Universidad Nacional Autónoma de México

Facultad de Ciencias

El problema del clique máximo: Análisis, resolución e implementación

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
ACTUARIA

PRESENTA:
GUADALUPE VILLEDA GÓMEZ

DIRECTORA DE TESIS:
M. EN I.O. MARÍA DEL CARMEN HERNÁNDEZ AYUSO

Ciudad Universitaria, Cd. Mx., 2019

1. **Datos del alumno**
   Villeda
   Gómez
   Guadalupe
   5567910315
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Actuaría
   406086525

2. **Datos del tutor**
   M. en I.O.
   María del Carmen
   Hernández
   Ayuso

3. **Datos del sinodal 1**
   Dra.
   Claudia Orquídea
   López
   Soto

4. **Datos del sinodal 2**
   Mat.
   Ana Lilia
   Anaya
   Muñoz

5. **Datos del sinodal 3**
   Mat.
   Laura
   Pastrana
   Ramírez

6. **Datos del sinodal 4**
   M. en C.
   David Chaffrey
   Moreno
   Fernández

7. **Datos del trabajo escrito**
   El problema del clique máximo: Análisis, resolución e implementación
   206 p
   2019

Wahrlich es ist nicht das Wissen,
sondern das Lernen, nicht das Besitzen
sondern das Erwerben, nicht das Da-Seyn,
sondern das Hinkommen,
was den grössten Genuss gewährt.

No es el conocimiento,
sino el acto de aprendizaje;
y no la posesión,
sino el acto de llegar allí,
lo que concede el mayor placer.

Johann Carl Friedrich Gauss

Agradecimientos

A mis padres, Estela y Luis por ser la principal razón para terminar esta tesis, su apoyo incondicional y ser mis modelos de trabajo y perseverancia.

A mi asesora María del Carmen Hernández Ayuso por aceptar este trabajo, su tiempo, paciencia, dedicación y todas las enseñanzas. Su invaluable apoyo llevó a buen término a este escrito.

A mis sinodales por las correcciones, comentarios y todo el tiempo dedicado para nutrir y enriquecer esta tesis.

A Ivan por estar siempre para mí, por todas las horas que me escuchaste hablar de esto, por tu apoyo en cuestiones técnicas, pero sobre todo gracias por no dejarme abandonar esto que tanto amo.

Índice

Introducción

Con este trabajo se propone el análisis y desarrollo del algoritmo para el problema del clique máximo de David R. Wood, publicado en la revista Operations Research Letters en su volumen 21 de 1997.

En el primer capítulo se dan los conceptos básicos que serán utilizados a lo largo del trabajo. Se presenta el problema del clique máximo como un problema de optimización y su versión como problema de decisión. Además se dan algunas aplicaciones que ha tenido para resaltar la importancia práctica del problema aquí analizado.

Posteriormente se da una breve introducción a la complejidad algorítmica, a los problemas NP Completos y al Teorema de Cook. Todo esto nos permitirá clasificar y demostrar que el problema del clique máximo en su versión de decisión es un problema NP Completo. Además se abordan las diferentes técnicas que se han utilizado para resolver el problema del clique máximo, ya sea de manera exacta o de forma aproximada. Dentro de los algoritmos exactos, encontramos al método de ramificación y acotamiento, el cual será utilizado por nuestro algoritmo. Veremos como su eficacia dependerá de encontrar buenas cotas, que permitan encontrar cuáles soluciones pueden no ser óptimas y de esta manera reducir el espacio solución.

En el capítulo tres se presentan diferentes resultados de teoría de gráficas que encuentran cotas para la cardinalidad del clique máximo, usando problemas relacionados con este como el problema de la

coloración mínima. Además se presenta una clase especial de gráficas llamadas perfectas, en donde la cardinalidad del clique máximo es igual al mínimo número de colores para colorear una gráfica, este último es conocido como el número cromático.

En el capítulo cuatro se presentará el algoritmo del clique máximo (ACM), que hace uso del método de ramificación y acotamiento, por lo que es necesario generar cotas sobre el tamaño del clique máximo. Para lograrlo utiliza como cotas superiores aproximaciones heurísticas del número cromático y del número crómatico fraccional. Mientras que para las cotas inferiores utiliza el heurístico del CLIQUE. Además que la cota inferior inicial en el nodo raíz, es el tamaño de un clique máximo de una subgráfica triangulada de aristas maximal.

Por último, el algoritmo se implementará en el lenguaje de programación Ruby para mostrar que es correcto y medir el tiempo de ejecución con diferentes gráficas.

Capítulo 1

Problema del clique máximo

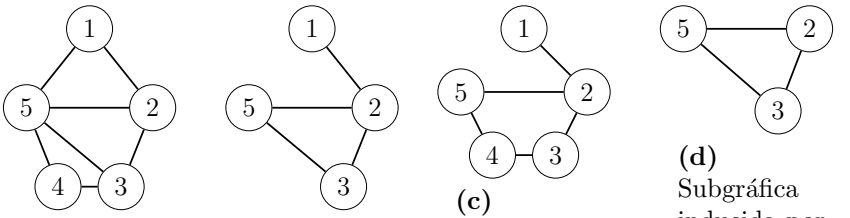
1.1. Conceptos básicos

Una gráfica simple $G = (V, E)$ es un par de conjuntos ajenos, donde $V = \{1, 2, \ldots, n\}$ es el conjunto de vértices de G distinto del vacío y $E \subseteq V \times V$ el conjunto de aristas. Los elementos de E serán de la forma $e = (u, v) = (v, u)$, con $u, v \in V$, y $u \neq v$. El conjunto de vértices de una gráfica G lo denotaremos como V_G, mientras que el de aristas como E_G; en ocasiones, cuando quede claro, utilizaremos la notación abreviada V y E.

El orden de una gráfica G es el número de vértices el cual será denotado como $|G| = |V_G|$ y el tamaño de G es el número de aristas denotado como $||G|| = |E_G|$. Las gráficas pueden ser finitas o infinitas de acuerdo a su orden; todas las gráficas consideradas en este trabajo son finitas.

Si (u, v) es una arista de G, entonces se dice que u y v son vértices adyacentes, además que u y v se conocen como vértices incidentes en la arista. La vecindad de un vértice $v \in V_G$ es el conjunto $N_G(v) = \{u \in V_G \mid (v, u) \in E_G\}$. El grado de v es el número de vértices adyacentes a v; es decir, $d_G(v) = |N_G(v)|$. Un vértice de grado cero se denomina aislado. El mínimo grado y el máximo grado están definidos como $\delta(G) = \text{mín}\{d_G(v) \mid v \in V_G\}$ y $\Delta(G) = \text{máx}\{d_G(v) \mid v \in V_G\}$

respectivamente.



(a) Gráfica G

(b) Subgráfica de G

(c) Subgráfica expandida de G

(d) Subgráfica inducida por $\{2, 3, 5\}$

Figura 1.1: Ejemplos de subgráficas

Una subgráfica [1] de una gráfica G es una gráfica H tal que $V_H \subseteq V_G$ y $E_H \subseteq E_G$. Una subgráfica $H \subseteq G$ es una subgráfica expandida o generadora de G, si todo vértice de G está en H; es decir, $V_H = V_G$. Una subgráfica $H \subseteq G$ es una subgráfica inducida de G si $E_H = E_G \cap (V_H \times V_H)$, o de otra manera, el conjunto E_H de aristas consiste de todas las $e \in E_G$ tal que $e \in E_H$. A cada subconjunto no vacío $A \subseteq V_G$ le corresponde una única subgráfica inducida $G(A) = (A, E_G \cap E(A))$. Ver Figura 1.1.

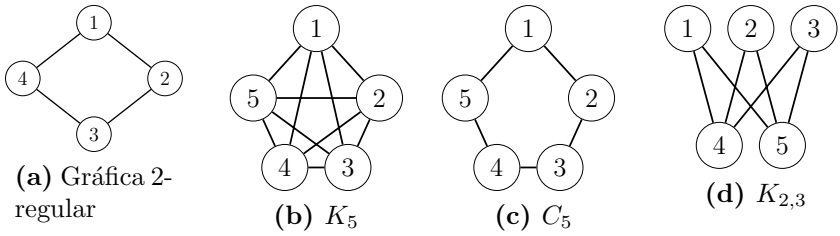
Dos gráficas G y H son isomorfas, si existe $f : V_G \to V_H$ biyectiva, tal que $(u, v) \in E_G$ si y sólo si $(f(u), f(v)) \in E_H$.

Una gráfica $G = (V, E)$ es completa si todos sus vértices son pares adyacentes; es decir, para todo $i, j \in V$, la arista $(i, j) \in E$; la gráfica completa con n vertices es denotada como K_n.

Una gráfica es regular si cada vértice tiene el mismo grado. Llamaremos a una gráfica $k-$regular si cada uno de sus vértices tiene grado k.

[1] Algunos autores utilizan el término gráfica parcial para referirse a las subgráficas.

Una sucesión de vértices $(v_1, v_2, \ldots, v_n, v_1)$, es llamado un ciclo C_n de longitud n, si $(_i, v_{i+1}) \in E$ para $i = 1, 2, \ldots, (n-1)$ y $(v_n, v_1) \in E$ con $n \geq 3$.

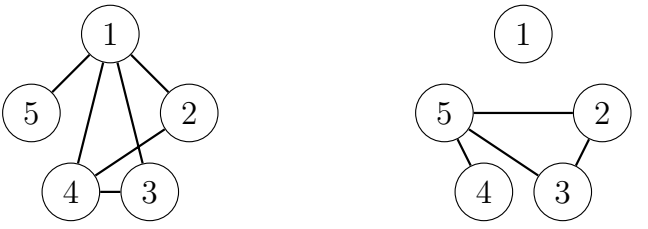


(a) Gráfica 2-regular **(b)** K_5 **(c)** C_5 **(d)** $K_{2,3}$

Figura 1.2: Diferentes tipos de gráficas

Una gráfica $G = (V, E)$ es llamada bipartita si V_G tiene una partición en dos subconjuntos X y Y, donde $X \cap Y = \emptyset$ y $X \cup Y = V$; tal que para cada arista $(u, v) \in E$, $u \in X$ y $v \in Y$. Una gráfica bipartita es completa si para todo $u \in X$ y $v \in Y$ tenemos que $(u, v) \in E$; y se denota por $K_{n,m}$ si $|X| = n$ y $|Y| = m$. En la Figura 1.2 podemos ver ejemplos de los conceptos anteriores.

Si U es algún conjunto de vértices (normalmente de G), escribimos $G - U$ para $G[V - U]$. En otras palabras, $G - U$ es obtenido al borrar de G todos los vértices de $V \cap U$ y sus aristas incidentes. Si $U = \{v\}$, escribiremos $G - v$ en lugar de $G - \{v\}$. Para $F \subseteq E$, escribimos $G - F := (V, E \setminus F)$ y $G + F := (V, E \cup F)$; como en los casos anteriores $G - \{e\}$ y $G + \{e\}$ son abreviados como $G - e$ y $G + e$. Llamaremos a G de aristas maximal *(edge-maximal)*, si G cumple una propiedad dada pero no la gráfica $G + (x, y)$, para vértices no adyacentes $x, y \in V_G$. En forma general, un conjunto S es maximal o minimal en relación a una determinada propiedad P si S satisface P y todo conjunto S' tal que $S \subset S'$ no satisface P.



(a) G **(b)** $\bar{G}$

Figura 1.3: Gráfica G y su complemento

El complemento de una gráfica $G = (V, E)$ es la gráfica $\overline{G} = (V, \overline{E})$, donde $\overline{E} = \{(i, j) | i, j \in V,\ i \neq j\ y\ (i, j) \notin E\}$. Ver Figura 1.3.

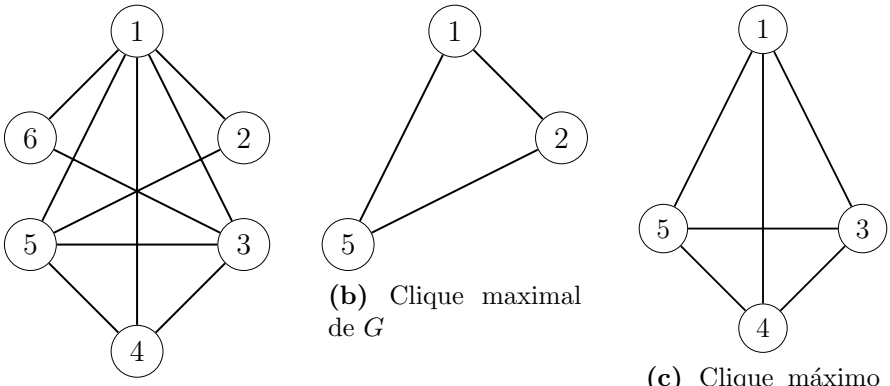
Un clique de $G = (V, E)$ es un conjunto de vértices $S \subseteq V$ tal que para todo par de vértices $x, y \in S$, $(x, y) \in E$. Es decir, la gráfica inducida de G por S es una gráfica completa.

Un clique es llamado maximal si este no es un subconjunto de un clique de cardinalidad mayor. Un clique máximo es un clique maximal con cardinalidad máxima. Ver Figura 1.4. La cardinalidad del clique máximo se denota como $\omega(G)$.

Dada una grafica $G = (V, E)$, un conjunto $H \subseteq V_G$ se llama conjunto independiente o estable si ninguno de los vértices de H es adyacente a otro en el conjunto. El número de independencia de G es el tamaño del conjunto independiente de cardinalidad máxima, el cual será denotado como $\alpha(G)$.

Un camino en una gráfica $G = (V, E)$ es una sucesión de vértices $W_k = (v_1, v_2, \ldots, v_k)$, tal que existe una arista de x_i a x_{i+1} en G para $i \in \{1, \ldots, k-1\}$. Un camino es cerrado si $v_1 = v_k$. La longitud de un camino es el número de aristas que recorre.

Una trayectoria P_k es un camino de longitud k que no repite vértices, por lo tanto tampoco repite aristas.



(a) G

(b) Clique maximal de G

(c) Clique máximo de G

Figura 1.4: Diferencia entre clique máximal y máximo de una gráfica

Una gráfica G es conexa si para todo par de vértices $u, v \in V_G$ existe una trayectoria entre u y v.

Una componente conexa de una gráfica G, es una subgráfica conexa maximal de G.

Una gráfica inconexa está compuesta por dos o más componentes conexas, donde en cada componente hay una trayectoria entre cualesquiera dos vértices.

Una arista $e \in E_G$ es un puente de la gráfica G si $G - e$ tiene más componentes conexas que G.

Un vértice $v \in V_G$ es un vértice de corte si $G - v$ tiene más componentes conexas que G. Un subconjunto $S \subset V_G$ es un conjunto separador si $G - S$ es inconexo. También diremos que S separa los vértices u y v, y que es un $(u, v)-$separador, si u y v pertenecen a diferentes componentes conexas de $G - S$. Ver figura 1.5.

Si G es conexa, entonces v es un vértice de corte si y solo si $G - v$

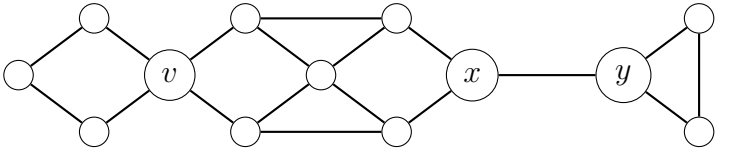


Figura 1.5: v, x, y son vértices de corte de la gráfica y la arista (x, y) es un puente

es inconexa, entonces $\{v\}$ es un conjunto separador.

G es $k-$conexa para $k \in \mathbb{N}$ si $|G| > k$ y $G - X$ es conexa para cualquier conjunto $X \subset V$ tal que $|X| < k$. El entero máximo k tal que G es $k-$conexa es la conexidad $\kappa(G)$.

Un bloque de una gráfica G es una subgráfica maximal, conexa y sin vértices de corte.

Un árbol T es una gráfica conexa y acíclica.

1.2. Problema del clique máximo

Un problema es una cuestión para la cual buscamos una respuesta, este puede contener datos (variables) que no poseen valores específicos al enunciar el problema. Tales datos son llamados parámetros del problema.

Los parámetros determinan una clase de problemas para cada asignación de valores que puedan tomar. A cada asignación específica de valores para los parámetros se le llama instancia del problema.

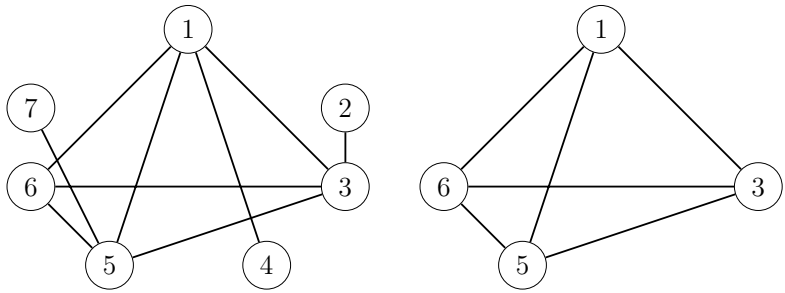
Una solución para una instancia de un problema es una respuesta a la cuestión hecha por el problema. Con estos conceptos ya podemos definir el problema del clique máximo.

Definición 1 (Problema del clique máximo)**.** *Dada una gráfica* $G = (V, E)$ *encontrar un clique máximo.*

Ejemplo 1. *Encontrar el clique máximo para la siguiente instancia:* $G = (V, E)$ *con* $V = \{1, 2, 3, 4, 5, 6\}$ *y* $E = \{(1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (3, 5), (3, 6), (5, 6), (5, 7)\}$

Solución: $S = \{1, 3, 5, 6\}$ y $\omega(G) = 4$.

Llamemos $G(S) = (S, E \cap S \times S)$ a la subgráfica inducida por S, $G(S)$ es el clique máximo de G. En la Figura 1.6b podemos ver el clique máximo de G.



(a) G **(b)** Clique máximo de G

Figura 1.6: Ejemplo 1

Un problema que está relacionado con el del clique máximo es el del máximo conjunto independiente, el cual será definido a continuación.

Definición 2. *(Problema del máximo conjunto independiente). Dada una gráfica* $G = (V, E)$ *encontrar un máximo conjunto independiente.*

Un clique de una gráfica, es un conjunto independiente en el complemento de la gráfica. En particular, si un clique es máximo en G, entonces este será un máximo conjunto independiente en $\overline{G}$ y por lo tanto, $\omega(G) = \alpha(\overline{G})$.

