de Turing determinista.

Funcionamiento:
Inicialmente una máquina de Turing *M*:

- Tiene al símbolo *b* almacenado en las celdas de su cinta, excepto para un número determinado de casillas, en donde es colocada la cadena de entrada *x*, $x \in \Sigma^*$. La cadena *x* es colocada en las celdas de la cinta etiquetadas con los números de 1 a |*x*|, colocando un símbolo por celda.
- La cabeza de lectura y escritura se encuentra inicialmente examinando al símbolo más a la izquierda de *x* (celda 1), y el estado del control finito es el estado inicial $q_0$.

En cada instante: El comportamiento de la mT *M*, se rige por el programa codificado a través de la función de transición *t*, si *M* lee el símbolo $a_j \in \Sigma$, el control finito está en el estado $q_i \in Q$ y se tiene definida *t* para $q_i$ y $a_j$, por ejemplo, $t(q_i,a_j) = (q_{ij},a_{ij},m_{ij})$ , significa que entonces:

- Se sustituye en la cinta el símbolo $a_i$ por el $a_{ij}$
- El control finito pasa al estado $q_{ij}$
- La cabeza de lectura escritura pasa a examinar o a la celda que está a su izquierda o no se mueve o a la celda que está a su derecha, según el valor de $m_{ij}$ (-1, 0 o 1 respectivamente).

Una *descripción instantánea* es una palabra *XqY*, donde *X* e *Y* están en $\Sigma^*$, y $q$ es un estado de *Q*. *XqY* significa que en la cinta está escrita la palabra *XY*, el control de la mT está en el estado *q*, y su cabeza de lectura-escritura se encuentra examinando al primer símbolo de *Y*.

Un *paso de cómputo* de una mT *M*, se denota por un par ordenado $(\delta_1, \delta_2)$ y suele escribirse $\delta_1 \xrightarrow{pasa} \delta_2$, donde $\delta_1$ y $\delta_2$ son descripciones instantáneas de *M* y denota que en un paso, *M* pasa de la descripción instantánea $\delta_1$ a la $\delta_2$. Si $\delta_2$ se sigue de $\delta_1$ por uno o más pasos de cómputo entonces se denotará por: $\delta_1 \xrightarrow{sigue} \delta_2$.

A una secuencia de pasos de cómputo también es común llamarle una *computación*. Una computación que lleva a que la mT *M* acepte a la entrada *x*, es una secuencia $C_0, C_1, ...., C_t$ de pasos de cómputo, tal que $C_0 = q_0 x$ es la descripción instantánea de arranque, y para $0 \leq i < t$, $C_i \xrightarrow{\text{pasa}} C_{i+1}$, $q$ en $C_i$, $q$ no es un estado final, y el estado $q$ en $C_t$ es el estado final $q_y$.

El *lenguaje aceptado* por una mTd *M*, representado por $L_M$, es el conjunto de palabras $x \in \Sigma^*$, que ocasionan que *M* llegue al estado de aceptación $q_y$, cuando *M* arranca con *x* en su cinta de entrada: $L_M = \left\{ x \in \Sigma^* \mid q_0 x \xrightarrow{sigue} y q_Y z,\ y, z \in \Sigma^*, q_Y \in F \right\}$.

Dada una mTd *M* que reconoce a $L_M$, podemos suponer que *M* se detiene para toda *x* en $L_M$. Sin embargo, para palabras no aceptadas, *M* puede parar en el estado $q_n$ o bien, es posible que *M* nunca se detenga.

Para que el programa de una mT *M* corresponda a la noción de *algoritmo*, es entonces necesario que este programa siempre se detenga al procesar cualquier cadena del alfabeto de entrada.

La definición anterior está orientada hacia el reconocimiento de un *lenguaje* (conjunto de palabras). Se puede hacer corresponder tal definición para abarcar la resolución de los problemas de decisión. En este sentido, se dice que una mT *M* resuelve al problema de decisión *PD*: <*D*,*S*> si *M* se detiene para toda *x* en *D* y $L_M = S$.

Usando el modelo de máquinas de Turing, diremos que el tiempo de una computación de aceptación $C_0, C_1, ..., C_t$ es *t*, el espacio de la computación es el número de celdas utilizadas por la máquina de Turing durante el cómputo de aceptación.

Sea *F*: $\mathbf{N} \rightarrow \mathbf{R}$ una función y *M* una mT, se dice que *M* acepta su entrada en tiempo (espacio) $F(n)$ si para cada *x* en $L_M$ hay una computación que lleva a aceptación de *M* sobre *x* tal que el tiempo (espacio) de la computación de aceptación no excede $F(n)$, con $n = long(x)$ - longitud de la entrada *x* [10].

O en términos de problemas de decisión *PD*: <*D*,*S*> tal que $L_M = S$, la función de complejidad de tiempo $f_M$: $\mathbf{Z}^+ \rightarrow \mathbf{Z}^+$ se define como:
$f_M(n)=\max\{m \mid \exists\, x \in S,\ n=long(x)$, y $m$ es el tiempo de la computación de aceptación de $x\}$.

En este sentido, se dice que el programa de una mT tiene *complejidad polinomial en tiempo,* si existe un polinomio $p$ tal que, para todo $n \in \mathbf{Z}^+$, $f_M(n) \leq p(n)$.

En términos de los algoritmos, se dice que un algoritmo *A* es e*ficiente* cuando su función de complejidad en tiempo se puede acotar por un polinomio. Es decir, si $f_A$(n) es la función de complejidad en tiempo del algoritmo *A* y existe un polinomio $p$ tal que, para todo $n \in \mathbf{Z}^+$ $f_A(n) \leq p(n)$, entonces se conviene en decir que *A* es un algoritmo *eficiente*.

*CLASES DE COMPLEJIDAD DEFINIDAS POR MTD*

Con el fin de clasificar el esfuerzo computacional que se requiere al resolver los problemas de decisión, originalmente se usó a las máquinas de Turing como el modelo formal de computación [9]. El concepto clave fue definir una *clase de complejidad* en términos de los lenguajes que reconoce una máquina de Turing con recursos acotados (consideraremos aquí sólo los recursos de tiempo y espacio). Las clases de complejidad permiten clasificar los lenguajes (problemas) según la complejidad intrínseca computacional requerida para reconocerlos(resolverlos). De particular interés, son las clases de complejidad, definidas por recursos acotados que crecen logaritmicamente (salvo se indique lo contrario, estaremos hablando de logaritmos base 2) o polinomios sobre la variable *n*, donde *n* es la longitud de las instancias de entrada.

$$\text{DLOG} = \left\{ L \subseteq \Sigma^* \mid \forall x \in L,\ \text{x es aceptada en espacio logarítmico} \right\}$$
$$\text{P} = \left\{ L \subseteq \Sigma^* \mid \forall x \in L,\ \text{x es aceptada en tiempo polinomial} \right\}$$
$$\text{PSPACE} = \left\{ L \subseteq \Sigma^* \mid \forall x \in L,\ \text{x es aceptada en espacio polinomial} \right\}$$

O en términos de los problemas de decisión, podemos definir a las clases de complejidad, de la manera siguiente:

$$\text{DLOG} = \left\{ PD \mid \exists M \text{ mTd que resuelve } PD \text{ usando espacio logarítmico} \right\}$$
$$\text{P} = \left\{ PD \mid \exists M \text{ mTd que resuelve } PD \text{ en tiempo polinomial} \right\}$$
$$\text{PSPACE} = \left\{ PD \mid \exists M \text{ mTd que resuelve } PD \text{ usando espacio polinomial} \right\}$$

Contrariamente a la simplicidad del modelo de mT, hasta ahora, todos los modelos reales de computación estudiados, tales como: mT de una cinta, mT de múltiples cintas, máquinas de acceso aleatorio (RAM), Sistemas de Post, Intérpretes de lenguajes como Basic, Pascal, Lisp, etc.,.., son equivalentes con respecto al comportamiento de complejidad polinomial en tiempo. Se cree entonces que cualquier otro modelo razonable de cálculo, comparta esta equivalencia. Por tanto, la clase de complejidad P definida anteriormente no será afectada si

es cambiado el modelo de computación, así que se puede seleccionar uno u otro modelo de cómputo según convenga sin sacrificar la generalidad de los resultados.

La hipótesis de que el concepto intuitivo de *computabilidad* puede identificarse con lo que una mT puede realizar, es conocida como la *tesis de Church-Turing*. Y aunque no podemos esperar una demostración de esta tesis, puesto que el concepto de *calculable* sigue siendo algo informal, se puede proporcionar evidencia de su carácter razonable [10].

### 2.4.2 Máquina de Turing No determinista

Por simplicidad, usaremos el modelo de máquina de Turing no determinista (que abreviaremos con mTnd) definido en [8], tal modelo tiene la misma estructura que una mTd, excepto que se le adiciona un módulo de adivinanza, el cual tiene su propia cabeza de sólo escritura, tal y como se muestra en la figura 2.9. El módulo de adivinanza se usa sólo con el propósito de que escriba la cadena que adivina y sólo bajo tal propósito es usado.

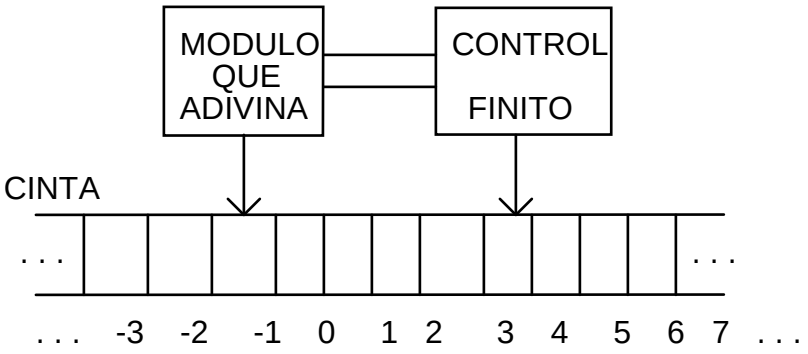



Figura 2.9. Representación esquemática de una máquina de Turing no determinista.

Inicialmente una cadena $x \in \Sigma^*$, es escrita en las celdas numeradas de 1 hasta $|x|$ de la cinta (en tanto que las restantes celdas contienen el símbolo blanco), la cabeza del control finito está posicionada en la celda 1 y el control finito está inactivo. La principal diferencia de este modelo con el de la mTd, es que el reconocimiento de la cadena *x* toma lugar en dos fases.

*Fase de adivinación*: El módulo que adivina mueve su cabeza de sólo escritura una celda a la vez para dejar escrito a partir de las celdas etiquetadas desde -1 hasta $-|w|$, una cadena $w \in \Sigma^*$, pasando posteriormente a la fase de revisión en donde ahora el módulo que adivina permanecerá inactivo. La elección de la cadena *w* a escribir, es realizada por el módulo que adivina de una forma totalmente arbitraria, y por tal razón, puede inclusive nunca parar al estar generando a *w*.