Se puede encontrar fácilmente la solución de un problema cuando la instancia es pequeña y manipulable. Sin embargo, una instancia puede tener un valor muy grande para n y ya no resulta fácil utilizar simple

inspección, se requiere entonces formalizar un proceso de solución para el problema, independientemente de su tamaño.

A este proceso que se describe paso a paso es conocido como algoritmo.

1.2.1. Tipos de problemas

Podemos clasificar los problemas computacionales en problemas de requerimientos y de dificultad.

Los problemas de requerimientos se dividen en seis problemas computacionales:

1. **Problemas de búsqueda**. Encontrar los valores X en los datos de entrada, que satisfagan la propiedad P.

2. **Problemas de estructura**. Transformar los datos de entrada para satisfacer la propiedad P.

3. **Problemas de construcción**. Construir un dato X que satisfaga la propiedad P.

4. **Problemas de optimización**. Encontrar, en los datos de entrada, la mejor X que satisfaga la propiedad P.

5. **Problemas de decisión**. Decidir si una entrada satisface o no la propiedad P.

6. **Problemas adaptativos**. Mantener la propiedad P todo el tiempo.

Se definen cuatro categorias para clasificar a los problemas según su dificultad:

1. **Problemas conceptualmente difíciles**. No se tiene un algoritmo que resuelva el problema, ya que no es posible entender suficientemente el problema.

2. **Problemas analíticamente difíciles**. Se tiene el algoritmo que resuelve el problema, pero no se sabe cómo analizarlo ni cómo se resuelve cada instancia.

3. **Problemas computacionalmente difíciles**. Se tiene el algoritmo, el cual es posible analizar, pero el análisis indica que resolver una instancia requiere un tiempo poco razonable. Esta categoria se divide en dos grupos:

   a) Problemas que se sabe son computacionalmente difíciles.

   b) Problemas que se sospecha son computacionalmente difíciles.

4. **Problemas computacionalmente sin solución.** No se tiene un algoritmo que resuelva el problema, ya que no es factible construir tal algoritmo.

Los problemas de dificultad serán importante en el siguiente capítulo donde se hablará de complejidad, por el momento se abordarán algunas características de los problemas de optimización, ya que el problema del clique pertenece a esta clasificación.

1.2.1.1. Problemas de optimización combinatoria

Desde un punto de vista matématico, un problema de optimización P se puede formular como una 3-tupla $P = (f, SS, F)$, definida como:

$$P = \begin{cases} \text{Máx (o Mín)} & f(x) & \text{Función Objetivo} \\ \text{s. a. (sujeto a),} & & \\ x \in F \subset SS & & \text{Restricciones} \end{cases}$$

Donde f es la función a optimizar (maximizar o minimizar), F es el conjunto de soluciones factibles y SS es el espacio de soluciones.

La resolución de este tipo de problemas ha atraído la atención de numerosos investigadores, principalmente desde la II Guerra Mundial con el florecimiento de la Investigación de Operaciones, esto se debe a las muchas aplicaciones que tienen en la industria, la tecnología y la ciencia.

Los problemas de optimización se pueden dividir de forma natural en dos categorías: aquellos en los que la solución está dada mediante valores reales y aquellos cuyas soluciones son valores enteros.[13]

Dentro de los problemas de optimización con solución entera, se encuentran un tipo particular de problemas denominados *problemas de optimización combinatoria*. De acuerdo con [61] un problema de optimización combinatoria consiste en encontrar un objeto entre un conjunto finito, o al menos numerable de posibilidades. Este objeto suele ser un número natural, un conjunto de naturales, una permutación o una estructura de gráfica (o subgráfica).

Existen muchos ejemplos de problemas de optimización combinatoria, algunos de los más conocidos son: el problema del agente viajero, el problema de la mochila, el problema de coloración de vértices y por supuesto, el problema del clique máximo.

Muchos de estos problemas se han intentado resolver formulándolos como modelos de programacion lineal entera, donde la pertenencia o no de una variable al subconjunto buscado, se representa a través de variables enteras $0 - 1$.

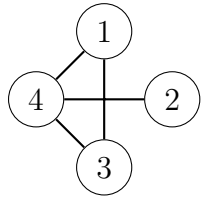


Figura 1.7: Gráfica del ejemplo 2

El problema del clique máximo es formulado como un problema de programación entera de la siguiente manera:

$$\begin{array}{rrcl} Máx & w & = & 1x \\ s.a. & & & \\ & Ax & \leq & 1 \\ & x_i & \in & \{0,1\},\ i \in V(G) \end{array}$$

La variable de decisión es x_i donde,

$$x_i = \begin{cases} 1 & \text{Si el vértice } i \text{ forma parte del clique} \\ 0 & \text{En otro caso} \end{cases}$$

Mientras que A es una matriz para la cual cada columna corresponde a cada uno de los vértices de G y las filas a cada conjunto independiente de G. Para cada entrada de la matriz, $a_{i,j} = 1$ si el vértice correspondiente a la columna j está en el conjunto independiente de la fila i, en caso de no ser así, $a_{i,j} = 0$.

Ejemplo 2. *Formular el problema del clique máximo como uno de programación entera para la gráfica de la Figura 1.7.*

Sea $I(G)$ el conjunto de todos los conjuntos independientes de $V = \{1, 2, 3, 4\}$, por lo que $I(G) = \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{2, 3\}\}$. Entonces A es una matriz de 6×4.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

$$x_j \in \{0,1\}, \quad j \in \{1,2,3,4\}.$$

Por lo tanto, el problema de programación entera queda de la siguiente manera:

$$Máx\ w = x_1 + x_2 + x_3 + x_4$$

$$\begin{array}{lccccccl} s.a. & & & & & & & \\ x_1 & & & & & & \leq & 1 \\ & & x_2 & & & & \leq & 1 \\ & & & & x_3 & & \leq & 1 \\ & & & & & x_4 & \leq & 1 \\ x_1 & + & x_2 & & & & \leq & 1 \\ & & x_2 & + & x_3 & & \leq & 1 \end{array}$$

$$x_j \in \{0,1\}, \quad j \in \{1,2,3,4\}.$$

La solución óptima es $x = (1,0,1,1)$ con $w = 3$. Por lo que el clique máximo lo forman los vértices 1, 3 y 4.

Para el ejemplo anterior es muy fácil encontrar una solución al problema, sin embargo no lo será cuando se trate de solucionar de manera general, ya que como explicaremos más adelante, los problemas de programación entera son uno de los tantos ejemplos NP-completos.

1.2.2. Aplicaciones

El problema del clique máximo tiene múltiples aplicaciones en diversas áreas, tales como: selección de proyectos, teoría de la clasificación, tolerancia de fallos, teoría de códigos, visión computacional,

recuperación de información, teoría de trasmisión de señales, secuenciación de proteínas y ADN, redes sociales, entre otros [64] [37].

Una aplicación interesante se realizó en 1998 en Estados Unidos [71], se trata de un programa desarrollado para el Servicio de Impuestos Internos (*IRS* por sus siglas en inglés), que permite detectar organizaciones dedicadas al fraude fiscal a través de declaraciones de impuestos falsas con el de fin obtener un reembolso. El *ISR* construyó gráficas donde cada vértice representa una declaración de impuestos y se trazaron aristas entre declaraciones sospechosamente similares. Estas gráficas son ingresadas al programa que devuelve los cliques más grandes, por lo que las declaraciones involucradas son posibles organizaciones de fraude, e inmediatemente son objeto de investigación.

Como se puede notar el problema del clique máximo tiene importantes aplicaciones, sin embargo, como veremos en el siguiente capítulo este problema no es fácil de resolver debido a que es un problema NP-Completo.

Capítulo 2

Complejidad algorítmica

2.1. Complejidad temporal de un algoritmo

Como ya mencionamos en el capítulo anterior, un algoritmo es un proceso paso a paso que resuelve a un problema X, si dicho algoritmo puede ser aplicado a alguna instancia E de X y se garantiza la generación de una solución para la instancia.

Suele decirse que un algoritmo es bueno, si para ejecutarlo se necesita poco tiempo y si requiere poco espacio de memoria. Sin embargo, por tradición, un factor más importante para determinar la eficacia de un algoritmo es el tiempo necesario para ejecutarlo.

Muchas veces para medir la complejidad temporal de un algoritmo, se escribe el programa para este y vemos que tan rápido se ejecuta. Sin embargo existen varios factores no relacionados con el algoritmo que afectan el desempeño del programa. Por ejemplo, la habilidad del programador, el lenguaje usado, el sistema operativo e incluso el compilador del lenguaje utilizado, todos estos factores afectan el tiempo necesario para ejecutar el programa. Por lo que es importante hacer un análisis del algoritmo.

El propósito del análisis de algoritmos es predecir el comportamien-

to de un algoritmo sin implantarlo en una máquina específica [55].

El tiempo de ejecución de un algoritmo no es más que el cálculo de operaciones elementales que realiza. Las operaciones elementales son: asignación, comparación y operaciones aritméticas básicas $(+, -, *, /)$. Para calcular el tiempo de ejecución de un algoritmo debemos preguntarnos primero si el algoritmo termina; después, si existe alguna caracterización precisa sobre la cantidad de tiempo (número de pasos) que tomará ejecutarlo.

Siempre se comenzará escogiendo un tipo de operación particular que use el algoritmo y se realizará un análisis matemático con el fin de determinar el número de operaciones necesarias para completarlo.

Suele decirse que el costo de ejecución de un algoritmo depende del tamaño de la instancia del problema, ya que refleja la cantidad de datos de entrada, el cual es denotado con n.

Formalmente, el tiempo de ejecución de un algoritmo A, aplicado a una instancia E, es el número de pasos a efectuar en A para encontrar la solución de E de tamaño n, y lo denotaremos como $\mathrm{T_A}(n)$.

Si $\mathrm{T_A}(n)$ está asintóticamente acotada por $\mathrm{g}(n)$, decimos que $\mathrm{T_A}(n)$ es de orden $\mathrm{g}(n)$ y lo denotamos como: $\mathrm{T_A}(n)$ es $\mathrm{O}(\mathrm{g}(n))$.

De manera más formal tenemos la siguiente definición.

Definición 3. *Sea* $\mathrm{f} : \mathbb{Z}^+ \to \mathbb{R}$ *y* $\mathrm{g} : \mathbb{Z}^+ \to \mathbb{Z}$. *Se dice que* $\mathrm{f}(n)$ *es* $\mathrm{O}(\mathrm{g}(n))$ *si existen constantes* $c > 0$ *y* $n_0 \geq 0$, *tal que* $0 \leq \mathrm{f}(n) \leq c \cdot \mathrm{g}(n)$ *para toda* $n \geq n_0$.

Ejemplo 3. *Mostremos que la función* $n^3 + n$ *es* $\mathrm{O}(n^3)$

$$n^3 + n \;=\; \left(1 + \frac{1}{n^2}\right) n^3 \;\leq\; 2n^3 \;\textit{para}\; n \geq 1$$

Por lo tanto, puede afirmarse que la complejidad temporal es $\mathrm{O}(n^3)$ *porque es posible que* c *y* n_0 *sean* 2 *y* 1, *respectivamente.*

Dos notaciones relacionadas son Ω y Θ.

Definición 4. *Sea* $\mathrm{f} : \mathbb{Z}^+ \to \mathbb{R}$ *y* $\mathrm{g} : \mathbb{Z}^+ \to \mathbb{R}$. *Se dice que* $\mathrm{f}(n)$ *es* $\Omega(\mathrm{g}(n))$ *si existen constantes* $c > 0$ *y* $n_0 \geq 0$ *tal que* $0 \leq c \cdot \mathrm{g}(n) \leq \mathrm{f}(n)$ *para toda* $n \geq n_0$.

Definición 5. *Sea* $\mathrm{f} : \mathbb{Z}^+ \to \mathbb{R}$ *y* $\mathrm{g} : \mathbb{Z}^+ \to \mathbb{R}$. *Se dice que* $\mathrm{f}(n)$ *es* $\Theta(\mathrm{g}(n))$ *si existen constantes* $c, c' > 0$ *y* $n_0 \geq 0$ *tal que* $0 \leq c \cdot \mathrm{g}(n) \leq f(n) \leq c' \cdot \mathrm{g}(n)$ *para toda* $n \geq n_0$.

Si $\mathrm{f}(n)$ es $\Theta(\mathrm{g}(n))$, entonces se dice que f y g tienen la misma tasa de crecimiento.

El desempeño computacional de un algoritmo puede ser analizado como una función del peor de los casos, del caso promedio o del mejor de los casos.

Definición 6 (Peor de los casos)**.** *El desempeño computacional* $\mathrm{T_A}(n)$ *de un algoritmo A sobre una instancia E, de tamaño n, se define como una función del peor de los casos de la siguiente forma:*

$$\mathrm{T_A}(n) = \mathrm{T_A}(E^*) \quad \textit{donde} \quad \mathrm{T_A}(E^*) = \mathrm{máx}\{\mathrm{T_A}(E) : |E| = n\}$$

Es decir, el análisis del peor de los casos es la función definida por el máximo número de pasos tomados por el algoritmo para resolver cualquier instancia de tamaño n. Este análisis nos proporciona una ejecución garantizada del algoritmo. Esto es, el algoritmo nunca requerirá más tiempo del que ha sido especificado por la cota dada.

Definición 7 (Mejor de los casos)**.** *El desempeño computacional* $\mathrm{T_A}(n)$*, de un algoritmo A sobre una instancia E, de tamaño n, se define como una función del mejor de los casos, de la siguiente forma:*

$$\mathrm{T_A}(n) = \mathrm{T_A}(E^*), \quad \text{donde} \quad \mathrm{T_A}(E^*) = \mathrm{mín}\{\mathrm{T_A}(E) : |E| = n\}$$

El análisis del mejor de los casos es la función definida por el mínimo número de pasos tomados por el algoritmo para resolver cualquier instancia de tamaño n. En general este tipo de análisis no es mucha utilidad, salvo para comparaciones especificas.

Definición 8 (Caso promedio)**.** *El análisis del caso promedio nos proporciona una función que depende de la esperanza de $T_A(n)$, y donde $P(E)$ es la probabilidad de que ocurra E.*

$$\mathrm{E}[\mathrm{T_A}(n)] = \sum_{\forall |E|=n} \mathrm{P}(E) \cdot \mathrm{T_A}(E)$$

Es decir al análisis del caso promedio es la función definida por el número de pasos que se espera tome el algoritmo para resolver cualquier instancia de tamaño n. Este análisis depende de la función de probabilidad con la que se distribuyan los datos y resulta ser un análisis más preciso, pero es más complicado y específico. Al cambiar la forma como se distribuyen los datos habrá que rehacer el análisis.

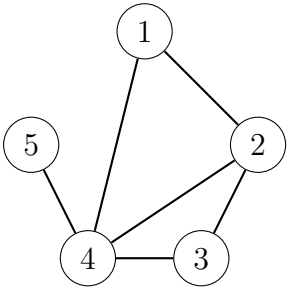


Figura 2.1

En problemas de optimización combinatoria, la entrada del algoritmo es un objeto combinatorio: una gráfica, un conjunto de enteros (posiblemente en vectores o matrices) o una familia de conjuntos finitos, por mencionar algunos.

Para el problema del clique máximo necesitamos una gráfica como entrada para el algoritmo. Ésta puede ser representada de muchas maneras, por ejemplo, se puede asociar a la gráfica $G = (V, E)$ su matriz de adyacencias de tamaño $|V| \times |V|$, definida de la siguiente forma:

$$a_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{en otro caso} \end{cases}$$

La matriz de adyacencias de la gráfica de la figura 2.1 es la siguiente:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Sin embargo, este tipo de representación no siempre es la manera más «económica» para describir una gráfica. Podemos tener una gráfica con $\binom{|V|}{2} = \Theta(|V|^2)$ aristas. Por ejemplo, para una gráfica con 100 nodos y 500 aristas, la representación a través de su matriz de adyacencias requeriría $10,000$ dígitos para guardar todas las entradas.

Una manera más usada para representar una gráfica es mediante sus listas de adyacencias. Para cada nodo $v \in V$ guardamos el conjunto $A(v) \subseteq V$ de nodos adyacentes a éste.

Para la gráfica de la figura 2.1 tenemos que sus listas de adyacencias son las siguientes:

$$\begin{aligned} A(v_1) &= \{v_2, v_4\} \\ A(v_2) &= \{v_1, v_3, v_4\} \\ A(v_3) &= \{v_2, v_4\} \\ A(v_4) &= \{v_1, v_2, v_3, v_5\} \\ A(v_5) &= \{v_4\} \end{aligned}$$

El tamaño de esta representación depende de la suma de la longitud de las listas. En este caso tenemos $2|E|$ entradas. Para la aplicación del algoritmo se trabajará con listas de adyacencias.

2.2. Algoritmos polinomiales y problemas intratables

Podemos encontrar una gran variedad de funciones de desempeño computacional para diferentes algoritmos y los podemos clasificar de acuerdo a su eficacia en problemas eficientes y no eficientes. De hecho, los científicos en ciencias de la computación clasifican los algoritmos en dos grandes grupos, los de tiempo polinomial y los de tiempo exponencial.

Definición 9. *Un algoritmo es eficiente si su desempeño computacional requiere tiempo* $O(P(n))$*, donde* $P(n)$ *es un polinomio sobre el tamaño de la entrada* n*.*

Definición 10. *La clase de todos los problemas que pueden ser resueltos por algoritmos eficientes se denota por* P*, de tiempo polinomial.*

Definición 11. *Un problema es llamado intratable si resulta imposible resolverlo con un algoritmo polinomial.*

Dentro de los problemas intratables tenemos a los algoritmos con desempeño computacional exponencial.

La distinción entre problemas polinomiales y exponenciales tienen particular significancia cuando consideramos la solución de instancias grandes del problema. El Cuadro 2.1 nos muestra las diferentes tasas de crecimiento entre varias funciones de desempeño computacional. Se puede observar como las tasas de crecimiento son mucho más grandes para las funciones exponenciales. Inclusive pudiendo mejorar significativamente las capacidades de cómputo actuales, no se tendría un gran impacto sobre el tiempo de ejecución de los algoritmos. El Cuadro 2.2 muestra como cambiaría el tamaño de la instancia del problema más grande resuelto en una hora si tuvieramos una computadora 100 ó $1,000$ veces más rápida que nuestra máquina actual. Notemos que el tamaño de la instancia más grande resuelto en una hora con el algoritmo con una función de complejidad 2^n, en una computadora $1,000$ veces más rápida, crecería en casi 10, mientras que para el algoritmo con función n^5 el tamaño se cuadruplicaría.

Las funciones de complejidad o funciones de desempeño computacional están definidas como una medida en el peor de los casos, por lo que un algoritmo con función de complejidad 2^n significa que al menos una instancia del problema de tamaño n requiere este tiempo. Muchos instancias de este mismo problema van a requerir menos tiempo. Un ejemplo de un algoritmo con complejidad exponencial es el algoritmo Simplex como lo mostraron Klee y Minty [14], este algoritmo sirve para resolver problemas de programación lineal y ha resultado muy eficiente en la práctica a pesar de su complejidad. Otro caso es el algoritmo de

Función \ Tamaño	10	20	30	40	50	60
n	0.00001 s.	0.00002 s.	0.0003 s.	0.00004 s.	0.00005 s.	0.0006 s.
n^2	0.0001 s.	0.0004 s.	0.0009 s.	0.0016 s.	0.0025 s.	0.0036 s.
n^3	0.001 s.	0.008 s.	0.027 s.	0.064 s.	0.125 s.	0.216 s.
n^5	0.1 s.	3.2 s.	24.3 s.	1.7 min.	5.2 min.	13 min.
2^n	0.001 s.	1.0 s.	17.9 min.	12.7 días	35.7 años	366 siglos
3^n	0.059 s.	58 min.	6.5 años	3855 siglos	2×10^8 siglos	1.3×10^{13} siglos

Cuadro 2.1: Comparación de funciones de desempeño computacional polinomial y exponencial.

Fuente: Michael R. Garey y David S. Johson, Computers and Intractability: A Guide to the Theory of NP-Completeness.
USA: W.H. Freeman and Company 1979, pag. 7

	Tamaño de la instancia más grande resuelto en una hora		
Función de complejidad	con una computadora actual.	con una computadora 100 veces más rápida.	con una computadora 1000 veces más rápida
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Cuadro 2.2: Efecto del mejoramiento tecnológico sobre el tamaño de la instancia para algoritmos con desempeño polinomial y exponencial.

Fuente: Michael R. Garey y David S. Johson, Computers and Intractability: A Guide to the Theory of NP-Completeness.
USA: W.H. Freeman and Company 1979, pag. 8

Ramificación y Acotamiento para el problema de la mochila el cual, dependiendo de la instancia , lo resuelve de manera exitosa, aunque para este algoritmo así como también para el anterior, su función de complejidad es exponencial.

Hay muchos problemas para los cuales se conocen algoritmos de tiempo no polinomial. Algunos de ellos pueden ser resueltos por algoritmos eficientes aún no conocidos, sin embargo, se tiene la certeza de que muchos problemas no pueden ser resueltos con algoritmos eficientes (polinomiales), por lo que debemos ser capaces de identificar y clasificar tales problemas, para no malgastar tiempo buscando algoritmos inexistentes.

Los problemas se pueden clasificar en tres categorías:

1. Problemas para los cuales sí se han encontrado algoritmos de tiempo polinomial.

2. Problemas que se han desmostrado ser intratables.

   Hay dos tipos de problemas en esta categoría. El primer tipo consiste en los problemas que requieren una respuesta no polinomial, como lo es determinar circuitos hamiltonianos: si tuvieramos una arista de un vértice a cada uno de los otros vértices, habría $(n-1)!$ circuitos halmitonianos, el algoritmo que solucione este problema debería mostrar todos estos circuitos, lo que significa que tal requerimiento no es razonable. El segundo tipo de intratabilidad ocurre cuando los requerimientos del problema son razonables y podemos probar que el problema no puede ser resuelto en tiempo polinomial. Se han encontrado relativamente pocos problemas de este tipo, los llamados problemas no-decidibles, *undecidable problems*. El más famoso de este tipo es el *Halting Problem*, donde se toma como entrada cualquier algoritmo A y cualquier entrada E para A, el problema consiste en decidir si el algoritmo A se interrumpirá al aplicarlo sobre E. En 1936, Turing demuestra que este problema es no decidible. Durante los años 50 y 60 se construyeron problemas de forma artificial y con características específicas para los cuales se demostró su intratabilidad. A

principios de la década de los 70's se desmuestra que algunos problemas de decisión no construidos artificialmente, son intratables.

3. Problemas para los cuales no se ha podido demostrar que son intratables, pero que no se ha encontrado un algoritmo polinomial que los solucione.

   Esta categoría incluye cualquier problema para el cual un algoritmo en tiempo polinomial no se ha descubierto y que además no se ha podido demostrar que tal algoritmo no es posible. Algunos de estos problemas son: el problema del agente viajero, el problema del clique máximo y el problema de la m-coloración para $m > 2$. Se han encontrado heurísticos para estos problemas, que como veremos más adelante resultan ser eficientes sin embargo no siempre obtienen el resultado óptimo. Esto es existe un polinomio en n que acota el número de veces el número de operaciones elementales que se efectúan sobre instancias tomados de un conjunto restringido. Sin embargo, tal cota polinomial no existe para todo el conjunto de instancias. Para mostrar esto, es necesario encontrar alguna secuencia infinita de instancias para los cuales ningún polinomio en n acote el número de operaciones elementales que se ejecutan durante el algoritmo.

Existe una interesante relación entre muchos de los problemas en esta categoría. El desarrollo de tales relaciones llevo a la creación de la Teoría NP [22].

2.3. Problemas NP-Completos

Quizás esta teoría sea una de las más importantes en ciencias de la computación. El investigador más importante en este campo, el profesor Stephen Arthur Cook de la Universidad de Toronto, fue galardonado con el Premio Turing por sus aportaciones en esta área de investigación. Sin duda, la teoría de los problemas *NP-Completos* constituye una de las teorías más apasionantes y desconcertantes de las ciencias

de la computación.

Desarrollaremos las nociones básicas de la teoría de los problemas *NP-Completos*, para que al final sea posible demostrar que el problema del clique máximo corresponde a esta clase.

Lo más conveniente para desarrollar la teoría es restringirnos a los problemas de decisión. Un problema de decisión es aquel cuya salida es simplemente si o no. Cada problema de optimización tiene un correspondiente problema de decisión.

Para el problema de optimización del clique máximo el problema de decisión asociado es:

Definición 12 (Problema de decisión del clique máximo)**.** *Dada una gráfica $G = (V, E)$ y un entero positivo k, determinar si existe un clique con al menos k vértices.*

2.3.1. Reducciones en tiempo polinomial

Sean P y Q dos problemas, supóngase que una instancia arbitrario E de P puede solucionarse transformando a la instancia E en una instancia E' de Q, resolviendo para E' y reduciendo la solución S' en la solución S para E.

Definición 13. *Si existen algoritmos, C_{E_j} para transformar una instancia E en otro E' y algoritmos C_S para convertir la solución S' en S se dice entonces que existe una reducción de P en Q, la cual se denota $P \propto Q$.*

Generalmente se realiza una transformación de P en Q cuando P es muy difícil de resolver de manera directa y Q es más fácil.

Si los algoritmos para convertir los instancias y las soluciones tienen desempeño computacional de orden polinomial, se dice que la transformación es polinomial.

Ejemplo 4 (Transformación polinomial del problema de decisión del máximo conjunto independiente al problema de decisión del clique

máximo). *Un conjunto independiente en una gráfica $G = (V, E)$ es un subconjunto $S \subseteq V$ tal que $(x, y) \notin E$ para toda $x, y \in S$. El problema de decisión del máximo conjunto independiente esta definido de la siguiente manera:*

Definición 14 (Problema de decisión del máximo conjunto independiente). *Dada una gráfica $G = (V, E)$ y un entero k, se debe verificar si existe un conjunto independiente de tamaño al menos k.*

Por lo que queremos mostrar que el problema del máximo conjunto independiente (decisión) $\propto$ problema del clique máximo (decisión).

Sea $I = (G, K)$ un instancia del problema del máximo conjunto independiente, donde $G = (V, E)$ es una gráfica y K es un entero positivo. El algoritmo de transformación construye un instancia $I' = (\overline{G}, K)$ del problema del clique máximo, donde $\overline{G} = (V, F)$ es la gráfica y su conjunto F esta definido de la siguiente forma:

$$(x, y) \in F \Leftrightarrow (x, y) \notin E$$

para toda $x, y \in V$, $x \neq y$. $\overline{G}$ es conocida como la gráfica complemento de G. $\overline{G}$ puede ser construida en un tiempo $O(n^2)$, donde $n = |V|$. Es fácil ver que $\overline{G}$ tiene un clique de tamaño k si y sólo sí G tiene un conjunto independiente de tamaño k. Por lo tanto el algoritmo de transformación es una transformación polinomial.

Muchos ejemplos de reducciones polinomiales se han hecho en problemas combinatorios. Por ejemplo, Dantzig en 1960 [23] redujo muchos problemas de optimización combinatoria al problema general de programación entera binaria. Edmonds (1962) [26] redujo el problema de cobertura de aristas con el mínimo número de vértices y el problema del máximo conjunto independiente al problema general del conjunto de cobertura (*set covering problem*) [34].

El objetivo de este método es buscar problemas equivalentes cuando no es posible encontrar algoritmos eficientes en tiempo polinomial. La clase de los problemas $NP - Completos$ abarca cientos de problemas de este tipo que resultan ser equivalentes.

2.3.2. El no determinismo y el teorema de Cook

Los fundamentos de la teoría de los problemas $NP-Completos$ inician con el artículo de Stephen Cook presentado en 1971, titulado «*The Complexity of Theorem Proving Procedures*» [22], donde aparece el famoso Teorema de Cook que es parte de una teoría denominada complejidad computacional.

Ahora hablaremos de la noción del no determinismo, la cual no es muy intuitiva. Debemos pensar en un algoritmo no determinista como una noción abstracta y no como una meta realista. El concepto de no determinismo es más importante para el desarrollo de la teoría y la explicación de la existencia de la clase de problemas $NP-Completos$ que para las técnicas que usa la teoría.

Un algoritmo no determinístico tiene todas las operaciones clásicas de uno determinístico y posee una propiedad no determinística muy importante denominada elección-nd *(nd-choise)*, que es usada para manipular selecciones de una forma poco usual. La propiedad elección-nd también es llamada propiedad adivinadora *(guessing)*.

Sea L un lenguaje que nos sirva para reconocer. Dada una entrada x, un algoritmo no determínistico efectúa pasos no determinísticos intercalando el uso de la elección-nd y al final decide si acepta o no a la entrada x. La diferencia clave entre un algoritmo determinístico y uno no determinístico está en la forma en que ellos reconocen el lenguaje.

Definición 15. *Un algoritmo reconoce un lenguaje L si la siguiente condición se satisface: Dada una entrada x, es posible convertir cada elección-nd encontrada durante la ejecución del algoritmo en una elección real tal que la salida del algoritmo acepte a x si y sólo si $x \in L$.*

En otras palabras, el algoritmo debe proveer al menos un posible camino para las entradas al lenguaje L para que sean salidas aceptables, y no debe generar o proveer ningún camino para que entradas que no pertenezcan a L lleguen a ser salidas aceptables.

Una entrada $x \in L$ puede tener varias rutas razonables. Requerimos que el algoritmo tenga al menos una buena secuencia de elecciones

para $x \in L$. Por otro lado, para cada entrada $x \notin L$, debemos tener una salida rechazada sin importar cual selección sea sustituida por la elección-nd. El tiempo de ejecución para una entrada $x \in L$ es la longitud de la mínima ejecución sobre las secuencias de elecciones que lleve a una salida aceptable. El desempeño computacional de un algoritmo no determinístico se refiere al tiempo de ejecución en el peor de los casos, para las entradas $x \in L$, las entradas que no pertenecen a L son ignoradas.

Ejemplo 5 (Un algoritmo no determinístico)**.** *Se dice que un conjunto M de aristas es un acoplamiento si cualesquiera dos aristas en M no tienen un vértice en común. Un acoplamiento M en una subgráfica es máximo si no existe otro acoplamiento M' tal que $|M'| > |M|$. Un acoplamiento en una gráfica es perfecto si cada vértice de la gráfica es incidente a una arista en el acoplamiento.*

Considere el problema de decidir si una gráfica dada $G = (V, E)$ tiene un acoplamiento perfecto. A continuación daremos un algoritmo no determinístico para este problema.

1. *Hacemos M un conjunto de aristas, el cual está inicialmente vacío.*
2. *Examinamos todas las aristas de G, una arista $\boldsymbol{e}$ a la vez.*
3. *Usamos la elección-nd para decidir si incluimos o no a la arista $\boldsymbol{e}$ en el acoplamiento M.*
4. *Una vez examinadas todas las aristas de la gráfica, verificamos si M es un acoplamiento perfecto.*

*La verificación puede hacerse en tiempo polinomial, pues sólo tenemos que determinar si M contiene exactamente $|V|/2$ aristas y si cada vértice incide únicamente en una arista de M. La salida del algoritmo será **sí**, si M es un acoplamiento perfecto y **no** en otro caso. Este resulta ser un algoritmo no determinístico correcto para el problema del acoplamiento perfecto, ya que:*

1. *Si un acoplamiento perfecto existe, entonces hay una secuencia de elecciones que llevará a la salida exitosa en M.*

2. *El algoritmo dice **sí**, solamente si la existencia de un acoplamiento perfecto fue probada por la verificación.*

Observemos que la elección-nd ayuda a seleccionar un candidato a ser conjunto que forme el acoplamiento perfecto, no construyó el acoplamiento perfecto. De hecho no estamos solucionando el problema, sólo estamos proponiendo un candidato para ser la solución. Si el candidato propuesto resulta ser la solución, el algoritmo de verificación deberá ser capaz de reconocerlo como una solución al problema.

Los algoritmos no determinísticos son muy poderosos pero su fuerza no es ilimitada. No todos los problemas pueden ser resueltos eficientemente por algoritmos no determinísticos.

Definición 16 (Clase NP)**.** *La clase de problemas para los cuales existe un algoritmo no determínistico cuyo desempeño computacional es polinomial en el tamaño de la entrada, es llamada clase NP.*

Parece razonable creer que los algoritmos no determinísticos son más poderosos que los determinísticos, pero ¿en realidad lo son? Una manera de probarlo es exhibiendo un problema NP que no esté en P. Nadie ha sido capaz de hacerlo. En contraste, si queremos probar que las dos clases son iguales, P = NP, entonces tenemos que mostrar que todo problema que pertenece a NP puede ser resuelto por un algoritmo determínistico en tiempo polinomial. Nadie lo ha probado tampoco, además pocos creen que sea cierto. El problema de determinar la relación entre P y NP es conocido como el problema ¿P = NP? [1].

Ahora definiremos dos clases, las cuales contienen importantes problemas dentro de la misma clase, todos equivalentes unos con otros.

Definición 17 (Problema NP-Duro)**.** *Un problema X es llamado problema NP-Duro, (NP-Hard), si cada problema en NP es polinomialmente reducible a X.*

[1] Nadie ha sido capaz aún de dar una respuesta final a este problema, haciéndolo uno de los grandes problemas no resueltos de la matemática. El Clay Mathematics Institute está ofreciendo una recompensa de un millón de dólares a quien logre dar una demostración de que P = NP o P $\neq$ NP.

Definición 18 (Problema NP-Completo)**.** *Un problema X es llamado problema NP-Completo, (NP-Complete), si*

1. *El problema X pertenece a* NP
2. *El problema X es NP-Duro.*

La definición de NP-Dureza *(NP-Hardness)* implica que si se prueba que cualquier problema NP-Duro pertenece a P, entonces la prueba debería implicar que P = NP. La clase NP-Duro puede ser descrita como el conjunto de problemas que son al menos tan difíciles como un problema de NP. Dentro de la clase NP tenemos que los problemas más costosos, son los que pertenecen a los NP-Completos, esta clase la podemos definir alternativamente como la intersección entre NP y NP-Duro.

Los problemas que pertenecen a la clase NP-Duro, no necesariamente pertenecen a la clase NP, por tanto, éstos no necesariamente son problemas de decisión. Además tenemos que si para un problema de optimización X su versión de decisión es NP-Completo, entonces X es NP-Duro.

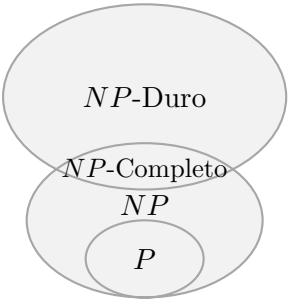


Figura 2.2: Diagrama de clases de complejidad

Cook en 1971 [22] demuestra que el problema de satisfacibilidad boleana, también llamado SAT, es NP-Completo, por lo que lo convierte en el primer problema perteneciente a esta clase. A partir de este problema, fue fácil demostrar que otro problema X también lo es, reduciendo polinomialmente el problema SAT (o cualquier otro problema NP-Completo) a X.

Lema 19. *Un problema X es NP-Completo si*

1. *X pertenece a NP*
2. *Y es polinomialmente reducible a X, para algún problema Y que sea $NP - Completo$*

Demostración. Para probar que X es NP-Completo se deben cumplir las condiciones de la definición 18. La primer condición se cumple por hipótesis. Por lo que sólo nos queda demostrar la condición 2. Tenemos que Y es NP-Completo, entonces Y está en NP-Duro, por lo que cada problema en NP es polinomialmente reducible a Y. Además tenemos que Y es polinomialmente reducible a X. Entonces, por transitividad, cada problema en NP es polinomialmente reducible a X. Por lo que X es NP-Duro, cumpliendose así la condición 2 de la definición de NP-Completo. Por lo tanto X es NP-Completo. □

Es mucho más fácil probar que dos problemas son polinomialmente reducibles que probar la condición 2 de la definición 18 directamente.

Poco después de que Cook dio a conocer su resultado, Karp en 1972 [47] encontró y demostró que 24 importantes problemas son NP-Completos. Desde entonces, a la fecha, cientos o quizá miles de problemas han sido clasificados como NP-Completos.

2.3.2.1. El teorema de Cook y la demostración de que el clique máximo es NP-Completo

Ahora describiremos el problema que Cook probó como NP-Completo [22] y mencionaremos la idea principal de la prueba. El problema es el de satisfacibilidad boleana, SAT *(Satisfiability)*.

Sea S una expresión lógica en Forma Normal Conjuntiva *(Conjunctive Normal Form)*, CNF. Esto es una conjunción de cláusulas, donde cada cláusula es una disyunción de variables, donde una variable y su negación no pueden aparecer en la misma cláusula.