*Fase de Revisión*: La fase de revisión inicia cuando el control de estados finitos es activado en el estado $q_0$. A partir de este momento, la computación continúa en exactamente la misma forma en que trabaja la mTd, puesto que no interviene en esta fase el módulo que adivina. Aunque bien es posible que la cabeza del control finito visite la cadena *w* escrita por el módulo que adivina.

La computación de la mTnd *M*, se detiene cuando el control finito pasa a un estado final, y se dice que es una *computación de aceptación* si el control finito se detiene en el estado de aceptación $q_y$. Nótese que para alguna combinación de las cadenas *wbx*, *M* puede nunca parar. En general *M* tiene un número infinito de posibles computaciones para una entrada *x* dada, una computación por cada cadena *w* generada por el módulo que adivina.

Se dice que *M acepta a x* si al menos existe una cadena *w*, que hace que *M* llegue a un estado de aceptación. El lenguaje reconocido por *M* es entonces:

$L_M = \{x \in \Sigma^* | \exists w$ generado por el módulo que adivina, tal que *M* acepta a $x\}$

El tiempo requerido por una mTnd *M* que acepta a una cadena $x \in L_M$, es definido como el mínimo (sobre todas las computaciones de aceptación de *M* sobre *x*), del número de pasos que ocurren en las fases de adivinación y revisión, hasta que esta última fase llegue a un estado de aceptación [10]. La función de complejidad de tiempo $T_M$: $\mathbf{Z}^+ \rightarrow \mathbf{Z}^+$ para *M* es:

$T_M(n) = max\{1 \cup \{m | \exists x \in L_M, long(x) = n,$ *M* acepta a *x* en *m* pasos de computación $\}\}$

Nótese que la función de complejidad de tiempo para *M* depende sólo del número de pasos que ocurren en una computación de aceptación, y que por convención $T_M(n)$ es igual a 1 cuando ninguna entrada de longitud igual a *n* es aceptada por *M*.

Se dice que más que hallar una solución al problema de decisión *PD*, *M comprueba soluciones* para el *PD*, ya que dada una instancia *x* del *PD*, el módulo que adivina propone una propuesta de solución *w*, y *M* debe comprobar si *w* es realmente una solución. En este sentido, la mTnd *M* es usada como un mecanismo de verificación o comprobación de propuestas a soluciones.

Se dice que un programa corre en *tiempo polinomial* para la mTnd *M*, si existe un polinomio *p* tal que $T_M(n) \leq p(n)$ para todo $n \geq 1$.

*CLASES DE COMPLEJIDAD DEFINIDAS POR UNA MTND*

Los problemas de decisión se agrupan en conjuntos de complejidad comparable que son llamadas *clases de complejidad* [7]. Un problema está dentro de la clase de complejidad NP si y sólo si puede resolverse en tiempo polinomial mediante un algoritmo no determinista. Se considera que los problemas NP-completos son los mas difíciles de la clase NP y la razón es que de tenerse una solución de tiempo polinomial para un problema

NP-completo entonces todos los problemas de la clase NP tendrían también una solución en tiempo polinomial.

$$\begin{aligned}
\text{NLOG} &= \{PD | \exists M \text{ mTnd que comprueba soluciones de PD usando espacio logarítmico}\} \\
\text{NP} &= \{PD | \exists M \text{ mTnd que comprueba soluciones de PD en tiempo polinomial}\} \\
\text{NPSPACE} &= \{PD | \exists M \text{ mTnd que comprueba soluciones de PD usando espacio polinomial}\}
\end{aligned}$$

Sólo reescribiremos en términos de lenguajes a la clase NP, ya que las definiciones para las clases de complejidad NLOG y NPSPACE son equivalentes.

$$NP = \left\{ \mathrm{L} \subseteq \Sigma^* | \exists M \text{ mTnd: } \left(L = L_M\right) \& \left(\forall x \in L, x \text{ es aceptada por M en tiempo polinomial}\right) \right\}$$

Nótese que *verificación* o *comprobación* en tiempo polinomial, no implica *resolubilidad* en tiempo polinomial. Al decir que se puede comprobar que una cadena *w* es o no solución a un problema, no se está realmente considerando el tiempo gastado en encontrar dentro de un espacio potencialmente grande de posibilidades a la cadena *w* que sea solución al problema.

Correspondiendo a esta noción de modelos no deterministas, un algoritmo que corre en tales modelos, será no determinístico, y en este sentido estará compuesto de dos fases separadas:
a) La fase que *adivina*
b) La fase que *revisa*

Dada una instancia *x* de un problema de decisión *PD*, la primer fase, *adivina* una estructura *w* que será una propuesta de solución. Entonces la instancia *x* y la cadena *w* se toman como entrada a la segunda fase (fase de revisión) que procede en forma determinista para terminar con una respuesta si o no o, caer en un ciclo infinito.

Un algoritmo no determinista *resuelve* un problema de decisión *PD*: <*D*,*S*> si se cumplen las siguientes propiedades para todas las instancias *I* del problema de decisión:

1. Si *I* tiene una solución, entonces existe alguna cadena *w* que es adivinada y que conduce a la fase de revisión a que responda 'SI'.
2. Si *I* es una instancia del *PD* sin solución, entonces no existe cadena *w* que el módulo que adivina pueda generar tal que conduzca a la fase de revisión a responder 'SI' [10].

Por ejemplo, un algoritmo no determinista para el problema del agente viajero puede construirse usando un estado de adivinación, que simplemente adivine una secuencia arbitraria sobre las ciudades. El estado de revisión, sólo se encargará de verificar que la suma de las distancias de las ciudades en la secuencia dada, cumplan, la cota exigida.

Un algoritmo no determinista que resuelve el problema de decisión *PD* se dice que es de tiempo polinomial si existe un polinomio *p* tal que para toda instancia *I* del *PD*, existe una

cadena *w* que conduce a la fase de revisión a responder *SI*, en tiempo acotado por $p(long(I))$. Nótese que esta cota impone también límites en la longitud de *w*.

Puesto que a cualquier modelo de cómputo se le puede adicionar el módulo que adivina con posibilidad de sólo escritura, tal y como se ha realizado en este apartado para las mTd´s, se podría reproducir la definición de no determinismo con cualquier otro modelo de computación. Las versiones resultantes de definición de clases no deterministas serán equivalentes en los otros modelos de computación y particularmente todos estos modelos son equivalentes en lo que respecta al tiempo polinomial, y por tal a la definición de la clase NP [7].

DTIME($F(n)$) (DSPACE($F(n)$)) denota la clase de problemas que son aceptados por máquinas de Turing deterministas que reconocen su entrada dentro de un tiempo (espacio) $F(n)$. En tanto que NTIME($F(n)$) (NSPACE($F(n)$) ) denota la clase de problemas aceptados por máquinas de Turing no deterministas que aceptan dentro de tiempo (espacio) $F(n)$.

Al especificar las cotas de clases, muchas veces se usa la notación de la función de orden O. Si $G(n)$ es una función, $G$: $\mathbf{N} \rightarrow \mathbf{R}$, O($G(n)$) es el conjunto de funciones $G'$ que satisfacen $G'(n) \leq c \cdot G(n)$ para alguna constante real positiva $c$ y para toda $n \geq n_0$, donde $n_0$ depende de $G$. Nótese que si $n_0$ es cero, entonces se cumple para todo $n \in \mathbf{N}$.

La notación O es usada dentro de las clases NTIME, DTIME, etc.,... Por ejemplo, DTIME($2^{O(n)}$) denota la unión de DTIME($2^{cn}$) tomadas sobre todas las constantes $c$. También se usa la notación *poly*($G(n)$) que abrevia a O($G(n)^{O(1)}$), esta es la clase de funciones acotadas superiormente por algún polinomio de función de $G$, ya que O(1) denota a cualquier constante real positiva. Usando esta notación, podemos reescribir a las clases de complejidad como:

- DLOG = DSPACE($log(n)$),
- NLOG = NSPACE($log(n)$),
- P = DTIME($poly(n)$),
- NP = NTIME($poly(n)$) y
- PSPACE= DSPACE($poly(n)$)

Las clases P y NP las podemos también identificar de la siguiente manera:

- P es la clase de problemas que poseen algoritmos deterministas de resolución que tardan tiempo polinomial.
- NP es la clase de problemas que tienen un algoritmo determinista de resolución que corre en tiempo exponencial, pero para los cuales también existe un algoritmo no determinista que corre en tiempo polinomial.

A los problemas de la clase P, se les reconoce como problemas *tratables*, puesto que para cualquiera de sus instancias, éstas se resuelven por algoritmos que corren en un tiempo acotado de cómputo. Se dice que un problema no ha sido *bien resuelto* hasta que es hallado un algoritmo determinista de tiempo polinomial que lo resuelve [8]. Se le asigna a un problema el término de *intratable*, si este se ha mostrado tan difícil que no se ha encontrado

algoritmo determinista con complejidad polinomial en tiempo que lo resuelva, es decir, no se ha hallado algoritmo eficiente que lo resuelva. Entonces una primera clase de complejidad que contiene problemas intratables, es la clase NP.

Todo algoritmo cuya función de complejidad de tiempo, no pueda acotarse por una función polinomial, se dice que es un algoritmo de complejidad exponencial en tiempo, aún cuando debe notarse, que esta definición incluye ciertas funciones de complejidad de tiempo, tales como, $n^{log(n)}$, las cuales normalmente no son consideradas como funciones exponenciales.

El término *intratable*, refleja el punto de vista de que los algoritmos de tiempo exponencial no son considerados 'buenos' algoritmos y que generalmente tal clase de algoritmos refleja variaciones de búsquedas exhaustivas, mientras que algoritmos de tiempo polinomial generalmente son construidos sólo a través de explotar la estructuras internas e inherentes del problema.

Existen algoritmos de tiempo exponencial, que han sido muy útiles en la práctica. La complejidad de tiempo de un algoritmo tal y como la hemos definido, es una medida para el peor de los casos, y el hecho de que un algoritmo tenga complejidad exponencial, significa que existe al menos una instancia del problema, que requiere mucho tiempo, aún y cuando muchas de las instancias del mismo problema requieran en la práctica de mucho menos tiempo que un valor exponencial.

Aún y cuando ciertos algoritmos de complejidad exponencial para determinados problemas tienen un buen comportamiento en la práctica, esto no ha detenido el avance de las investigaciones por buscar algoritmos de tiempo polinomial que resuelvan los mismos problemas. Y de hecho, el gran éxito de tales algoritmos exponenciales, lleva a la sospecha de que hace falta capturar alguna propiedad crucial del problema cuyo refinamiento pueda llevarnos a mejores métodos. Un área de gran interés es sobre técnicas generales que permitan diseñar algoritmos deterministas de tiempo polinomial ya sea para los problemas intratables, o para variaciones de éstos.