Ejemplo 6. *Considere la expresión lógica* $S = (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$*, es una expresión en Forma Normal Conjuntiva. La variable* $\bar{x}$ *representa la negación de* x*. Cada variable tiene valor* 0 *o* 1*, falso y verdadero respectivamente.*

Se dice que una expresión lógica se satisface si existe una asignación de valores de verdad, ceros y unos, a sus variables tal que el valor de la expresión sea 1, es decir verdadero.

Definición 20 (Problema SAT)**.** *Consiste en determinar si una expresión lógica dada se satisface sin necesidad de encontrar la asignación que la satisface.*

Para el ejemplo 6, S si se satisface, ya que la asignación $x = 1$, $y = 1$ y $z = 0$ la satisface.

Teorema 21 (Teorema de Cook)**.** *El problema SAT es NP-Completo [22].*

El problema del SAT es NP ya que es posible adivinar una asignación de verdad y verificar en tiempo polinomial que la expresión se satisface. La idea de la prueba de que el SAT es NP-Duro es que una máquina de Turing, aún una no determinista, y todas sus operaciones sobre una entrada dada, puede ser descrita como una expresión lógica. Por descrita queremos decir que la expresión será satisfecha si y sólo si la máquina de Turing termina con un estado aceptable para tal entrada. Esto no es fácil de hacer, pues tal expresión puede llegar a ser muy complicada, aunque su tamaño sea polinomial con respecto al número de pasos que la máquina de Turing realiza. Por lo tanto, cualquier algoritmo NP puede ser descrito por un instancia del problema SAT.

Teorema 22. *El problema del clique máximo es NP-Completo*

Demostración. El problema del clique máximo pertenece a la clase NP ya que podemos sugerir un subconjunto de al menos k vértices y verificar si es un clique en tiempo polinomial.

Reduciremos el problema del SAT al problema del clique máximo. Sea E una expresión lógica arbitraria en CNF: $E = E_1 \wedge E_2 \wedge \cdots \wedge E_m$. Sea $E_i = (x \vee v \vee z)$, usaremos tres variables para ilustrar. Asociamos una columna de tres vértices con las variables en E_i, aún si ellas aparecen en otras cláusulas. Es decir; asociamos una columna de vértices por cada cláusula, dicha columna tendrá tantos vértices como variables en la cláusula. El problema es cómo conectar estos vértices tal que G contenga un clique de tamaño mayor o igual a k si y sólo si E se satisface.

Nótese que tenemos libertad de elegir el valor de k, ya que queremos reducir el SAT al problema del clique, lo cual significa resolver el

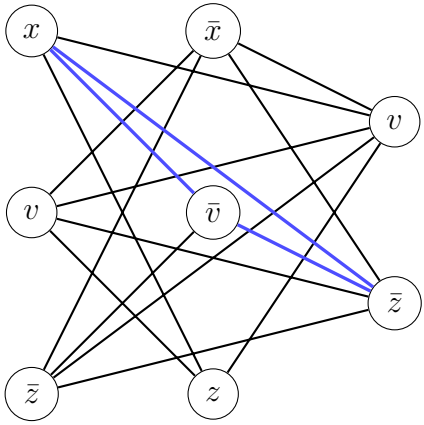


Figura 2.3: Gráfica para el ejemplo 7, el clique está formado por los vértices x, $\bar{v}$ y $\bar{z}$

SAT usando una solución del problema del clique. Una solución para el clique deberá funcionar para toda k. Esta es una importante flexibilidad que es usada frecuentemente en las pruebas de NP-Completos. Elegimos $k = m$ como el número de cláusulas.

Las aristas de G se definen de la siguiente forma: vértices de la misma columna son vértices asociados con variables de la misma cláusula y estos no están conectados. Vértices de diferentes columnas están casi siempre conectados, al menos que correspondan a la misma variable forma complementaria. Esto es, la única forma en que no conectamos dos vértices de diferentes cláusulas es cuando uno corresponde a la variable x y el otro a $\bar{x}$. G se construye en tiempo polinomial.

Ejemplo 7. *Para el instancia* $E = (x \vee v \vee \bar{z}) \wedge (\bar{x} \vee \bar{v} \vee z) \wedge (v \vee \bar{z})$ *se tiene la gráfica de la figura 2.3.*

Tomemos la asignación $x = 1$, $v = 0$ *y* $z = 0$ *con la cual la expresión es verdadera. Entonces tenemos un clique de tamaño* $k = 3$, *formado por los vértices* $x, \bar{v}$ *y* $\bar{z}$

Afirmamos que G tiene un clique de tamaño mayor o igual a m si y sólo si la expresión E se satisface. En efecto, la construcción garantiza que el clique de tamaño máximo no excede m.

⇐) Supongamos que E se satisface. Entonces existe una asignación de verdad tal que cada cláusula contiene al menos una variable cuyo valor es 1. Elegiremos el vértice correspondiente a esa variable para el clique. Si más de una variable en una cláusula tiene valor 1, elegimos arbitrariamente. El resultado es un clique, ya que la única vez que dos vértices de dos columnas no están conectados es cuando ellos son el complemento uno del otro, lo cual, por supuesto, no puede suceder en una asignación de verdad consistente.

⇒) Asumimos que G contiene un clique de tamaño mayor o igual a m. El clique debe contener exactamente un vértice de cada columna, ya que dos vértices de una misma columna nunca están conectados. Asignamos a las correspondientes variables el valor de 1. Si algunas variables no están asignadas de esta manera, pueden ser asignadas arbitrariamente. Como todos los vértices en el clique están conectados entre sí, podemos asegurar x y $\bar{x}$ nunca están conectados, y esta asignación de verdad es consistente.

□

2.4. Tratando con el problema del clique máximo

En el sección anterior demostramos que el problema de decisión del clique máximo es NP-Completo, por lo que su versión de optimización es NP-Duro, es decir, aún no lo podemos resolver con un algoritmo en tiempo polinomial y probablemente no exista tal algoritmo; sin embargo requiere una solución debido a sus numerosas aplicaciones en la vida cotidiana.

Tenemos dos maneras para resolver problemas de la clase NP-Duros, una es desarrollar un algoritmo que nos proporcione una solución exacta y la otra es un algoritmo cuya salida sea una solución aproximada. Estas dos maneras de hacerlo serán descritos en lo largo de esta sección, así como también se hará un pequeño recorrido por algunos de los algoritmos que se han utilizado para tratar de resolver el problema del clique máximo y el del máximo conjunto independiente.

2.4.1. Algoritmos exactos

Los algoritmos exactos dan una solución óptima para problemas NP-Duros, aunque el tiempo de ejecución es inevitablemente exponencial. Sin embargo, esta forma de resolver un problema no es una mala idea si el tamaño de la entrada no es grande. Dentro de estos algoritmos encontramos a los algoritmos de enumeración, estos a su vez se clasifican en algoritmos enumerativos explícitos y enumerativos implícitos o parciales.

2.4.1.1. Algoritmos enumerativos explícitos

En general, un conjunto de soluciones de un problema entero o combinatorio se puede suponer con un número finito de soluciones factibles. Un método directo para resolverlos es enumerar exhaustivamente (o explícitamente) todas y cada una de las soluciones factibles. En este caso, la solución óptima está determinada por la solución o las soluciones que den el mejor (máximo o mínimo) valor de la función objetivo.

El inconveniente de esta técnica es que el número de soluciones factibles puede llegar a ser demasiado grande y es probable no encontrar la solución óptima en un tiempo de cómputo razonable.

El primer algoritmo para enumerar todos los cliques de una gráfica arbitraria es el trabajo de Harary y Ross [40] en 1957, el cual estaba motivado por un problema con datos sociométricos.

Al trabajo de Harary y Ross le siguieron muchos otros como el de Maghout [54], Paull y Unger [65], Bonner [15], Marcus [56], y Bednarek y Taulbee [9]. Aunque todos estos problemas son de diferentes campos de investigación, cada uno de elllos enumeran todos los cliques de una gráfica.

En 1973, dos nuevos algoritmos son propuestos usando el método

de vuelta atrás o retroceso *(backtracking)*, uno de ellos es el propuesto por Akkoyunlu [1] y el otro por Bron y Kerbosch [17]. La ventaja de esta técnica de enumeración es la eliminación de la redundancia en la generación del mismo clique, además de que su capacidad de almacenamiento es polinomial.

Tsukiyama *et al* [76] propusierón un algoritmo enumerativo que combinaba los estudios usados por Auguston y Minker [2], Bron y Kerbosch [17] y Akkoyunlu [1]. El resultado fue un algoritmo con un tiempo de complejidad de $O(nm\mu)$, donde n, m, μ son el número de vértices, aristas y cliques maximales en una gráfica.

En 1980 Loukakis y Tsouros [51] proponen un algoritmo enumerativo llamado, primero en profundidad *(depth-first)*, que genera lexicográficamente todos los conjuntos independientes. Ellos compararón su algoritmo con el de Bron y Kerbosch [17] y el algoritmo de Tsukiyama *et al.* [76]. Sus resultados computacionales sobre gráficas de 220 vértices sugirierón una mayor eficiencia. Su algoritmo fue 30 veces más rápido que el de Bron y Kerbosch y tres veces más veloz que el de Tsukiyama *et al.*

En 1988, Johnson *et al* [46] propusieron un algoritmo que enumera todos los conjuntos maximales indepedientes en orden lexicográfico. El algoritmo tiene un orden $O(n^3)$ entre la generación de dos conjuntos sucesivos independientes. Ellos también muestran que no existe un algoritmo de orden polinomial para enumerar los cliques maximales en orden lexicográfico inverso (si P $\neq$ NP).

El algoritmo de Chiba and Nishizeki [19] lista todos los cliques con tiempo de complejidad de $O(a(G)m\mu)$, donde $a(G)$ es la arborescencia de la gráfica G; mejorando así el tiempo de complejidad de Tsukiyama *et al.*

También en 1988, Tomita *et al* [75] propone una modificación al algoritmo de Bron y Kerbosch, obteniendo un tiempo de complejidad de $O(3^{n/3})$.

2.4.1.2. Ramificación y acotamiento (enumeración implicita)

Dentro de los métodos de enumeración encontramos los de enumeración implícita o parcial, para los cuales la idea principal es considerar solamente una porción de todas las soluciones factibles mientras que descartamos automáticamente las restantes como «*no prometedoras*».

El método de ramificación y acotamiento *(branch and bound)* es un tipo de estrategia de solución de enumeración parcial, que permite, a algunos problemas de optimización combinatoria, ser resueltos sin explícitamente enumerar todas las soluciones factibles.

Estos algoritmos generan soluciones parciales sobre un árbol de búsqueda utilizando métodos para acotar aquellas ramas que ya no pueden ser extendidas, a través de su eliminación.

Un árbol consta de un conjunto finito de elementos llamados nodos, además de un conjunto finito de líneas denominadas ramas que se encargan de conectar a los nodos. Se dice que un nodo x es padre si tiene nodos sucesores. Estos nodos sucesores se llaman hijos. Sólo puede haber un único nodo sin padres, que llamaremos raíz. Por ejemplo, en la figura 2.4 el nodo a es padre del nodo c y d. Un nodo sin hijos, como los nodos g, h, i, e, j y k se llaman nodos hoja. Los nodos que no son hojas se denominan nodos internos.

Si $n_1, n_2, \ldots, n_k$ es una sucesión de nodos en un árbol tal que n_i es el padre de n_{i+1} para $1 \leq i < k$, entonces la sucesión es llamada un camino del nodo n_1 al nodo n_k. La longitud del camino es igual al número de nodos del camino menos uno.

Si existe un camino del nodo a a otro b, entonces a es un antecesor de b, y b es un descendiente de a. Cualquier nodo es antecesor y descendiente de el mismo.

Un antecesor o descendiente de un nodo, diferente a él mismo, es llamado antecesor propio o descendiente propio respectivamente. En un árbol, la raíz es el único nodo sin antecesor propio. Un nodo sin

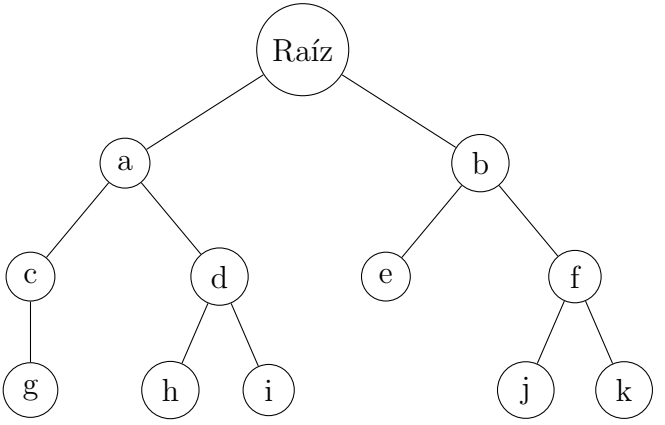


Figura 2.4: Árbol

descendientes propios se denomina hoja.

La altura de un nodo en un árbol es la longitud del camino más largo de ese nodo a una hoja. La altura de un árbol es la altura de la raíz. La profundidad de un nodo es la longitud del único camino de la raíz a ese nodo.

A menudo es útil asociar una etiqueta, o valor, a cada nodo de un árbol. Esto es, la etiqueta de un nodo no será el nombre del nodo, sino un valor almacenado en él. En algunas aplicaciones, incluso se cambiará la etiqueta de un nodo sin modificar su nombre.

En la técnica de ramificación y acotamiento son de gran utilidad los árboles, ya que se hace la búsqueda de la solución con la información guardada en sus nodos.

Hay muchos problemas que exigen encontrar una configuración máxima o mínima de alguna clase que son susceptibles de resolución por medio de la búsqueda de retroceso de un árbol de todas las posibilidades. Los nodos del árbol pueden considerarse como conjuntos de configuraciones y los hijos de un nodo n representan cada uno un subconjunto de las configuraciones que n representa. Finalmente, cada hoja representa configuraciones sencillas o soluciones al problema, y se

puede evaluar cada una de las configuraciones con el fin de saber si es la mejor solución encontrada hasta ese momento.

Si se es hábil en el diseño de la búsqueda, los hijos de un nodo representarán muchas menos configuraciones que el nodo mismo, así que no es necesario profundizar mucho para alcanzar las hojas.

El proceso de ramificación y acotamiento requiere de dos herramientas. La primera es un proceso de expansión, que dado un conjunto fijo S, devuelve dos o más conjuntos más pequeños $S_1, S_2, \ldots, S_n$ cuya unión cubre S. Este paso se llama ramificación, ya que su aplicación es recursiva, esta definirá una estructura de árbol cuyos nodos seran subconjuntos de S. La idea clave del algoritmo es si la menor rama para algún árbol nodo A es mayor que la rama padre para otro nodo B, entonces A debe ser descartada con seguridad de la búsqueda. Este paso es llamado acotamiento, y usualmente es implementado manteniendo una variable global m que guarda el mínimo nodo padre visto entre todas las subregiones examinadas hasta entonces. Cualquier nodo cuyo nodo hijo es mayor que m puede ser descartado. La recursión para cuando el conjunto candidato S es reducido a un solo elemento, o también cuando el nodo padre para el conjunto S coincide con el nodo hijo. De cualquier forma, cualquier elemento de S va a ser el mínimo de una función dentro de S.

Se suele interpretar como un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la actual. La característica de esta técnica con respecto a otras, y a la que debe su nombre, es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo prometedoras, para «podar» esa rama del árbol y no continuar mal gastando recursos y procesos en casos que se alejan de la solución óptima.

De acuerdo a Murty [57], el método de ramificación y acotamiento para problemas de optimización fue desarrollado independientemente por Land y Doig [50] en 1960 y, Murty, Karel y Little [58] en 1962. Estos desarrollos simultaneos se dierón de manera diferente: Land y Doig se enfocarón a problemas de programación entera y mixta, mientras

Murty *et al.* se dedicaron al estudio de un tipo especifico de problema de optimización entera, el problema del agente viajero.

Para el problema del clique máximo, el método de enumeración implicita mejor conocido y más usado, es el de ramificación y acotamiento. Las preguntas claves que propusierón Pardalos y Xue [64] para identificar un buen algoritmo de ramificación y acotamiento, que resuelva el problema del clique máximo, son las siguientes:

1. ¿Cómo encontrar una buena cota inferior?, es decir, un clique de tamaño grande.

2. ¿Cómo encontar una buena cota superior sobre el tamaño del clique máximo?

3. ¿Cómo ramificar?, es decir, dividir un problema en subproblemas más pequeños.

Esta técnica será utilizada por el algoritmo del clique máximo, estas tres preguntas serán claves para acotar y ramificar, en el capítulo 4 se describirá.

Los primeros algoritmos de enumeración implícita para el problema del clique máximo y el del conjunto independiente máximo, aparecierón en la decada de los 70. Dentro de estos primeros trabajos podemos mencionar el de Desler y Hakimi [24], Tarjan [72] y Houck [42]. Los algoritmos fueron mejorados en 1977 por Tarjan y Trojanowski [73], y por Houck y Vemuganti [43]. En Tarjan y Trojanowski, proponen un algoritmo recursivo para el problema del máximo conjunto independiente, mostrando que su algoritmo tenia un tiempo de complejidad de $O(2^{n/3})$. Esta cota sobre el tiempo, mostró que es posible resolver un problema NP-Duro mucho más rápido que con enumeración explicita. En el mismo año, Chvátal [21] mostró que haciendo pruebas recursivas sobre una cota superior, el número estable, tiene al menos un orden $O(c^n)$, donde $c > 1$ es una constante. En el trabajo de Houck y Vemuganti hacen uso de la relación entre el máximo conjunto independiente y una clase especial de gráficas bipartitas, y usan esta relación para encontrar una solución inicial en su algoritmo para el problema del

máximo conjunto independiente.

Muchos algoritmos fueron propuestos en la decada de los ochenta, uno de los más importantes fue el de Balas y Yu [7]. En su algoritmo, la enumeración implicita fue implementada de manera diferente. Primero, encontraron una subgráfica triángular maximal inducida T de G. Una vez encontrada T, buscaron un máximo clique de T. Este clique funciona como una cota inferior y una solución factible para el problema del máximo clique. Después, usan un procedimiento de coloración heurística para extender T a una subgráfica más grande (maximal). La importancia de este segundo paso es que ayuda a reducir el número de subproblemas generados desde cada nodo del árbol de búsqueda y por lo tanto el del árbol de búqueda completo. Resolvierón el problema sobre gráficas de 400 vértices y $30,000$ aristas. Compararon su algoritmo con otros, encontrando que su algoritmo no solo generaba un árbol de búsqueda más pequeño, también requería menos tiempo de procedimiento.

En 1986, Kikusts [48] propuso un algoritmo de ramificación y acotamiento para el problema del máximo conjunto independiente, basado en una nueva relación recursiva para el número estable $\alpha(G)$ de una gráfica G. A saber,

$$\alpha(G) = max\{+\alpha(G - v - N(v)), \alpha'_v\},$$

donde $\alpha'_v = max\{|I||I \subseteq G - v \text{ es independiente}, |I \cap N(v)| \geq 2\}$. Esta relación es diferente de la relación recursiva

$$\alpha(G) = max\{1 + \alpha(G - v - N(v)), \alpha(G - v)\},$$

que es tradicionalmente usada en algoritmos de ramificación y acotamiento para el problema del máximo conjunto independiente.

También en 1986, Robson [69] propuso una modificación al algoritmo recursivo de Tarjan y Trojanowski [73]. Robson mostró, a través de un análisis detallado, que su algoritmo tiene un tiempo de complejidad de $O(2^{0.276n})$. Este tiempo es mejor que el tiempo de complejidad de $O(2^{n/3})$ de Tarjan y Trojanowski [73].

Al final de la década de los ochenta, nuevos algoritmos fueron propuestos, por ejemplo Pardalos y Rodgers [63] (el cual fue publicado en 1992), Gendreau *et al* [36] y Tomita *et al* [74]. El algoritmo de Pardalos y Rodgers está basado en una formulación de programación cuadrática cero-uno sin restricciones del problema del máximo clique. En su trabajo, prueba dos diferentes reglas de ramificación, algoritmos glotones *(greedy)* y no glotones *(nongreedy)*. Reporantando interesantes resultados concernientes al comportamiento de los algoritmos usando las diferentes técnicas mencionadas. El algoritmo de Gendreau *et al* fue un algoritmo enumerativo implícito. Su regla para ramificar (la selección del siguiente vértice para que sea una nueva rama), está en el número de triángulos a los que un vértice pertenece. El algoritmo de Tomita *et al* usa un algoritmo de coloración glotón para obtener una cota superior sobre el tamaño de el clique máximo. Algunos resultados computacionales pueden encontrarse en [36][74].

En la década de 1990, muchos algoritmos fueron propuestos, por ejemplo, los algoritmos de Pardalos y Phillips [62], de Friden *et al* [29], Carraghan y Pardalos [18], Babel y Tinhofer [4], Babel [3] y Xue [79].

Pardalos y Phillips [62], formularon el problema del clique máximo como un problema de optimización global cuadrática indefinida con restrcciones lineales. Además, dieron cotas superiores teóricas sobre el tamaño del clique máximo. El algoritmo de Friden *et al* [29], fue un algoritmo de ramificación y acotamiento para el problema del máximo conjunto independiente. Ellos usaron la técnica de *búsqueda tabú* para encontar cotas superiores e inferiores en su algoritmo.

Carraghan y Pardalos [18] propusieron un algoritmo de enumeración implícita. Su algoritmo es muy fácil de entender y muy eficiente para gráficas de poca densidad. Su regla de ramificación corresponde a la técnica no glotona *(nongreedy)* descrita por Pardalos y Rodgers [63]. Usando este algoritmo, fueron capaces de resolver problemas sobre gráficas de 500 vértices. Ellos también probaron su algoritmo sobre ejemplares de gráficas con $1,000$ y $2,000$ vértices. Ya que su algoritmo es fácil de entender y su codigo está disponible, puede servir como referencia para comparar diferentes algoritmos.

Babel y Tinhofer [4] propusieron un algoritmo de ramificación y acotamiento para el problema del clique máximo. El ingrediente principal de su algoritmo es el uso de un heurístico rápido y relativamente bueno para el problema de coloración mínima propuesto por Brelaz [16]. El heurístico de coloración es llamado el *degree of saturation largest first* (DSATUR). Aplicando DSATUR a una gráfica, se encuentra una cota superior sobre el tamaño del máximo clique y además también es cota para los cliques maximales y por lo tanto, una cota inferior para estos últimos. Babel y Tinhofer aprovecharon esta característica y aplicaron DSATUR a cada nodo del árbol de búsqueda. Probaron su algoritmo sobre gráficas de 100 a 400 vértices con variadas densidades. En 1991, Babel [3] mejoró el algoritmo de Babel y Tinhofer.

También en 1991, basado en el hecho de que una coloración fraccional proporciona una cota superior más cercana que una solución de coloración entera para el problema del clique máximo, un heurístico para el problema de coloración fraccional es propuesto por Xue [79] y usado en un algoritmo de ramificación y acotamiento para el problema del clique máximo, reduciendo sustancialmente el árbol de búsqueda.

2.4.2. Algoritmos de aproximación o heurísticos

En contraposición a los métodos exactos que proporcionan una solución óptima del problema, los métodos aproximados o heurísticos se limitan a proporcionar una buena solución del problema, aunque no necesariamente óptima. Lógicamente,el tiempo invertido por un método exacto para encontrar la solución óptima de un problema NP-Completo, si es que existe tal método, es de un orden de magnitud muy superior al del algoritmo de aproximación, pudiendo llegar a ser tan grande en muchos casos, haciéndolo inaplicable.

El término heurística proviene del vocablo griego *heuriskein*, que puede traducirse como encontrar, descubrir o hallar.

Desde un punto de vista científico, el término heurística se debe al matemático George Polya quien lo empleó por primera vez en su libro

How to solve it [66]. Con este término, Polya englobaba las reglas con las que los humanos gestionan el conocimiento común y que, a grandes rasgos, se podía simplificar en:

1. Buscar un problema parecido que ya haya sido resuelto
2. Determinar la técnica empleada para la resolución así como la solución obtenida
3. En el caso en el que sea posible, utilizar la técnica y solución descrita en el punto anterior para resolver el problema planteado

Actualmente existen dos interpretaciones posibles para el término heurística. La primera de ellas concibe las heurísticas como un procedimiento para resolver problemas. Para esta interpretación, las definiciones más interesantes que se extraen de la literatura son las siguientes:

- T. Ñicholson [60]: «*Procedimiento para resolver problemas por medio de un método intuitivo en el que la estructura del problema puede interpretarse y explotarse inteligentemente para obtener una solución razonable*».
- H. Müller-Merbach [59]: «*En investigación operativa, el término heurístico normalmente se entiende en el sentido de un algoritmo iterativo que no converge hacia la solución, óptima o factible, del problema*».
- Barr *et al.* [8]: «*Un método heurístico es un conjunto bien conocido de pasos para identificar rápidamente una solución de alta calidad para un problema dado*».

La segunda interpretación de heurística entiende que éstas son una función que permiten evaluar la bondad de un movimiento, estado, elemento o solución. Un ejemplo, de este tipo de definición, es el siguiente.

- Rich *et al.* [68]: «*Una función heurística es una correspondencia entre las descripciones de los estados del problema hacia alguna medida de idoneidad, normalmente presentada por números. Los aspectos del problema que se consideran, cómo se evalúan estos*

aspectos y los pesos que se dan a los aspectos individuales, se eligen de forma que el valor que la función da a un nodo del proceso de búsqueda sea una estimación tan buena como sea posible para ver si ese nodo pertenece a la ruta que conduce a la solución».

De todas las definiciones , quizá las más intuitiva es la presentada por Zanakis *et al.* [80]:

«*Procedimientos simples a menudo basados en el sentido común que se supone que obtendrán una buena solución, no necesariamente óptima, a problemas difíciles de un modo sencillo y rápido*».

Son varios los factores que pueden hacer interesante la utilización de algoritmos heurísticos para la resolución de un problema:

1. *Cuando no existe un método exacto de resolución o éste requiere mucho tiempo de cálculo o memoria.* Ofrecer entonces una solución que sólo sea aceptablemente buena resulta de interés frente a la alternativa de no tener ninguna solución en absoluto.

2. *Cuando no se necesita la solución óptima.* Si los valores que adquiere la función objetivo son relativamente pequeños, puede no merecer la pena esforzarse, con el consiguiente coste en tiempo y dinero, en hallar una solución óptima que, por otra parte, no representará un beneficio importante respecto a una que sea simplemente sub-óptima. En este sentido, si puede ofrecer una solución mejor que la actualmente disponible, esto puede ser ya de interés suficiente en muchos casos.

3. *Cuando los datos son poco fiables.* En este caso, o bien cuando el modelo es una simplificación de la realidad, puede carecer de interés bucar una solución exacta, dado que de por sí ésta no será más que una aproximación de la real, al estar basado en datos que no son reales.

4. *Cuando limitaciones de tiempo, espacio, para almacenamiento de datos, etc., obliguen el empleo de métodos de rápida respuesta, aun a costa de la precisión.*

5. *Como paso intermedio en la aplicación de otro algoritmo.* A veces son usadas soluciones heurísticas como punto de partida de algoritmos exactos de tipo iterativo. En el caso del algoritmo de clique máximo utilizaremos heurísticas como cotas para el método de ramificación y acotamiento.

2.4.2.1. Heurísticos usados para el problema del clique máximo

Ahora presentaremos algunos de los algoritmos que se han usado para dar solución al problema del clique máximo. En la literatura podemos encontrar diversos heurísticos que van desde los más simples como los voraces, hasta los más sofisticados como los basados en redes neuronales.

La mayoría de los algoritmos de aproximación para el problema del clique máximo son los llamados heurísticos voraces secuenciales (*sequential greedy heuristics*). Estos heurísticos generan un clique maximal a traves de la unión de un vértice a un clique parcial, o de la eliminación repetida de un vértice de un conjunto que no es clique. Kopf y Ruhe [49] nombraron dos clases de heurísticos, el *Best in* y el *Worst out*. Las decisiones sobre las cuales se permite a un vértice agregarlo o moverlo, se basan en ciertos indicadores asociados con los vértices candidatos. Por ejemplo, un posible heurístico *Best in* construye un clique maximal agregando repetidamente el vértice que tenga el grado más alto entre los vértices candidatos. En este caso, el indicador es el grado del vértice. De otra forma, un posible heurístico *Worst out* puede comenzar con el conjunto de vértices V completo, removiendo repetidamente vértices de V hasta que llegue a formar un clique.

Una característica común de los heurísticos secuenciales es que todos ellos encuentran sólo un clique maximal, una vez que lo encontraron la búsqueda se detiene. Podemos ver este tipo de heurísticos desde un punto de vista diferente, definamos S_G como el espacio que consiste de todos los cliques maximales de G. Lo que un heurístico voraz secuencial hace es encontrar un punto en S_G, esperando estar muy cerca del punto óptimo. Esto sugiere una manera de mejorar la

solución aproximada, expandiendo la búsqueda en S_G. Por ejemplo, una vez que encontremos un punto $x \in S_G$, podemos encontrar sus vecinos para mejorar x. Esto nos lleva a los heurísticos de búsqueda local.

En un heurístico de búsqueda local, los vértices que tengan más vecinos para la búsqueda, tendrán mayor probabilidad de encontrar una mejor solución. Una clase bien conocida de heurísticos de búsqueda local es la de los heurísticos k-intercambio *(k-interchange)*. Estos algoritmos están basados en el k-vecino de una solución factible. En el caso del problema del máximo clique, $y \in S_G$ es un k-vecino de $x \in S_G$ si $|x - y| \leq k$, donde $k \leq |x|$. Lo primero que hace el algoritmo es encontrar un clique maximal $x \in S_G$, luego busca todos los vecinos de x y regresa el mejor clique encontrado. Como es de esperarse, el principal factor para la complejidad de esta clase de heurísticos es el tamaño de la vecindad y las búsquedas involucradas. Por ejemplo, en el heurístico de k-intercambio, la complejidad crece aproximadamente con $\mathrm{O}(n^k)$.

La calidad de la solución de un heurístico de búsqueda local dependerá directamente del punto de inicio $x \in S_G$ y la vecindad de x. Para mejorar la calidad de las soluciones, necesitamos incrementar la vecindad de x (el punto de inicio) para incluir un mejor punto. Si quisieramos probar con varios puntos sobre S_G, entonces tendriamos una vecindad muy grande. El problema es que cuando el tamaño de los vecinos se incrementa, hace que el tiempo de búsqueda se incremente.

Una clase de heurísticos diseñados para búscar varios puntos de S_G son llamados heurísticos aleatorios *(randomized heuristics)*. La función principal de esta clase de heurísticos es encontrar un punto al azar en S_G. Una manera de hacerlo es incluyendo un factor aleatorio en la generación del punto. Un heurístico aleatorio ejecuta otro heurístico con factores aleatorios incluidos, cierto número de veces para encontrar diferentes puntos sobre S_G. Por ejemplo, podemos aleatorizar un heurístico voraz secuencial y ejecutarlo N veces. La complejidad de un heurístico aleatorio depende de la complejidad del heurístico y del número N. Una implementación de un heurístico aleatorio para el problema del conjunto independiente máximo puede ser encontrado en Feo *et al* [27], donde la búsqueda local es combinada con heurísticos

aleatorios. Sus resultados computacionales indican que su trabajo es efectivo para encontrar cliques grandes de gráficas generadas aleatoriamente. Esto es, para gráficas generadas aleatoriamente con $1,000$ vértices y $50\,\%$ de densidad, su algoritmo encontro cliques de tamaño 15 o más grandes en algunos casos.

A partir de la década de los noventa, los métodos heurísticos de búsqueda tabú *(Tabu Search)* y redes neuronales *(Neural Networks)* han sido usados para encontrar una solución aproximada al problema del clique máximo. Friden *et al* [28] propuso un heurístico basado en búsqueda tabú. Una implementación diferente de búsqueda tabú para el mismo problema fue propuesto por Gendreau *et al* [36]. Ramanujam y Sadayappanv [67], Jagota [44], y Jagota y Regan [45] propusieron heurísticos basados en redes neuronales. Lo que estos métodos hacen es buscar varios puntos de S_G. Sin embargo, ya que hay muchos parámetros afectando el patron de búsqueda y el resultado, es díficil reconocer que punto en S_G se está búscando. Aún así, algunos resultados son alentadores.

Otro tipo de búsqueda que encuentra un clique maximal de G es llamado *subgraph approach* que está basado en el hecho de que un máximo clique C de una subgráfica $G' \subseteq G$ es también un clique de G. El *subgraph approach* primero encuentra una subgráfica $G' \subseteq G$ tal que el clique máximo pueda ser encontrado en tiempo polinomial, entonces encuentra un máximo clique de G' y lo utiliza como una solución aproximada. La ventaja de este método es que en la búsqueda del clique máximo $C \subseteq G'$, se encuentran implícitamente otros cliques de G' ($S_{G'} \subseteq S_G$). Debido a la estructura especial de G', esta búsqueda se puede hacer eficientemente. En Balas y Yu [7], G' es una subgráfica triángular de aristas maximal de G. Se puede aplicar la misma idea para las clases de gráficas que tienen algoritmos polinomiales para el problema del clique máximo.

Capítulo 3

Cotas para la cardinalidad del clique máximo $\omega(G)$

Como se vio en los capítulos anteriores, todo problema dentro de la clase NP-Completos se puede reducir en tiempo polinomial a cualquier otro dentro de esta. Dentro de esta clasificación encontramos problemas que están íntimamente unidos al problema del clique máximo, ya que la solución de estos problemas nos dan una cota para la cardinalidad del clique máximo, uno de estos problemas es el de coloración mínima. Otro tipo de problema, que también se abordará en este capítulo, será la coloración fraccional($\chi_f(G)$) que nos proporcionará una mejor cota superior, además se analizará su relación con el problema del clique fraccional mediante la teoría de dualidad. Estas dos cotas nos serán de utilidad más adelante, ya que se usarán en el algoritmo de ramificación y acotamiento mediante aproximaciones heurísticas.

Por último se estudiarán algunos resultados de gráficas perfectas, las cuales tienen la propiedad de que la cardinalidad del clique máximo es igual al mínimo número de colores para colorear una gráfica ($\chi(G)$), cumpliéndose también para cualquier subgráfica inducida. Una clase especial de estas gráficas son las trianguladas, en ellas determinaremos un clique máximo y la cardinalidad de este será la cota inferior para $\omega(G)$ en el nodo raíz del algoritmo de ramificación y acotamiento.

3.1. Coloraciones

Una k-coloración de vértices de una gráfica $G = (V, E)$ es una partición de V en conjuntos $C_1, C_2, \ldots, C_k$ llamados clases de coloración. Una k-coloración es propia si para todo par de vértices adyacentes, estos tienen diferente color, de esta manera $C_1, C_2, \ldots, C_k$ son conjuntos independientes de vértices. El problema de coloración mínima consiste en encontrar en una gráfica G la coloración propia con el mínimo número de colores.

El número cromático $\chi(G)$ de G, está definido como

$$\chi(G) = \text{ mín } \{k \mid \text{existe una } k\text{-coloración propia de G}\}.$$

Por lo que el problema de coloración mínima consiste en determinar el número cromático de G. Si $\chi(G) = k$, entonces G es k-cromática; G es k-coloreable si $\chi(G) \leq k$. Por ejemplo, en la gráfica de la Figura 3.1 se puede ver que es 3-crómatica; es decir, $\chi(G) = 3$, también es posible colorearla propiamente con 5 o 4 colores, por lo que también es 5-coloreable o 4 coloreable.

Otra manera de ver una k-coloración de una gráfica es definiéndola como una función $f : V_G \to [1, k]$ y si esta coloración es propia entonces para todo $(u, v) \in E$, tenemos que $f(u) \neq f(v)$.

Esta definición es equivalente a la primera, ya que para cada coloración de vértices $f : V_G \to [1, k]$ proporciona una partición $C_1, C_2, \ldots, C_k$ del conjunto de vértices V, donde $C_i = \{v \mid f(v) = i\}$.

Nuestro objetivo es demostrar que el número crómatico $\chi(G)$ es una cota superior para $\omega(G)$.