La habilidad de algoritmos no deterministas, de poder revisar un número exponencial de posibilidades en tiempo polinomial, genera la sospecha, de que este tipo de algoritmos son estrictamente más poderosos que los algoritmos deterministas de complejidad polinomial. Sin embargo, contra todos los esfuerzos realizados, no se ha podido demostrar esta especulación. La pregunta de si ¿ $P = NP$ ?, es uno de los problemas abiertos centrales en la ciencia de la computación [7].

Aún más, sabiendo que se cumple la siguiente jerarquía entre las clases de complejidad:
$DLOG \subseteq NLOG \subseteq P \subseteq NP \subseteq PSPACE$, (*)
sigue siendo una pregunta abierta, decidir cuáles de estas contenciones son propias, aún cuando es sabido que: $DLOG \neq PSPACE$ [11]

Las siguientes relaciones se cumplen:

$\text{DTIME}(T(n)) \subseteq \text{NTIME}(T(n))$
$\text{DTIME}(T(n)) \subseteq \text{DSPACE}(T(n)/\log(T(n)))$

$\text{NTIME}(T(n)) \subseteq \text{DTIME}(2^{O(T(n))})$
$\text{NTIME}(T(n)) \subseteq \text{DSPACE}(T(n))$

$\text{NSPACE}(S(n)) \subseteq \text{DSPACE}(S(n)^2)$
$\text{NSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})$

## 2.5 REDUCIBILIDAD ENTRE PROBLEMAS

Un programa para una mT puede también ser usado para calcular funciones. Supongamos que $M$ es un programa de una mT con alfabeto de entrada y salida $\Sigma$, que se detiene para toda $x$ en $\Sigma^*$. Entonces $M$ calcula la función $f_M$: $\Sigma^* \rightarrow \Sigma^*$ donde $f_M(x)$ es la cadena que al detenerse $M$, queda en la cinta de la mT, al arrancar $M$ teniendo a la cadena $x$ como entrada.

*Logspace* (respectivamente *polytime*) es la clase de funciones computables por mTd's que requieren espacio $\log(n)$ (tiempo $poly(n)$, resp.). Sean $A$: $<D_A,S_A>$ y $B$: $<D_B,S_B>$ dos problemas de decisión, $A$ es reducible en espacio logarítmico a $B$, y se escribe $\boldsymbol{A \leq_{log} B}$ (respectivamente $A$ es reducible en tiempo polinomial a $B$, denotado por $A \leq_p B$) si existe una función $f$: $D_A \longrightarrow D_B$ que está en la clase *logspace* (resp. $f$ está en la clase *polytime*) tal que para toda $x$, $x$ está en $S_A$ si y solo si $f(x)$ está en $S_B$.

Si hay una constante $b > 0$ tal que $f(x) \leq b \cdot long(x)$ para toda $x$ con $long(x) > 0$ se escribirá $A \leq_{log\text{-}lin} B$ (respectivamente $A \leq_{p\text{-}lin} B$) [11].

Se denotará con: $\leq_{eff}$ a cualquiera de: $\leq_{log}$, $\leq_{log\text{-}lin}$, $\leq_p$, $\leq_{p\text{-}lin}$. Entonces $\boldsymbol{A \leq_{eff} B}$ significa que $A$ es reducible a $B$ vía una función $f$ que es computable determinísticamente en tiempo polinomial.

Sean $B$ y $D$ dos problemas, y sea $C$ una clase de problemas:

1. Se denotará con $C \leq_{eff} B$ si es que: $A \leq_{eff} B$ para todo problema $A$ en $C$, y en tal caso se dice que $B$ es *C-Difícil* con respecto a $\leq_{eff}$.
2. $B$ es *C-completo* con respecto a $\leq_{eff}$, si $C \leq_{eff} B$ y además $B$ está en la clase $C$.
3. $D$ es *Co-C* si su problema complemento, $D^C$ está en la clase $C$.

Otras clases de complejidades que han sido de vital importancia en la Teoría de Complejidad, son:

- NP-completo - son problemas que representan a la clase NP, en el sentido de que cualquier otro problema NP puede reducirse en tiempo polinomial a estos.
- Co-NP - es la clase de problemas cuyos complementos también están en la clase NP.
- NP-Difícil - son problemas que en principio parecen ser más difíciles de resolver que los problema de la clase NP, ya que cualquier problema en NP puede reducirse en tiempo polinomial a un problema de esta clase [11].

Se define a dos lenguajes $L_1$ y $L_2$ (o dos problemas de decisión $\Pi_1$ y $\Pi_2$) como *polinomialmente equivalentes,* cuando tanto $L_1 \leq_{eff} L_2$ como $L_2 \leq_{eff} L_1$ ( tanto $\Pi_1 \leq_{eff} \Pi_2$ como $\Pi_1 \leq_{eff} \Pi_2$ ). Puede comprobarse que la relación $\leq_{eff}$ es una relación de equivalencia legítima y que además impone un orden parcial sobre clases de lenguajes polinomialmente equivalentes (o sobre los problemas de decisión polinomialmente equivalentes).

De hecho la clase P será la menor clase de equivalencia bajo la relación de orden parcial $\leq_{eff}$ y por lo tanto puede pensarse como la clase de complejidad consistente de los problemas de decisión más *fáciles*. La clase de los problemas NP-completo formará otra clase de equivalencia, distinguida por la propiedad de que esta clase contiene a los problemas de decisión más *difíciles* de la clase NP [8].

Sigue siendo un problema abierto hasta ahora, el conocer con exactitud las líneas de acotación que existe entre las clases de complejidad de los problemas P, NP, NP-completo y los Co-NP.

# 3. RECORRIDOS EN UN GRAFO

Tomando como referencia que el primer problema planteado sobre grafos fue el de los puentes de Königsberg, podemos entonces decir que estos problemas plantean un recorrido a través del grafo.

La mayoría de los problemas que consideran un grafo, requieren de definir una estrategia para recorrer todas las partes del mismo, es decir, definir un orden para visitar cada nodo y arista del grafo y, de preferencia, no visitar más de una vez cada elemento del grafo. Así, dependiendo del tipo de aplicación podemos establecer un recorrido para lograr obtener el resultado deseado.

Los dos tipos más usados de recorrido sobre un grafo son; el recorrido a lo ancho y el recorrido a lo profundo. Dado que estamos hablando de un recorrido entonces eso no lleva a establecer un camino dentro del grafo, el cual ya hemos definido en el capitulo anterior.

## 3.1. RECORRIDO A LO PROFUNDO

En un grafo, ya sea dirigido o no dirigido, se puede aplicar el recorrido en profundidad. Consideremos un grafo $G = (V,A)$ y sea un vértice $v \in V$, la estrategia de recorrido en profundidad explora sistemáticamente las aristas de $G$ de manera que primero se visitan los vértices adyacentes a los visitados mas recientemente. De esta forma, se va profundizando en el grafo; es decir, alejándose progresivamente de $v$.

Esta estrategia admite una implementación simple de forma recursiva, utilizando globalmente un contador $n$ y un vector que va marcando los nodos ya *visitados*, también es común ir almacenando el orden del recorrido. La búsqueda a lo profundo puede analizarse en base al siguiente algoritmo.

**Procedimiento *dfs*(*u*)**

1.- Marcar $u$ como visitado.
2.- Para cada uno de los nodos $w \in N(u)$
    a) Si $w$ esta como no visitado entonces $dfs(w)$.
    b) Sino marca la arista $\{w,v\}$ como arista de retroceso.
3.- Marcar $u$ como terminado.
4.- Return

```
 main()
{ for v=1 to n
   Visitados [v]=0; /*0 representa no visitado */
       for v=1 to n
if (visitados[v]= =0) then dfs (v);
}

Procedure dfs (v:nodo)
nodo w
{ visitados[v]=1; /*1 representa visitado */
   for  cada nodo w en Ady[v]
      if (visitados[w]=0) then  dfs (w);
}
```

El procedimiento *dfs* se ejecuta en un tiempo de $O(m+n)$ donde $n$ y $m$ son el número de nodos y el número de aristas de entrada del grafo $G$, respectivamente [2]. Hasta aquí, *dfs* es un procedimiento en tiempo lineal en base al tamaño del grafo de entrada: $G$.
Esta búsqueda a lo profundo será usado dentro del proceso de coloreo del grafo.

Cada arista de retroceso detectada durante la búsqueda *dfs* marca el inicio y fin de un ciclo base o ciclo fundamental.

## 3.2. RECORRIDO A LA ANCHO

Dado un grafo $G = (V,A)$ y un vértice $v \in V$, la estrategia de recorrido en amplitud explora sistemáticamente las aristas de $G$ de manera que primero se visitan los vértices mas cercanos a $v$.

La búsqueda a lo ancho se determina de la búsqueda a lo profundo en términos en el cual los vértices de un grafo son visitados. Algunas veces, la búsqueda a lo ancho es visualizada como si simultáneamente la exploración de los nodos inicia en un punto en común y se da la bifurcación de sus aristas de forma independiente y simultánea.

La función recorrido en amplitud obtiene un recorrido en anchura de $G$ llamando a la rutina *bfs*. Se marca (con un color) los tres momentos de un nodo durante el recorrido, cuando no ha sido visitado es blanco, cuando se visita por vez primera pasa a color gris y cuando han sido visitados todos sus nodos adyacentes queda finalmente en color negro. *bfs* utiliza una cola auxiliar de *pendientes* donde se almacenan los vértices no visitados.

El siguiente algoritmo hace referencia al proceso de búsqueda a lo ancho:

**Procedimiento *bfs(u)***

1. Crea una lista vacía $L$
2. Asigna $u = v$

3. Marca *u* como visitado
4. Añade a cada vértice no marcado en T(*v*) al fin de L
5. Si L esta vacía, termina
6. Asigna a *u* el primer vértice en L
7. Quita el primer vértice de L
8. Return paso (3)

```
Procedure breadth (S)
Parents[ S ]= -1; color[S]=gris; encolar(pendientes, S);
While (pendientes is not empty)
{
      V=desencola(pendientes);
      For each node W en la lista de Adyacentes[V];
            If(color[W]= =blanco)
            {
            color[W]=gris; encolar(pendientes, W); padre[W]=V;
            }
      color[V]=negro;
}
Return;
```

Si consideramos un grafo $G=(V,E)$, donde $|V|=n$, $|E|=m$, los recorridos a lo profundo y a lo ancho son procedimientos óptimos en el tiempo requerido para visitar todos los nodos y aristas del grafo. En términos del tiempo que requieren estos algoritmos, ambos son de complejidad lineal de acuerdo al tamaño de su entrada, es decir, en base a la suma del número de nodos y aristas que hay en el grafo, lo que se denota como complejidad de orden $O(n+m)$. El ser de orden $O(n+m)$ indica que los recorridos tardan un tiempo lineal con respecto al número de nodos y aristas del grafo [2].

## 3.3. ARBOL GENERADOR DE UN GRAFO.

Todo grafo *G* conexo tiene un árbol generador *T*. Si *G* es un árbol entonces es en si mismo su árbol generador. Sea *G-T* la notación para considerar el grafo con aristas *E(G) – E(T)*. Nótese que cualquier arista $\{v,w\} \in E(G-T)$ que se adicione a *T* nos conforma un ciclo único denotado por $C_{v,w}$. Esto es porque *v* y *w* están conectados por un camino único $P_{v,w}$ en *T*. Así $P_{v,w}$ + {*v*,*w*} crean el ciclo $C_{v,w}$ llamado ciclo fundamental de *v*,*w* con respecto a *T*.

- Un árbol generado $T_G$ donde $G=(V,E)$, puedo formarse al aplicar la búsqueda a lo profundo sobre *G*, y al no considerar las aristas de retroceso.
- Toda arista de retroceso encontrada durante la búsqueda a lo profundo nos conforma un ciclo fundamental.

- Todo ciclo de *G* puede ser formado a partir de los ciclos fundamentales del grafo.

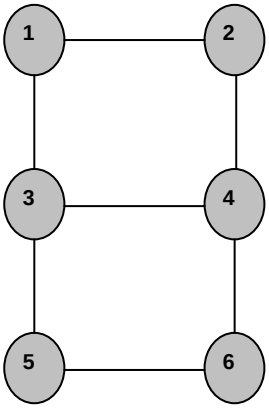



Fig. 3.1. Grafo *G=(V,E)*

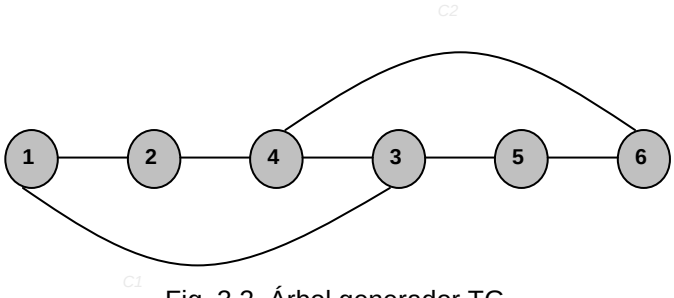



Fig. 3.2. Árbol generador TG

## 3.4. CICLOS FUNDAMENTALES

Cada arista de retroceso que es detectada durante la búsqueda a lo profundo: *dfs* marca el inicio y el fin de un ciclo base o ciclo fundamental.

El determinar el numero de aristas de retroceso en un grafo, nos esta dando al mismo tiempo el número de ciclos básicos que existe en el grafo.

Sea $C = \{C_1, C_2, \ldots, C_k\}$ el conjunto de ciclos base encontrados en un grafo después de realizada la búsqueda a lo profundo. Para dos distintos ciclos base $C_i$ y $C_j$ de *C*, si $C_i$ y $C_j$ no tienen aristas en común, decimos que ambos ciclos son independientes uno de otro. Y si un ciclo $C_i$ no tiene aristas comunes con ningún otro ciclo de *C*, entonces $C_i$ es un ciclo independiente en el grafo.

Además, se puede demostrar que cualquier otro ciclo que se visualiza en el grafo original *G* se conforma de la combinación lineal de dos o más ciclos básicos. Por lo que es común denotar que cualquier ciclo de un grafo se puede representar como: $C_1 \oplus C_2 \oplus \ldots \oplus C_n$, donde cada *Ci*, *i*=1,…,*n* es un ciclo básico y el operador ($A \oplus B$) indica que se consideran aristas disjuntas del conjunto de aristas que aparecen tanto en *A* como en *B* [1].

# 4. PROCEDIMIENTOS BASICOS PARA EL 3-COLOREO

En este capítulo se hará énfasis al coloreo de un grafo, problema que por mucho tiempo estuvo ligado al problema de colorear un mapa. Como se indicó en capítulos anteriores, este último problema pregunta: ¿Cuántos colores son necesarios para colorear un mapa de tal forma que los vecinos de cada estado tengan un color diferente? Este problema se puede resolver de distintas maneras, pero lo óptimo es encontrar un procedimiento que nos pueda decir el número mínimo de colores a utilizar.

## 4.1. COLOREO DE UN GRAFO

El coloreo de un grafo $G=(V, E)$ es una asignación de color a cada uno de los vértices de $G$. Un coloreo es propio si los vértices adyacentes entre sí, siempre tienen un color diferente. Un $k$-coloreo de $G$ es un mapeo de $V$ en el conjunto $\{1,2,3,\ldots,k\}$ de $k$ "colores". El numero cromático de un grafo $G$ es denotado por $\chi(G)$ y representa al mínimo valor de $k$ tal que $G$ tiene un $k$-coloreo propio. Se sabe que $\chi(G)$ es un problema computable en tiempo polinomial cuando $\chi(G)\leq 2$, pero determinar para cualquier grafo $G$, si $\chi(G)\geq 3$ se convierte en un problema NP-completo[8] .

El problema de determinar el número mínimo necesario para colorear un grafo es un problema NP-completo, aún para grafos $G$ de grado 3 ($\Delta(G)=3$). Una consecuencia de lo anterior, es que no se tiene una teoría completa sobre el coloreo de un grafo[3].

Los problemas de decisión se clasifican en conjuntos de complejidad comparables llamados *clases de complejidad*. Un problema se encuentra dentro de la clase de complejidad NP si y sólo si puede resolverse en tiempo polinomial mediante un algoritmo no determinista. Se considera que los problemas NP-completos son los mas difíciles de la clase NP y la razón es que de tenerse una solución de tiempo polinomial para un problema NP-completo entonces todos los problemas de la clase NP tendrían también una solución en tiempo polinomial

En el problema del coloreo de un grafo, se desea asignar a cada vértice del grafo un color único. Con la restricción de que a cada par de vértices conectados por una misma arista se les debe asignar colores diferentes. También, podemos ver el coloreo de un grafo $G$ como una función $V(G)$ del conjunto N de los enteros positivos (o números naturales) tal que los vértices adyacentes presentan distintos valores funcionales , esto es, el coloreo de $G$ es una función:

$c : V(G) \rightarrow$ N tal que para todo $\{u,v\} \in E(G)$ se tiene que $c(u) \neq c(v)$

Al realizar el coloreo del grafo se pueden utilizar *n* colores, pero nosotros centramos nuestra atención en el 3-coloreo, y proponemos un procedimiento para dado un grafo de entrada determinar si es posible o no, colorearlo con 3 colores.

Presentamos aquí algunas condiciones de suficiencia para que se realice satisfactoriamente el procedimiento del 3-coloreo de un grafo.

## 4.2. DOS COLOREO DE UN ARBOL

Un *árbol* es un grafo conexo que no tiene ciclos. Los árboles son los grafos conectados más pequeños, ya que si se elimina alguna arista del árbol, este queda desconectado. Una propiedad fundamental de un árbol es que todo árbol con *n* vértices tiene (*n*-1) aristas.

Si $G=(V,E)$ es un árbol entonces $|v|=|E|-1$. Por prueba de inducción sobre el numero de vértices $|v|=n$.

1. Si $n=1$ entonces $G=K$ el cual es un grafo conectado sin ciclos y así que $|E|=0$, $K_1=0$.
2. Si $n=2$ entonces $G=K_2$ y se tiene que $|E|=1$. Asumamos que el resultado se cumple para cualquier $|G|\leq t$ y consideremos ahora en un árbol $G$ con $|v|=t+1$.
3. Sea $G=(V,E)$ tal que $|v|=t+1$, G debe tener un vértice de grado 1 porque si no $G$ contendrá un ciclo. Sea $v\in V$ tal que $\delta(v)=1$ y sea $G'=G-\{v\}$ lo cual aún es un grafo conectado sin ciclos, así que $G$ es un árbol con $G$ tonos.
   Por tanto $|E(G')|= |v'|-1=|v|-2$ y se sigue que $|E(G)|= |v|+1$ así el resultado es verdadero para $|V|= n=t+1$ [1].

Entonces cualquier árbol *T*, se puede colorear por niveles y alternando colores, usando sólo dos colores.

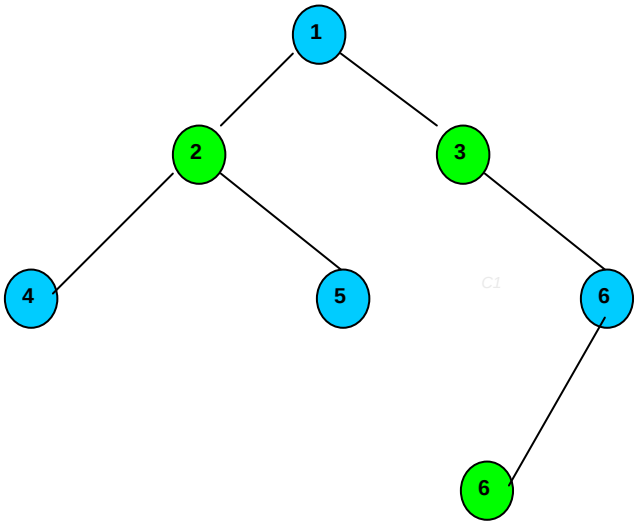



Fig. 4.1 Coloreo de un árbol por niveles

## 4.3. 2-COLOREO DE GRAFOS

La clase principal de grafos reconocida por ser 2-coloreable, es la clase de grafos bipartitos. Por otro lado, es conocido que se puede identificar en tiempo polinomial si un Grafo *G* es bipartito o no. En base a los ciclos fundamentales del grafo *G* se tiene que *G* es un grafo bipartito si y sólo si *G* no contiene ciclos de longitud impar [6].

Se sabe que un grafo *G* tiene un número cromático igual a 2 si y solo si *G* es un grafo bipartito no vació. Dadas las definiciones anteriores un grafo *G* es bipartito si y solo si *G* no tiene ciclos impares. Si una grafo *G* contiene al menos un ciclo impar, entonces $\chi(G) \geq 3$.

Por supuesto, si $G \cong C_n$ para algún entero n $\geq$ 4, entonces $\chi(G) = 2$. Por otro lado si $n \geq 3$ es un ciclo impar entero, entonces $\chi(C_n) = 3$. Nosotros ya sabemos que $\chi(C_n) \geq 3$ cuando $n \geq 3$ es impar. Para mostrar que $\chi(C_n) = 3$, necesitamos mostrar que existe un 3-coloreo de $C_n$. Así podemos colorear algunos vértices de $C_n$ con el tercer color y alternar el primero y el segundo color para los vértices faltantes.