Proposición 23. *Si H es una subgráfica de G, entonces $\chi(H) \leq \chi(G)$.*

Demostración. Supongamos que $\chi(G) = k$, entonces existe una k-coloración propia C de G; es decir, $C : V_G \to [1, k]$. Como C es una coloración propia, entonces asigna colores distintos a todo par de vértices adyacentes de G, si ahora aplicamos a H la coloración C, entonces

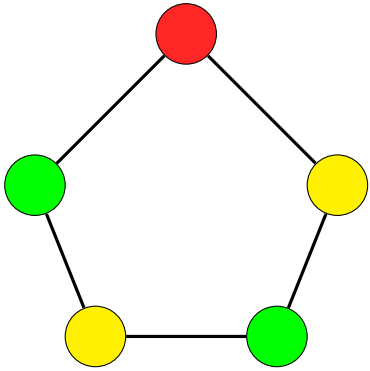


Figura 3.1: Gráfica 3-cromática

también le asignará colores distintos a todo par de vértices adyacentes de H. Por lo tanto H es k-coloreable y $\chi(H) \leq \chi(G)$. □

La proposición anterior nos permitirá demostrar que el número cromático de G es una cota superior de $\omega(G)$, el cual es enunciado en la siguiente proposición.

Proposición 24. *Para toda gráfica G, $\chi(G) \geq \omega(G)$.*

Demostración. Supongamos que $\omega(G) = k$, esto quiere decir que tenemos una gráfica S que es el clique máximo de G, de orden k y $S \subseteq G$. Entonces por la proposición 23, tenemos que $\chi(S) \leq \chi(G)$. Y ya que S es un clique, entonces $\chi(S) = k = \omega(G)$. Por lo tanto $\chi(G) \geq \omega(G)$. □

Con este último resultado demostramos que la cardinalidad del clique máximo está acotado por el número cromático.

Recordemos que un conjunto independiente *(stable vertex)* es un subconjunto de V donde para cada par de vértices se cumple que no son adyacentes. El problema del máximo conjunto independiente, consiste en encontrar el conjunto independiente de máxima cardinalidad. El tamaño del máximo conjunto independiente es el número estable *(stability number)* de G, denotado por $\alpha(G)$. Por lo que se cumple que $\omega(G) = \alpha(\bar{G})$, donde $\bar{G}$ es la gráfica complemento.

Como hemos visto hasta ahora el número cromático está relacionado con la cardinalidad del clique máximo y este a su vez con el número estable. De igual manera podemos encontrar una relación entre el número estable y el número cromático. Veamos que una k-coloración propia hace una partición de V entre conjuntos independientes $C_1, C_2, \ldots, C_k$, si el número de clases de coloración es el máximo que podemos lograr, entonces $k = \alpha(G)$, de otro modo $k < \alpha(G)$. La relación entre el número estable y el número cromático está dada por la siguiente proposición.

Proposición 25. *Si G es una gráfica de orden n, entonces $\frac{n}{\alpha(G)} \leq \chi(G)$.*

Demostración. Supongamos que $\chi(G) = k$ y sea una k-coloración dada de G con clases de color resultantes $C_1, C_2, \ldots, C_k$. Ya que

$$n = |V(G)| = \sum_{i=1}^{k} |C_i| \leq k\alpha(G)$$

por lo tanto

$$\frac{n}{\alpha(G)} \leq \chi(G)$$

. □

Las cotas de las proposiciones 24 y 25 cumplen la igualdad cuando la gráfica G es completa.

Ahora demostraremos que el número cromático está acotado superiormente por el grado máximo de la gráfica ($\Delta(G)$) más uno, por lo que se convertirá también en una cota para $\omega(G)$. Esta útima cota será importante para los heurísticos que se utilizarán en el algoritmo. Pero antes daremos un algoritmo glotón para la coloración propia de vértices que será utilizado en las demostraciones.

Algorimo de coloración glotón

Sea $G = (V, E)$ con $V = \{v_1, v_2, \ldots v_n\}$ vértices ordenados de alguna manera. Colorearemos los vértices con la siguiente función,
$f : V \to \mathbb{N}$ tal que $f(v_1) = 1$ y

$$f(v_i) = \text{mín}\{j \mid f(v_t) \neq j \text{ para toda } t < i \text{ con } (v_i, v_t) \in E\}.$$

La relación entre $\Delta(G)$ y $\chi(G)$ está dada en la siguiente proposición.

Proposición 26. *Para toda gráfica G, $\chi(G) \leq \Delta(G) + 1$.*

Demostración. Utilizando el algoritmo glotón tenemos que $f(v_i) \leq d_G(v_i) + 1$, debido a que $f(v_i)$ a lo más puede tomar $d_G(v_i) + 1$ si es que todos sus vecinos ya estan coloreados por el algoritmo. Además, cada vértice tiene a lo más $\Delta(G)$ vecinos y por el algoritmo a lo más utilizaremos $\Delta(G) + 1$. Entonces $f : V_G \to [1, \Delta(G) + 1]$. Por lo tanto $\chi(G) \leq \Delta(G) + 1$ □

Como consecuencia inmediata de esta proposición, tenemos que $\omega(G)$ también está acotada superiormente por $\Delta(G) + 1$. Ahora si nosotros conocemos que la gráfica con la que trabajamos, no es una gráfica completa y además no es un ciclo impar entonces esta relación se convierte en $\chi(G) \leq \Delta(G)$, como se enuncia y se demuestra acontinuación.

Teorema 27 (**Teorema de Brooks**). *Si G es una gráfica conexa que no es una gráfica completa ni un ciclo impar, entonces $\chi(G) \leq \Delta(G)$.*

Demostración. Sea G una gráfica conexa y sea $\Delta(G) = k$. Si $k \leq 1$ entonces G sería una gráfica completa; si $k = 2$ entonces G por ser conexa seria un ciclo. Así G sería un ciclo impar o una gráfica bipartita (si G es un ciclo par), en cuyo caso $\chi(G) = 2 = k$. Ahora para $k \geq 3$ queremos probar que es posible ordenar los vértices de manera que cada uno tenga a lo más $k - 1$ vecinos con etiqueta menor; si conseguimos esta ordenación, el algoritmo glotón utilizaría a lo más k colores, cumpliendo la desigualdad.

Caso 1. Supongamos primero que G no es k-regular, entonces existe un vértice v, tal que $d(v) < k$. Etiquetamos a este vértice con la etiqueta n. Ya que G es conexa, entonces podemos formar un árbol de expansión T de G a partir de v_n. Etiquetando en forma decreciente a cada vértice de G con su distancia de v_n en T, obteniendo la ordenación $(v_1, v_2, \ldots, v_n)$ donde para cada v_i, $i \neq n$, se cumple que tiene vecinos con etiquetas más grandes a lo largo de la trayectoria a v_n en el árbol. Por lo que cada vértice tiene a lo más $k-1$ vecinos con etiqueta menor, aplicando el algoritmo glotón a esta ordenación utilizará a lo más k colores.

Caso 2. Supongamos que G es k-regular. Distinguiremos dos subcasos: G puede tener un vértice de corte o ser 2-conexo.

Supongamos primero que G tiene un vértice de corte x, y sea G' la subgráfica que consiste de una componente C de $G - x$ junto con las aristas que unen dicha componente a x, es decir, $G' = C + x$. En G' tenemos que $d(x) < k$, utilizando el método del caso 1 obtenemos una k-coloración de G'. De la misma manera aplicamos el método para cada una de las subgráficas de G de la forma $C + x$, donde C es una componente de $G - x$, de tal manera que en todas x tenga el mismo color. Uniendo las coloraciones de las distintas componentes hallamos una coloración para toda G con k-colores.

Ahora supongamos que G es 2-conexa. Por lo que para toda ordenación de vértices, el último vértice tiene k-vecinos. Sin embargo, la idea de la ordenación para la coloración utilizando el algoritmo glotón aún puede funcionar si aseguramos que dos vecinos de v_n tengan el mismo color. En particular, supongamos que para algún vértice v_n con vecinos v_1 y v_2, tal que v_1 no es adyacente a v_2 y $G - \{v_1, v_2\}$ es conexo. En este caso, etiquetamos a los vértices de un árbol de expansión de $G - \{v_1, v_2\}$ usando $3, \ldots, n-1$, de tal manera que las etiquetas se incrementan a lo largo de la trayectoria hacia la raíz v_n. Por lo que cada vértice antes de v_n tiene a lo más $k-1$ vecinos con etiquetas más pequeñas. La coloración utilizando el algoritmo glotón también usa a lo más $k-1$ colores sobre los vecinos de v_n, ya que v_1 y v_2 recibieron el mismo color.

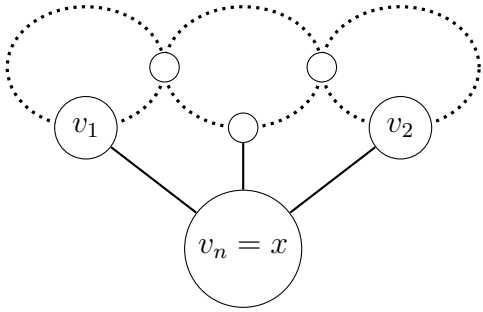


Figura 3.2: Gráfica con vértice v_n y vecinos no adyacentes v_1 y v_2 tal que $\kappa(G - v_n) = 1$

Ahora solo nos queda mostrar que toda gráfica 2-conexa y k-regular con $k \geq 3$, tiene vértices v_1, v_2 y v_n que cumplen con la ordenación anterior y por lo tanto cumplen con la desigualdad. Escogemos un vértice x, si $\kappa(G - x) \geq 2$, entonces etiquetamos a x con v_1 y con v_2 al vértice con distancia 2 desde x. El vértice v_2 existe porque G es regular y no es una gráfica completa. Con v_n etiquetamos a un vecino de v_1 y v_2.

Si $\kappa(G - x) = 1$, entonces $v_n = x$. Ver Figura 3.2. Ya que G no tiene vértices de corte, entonces x tiene un vecino en cada bloque de $G - x$. Los vecinos v_1, v_2 de x, cada uno en alguno de los dos bloques no son adyacentes. Por lo tanto $G - \{x, v_1, v_2\}$ es conexa, ya que los bloques no tienen vértices de corte. Como $k \geq 3$, entoces el vértice x tiene otro vecino y $G - \{v_1, v_2\}$ es conexa. □

Hasta esta parte tenemos que la cardinalidad del clique máximo $\omega(G)$ está acotado por el número cromático $\chi(G)$; sin embargo, muchas veces la separación entre estas cotas es grande, por lo que nos vemos obligados a encontrar una mejor cota para $\omega(G)$. Una manera de mejorar la cota es utilizando coloración fraccional, que no es más que una generalización de la coloración de gráficas.

3.1.1. Cliques y coloraciones fraccionales

Ahora generalizaremos la idea de coloración propia a la de coloración fraccional. Dada una gráfica $G = (V, E)$, una coloración fraccional propia hace una asignación de un conjunto de colores a cada vértice, de

tal manera que vértices adyacentes tienen asignados conjuntos ajenos. A cada color se le asignará un peso mayor que cero, de tal manera que la suma de ellos en cada vértice sea al menos uno. El peso de la coloración fraccional es la suma de los pesos de los colores. El número cromático fraccional de una gráfica, $\chi_f(G)$, es el mínimo peso posible tal que exista una coloración fraccional propia.

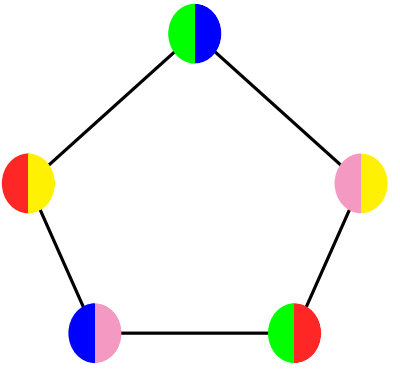


Figura 3.3: Gráfica C_5 con una $5/2$ coloración fraccional propia

Por ejemplo, para la gráfica C_5 una coloración fraccional es la que se presenta en la Figura 3.3, está coloración está asignando un conjunto de dos colores a cada vértice. Si a cada uno de estos colores le asignamos un peso de $1/2$ y debido a que utilizamos cinco colores, entonces el peso de la coloración es igual a $5/2$. En este ejemplo esta coloración resulta ser la óptima, es decir, $\chi_f(G) = 5/2$.

Una forma equivalente de definir la coloración fraccional es en términos de conjuntos independientes. Utilizaremos $I(G)$ para denotar al conjunto de todos los conjuntos independientes de G, y $I(G, u)$ para denotar a los conjuntos independientes que contienen al vértice u. Una coloración fraccional propia de una gráfica G es una función real no negativa f sobre $I(G)$ tal que para cualquier vértice x de G se cumple que:

$$\sum_{S \in I(G,x)} f(S) \geq 1.$$

El peso de una coloración fraccional es la suma de todos sus valores, es decir, $\sum_{I(G)} f(I(G))$, el mínimo peso posible será el número cromático $\chi_f(G)$. Llamaremos a una coloración fraccional regular, si para cada vértice u de G tenemos $\sum_{S \in I(G,u)} f(S) = 1$.

Ejemplo 8. *Para ilustrar los conceptos vistos hasta este momento, consideremos la gráfica de la Figura 3.4.*

En este caso el conjunto de todos los conjuntos independientes es:

$$I(G) = \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_4\}, \{v_3, v_5\}\}.$$

Definimos la función $f : I(G) \to \mathbb{R}^+ \cup \{0\}$ como:

$$f(S) = \begin{cases} 1/3 & \text{si } S \in \{\{v_1\}, \{v_3\}, \{v_4\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_4\}, \{v_3, v_5\}\} \\ 2/3 & \text{si } S \in \{\{v_2\}, \{v_5\}\} \\ 0 & \text{en otro caso} \end{cases}$$

Esta función cumple con ser una coloración fraccional propia ya que para cada vértice x se cumple que

$$\sum_{S \in I(G,x)} f(S) \geq 1.$$

Comprobando para cada uno de los vértices:

$$\begin{aligned}
\sum_{S\in I(G,v_1)} f(S) &= 1/3+1/3+1/3=1\\
\sum_{S\in I(G,v_2)} f(S) &= 1/3+2/3=1\\
\sum_{S\in I(G,v_3)} f(S) &= 1/3+1/3+1/3=1\\
\sum_{S\in I(G,v_4)} f(S) &= 1/3+1/3+1/3=1\\
\sum_{S\in I(G,v_5)} f(S) &= 1/3+2/3=1.
\end{aligned}$$

Como se puede observar para cada vértice la suma de los pesos de los vértices es uno, por lo que la coloración cumple con ser una coloración fraccional regular.

El peso de la coloración fraccional es igual a:

$$\sum_{S\in I(G)} f(S) = 7(1/3)+2(2/3) = \frac{11}{3}.$$

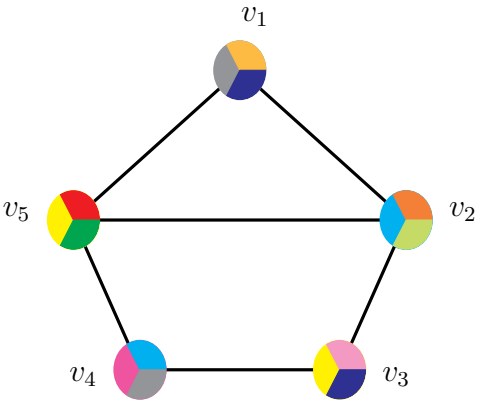


Figura 3.4: Gráfica con una coloración fraccional propia con peso igual a 11/3.

Es equivalente a la primera definición, ya que cuando la función $f(S) = 1/3$ se puede interpretar que a cada vértice de S se le asigna un único color, para el caso en el que $f(S) = 2/3$ entonces se le asignarán dos colores diferentes a cada vértice de S.

Entonces:

$f(v_1) = \frac{1}{3} \Rightarrow$ a v_1 le asigna el color Amarillo Ocaso

$f(v_3) = \frac{1}{3} \Rightarrow$ a v_3 le asigna el color Rosa

$f(v_4) = \frac{1}{3} \Rightarrow$ a v_4 le asigna el color Magenta

$f(v_1, v_3) = \frac{1}{3} \Rightarrow$ a v_1 y a v_3 le asigna el color Morado

$f(v_1, v_4) = \frac{1}{3} \Rightarrow$ a v_1 y a v_4 le asigna el color Gris

$f(v_2, v_4) = \frac{1}{3} \Rightarrow$ a v_2 y a v_4 le asigna el color Azul

$f(v_3, v_5) = \frac{1}{3} \Rightarrow$ a v_3 y a v_5 le asigna el color Amarillo

$f(v_2) = \frac{2}{3} \Rightarrow$ a v_2 le asigna los colores Naranja y Verde Limón

$f(v_5) = \frac{2}{3} \Rightarrow$ a v_5 le asigna los colores Rojo y Verde

Quedando la gráfica coloreada como en la Figura 3.4.

Ahora daremos un ejemplo de una coloración fraccional no regular para la misma gráfica. En este caso definimos a la función f sobre los conjuntos independientes $I(G)$ de la siguiente manera:

$f(v_1) = \frac{1}{2} \Rightarrow$ a v_1 le asigna el color Azul

$f(v_3) = \frac{1}{2} \Rightarrow$ a v_3 le asigna el color Sepia

$f(v_4) = \frac{1}{2} \Rightarrow$ a v_4 le asigna el color Magenta

$f(v_1, v_3) = \frac{1}{3} \Rightarrow$ a v_1 y a v_3 le asigna el color Amarillo

$f(v_1, v_4) = \frac{1}{3} \Rightarrow$ a v_1 y a v_4 le asigna el color Violeta

$f(v_2, v_4) = \frac{1}{3} \Rightarrow$ a v_2 y a v_4 le asigna el color Durazno

$f(v_3, v_5) = \frac{1}{3} \Rightarrow$ a v_3 y a v_5 le asigna el color Verde

$f(v_2) = \frac{3}{4} \Rightarrow$ a v_2 le asigna los colores Naranja, Verde Limón y Rojo

$f(v_5) = \frac{3}{4} \Rightarrow$ a v_5 le asigna los colores Morado, Gris y Rosa

Esta función cumple con ser una coloración fraccional propia no regular ya que para cada vértice x se cumple que

$$\sum_{S \in I(G,x)} f(S) \geq 1.$$

Esta coloración tiene un peso de 13/3 como se puede observar en la Figura 3.5. Ninguno de estas dos soluciones es la solución óptima.