La forma natural de construir los ciclos fundamentales de un grafo, es aplicando la búsqueda a lo profundo (o igualmente, la búsqueda a lo ancho). Una vez reconocidos los ciclos fundamentales, es relevante para el 3-coloreo, revisar si estos son de longitud par, o impar, o una combinación de ambos casos.

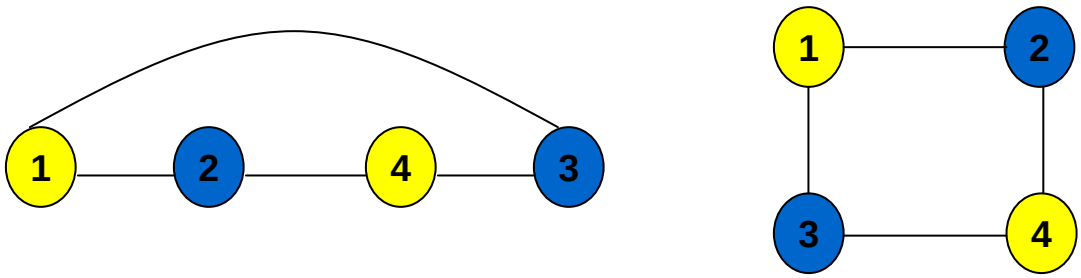



Fig. 4.2. Grafo Bipartito

## 4.4. 3-COLOREO PARA CICLOS EMBEBIDOS

Primero suponemos que tenemos como entrada un grafo general cualquiera *G*, al cual le aplicamos la búsqueda a lo profundo: $T_G = dfs(G)$ y verificamos si $T_G$ presenta ciclos embebidos.

Sean $C = \{C_1, C_2, \ldots, C_k\}$ y si dos ciclos $C_i$ y $C_j$ tienen aristas comunes, se puede revisar si pueden considerarse como un ciclo embebido uno dentro del otro. Dados dos diferentes

ciclos $C_i$ y $C_j$ de un grafo con aristas comunes, diremos que $C_i$ puede embeberse dentro de *Cj*, si:

a)$V(C_i) \subseteq V(C_{ij})$ : el conjunto de nodos de $C_i$ es un subconjunto de los nodos de $C_j$.

b) $C_i \oplus C_j = C_k$ , donde el operador $\oplus$ denota lo operación or-exclusivo entre el conjunto de aristas de los ciclos y $C_k$ es un nuevo ciclo del grafo, pero diferente a $C_i$ y $C_j$ [1].

En este caso, diremos que $C_i$ es el ciclo embebido interno y *Cj* es el ciclo embebido externo. La propiedad de que dos ciclos estén embebidos uno dentro del otro, puede extenderse a un conjunto de ciclos embebidos. Sea $D = (C_1, C_2, \ldots, C_k)$ una tupla de ciclos embebidos y ordenada de tal forma que $C_i$ está embebido en $C_{i+1}$, para $i$=1,…, $k$-1. En este caso, diremos que $C_1$ es el ciclo embebido más interno y que $C_k$ es el ciclo embebido más externo de la tupla.

Teniendo la premisa de coloreo de ciclos donde para colorear un ciclo par son suficientes 2 colores y, si tenemos un ciclo impar necesitaremos de un tercer color para el coloreo del ciclo. Entonces si $T_G$ presenta ciclos embebidos, podemos colorear el conjunto de ciclos embebidos usando sólo 3 colores, de la forma siguiente:

1. Para *i=1* hasta *k*
2. Colorear $C_i$
    a. Si $C_i$ es par, colorear con 2 colores
    b. Sino colorear con 3 colores
3. Terminar

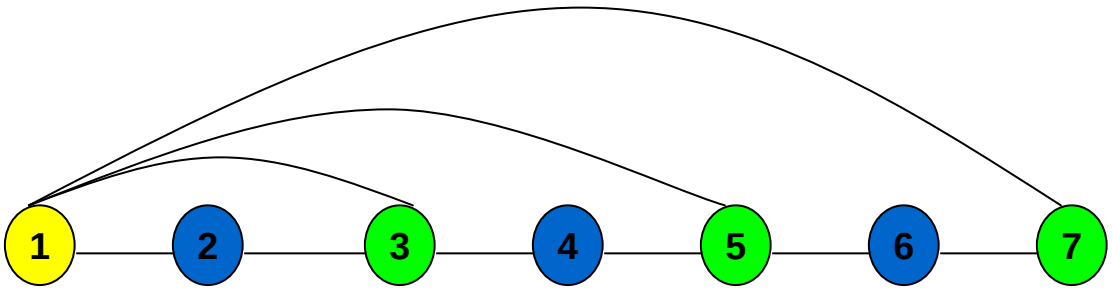



Fig. 4.3. Grafo Embebido

## 4.5. CONDICIONES DE SUFICIENCIA PARA EL 3-COLOREO

Anteriormente se ha planteado un algoritmo de coloreo para grafos bipartitos y se han presentado condiciones de suficiencia para el coloreo optimo de grafos embebidos. Además se ha mostrado que estos procedimientos trabajan en tiempo lineal.
Dado que tenemos la forma natural de construir los ciclos fundamentales de un grafo, a través de aplicar la búsqueda a lo profundo (o igualmente, la búsqueda a lo ancho). Una vez reconocidos los ciclos fundamentales, es relevante para el 3-coloreo, revisar si estos son de longitud par, o impar, o una combinación de ambos casos.

Sea $T_G = dfs(G)$ el grafo generado después de haberle aplicado la búsqueda a lo profundo. Procedemos a revisar si $T_G$ presenta alguno de los siguientes casos básicos:

1. Si $T_G$ no tiene ciclos fundamentales impares entonces $T_G$ es 2-coloreable. Esta opción considera el caso donde $T_G$ es un grafo bipartito. En este caso, la coloración por niveles resulta ser un procedimiento eficiente para el 2-coloreo.
2. Si algún ciclo básico en $T_G$ incluye ciclos impares, pero tales ciclos no se intersectan con algún otro ciclo entonces $T_G$ es 3-coloreable. Podemos colorear $T_G$ por niveles, usando el tercer color para colorear cada uno de los nodos finales de cada ciclo impar. El nodo final de un ciclo es el nodo donde la arista de retroceso fue encontrada durante la búsqueda a lo profundo.
3. Si existen dos ciclos intersectados $C_i$ y $C_j$ de $C$ en $T_G$ donde ambos ciclos comparten una sola arista en común entonces podemos transformar tales ciclos en ciclos embebidos mediante una transformación polinomial sobre el grafo. Ver la siguiente sección.
4. Cuando tenemos ciclos embebidos, se inicia el coloreo del ciclo más interno al más externo. Cuando se inicia el coloreo de un nuevo ciclo, usamos un color diferente para sus nodos vecinos, considerando para esto solo los nodos en los ciclos internos que ya han sido coloreados. En este caso, consideramos a $T_G$ como una instancia de series-paralelas del grafo, el cual se sabe que es 3-coloreable en tiempo lineal.
5. Si en $T_G$ sólo se tienen ciclos intersectados a pares, entonces $G$ es 3-coloreable. El coloreo de $T_G$ se hace de manera similar al paso anterior, elegimos algún ciclo y lo coloreamos, para colorear el siguiente ciclo usamos un color diferente al de sus nodos vecinos, considerando el ciclo que ya se ha coloreado anteriormente [2].

Entonces, si la estructura topológica de la entrada del grafo pertenece a alguno de de los anteriores casos básicos, entonces $G$ es 3-coloreable y el coloreo puede realizarse eficientemente.

# 5. REDUCCIONES POLINOMIALES SOBRE GRAFOS

La topología del grafo es una parte importante para la determinación del número cromático, ya que el procedimiento que proponemos para el coloreo de grafos embebidos trabaja con esta condición. En el capitulo anterior se menciona como se realiza el coloreo en grafos embebidos, ahora proponemos un procedimiento el cual trabaja sobre un grafo *G* el cual inicialmente no presenta una topología de un grafo embebido, sin embargo, veremos bajo que condiciones podemos transformar un grafo con ciclos intersectados en un grafo con ciclos embebidos [4].

Además, en caso de que las condiciones de suficiencia para el 3-coloreo no se cumplan, presentamos procedimiento eficiente para realizar el coloreo de un grafo con *n*-colores.

## 5.1. TRANSFORMACION DE CICLOS INTERSECCTADOS EN CICLOS EMBEBIDOS.

Veamos como trabaja nuestro procedimiento sobre grafos *G* de grado 3. Por ejemplo consideremos el grafo de *G* de la figura 5.1, que inicialmente no se observa como un grafo con ciclos embebidos, pero al realizar un recorrido sobre el obtenemos como resultado el grafo *G* de la figura 5.2.

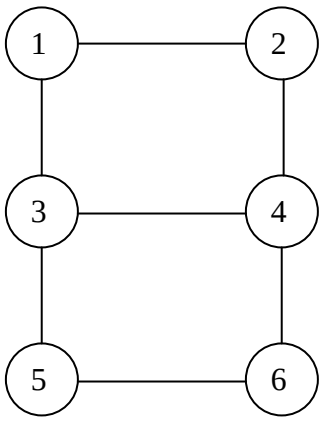



**Fig. 5.1**. Grafo inicial

El propósito principal de la reducción entre grafos es transformar un grafo el cual tenía en su estructura dos ciclos intersectados, después de realizar la transformación se tiene un

grafo equivalente pero donde las dos aristas de retroceso coinciden en un nodo común, lo que obligará a que los dos ciclos se reescriban como ciclos embebidos [4].

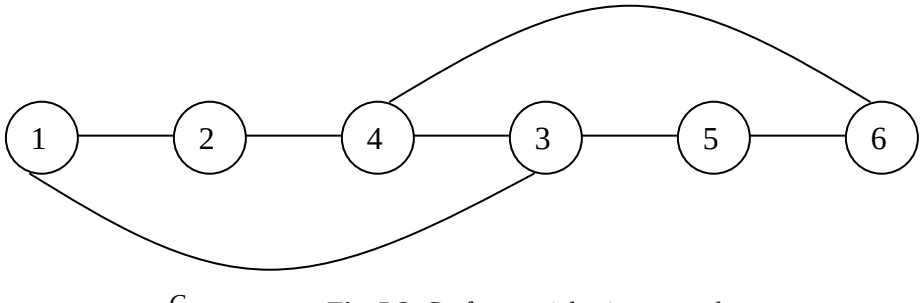



**Fig. 5.2**. Grafo con ciclos intersectados

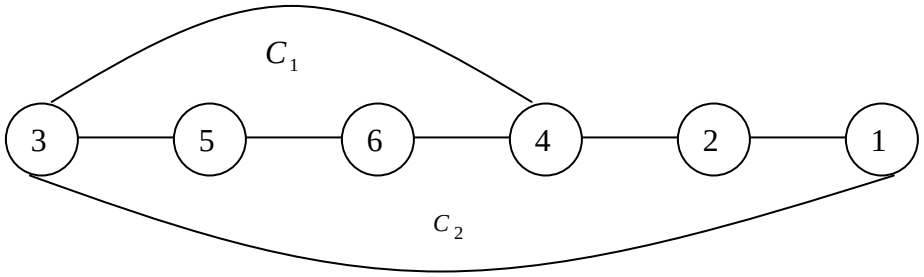



**Fig. 5.3**. Grafo con ciclos embebidos compartiendo nodo

El algoritmo para la transformación de un grafo con dos ciclos intersectados a un grafo con ciclos embebidos, se muestra a continuación.

**PROCEDIMIENTO Transf_Embebidos(G)**

**Entrada:** Un grafo $G$ resultante de aplicar el procedimiento de búsqueda a lo profundo y el conjunto de sus ciclos base $C=\{C_1,C_2,\ldots,C_k\}$.
**Salida:** Reescritura del grafo transformando ciclos intersectados y con arista común en ciclos embebidos.

**Procedimiento**

Para $i$=1 to $k$
a) Si ($C_i$ no está embebido en $C_j \in C$ and $|E(C_i) \cap E(C_j)|$=1) entonces
/* hay que reorganizar a $C_i$ y $C_j$ como ciclos embebidos */

Sea e= $E(C_i) \cap E(C_j)$ /* la arista común de los dos ciclos intersectados */
1) Selecciona el nodo *u* incidente a *e* y a la arista de retroceso de $C_i$
2) Redibuja a $C_j$ a partir de *u* y siguiendo por su arista incidente diferente a *e*.
3) Al terminar de reescribir a $C_j$, su último nodo *v* dibujado es común con $C_i$, ahora se redibuja a $C_i$ a partir de *v* siguiendo su arista incidente diferente a *e*.
4) Al terminar de reescribir a $C_i$, se redibujan todas las aristas faltantes, incluyendo la arista *e* que ahora se convierte en una arista de retroceso.
Continuar [4]

Este procedimiento es de complejidad polinomial en tiempo, puesto que recorre cada uno de los ciclos que existe en el grafo para ver si éste puede quedar embebido dentro de otro de los ciclos base.

## 5.2. COLOREO GENERAL DE UN GRAFO USANDO NODOS CRITICOS.

Primeramente revisaremos algunos resultados que se usaran como preámbulo para el proceso del coloreo de un grafo.

**Lema**. Si *G* es un grafo no regular conexo entonces $\chi(G) \leq \Delta(G)$. Si *G* es regular, entonces $\chi(G) \leq \Delta(G) + 1$.

Para probar el lema, asumimos que *G* es no regular, escogemos un vértice $v_1$ de grado mínimo en *V* como el nodo raíz del árbol generador. Recorremos el grafo resultante en preorden (realizando el recorrido primero por los nodos hijos y después los nodos padres). Cuando cada uno de los vértices $v_i \neq v_1$ es coloreado, el padre de $v_i$ ya no es coloreado. Por consiguiente podríamos decir que $\delta(v_i) - 1$vertices adyacentes ya han sido coloreados. En consecuencia $\chi(v_i) \leq \Delta(G)$. Cuando $v_i$ es coloreado, todos los vértices adyacentes ya han sido coloreados. En consecuencia $\delta(v_1) < \Delta(G)$, podemos concluir que $\chi(v_1) \leq \Delta(G)$. Por consiguiente $\chi(G) \leq \Delta(G)$ [2].

Si *G* es un grafo regular, el procedimiento es como se menciona anteriormente, excepto que $\chi(v_1) \leq \Delta(G)+1$. Por consecuencia obtenemos la conclusión del lema. Note que si *G* es regular, solo un vértice necesita ser coloreado con el color $\Delta(G) + 1$.

Si la búsqueda a lo profundo es utilizada para ordenar los nodos de un grafo completo $K_n$, y después aplicamos color a los nodos del árbol generador en preorden, entonces se puede mostrar que se necesitan usar exactamente *n* colores.

Dado un grafo $G=(V,E)$, y un nodo $v \in V$, el subgrafo inducido por $N(V)$ es denotado como $H(v)$, el cual consiste de $N(V)$ y todas las aristas en *E* entre los vértices de $N(v)$.

**Lema.** Si *G* es *k*-coloreable, entonces para algún $v \in V$, $H(v)$ es ($k$- 1)- coloreable [2].

Desde los grafos 2-coloreables o grafos bipartitos, pueden ser identificados y coloreados (con solo dos colores) en tiempo polinomial. Por el lema anterior, los vecinos de algún vértice en un grafo 3-coloreable pueden ser coloreados con dos colores en tiempo polinomial.

Sea $G = (V, E)$ un grafo con $\chi(G) = m$. Si nosotros removemos una arista $\{u,v\}$de $G$, entonces existen 2 posibilidades, ya sea que $\chi(G - \{u, v\}) = m$ or $\chi(G - \{u, v\}) = m-1$. En el ultimo de los casos, decimos que la arista $\{u, v\}$ es critica. Si un nodo $u$ tiene una arista critica, entonces podemos extender la definición y el nodo $u$ es critico también.

Un grafo $G$ es critico si $\chi(G - \{u, v\}) = \chi(G) - 1$ para todas la aristas $\{ u, v \} \in E$. Si $\chi(G) = m$, decimos que $G$ es $m$-critico.

Es fácil observar que para cada grafo $G$ hay un subgrafo crítico. Si $\chi(G - \{u, v\}) = \chi(G)$ para alguna de las aristas $-\{u, v\} \in E$, podemos eliminar dicha arista. Continuando con la eliminación de las aristas hasta que cada uno de las restantes aristas sean criticas. Entonces el resultado es un subgrafo crítico [2].

De acuerdo a estos últimos resultados, desarrollamos un algoritmo que se presenta a continuación. Sea $G = (V, E)$ un grafo con $|V| = n$, $|E| = m$ y sea $T_G$ un grafo generado al aplicar la búsqueda a lo profundo sobre $G$.

La estrategia general de esta propuesta para colorear $G$ consiste en:
Primero: Reconocer la topología del grafo $G$. Esto se realiza al aplicar la búsqueda a lo profundo sobre el grafo.
Segundo: Probar si $G$ puede ser coloreado con dos colores, un procedimiento de complejidad lineal en tiempo es ejecutado para determinar esta condición.
Tercero: Si $G$ no es 2-coloreable, entonces detectamos al nodo $v$ parte de un ciclo impar de grado máximo y con conflicto máximo para colorear $v$ dentro de los nodos del ciclo impar. Este procedimiento puede ser realizado en tiempo polinomial.
Cuarto: Se colorea $v$ con el color activo, $v$ y sus aristas adyacentes son borradas del grafo actual. Después de ésto, el proceso regresa al paso 1.

El procedimiento es ejecutado de manera iterativa mientras el en el grafo actual no se lleve o se reorganice de tal manera que llegue a ser bipartito. Aquí se muestra el pseudocódigo para este propósito.

PROCEDIMIENTO SELECT_NODE_CANDIDATE($T_G$, $NV$)

> Entrada: $T_G$ es un subgrafo, $NV$ es la vecindad de los vértices que no pueden ser colorea con $k$ colores.
> Salida: $v \in V$ un vértice a ser coloreado.

**Procedimiento**

1. *Vértices* = *minus*($T_G$ ,*NV* ); /* *Vértices* = *V* ($T_G$ ) − *NV* */
2. Escoger $v \in$ *Vértices* tal que /* *v* debe ser un nodo critico */
3. δ(*v*) and δ(*NV* (*v*)) son valores máximos sobre el conjunto de nodos del ciclo impar
4. En otro caso /* Si todos los ciclos impares ya han sido cubiertos por *NV* */
5. Escoger $v \in$ *Vértices* tal que /* *v* tiene el grado máximo en $T_G$ */
6. Si (*maximumOver*_$T_G$ (δ(*v*), $T_G$ ) == true) entonces
7. Return *v*

ALGORITMO SEEK_CHROMATIC_NUMBER

Entrada: *G* es un grafo no dirigido.
Salida: Una aproximación al valor de χ(*G*)

**Procedimiento**

*k* = 3; /* Inicializamos con la clase de color *k* = 3*/
$T_G$ =*dfs*(*G*) /*Los nodos del grafo están ordenados*/

1.- Mientras (is_bipartite($T_G$ )==false) /*Mientras exista un ciclo impar en *G*/
{*NV* = Ø; /**NV* es el conjunto para la clase de color *k* */
2.- Mientras (is_subset(*NV*, *V*($T_G$ ))==true)
a) { *v* = *Select_Candidate_Node*($T_G$ , *NV*);
b) *Color*(*v*) = *k* ; *delete*($T_G$ , *v*);
*add*(*NV*, *N(v)*); /**NV* = *NV* ∪ *N*(*v*)*/
c) *H* = Max_Component($T_G$ ); /*Si $T_G$ esta desconectado entonces hay que considerar el componente con el máximo valor para χ(*G*) */
d) $T_G$ = *dfs(H)* /*Mantiene ordenados los nodos restantes*/
}*k*++;
}
3.- Llam a 2-Coloring($T_G$ ) /*Al final, el grafo resultante es 2-coloreable*/
Return

El procedimiento *Seek_Chromatic_Number* [2] consiste de dos ciclos, uno dentro del otro. En cada iteración del ciclo externo, se determina un conjunto independiente al que se le asigna el color *k*. Esta clase de color *k* es formada por los nodos críticos del grafo con el que se esta trabajando (grafo actual) y se compone de un conjunto independiente $I_k$ de el grafo. Así tendremos, que cada uno de los nodos en el conjunto independiente $I_j$ es

coloreado con el color $j + 2$, es como se muestra en el primer y segundo grafo de la figura 5.5.

El ciclo interno se utiliza para realizar la búsqueda de cada uno de los nodos críticos *v* en el grafo con el que se esta trabando así como para la construcción de las vecindades *NV*(*v*) para el conjunto independiente $I_j$ donde v es como se muestra en el segundo grafo de la figura 5. 4.

Cuando un nodo critico es detectado, este es coloreado y eliminado del grafo al igual que las aristas incidentes a este nodo; como $T_G$ esta cambiando cada vez que hay una nueva iteración del ciclo interno, entonces en algún momento podría ser no conexo por lo que es necesario estar aplicando la búsqueda a lo profundo después de terminado el ciclo interno.