Las clases de color de una k-coloración propia de G forman una colección de k conjuntos independientes ajenos $V_1, \ldots, V_k$, donde la unión de todos ellos es V_G. La función f tal que $f(V_i) = 1$ y $f(S) = 0$ para cualquiera otros conjuntos independientes S es una coloración fraccional de peso k. Por lo tanto, es inmediato que

$$\chi_f(G) \leq \chi(G).$$

El cumplimiento de esta desigualdad será aún más clara cuando se presente la formulación del problema de coloración, como un problema lineal entero y el de coloración fraccional como un problema lineal, las cuales se presentarán más adelante.

De manera similar podemos definir un clique fraccional de una gráfica G, como una una función real no negativa sobre $V(G)$, tal que para

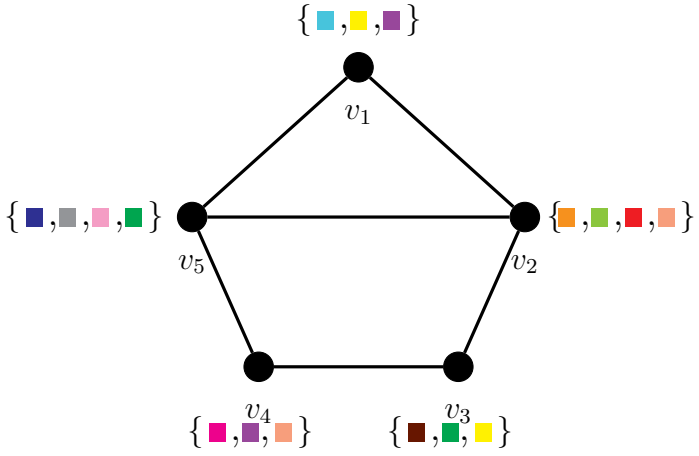


Figura 3.5: Gráfica con una coloración fraccional propia no regular de peso igual a 13/3

cualquier conjunto independiente de G, se cumple que $\sum_{\forall v \in I(G)} f(v) \leq 1$. El peso de un clique fraccional es la suma de sus valores, es decir, $\sum_{\forall v \in V(g)} f(v)$. La cardinalidad del clique fraccional de G es el máximo peso posible de un clique fraccional, y es denotado por $\omega_f(G)$.

Para cualquier clique de tamaño k, tenemos que es un caso especial de un clique fraccional, donde cada uno de los k vértices que conforman el clique toman el valor de uno, mientras que los otros vértices toman el valor de cero. Por lo que el peso de este clique fraccional es k y por lo tanto

$$\omega(G) \leq \omega_f(G).$$

Por ejemplo, si tomamos nuevamente C_5 y definimos la función que tome un 1/2 sobre cada vértice de C_5, con esta función se cumple que para cualquier conjunto independiente $\sum_{\forall v \in I(G)} f(v) \leq 1$. Por lo que tenemos un clique fraccional con peso igual a 5/2, por lo tanto $\omega_f(G)$ es estrictamente más grande que $\omega(G)$. En el ejemplo anterior, llegamos

a que $\chi_f(G) = 5/2$ el cual es igual a $\omega_f(G)$, esto no es casualidad, de hecho esto se cumple para cualquier gráfica como se demostrará más adelante.

Ya que $\omega(G) \leq \omega_f(G)$ y $\chi(G) \geq \chi_f(G)$, además de saber que se cumple $\omega(G) \leq \chi(G)$, entonces es natural suponer que

$$\omega_f(G) \leq \chi_f(G). \tag{3.1}$$

Para demostrar la desigualdad anterior, es necesario hacer una descripción alternativa de $\omega_f(G)$ y $\chi_f(G)$.

Primero, recordemos la formulación del problema del clique máximo como un problema lineal entero binario, el cual formulamos de la siguiente manera:

Sean:

$$x_i = \begin{cases} 1 & \text{si el vértice i se incluye en el clique} \\ 0 & \text{en otro caso} \end{cases}$$

$$\begin{array}{lrcl} \text{Max} & z & = & 1x \\ s.a. & & & \\ & Ax & \leq & 1 \\ & x_i & \in & \{0,1\},\ i \in V(G), \end{array}$$

donde A es una matriz, para la cual cada columna corresponde a cada uno de los vértices de G y las filas a cada conjunto independiente de G. Para cada entrada de la matriz, $a_{i,j} = 1$, si el vértice correspondiente a la columna i, está en el conjunto independiente de la fila j; en caso de no ser así, $a_{i,j} = 0$.

De la misma manera, podemos formular el problema de coloración como un problema linel entero binario. Para este caso recordemos que una coloración hace una partición del conjunto V entre conjuntos $C_1, C_2, \ldots, C_k$, donde la unión de estos conjuntos forman V. El problema de coloración mínima consiste en encontrar el menor k, es decir el menor número de conjuntos independientes de la gráfica, tal que cumplan con las condiciones de coloración.

De esta forma la variable de decisión es:

$$y_i = \begin{cases} 1 & \text{si el conjunto independiente i se incluye en la coloración} \\ 0 & \text{en otro caso} \end{cases}$$

Y el problema lineal entero binario para la coloración mínima es el siguiente:

$$\begin{array}{lrcl} \text{Min} & c & = & y1 \\ s.a. & & & \\ & yA & \geq & 1 \\ & y_i & \in & \{0,1\},\ i \in I(G). \end{array}$$

La matriz A es la misma que la del problema del clique, mientras que y es un vector renglón.

Ejemplo 9. *Daremos la formulación como un problema de programación linela entera para el problema del clique máximo y para el problema de coloración para el ciclo C_5.*

Para esta gráfica tenemos que

$$I(G) = \{\{v_1\},\{v_2\},\{v_3\},\{v_4\},\{v_5\},\{v_1,v_3\},\{v_1,v_4\},\{v_2,v_4\}, \{v_2,v_5\},\{v_3,v_5\}\}.$$

La matriz A, será aquella donde cada columna corresponde a cada vértices y cada fila a cada conjunto independiente. Por lo que A es una matriz de 10×5.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

El problema del clique máximo para C_5 como un problema de programación lineal entera es:

$$Max\ z = x_1 + x_2 + x_3 + x_4 + x_5$$

$$\begin{array}{lccccccccccc}
s.a. & & & & & & & & & & & \\
x_1 & & & & & & & & & & \leq & 1 \\
 & x_2 & & & & & & & & & \leq & 1 \\
 & & & x_3 & & & & & & & \leq & 1 \\
 & & & & & x_4 & & & & & \leq & 1 \\
 & & & & & & & x_5 & & & \leq & 1 \\
x_1 & & + & x_3 & & & & & & & \leq & 1 \\
x_1 & & & & + & x_4 & & & & & \leq & 1 \\
 & x_2 & & & + & x_4 & & & & & \leq & 1 \\
 & x_2 & & & & & + & x_5 & & & \leq & 1 \\
 & & & x_3 & & & + & x_5 & & & \leq & 1
\end{array}$$

$$x_j \in \{0,1\}, \quad j \in \{1,2,3,4,5\}.$$

En este caso el x_i corresponde a cada vértice, asi que si un vértice está incluido en el clique máximo entonces su peso correspondiente será cero, en caso contrario es uno.

El problema de la coloración máxima para C_5 como un problema de programación lineal entera es:

$$Min\ c = y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7 + y_8 + y_9 + y_{10}$$

$$\begin{array}{lcccccccccccccc}
s.a. & & & & & & & & & & & & & & \\
y_1 & & & & & + & y_6 & + & y_7 & & & & & & \geq & 1 \\
 & y_2 & & & & & & & & + & y_8 & + & y_9 & & \geq & 1 \\
 & & y_3 & & & + & y_6 & & & & & & & + & y_{10} & \geq & 1 \\
 & & & y_4 & & & & + & y_7 & + & y_8 & & & & \geq & 1 \\
 & & & & y_5 & & & & & & & + & y_9 & + & y_{10} & \geq & 1
\end{array}$$

$$y_j \in \{0,1\}, \quad j \in \{1,2,3,\ldots,10\}.$$

Ahora, si relajamos las formulaciones lineales, es decir, cambiamos la condición de valores binarios por reales, entonces obtenemos los las formulaciones lineales del clique fraccional y coloración fraccional siguientes:

$$\left.\begin{array}{rrcl} Max & z & = & 1x \\ s.a. & & & \\ & Ax & \leq & 1 \\ & x_i & \geq & 0,\ i \in V(G) \end{array}\right\} \text{ Problema primal (P)}$$

$$\left.\begin{array}{rrcl} Min & c & = & y1 \\ s.a. & & & \\ & yA & \geq & 1 \\ & y_j & \geq & 0,\ j \in I(G). \end{array}\right\} \text{ Problema dual (D)}$$

Por la teoría de la dualidad, todo problema lineal, llamado primal, tiene un dual asociado. Y justamente, el dual del problema del clique fraccional es el de coloración fraccional.

Ahora ya contamos con las condiciones necesarias para demostrar (3.1).

Lema 28. *Sean la pareja de problemas duales P y D, entonces para cualquier clique fraccional x y cualquier coloración fraccional y de G, se cumple que $z \leq c$.*

Demostración. Ya que x es un clique fraccional y y una coloración fraccional, entonces x y y son soluciones factibles de sus respectivos problemas por lo que

$$\begin{aligned} y1 - 1x &= y1 + (-yAx + yAx) - 1x \\ &= y(1 - Ax) + (yA - 1)x. \end{aligned}$$

Ya que x es un clique fraccional, entonces $1 - Ax \geq 0$. Y y es una coloración fraccional, entonces $y \geq 0$, por lo tanto $y(1-Ax) \geq 0$. De la misma manera, x y $(yA-1)$ son no negativos, por lo que $(yA-1)x \geq 0$. Entonces $y1 - 1x$ es la suma de dos números no negativos, por lo tanto $y1 \geq 1x$ para cualquier clique fraccional x y cualquier coloración fraccional y. □

La demostración que acabamos de hacer, es básicamente la prueba del teorema de dualidad debil de programación lineal. Una consecuencia inmediata del teorema anterior, establece que si se cumple la igualdad del lema 28, entonces estas soluciones son óptimas.

Lema 29. *Para cualquier gráfica G, si x' es un clique fraccional, y y' una coloración fraccional, tales que $1x' = y'1$, entonces x' y y' son soluciones óptimas de P y D respectivamente.*

Demostración. Sea x un clique fraccional de G, es decir, x es una solución factible de P. Como y' es una solución factible de D, por el teorema 28 se tiene:

$$1x \leq y'1,$$

por hipótesis tenemos que

$$y'1 = 1x',$$

por lo que

$$1x \leq 1x'$$

para cualquier clique fraccional x. Por lo tanto, x' es solución óptima de P. Análogamente y' es solución óptima de D. □

Ya que se demostró que x' y y' son soluciones óptimas, entonces $\omega_f(G) = 1x'$ y $\chi_f(G) = y'1$, debido a que $1x' = y'1$ es el valor óptimo para P y D. Por lo tanto, $\omega_f(G) = \chi_f(G)$.

3.2. Gráficas Perfectas

En esta sección daremos una pequeña introducción a gráficas perfectas, para definir y caracterizar una clase especial de ellas: las gráficas trianguladas. Este tipo de gráficas serán de gran utilidad en el algoritmo, ya que cumplen con $\omega(G) = \chi(G)$, la cual no sólo se cumple para la gráfica completa sino también para cualquier subgráfica inducida.

Antes de definir a las gráficas perfectas, recordaremos algunas conceptos.

$\omega(G)$ es la cardinalidad del clique máximo de G.

$\chi(G)$ es el mínimo número de colores para colorear propiamente a G, llamado número cromático.

$\alpha(G)$ es la cardinalidad del máximo conjunto independiente de G, conocido como el número estable o independiente.

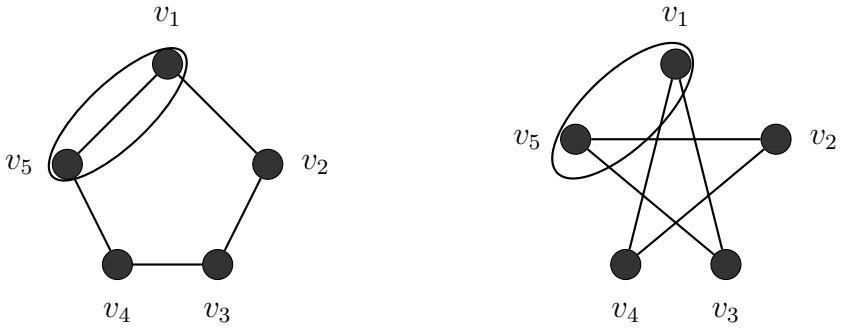
Otro concepto que está relacionado con los anteriores, pero que no había sido mencionado, es $\theta(G)$ que denota el óptimo del problema de la cobertura de cliques. Este al igual que el problema de coloración, el del máximo conjunto independiente y el del clique máximo es NP-Completo, y consiste en hacer una partición de los vértices en k conjuntos, de tal forma que cada conjunto forme un clique. El mínimo k que cumple con las condiciones es $\theta(G)$.

Los vértices que forman el clique máximo en una gráfica G, van a ser el máximo conjunto independiente en la gráfica complemento de G, $\overline{G}$, por lo que se cumple que: $\omega(G) = \alpha(\overline{G})$. Por ejemplo en la Figura 3.6, tenemos que el clique máximo lo forman la subgráfica inducida por los vértices $\{v_1, v_5\}$ (notemos que para esta gráfica el clique máximo no es único), por lo tanto $\omega(G) = 2$. Este mismo conjunto de vértices en la gráfica $\bar{G}$ cumple con ser el máximo conjunto independiente.

Lo mismo sucede con el problema de cobertura de cliques, los conjuntos de la partición de vértices óptima de cualquier gráfica G cada uno forman un clique, es decir, una subgráfica completa en G, su complemento formará un conjunto independiente en $\bar{G}$. Y por lo tanto, cada uno de estos conjuntos será una clase de coloración en $\bar{G}$, cumpliendose $\theta(G) = \chi(\bar{G})$. Ver Figura 3.7.

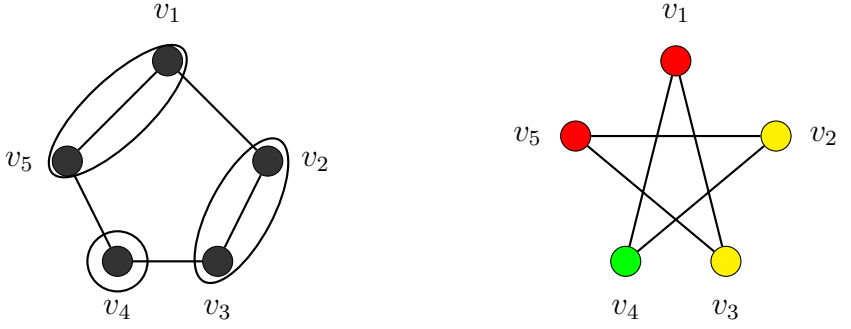
Como se demostró en la proposición 24 para toda gráfica G, $\omega(G) \leq \chi(G)$. Además $\alpha(G) \leq \theta(G)$, ya que un clique y un conjunto independiente a lo más comparten un vértice.

Con todos estos conceptos claros, así como su relación entre ellos, ya estamos en condiciones para comenzar.



(a) G **(b)** $\bar{G}$

Figura 3.6: Para las gráficas tenemos que $\omega(G) = 2 = \alpha(\bar{G})$



(a) G **(b)** $\bar{G}$

Figura 3.7: Para las gráficas tenemos que $\theta(G) = 3 = \chi(\bar{G})$

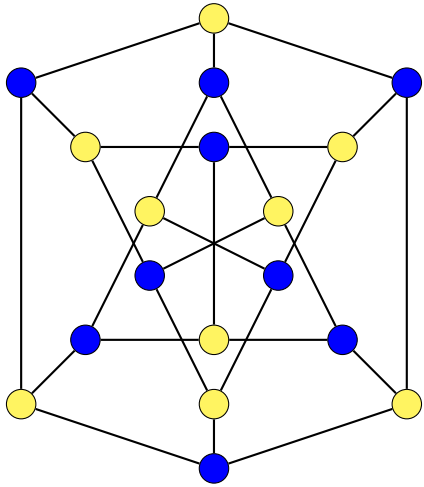


Figura 3.8: La gráfica de Pappus es un ejemplo de gráfica perfecta

El concepto de gráfica perfecta fue introducida por Claude Berge en 1960 [38]. Berge establece que una gráfica G es perfecta si cumple las siguientes dos propiedades:

$$\omega(G_A) = \chi(G_A) \qquad \forall A \subseteq V \qquad [P1] \tag{3.2}$$

y

$$\alpha(G_A) = \theta(G_A) \qquad \forall A \subseteq V \qquad [P2]. \tag{3.3}$$

Una gráfica que satisface la propiedad $P1$ es llamada $\chi - perfecta$ y es $\alpha - perfecta$ si satisface $P2$. Berge conjetura que χ-perfección y α-perfección son equivalentes. Esta conjetura fue trabajada por Fulkerson en 1969, 1971 y 1972 [30][31][32], y finalmente fue probada por Lovász en 1972 [53].

Teorema 30 (**El teorema de la gráfica perfecta**). *Una gráfica G es χ-perfecta $\iff$ es α-perfecta.*

Este teorema es equivalente a decir que G es perfecta si y sólo si $\overline{G}$ es perfecta.

Una tercera condición equivalente, debida a Lovász [52], dice que

$$w(G_A)\alpha(G_A) \geq |A|, \qquad \forall A \subseteq V.$$

Por lo que mostrar que una gráfica cumple con alguna de las propiedades será suficiente para afirmar que la gráfica es perfecta.

Fulkerson en 1973 [33] también prueba el teorema de la gráfica perfecta utilizando teoría poliedral.

En 1963, Berge hace una conjetura aún más fuerte de gráficas perfectas, la cual fue probada hasta el 2006 por Chudnovsky, Robertson, Seymour y Thomas[20]. Este teorema es conocido como el teorema fuerte de las gráficas perfectas, el cual es mencionado acontinuación, sin presentar la demostración.

Teorema 31. *[**Teorema fuerte de las gráficas perfectas**] Una gráfica G es perfecta si y sólo si G y $\overline{G}$ no tienen como subgráficas inducidas a C_{2k+1} o $\overline{C_{2k+1}}$, con $k \geq 2$.*

Podemos observar claramente que las gráficas C_{2k+1} con $k \geq 2$ no son perfectas, ya que $\chi(C_{2k+1}) = 3$ y $\omega(C_{2k+1}) = 2$; y por el teorema 31, $\overline{C_{2k+1}}$ tampoco es perfecta.