El ciclo externo finaliza cuando el subgrafo restante $T_G$ es bipartito, y entonces $T_G$ es 2-coloreable, así como se muestra en el tercer grafo de la figura 5.5.

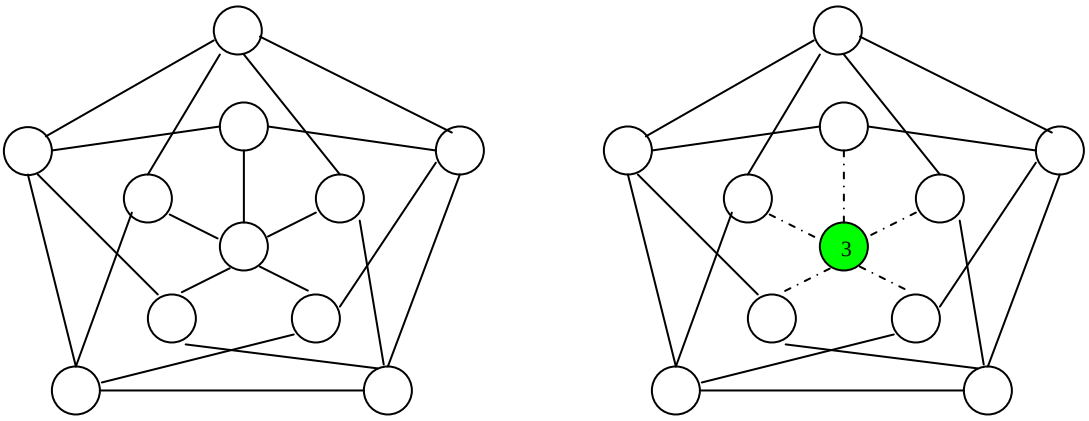



**Fig. 5.4**. Ejecución del algoritmo sobre el grafo de Grötzsch

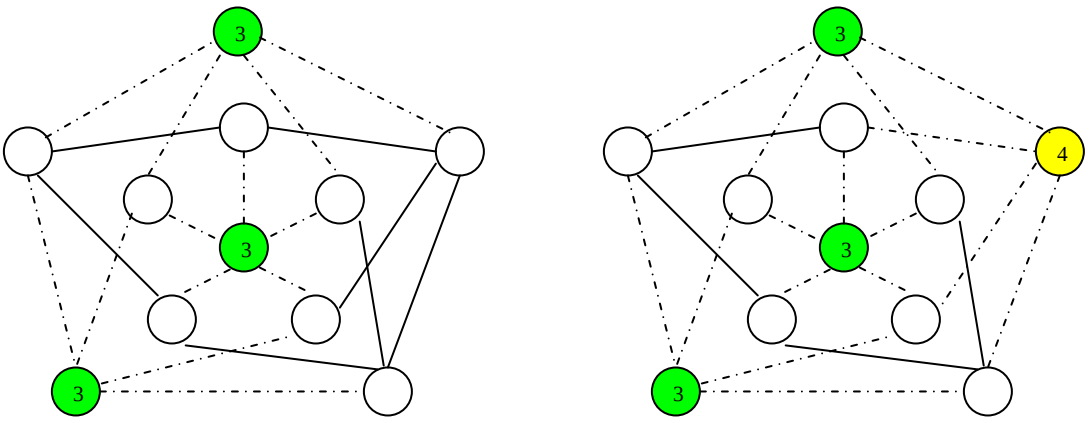



**Fig. 5.5**. Dos iteraciones del Ciclo principal

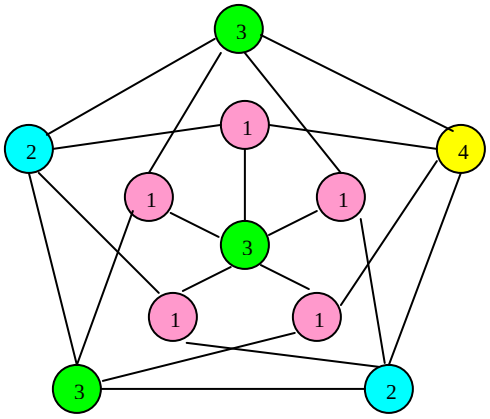



**Fig. 5.6**. Tercera iteración del Ciclo principal (Grafo Resultante)

## 5.3. ANALISIS DE LA COMPLEJIDAD EN TIEMPO DEL ALGORITMO

Dentro del algoritmo *Seek_Chromatic_Number* la operación mas costosa hablando de tiempo dentro del ciclo interno (paso 2); es determinar el nodo critico *v* a ser coloreado, línea (a). Este paso es realizado en tiempo $\Theta(m * n)$, y como ese ciclo itera por lo menos *n* veces, entonces la complejidad de tiempo del ciclo interno es $\Theta(m * n^2)$. Mientras que el ciclo externo (paso 1) es ejecutado en $(m - n)$ veces en el peor de los casos (cuando todos los ciclos de *G* son impares), y la instrucción más costosa es el mismo ciclo interno, tendremos que la complejidad total en tiempo del procedimiento es polinomial y es de orden:

$$\Theta((m - n) * m * n^2) \text{ [1]}$$

Nuestro interés principal será determinar el número promedio de colores usados por el algoritmo, más que el comportamiento en el peor de los casos. Sea $rel = m - n$ la variable usada para denotar la relación entre el numero de aristas y el numero de nodos en cualquier subgrafo dado. La variable *rel* denota el número de ciclos menos 1 en un grafo conectado. Dado que sabemos que para algún grafo *H*, si $m \leq n$ entonces *H* es bipartito o *H* es un ciclo impar simple y entonces *H* es 3-coloreable. Se analizará el valor que toma $rel_i$ para cada subgrafo $T_{G\,i} \subset G$ generado después de cada iteración *i* del ciclo principal (ciclo externo). Sea $K_i$ sea el conjunto independiente construido en tal iteración *i*.

Dado un grafo inicial $G = (V, E)$ con $|V| = n_0$ y $|E| = m_0$ en cada iteración del ciclo principal el numero de nodos y el número de aristas son actualizados como: $n_{i+1} = n_i - |K_i|$

y $m_{i+1} = m_i - |E(K_i)|$, puesto que cada iteración los nodos en $K_i$ son eliminados del grafo activo al igual que las aristas incidentes: $E(K_i)$ [2].

Sea $G_{i+1} = G_i - K_i$ el grafo restante generado de $G_i$ después de haber finalizado la iteración *i* dentro del ciclo principal. Como cada $K_i$ es un conjunto independiente de $G_i$, no hay arista de $G_i$ conectando alguno de los nodos de $K_i$. Además, las aristas en $E(K_i)$ cubren cada nodo de $G_i$, esto es que $E(K_i)$ es una arista que cubre a $G_i$, entonces nosotros tenemos que $|E(K_i)| \geq n_i$ y el numero de aristas restantes en $G_{i+1}$ es $m_{i+1} = m_i - |E(K_i)| \geq m_i - n_i$, así $m_i - n_i$ es una cota superior para $m_{i+1}$.

El comportamiento entre el número de aristas y el número de nodos para cada subgrafo $G_i$, cumple que:

$rel_0 = m_0 - n_0$

$rel_1 = m_1 - n_1 \leq (m_0 - n_0) - n_1 = (m_0 - n_0) - (n_0 - |K_0|) = m_0 - 2\,n_0 + |K_0|$.

$rel_2 = m_2 - n_2 \leq (m_1 - n_1) - (n_1 - |K_1|) = m_0 - 2\,n_0 + |K_0| - (n_0 - |K_0| - |K_1|) = m_0 - 3\,n_0 + 2|K_0| + |K_1|$.

$rel_3 = m_3 - n_3 \leq (m_2 - n_2) - (n_2 - |K_2|) = m_0 - 3\,n_0 + 2|K_0| + |K_1| - (n_0 - |K_0| - |K_1| - |K_2|) = m_0 - 4\,n_0 + 3|K_0| + 2|K_1| + |K_2|$.

.....

$rel_k \leq m_0 - (k+1)\,n_0 + k\,|K_0| + (k-1)\,|K_1| + \ldots + |K_{k-1}|$

Un valor promedio para $|K_i|$, $i = 0, \ldots, k\text{-}1$ es calculado al considerar que $\sum_{v \in K_i} \delta(v) \approx \sum_{j=1}^{|K_i|} \delta(G_i) \geq n_i$, donde $\delta(G_i)$ es el grado promedio del subgrafo $G_i$, y entonces $|K_i| * \delta(G_i) \geq n_i$ y además $|K_i| \geq n_i / \delta(G_i) \approx n/\delta$, siendo $\delta$ el grado promedio del grafo inicial *G*. Así, podemos aproximar el valor de $|K_i|$ por $n/\delta$.

Entonces $rel_k \leq m_0 - (k+1)\,n_0 + \sum_{i=0}^{k-1}(k-i) * (n/\delta) = m_0 - (k+1)n + (n/\delta) * ((k)(k+1)/2)$. Y nosotros queremos saber el numero promedio de iteraciones para el ciclo externo hasta que se llega al caso de $rel_k \leq 0$, esto es cuando: $m_0 + (k(k+1)/2) * (n/\delta) \leq (k+1)n_0$.

Sea $m = m_0$ y $n = n_0$. Como $m = (\delta * n) / 2$ entonces $[(\delta * n)/2 + (n/\delta) * (k(k+1))/2] / n \leq k+1$, así $\delta + (k(k+1))/\delta \leq 2 * (k+1)$. Así,

$$\delta/(k+1) + k/\delta \leq 2 \qquad (1)$$

Para resolver la ecuación (1) expresamos la ecuación en términos de la variable *k*, obteniendo: $0 \leq -(1/\delta)k^2 + (2 - (1/\delta))k + (2-\delta)$ y factorizando el polinomio en *k*, tenemos: $0 \leq -(1/\delta)(k + (\sqrt{4\delta+1} - 1 + 2\delta)/2)(k - (\sqrt{4\delta+1} + 2\delta - 1)/2)$, y el intervalo donde donde *k* *cumple* la ecuación (1), es:

$-(\sqrt{4\delta+1}+1-2\delta)/2 < k < (\sqrt{4\delta+1}-1+2\delta)/2$. Así, el máximo valor para *k* cumpliendo (1) y el cual representa el número aproximado de colores usados para nuestro algoritmo es: $\chi(G) \approx (\sqrt{4\delta+1}+2\delta-1)/2$.

Desde aquí este procedimiento utiliza $\tilde{O}(\delta)$ colores (donde la notación $\tilde{O}$ es utilizada para suprimir factores polilogarítmicos) para colorear el grafo de entrada *G*, siendo $\delta$ el grado promedio inicial del grafo *G*

# 6. CONCLUSIONES

El objetivo principal de este trabajo de tesis fue estudiar el problema del coloreo de un grafo y particularmente, el 3-coloreo. Esto con la finalidad de proponer un nuevo algoritmo competitivo con los actuales. Entre los objetivos particulares y que lograron cumplirse, se tienen los siguientes:

- Diseño de un nuevo algoritmo que trabaja eficientemente, esto es, tiene complejidad polinomial en tiempo an base al tamaño de su grafo de entrada. Este algoritmo realiza el 3-coloreo de su grafo de entrada, si es que es `posible tal 3-coloreo.
- Determinación de las condiciones de suficiencia para que nuestro algoritmo trabaje de forma efectiva (encuentre la solución cuando ésta exista).
- Diseño de una transformación que trabajando en tiempo polinomial transforma un grafo con ciclos intersectados en un grafo con ciclos embebidos, cuando esto es posible. Tal transformación coadyuva a que el grafo de entrada cumpla las condiciones de suficiencia para hacer el 3-coloreo.
- Diseño de un algoritmo de aproximación para el coloreo en general. Para cualquier grafo de entrada, incluyendo los que no son 3-coloreables, el algoritmo haya un coloreo aproximado al coloreo con el número óptimo (mínimo) de colores.
- Determinación del factor de aproximación que garantiza el algoritmo genera anteriormente comentado. Este algoritmo de aproximación también es eficiente, esto es, corre en tiempo polinomial sobre el tamaño del grafo de entrada.

Para la implementación computacional de los algoritmos antes mencionados se desarrolló un sistema con el lenguaje JAVA. Este sistema contiene una interfaz gráfica sencilla pero amigable al usuario.

Requerimientos del Sistema:

- Procesador 300 Mhz o superior
- 32 Mb en memoria RAM
- Instalación de la Maquina Virtual de Java (JVM)
- Espacio mínimo en disco duro (1 Mb)
- Programas desarrollados en este trabajo de tesis

Aportaciones

La teoría de grafos tiene múltiples aplicaciones en diversas áreas. Los grafos nos han permitido modelar situaciones como relaciones de enemistad, modelación de rutas en un plano, así como dentro del campo de la investigación en circuitos electrónicos y en el tráfico de datos en redes.

En nuestro trabajo, un resultado relevante, fue el demostrar que aunque el problema del 3-coloreo es un problema NP-Completo, se pueden construir algoritmos eficientes que

efectivamente colorean con 3 colores a un amplio espectro de los grafos 3-coloreables. Aunque como es de esperarse, no se tienen todas las condiciones necesarias y suficientes que pueden probarse en tiempo polinomial y que nos garanticen con un grafo de entrada es o no, 3-coloreable.

Limitaciones del trabajo

Dentro de los desarrollos alcanzados en la presente tesis, también podemos decir que existen limitaciones dado que solo estamos tratando un tema dentro del área del coloreo de grafos, en este caso, el 3-coloreo. Creemos que aún existen condiciones de suficiencia que se podrían determinar para realizar eficientemente un 3-coloreo, lo que permitiría extender la clase de grafos que pueden ser reconocidas en tiempo polinomial como 3-coloreables.

Trabajos a futuro

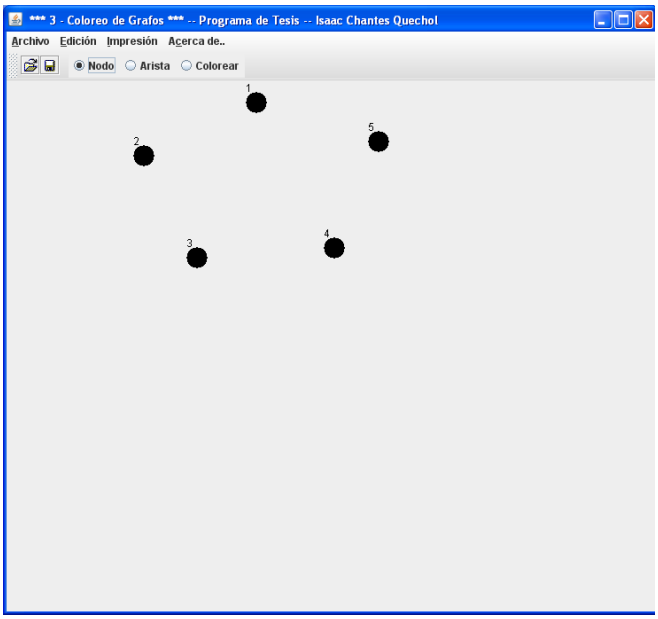
- Continuar extendiendo el algoritmo para un coloreo general tratando de mejorar su factor de aproximación.
- Continuar estudiando las condiciones de suficiencia que nos permita determinar en tiempo polinomial si un grafo de entrada es o no 3-coloreable.
- Estudiar si es posible que existan criterios estrictos para la determinación del 3-coloreo de un grafo.

Anexo A

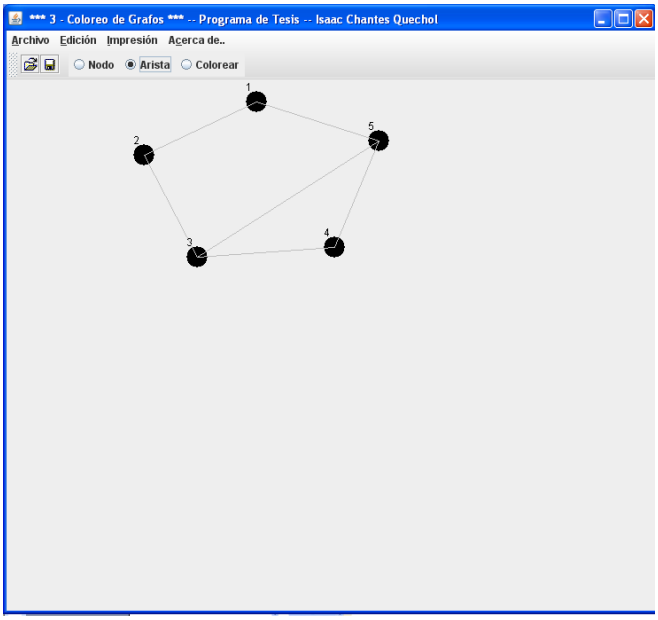
La teoría de grafos aún es un tema de mucho desarrollo y el planteamiento y esta tesis es una de ellas ya que se presentan avances en cuanto al tema del coloreo de grafos y en las reducciones polinomiales que trabajan sobre las topologías equivalentes de un grafo.

La interfaz grafica de nuestro sistema permite apreciar de una manera mas clara la topología del grafo como se muestra a continuación.

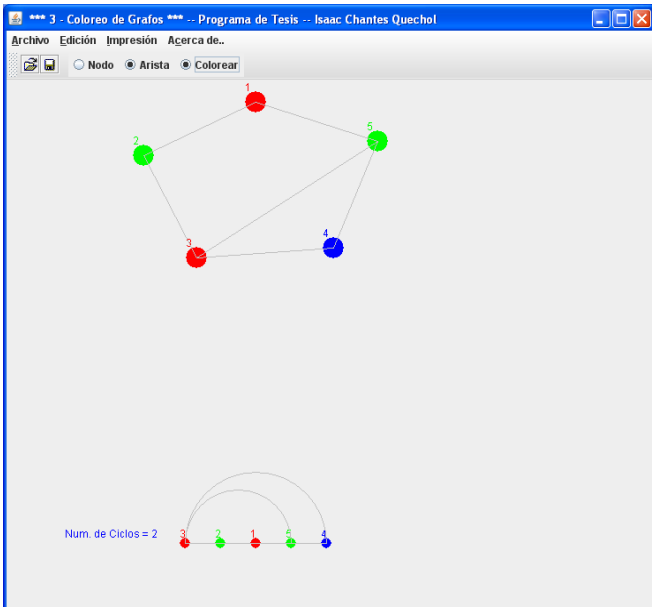
1.- Para poder dibujar los nodos se puede ir al menú *Edición -> Dibujar Nodo* en después dar clic en el área y aparecerá el nodo con su respectivo numero de referencia. O también dar clic en el radio button *Nodo* y ya se podrá dibujar el nodo de la misma manera que la explicada anteriormente




2.- Se puede dibujar una arista si vamos al menú *Edicion->Dibujar Arista* y posteriormente arrastrando el puntero del mouse de un nodo al otro que se desea unir; no importa si no es muy preciso al seleccionar el nodo ya que por default se seleccionara el nodo mas cercano al puntero. Otra manera es dando clic en el radio button *Arista* y arrastra el mouse como se explica.




3.- Finalmente para colorear el grafo tenemos que ir al menú *Edicion-> Colorear Grafo* y se nos mostrara el resultado del coloreo en la pantalla. También podemos dar clic en el radio button *Colorear* y arrojara el mismo resultado




Para Terminar la aplicación podemos cerrar la ventana normalmente. Como se muestra esta aplicación realiza el algoritmo de coloreo como se es requerido.

# 7. REFERENCIAS


[1] R. Beigel, D. Eppstein, 3-Coloring in Time O(1.3289n), Proceedings of the 36th Annual Symposium on Foundations of C.S., 1995.

[2] G. De Ita, M. Contreras, E. Vera, A New Efficient Algorithm for Chromatic Number, Advances in Computer Science and Engineering, Vol. 27, 2007, pp.113-122.

[3] William Kocay, Donald L. Kreher, Graphs, Algorithms, and Optimization, Chapman & Hall/CRC, 2005.

[4] G. De Ita, M. Contreras, A Polynomial Graphical Reduction to Speed Up the Counting of Models for Boolean Formulas, Workshop on Logic, Language and Computation, Nov. 2006.

[5] Gary Chartrand, Ping Zhang, Introduction to Graph Theory, Mc Graw-Hill Int. Edition, 2005.

[6] S. Arora, E. Chlamtac, M. Charikar, New Approximation Guarantee for Chomatic Number, Proceedings STOC, May 2006.

[7] David S. Johnson, The NP-Completeness Column: An Ongoing Guide, J. Algorithms 3, 1982, pp. 89-99.

[8] Garey M., Johnson D., Computers and Intractability a Guide to the Theory of NP-Completeness, W.H. Freeman and Co., 1979.

[9] Hartmanis J., Stearn R., On the Computational Complexity of Algorithms, Trans. American Math Soc., 177 285-306, 1965.

[10] Hopcroft J., Ullman J., Introducción a la teoría de Autómatas, Lenguajes y Computación, CECSA 1993.

[11] Stockmeyer L. J., Classifying the computacional complexity of problems, The Journal of Symbolic Logic, 1987.

[12] De Ita G., Morales G., Propuestas Algorítmicas en el Tratamiento de los Problemas de Satisfactibilidad, Publicaciones Técnicas del CInvestav – IPN, Serie Verde No. 54, 1996.