Ejemplos de gráficas que se ha mostrado que son perfectas son las gráficas bipartitas, las cordales o trianguladas, las de comparabilidad y muchas otras. Para los fines de nuestro algoritmo, se utilizará una clase especial de gráficas perfectas conocidas como gráficas trianguladas.

3.2.1. Gráficas trianguladas

Una de las primeras clases de gráficas que se reconocieron perfectas fue la clase de gráficas trianguladas[1]. Hajnal y Surányi [39] mostraron en 1958 que las gráficas trianguladas satisfacen la propiedad

[1]Las gráficas trianguladas también son conocidas como gráficas cordales *(Chordal Graphs)*.

P_2 (α-perfección), y Berge en 1960 [10] probó que satisfacen P_1 (χ-perfección). Estos dos resultados, en gran medida inspirarón la conjetura de que P_1 y P_2 son equivalentes, que ahora se conoce como verdadera; por lo que el estudio de las gráficas triánguladas, bien podría ser conocido como el inicio del estudio de las gráficas perfectas.

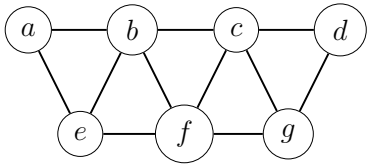
Una gráfica G es llamada triangulada si todo ciclo C_n, con $n > 3$, posee una cuerda *(chord)*, esto es, una arista adyacente a dos vértices no consecutivos en el ciclo. De manera equivalente, G es triangulada si no contiene una subgráfica inducida isomorfa a C_n para $n > 3$. Toda gráfica triángulada G tiene la propiedad de que toda subgráfica inducida de G también es triangulada.

Un vértice x de G es llamado simplicial si su vecindad es un clique.

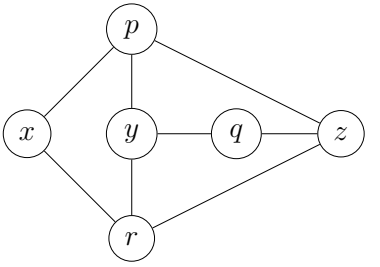
Una biyección $f : \{1, 2, \ldots, n\} \to V$, con $|V| = n$, es llamada una ordenación de G. Sea f^{-1} la inversa de f, por lo que $f^{-1}(v)$ denota la posición en la ordenación asignada al vértice v. El conjunto $N_f(v)$ está definido como el conjunto de todos los vecinos u de v, tal que $f^{-1}(u) > f^{-1}(v)$, en otras palabras, todos los vecinos de v que tienen la posición más alta en la ordenación f. Una ordenación f es llamada una ordenación de eliminación perfecta si para cada $v \in V$, $N_f(v)$ es un clique.

Sea $G = (V, E)$ una gráfica y sea $\sigma = (v_1, v_2, \ldots, v_n)$ una ordenación de los vértcies. Se dice que σ es una ordenación de eliminación perfecta si cada v_i es un vértice simplicial de la subgráfica inducida $G_{\{v_i,\ldots,v_n\}}$. En otras palabras, cada conjunto $X_i = \{v_j \in N(v_i) | j > i\}$ es un clique. Por ejemplo, la gráfica triangulada de la Figura 3.9 tiene una ordenación de eliminación perefecta $\sigma = (a, d, g, e, f, b, c)$. Para este caso, la ordenación σ no es única, de hecho existen 96 diferentes. A diferencia de la gráfica no triangulada de la Figura 3.9, para la cual no existen vértices simplicial, por lo tanto, no podemos comenzar a construir ninguna ordenación de eliminación perfecta.

Un $(u, v)-$separador de G es un conjunto $S \subseteq V$, tal que $G - S$ es no conexo y además u y v quedan en componentes conexas distintas. Es minimal si ningún conjunto menor tiene la misma propiedad.



(a) Gráfica triangulada



(b) Gráfica no triangulada

Figura 3.9

Consideremos nuevamente las gráficas de la Figura 3.9, para la gráfica no triangulada tenemos que el conjunto $\{y, z\}$ es un $(p, q)-$separador minimal, mientras que $\{x, y, z\}$ es un $(p, r)-$separador minimal. Para el caso de la gráfica triangulada, tenemos que todo separador de vértices minimal tiene cardinalidad dos, además que los dos vertices de cada separador son adyacentes en la gráfica triangulada, de hecho, este fenómeno ocurre en todas las gráficas trianguladas como veremos en el siguiente teorema.

Teorema 32. *Sea G una gráfica no dirigida. Los siguientes enunciados son equivalentes:*

i) G es triangulada

ii) G tiene una ordenación de eliminación perfecta. Además, cualquier vértice simplicial puede comenzar la ordenación de eliminación perfecta.

iii) Todo separador de vértices minimal induce una subgráfica completa de G.

Demostración. $(iii) \Rightarrow (i)$ Sea $[a, x, b, y_1, y_2, \ldots, y_k]$ con $k \geq 1$ un ciclo simple de $G = (V, E)$. Cualquier $(a, b)-$separador de vértices minimal debe contener al vértice x y y_i para alguna i, entonces $(x, y_i) \in E$, la cual es una cuerda del ciclo.

$(ii) \Rightarrow (iii)$ Supongamos que S es un $(a,b)-$separador minimal, con G_A y G_B las componentes conexas de G_{V-S} que contienen a a y b respectivamente. Ya que S es minimal, entonces cada $x \in S$ es adyacente a algún vértice en A y a algún vértice en B. Por lo que para cualquier par $x, y \in S$ existe una trayectoria $[x, a_1, \ldots, a_r, y]$ y $[y, b_1, \ldots, b_t, x]$, donde $a_i \in A$ y $b_i \in B$, además cada trayectoria es de la longitud más pequeña posible. Siguiendo ambas trayectorias tenemos $[x, a_1, \ldots, a_r, y, b_1, \ldots, b_t, x]$ un ciclo simple de longitud al menos 4, lo que implica que el ciclo debe tener una cuerda. Pero $(a_i, b_j) \notin E$ por la definición de separador de vértices, $(a_i, a_j) \notin E$ y $(b_i, b_j) \notin E$ ya que tomamos a r y t lo más pequeñas posibles. Por lo tanto, la única cuerda posible es $(x, y) \in E$.

Observación: De hecho $r = t = 1$, lo que implica que para todo $x, y \in S$ existen vértices en A y B tal que son adyacentes a x y y.

Antes de seguir con las demás implicaciones del teorema, probaremos un lema que nos ayudará a continuar.

Lema 33. *(Dirac [1961]). Toda gráfica triangulada $G = (V, E)$ tiene un vértice simplicial. Además, si G no es clique, entonces tiene dos vertices simplicial no adyacentes.*

Demostración. El lema es trivial si G es completa. En caso contrario, supongamos que G tiene dos vértices a y b no adyacentes, y que el lema es verdadero para todas las gráficas con menos vértices que G. Sea S un separador de vértices minimal para a y b, con G_A y G_B las componentes conexas de G_{V-S} que contienen a a y b respectivamente. Por inducción, G_{A+S} tiene dos vértices simplicial no adyacentes, uno de los cuales debe estar en A, ya que S induce una gráfica completa; o G_{A+S} es completa y cualquier vértice de A es simplicial en G_{A+S}. Además, ya que $N(A) \subseteq A + S$, un vértice simplicial de G_{A+S} en A es simplicial en toda G. Análogamente B contiene un vértice simplicial de G. Lo que prueba el lema. □

(Continuación de la demostración del teorema 32). $(i) \Rightarrow (ii)$ Por el lema anterior, si G es triangulada, entonces G tiene un vértice simplicial. Sea este vértice x. Ya que $G_{V-\{x\}}$ es triangulada y más pequeña

que G, ésta tiene por inducción una ordenación de eliminación perfecta; para la cual al agregar el vértice x al inicio, formamos una ordenación de eliminación perfecta para G.

$(ii) \Rightarrow (i)$ Sea C un ciclo simple de G y sea x el vértice con el número más pequeño en la ordenación de eliminación perfecta de G. Ya que $|N(x) \cap C| \geq 2$ y x es simplicial entonces garantizamos una cuerda en C. Ver Figura 3.10. □

Ahora sólo nos falta probar que las gráficas trianguladas son perfectas, antes de hacer la demostración necesitaremos de dos resultados que utilizan el principio de separación entre piezas *(principle of separation into pieces)* [11], aplicado al problema de coloración y al problema del clique máximo con el fin de simplificarlos.

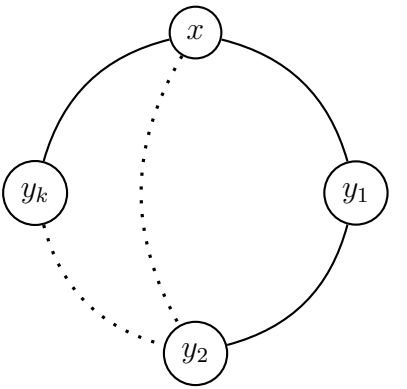
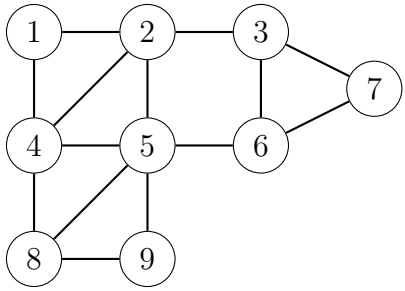


Figura 3.10

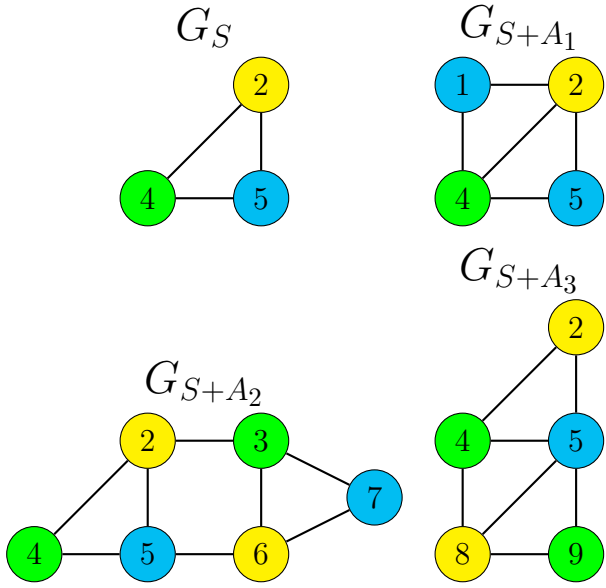
El principio de separación de piezas puede ser aplicado a gráficas que tengan una subgráfica S la cual forma un clique, donde al remover S de la gráfica genere componentes conexas A_1, A_2, A_3, ...; el principio consiste en colorear a S y después separadamente a cada una de las subgráficas G_{S+A_1}, G_{S+A_2}, G_{S+A_3}, ... llamadas piezas, de esta forma encontramos una coloración de la gráfica sin que los colores de los vértices de S entren en conflicto.

Ejemplo 10. *Principio de separación entre piezas [11]*

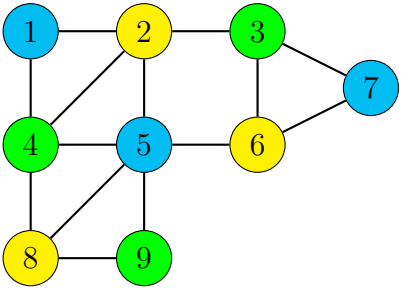
Supongamos que queremos colorear propiamente la siguiente gráfica:



En este caso, el conjunto S es el formado por los vértices $\{2, 4, 5\}$, mientras que las componentes conexas son $A_1 = \{1, 2, 4\}$, $A_2 = \{2, 3, 5, 6, 7\}$ y $A_3 = \{4, 5, 8, 9\}$. Ya identificadas las piezas, podemos aplicar el principio de separación, primero coloreando a S y después a las piezas formadas por las subgráficas $G + A_i$ para $i \in 1, 2, 3$.



Podemos concluir, uniendo cada una de las piezas, que la gráfica es 3−coloreable, ya que cada una de las subgráficas G_{S+A_i} para $i \in \{1, 2, 3\}$ es 3−coloreable.



Teorema 34. *Sea S un vértice separador de una gráfica no dirigida conexa $G = (V, E)$, y sean $G_{A_1}, G_{A_2}, \ldots, G_{A_t}$ las componentes conexas de G_{V-S}. Si S es un clique (no necesariamente maximal), entonces $\chi(G) = \max_i\{\chi(G_{S+A_i})\}$ y $\omega(G) = \max_i\{\omega(G_{S+A_i})\}$.*

Demostración. Claramente $\chi(G) \geq \chi(G_{S+A_i})$ para cada i, en particular $\chi(G) \geq \max_i\{\chi(G_{S+A_i})\} = k$. En realidad, G puede ser coloreado usando exactamente k colores. Primero coloreamos a G_S, luego extendemos independientemente la coloración a cada pieza G_{S+A_i}. Luego, se hace la composición de G con cada una de las coloraciones de las piezas, esto se puede hacer debido que S se mantuvo con la misma coloración para cada una de las componentes. Esta composición será una coloración de G. Por lo tanto $\chi(G) = k$. Para la segunda desigualdad, tenemos que $\omega(G) \geq \omega(G_{S+A_i})$ para cada i. En particular, $\omega(G) \geq \max_i\{\omega(G_{S+A_i})\} = m$. Sea X el máximo clique de G, es decir, $|X| = \omega(G)$. Por lo que para cualesquiera dos vértices en X es imposible que uno este en G_{A_i} y otro en G_{A_j} para $i \neq j$, ya que todo par de vértices están conectados. Por lo tanto, X esta contenido totalmente en alguna de las piezas, supongamos que está en G_{S+A_r}, por lo que, $m \geq \omega(G_{S+A_r}) \geq |X| = \omega(G)$. Por lo tanto, $\omega(G) = m$. □

Corolario 35. *Sea S un conjunto de separación de una gráfica conexa no dirigida $G = (V, E)$ y sean $G_{A_1}, G_{A_2}, \ldots, G_{A_t}$ las componentes conexas de G_{V-S}. Si S es un clique, y si cada subgráfica G_{S+A_i} es perfecta, entonces G es perfecta.*

Demostración. Supongamos que el resultado es verdadero para todas las gráficas con menos vértices que G. Usando el teorema 34 y la hipótesis de que cada G_{S+A_i} es perfecta, tenemos que

$$\chi(G) = \max_i\{\chi(G_{S+A_i})\} = \max_i\{\omega(G_{S+A_i})\} = \omega(G).$$

□

Ahora ya estamos preparados para demostrar el resultado principal.

Teorema 36. *(Berge[1960], Hajnal y Surányi[1958]). Toda gráfica triangulada es perfecta.*

Demostración. Sea G una gráfica triangulada, supongamos que el teorema es verdadero para todas las gráficas con menos vértices que G. Supongamos que G es conexa, en caso que no lo sea consideramos cada componente individualmente. Si G es completa, entonces es claro que G es perfecta. Si G no es completa, entonces sea S un separador de vértices minimal para algún par de vértices no adyacentes. Por el teorema 32, S es un clique. Además, por la hipótesis de inducción, cada una de las subgráficas trianguladas G_{S+A_i}, como se definió en el corolario 35, es perfecta. Por lo tanto, por el colorario 35, G es perfecta. □

Capítulo 4

Algoritmo del clique máximo (ACM)

4.1. Descripción general del algoritmo

En este capítulo presentaremos el algoritmo del clique máximo desarrollado por David R. Wood [78].

El algoritmo se basa en el método de ramificación y acotamiento *(branch and bound)*, que como se explicó en la sección 2.4, se encuentra dentro de los algoritmos de enumeración implícita, para los cuales la idea principal es considerar solamente una porción de todas las soluciones factibles mientras que descartamos las restantes como «no prometedoras».

Este tipo de métodos se puede representar mediante un árbol, donde cada nodo va a representar una solución al problema y cada rama nos lleva a otra posible posterior a la actual. Bajo ciertas reglas el algoritmo va a ser capaz de detectar en qué momento las solución futura ya no será prometedora, para descartar ese nodo y de esta forma no malgastar recursos.

Para el problema del clique máximo Pardalos y Xue [64] propusieron las siguientes preguntas clave para identificar un buen algoritmo

de ramificación y acotamiento:

1. ¿Cómo encontrar una buena cota inferior? Es decir, un clique de tamaño grande.

2. ¿Cómo encontrar una buena cota superior sobre el tamaño del clique máximo?.

3. ¿Cómo ramificar? Es decir, dividir un problema en subproblemas más pequeños.

Para el algoritmo que presentaremos las cotas superiores estarán dadas por $\chi(G)$ y $\chi_f(G)$ y serán aproximados utilizando algunos heurísticos que serán descritos más adelante. Mientras que la cota inferior será la cardinalidad del clique obtenido más grande hasta ese momento. Estas cotas nos permitirán saber si debemos ramificar.

Ahora vamos a describir el algoritmo del clique máximo de manera general, mediante un diagrama de flujo que se presenta en las Figuras 4.1 y 4.2.

Dada una gráfica $G = (V, E)$, el algoritmo mantiene las siguientes condiciones:

- Si h es la profundidad del árbol de búsqueda, el conjunto $\{v_1, v_2, \ldots, v_{h-1}\}$ consiste de vértices adyacentes entre sí.

- M es el clique más grande encontrado por el algoritmo y su cardinalidad es la cota inferior para $\omega(G)$; $1 \leq |M| \leq \omega(G)$.

- Para $1 \leq i \leq h$, el conjunto de vértices $S_i \subseteq \bigcap_{j=1}^{i-1} N_G(v_j)$ contiene candidatos para hacer más grande $\{v_1, v_2, \ldots, v_{i-1}\}$.

- Para $1 \leq i \leq h$, $(C_1^i, C_2^i, \ldots, C_k^i)$ es una coloración de vértices de $G(S_i)$, con cardinalidad k_i. Mientras que la cardinalidad de la coloración fraccional de $G(S_i)$ es k_i'. Ambas son cotas superiores para $\omega(G(S_i))$, con $k_i' \leq k_i$.

- Un nodo activo del árbol de búsqueda corresponde al subproblema de encontrar un clique más grande que M de la subgráfica:

$$G_i = G(\{v_1, v_2, \ldots, v_{i-1}\} \cup S_i),$$

para $1 \leq i \leq h$. Claramente, $\omega(G_i) \leq i - 1 + k'_i \leq i - 1 + k_i$.

- En cada estado de ramificación se activa exactamente un nodo en el árbol de búsqueda.

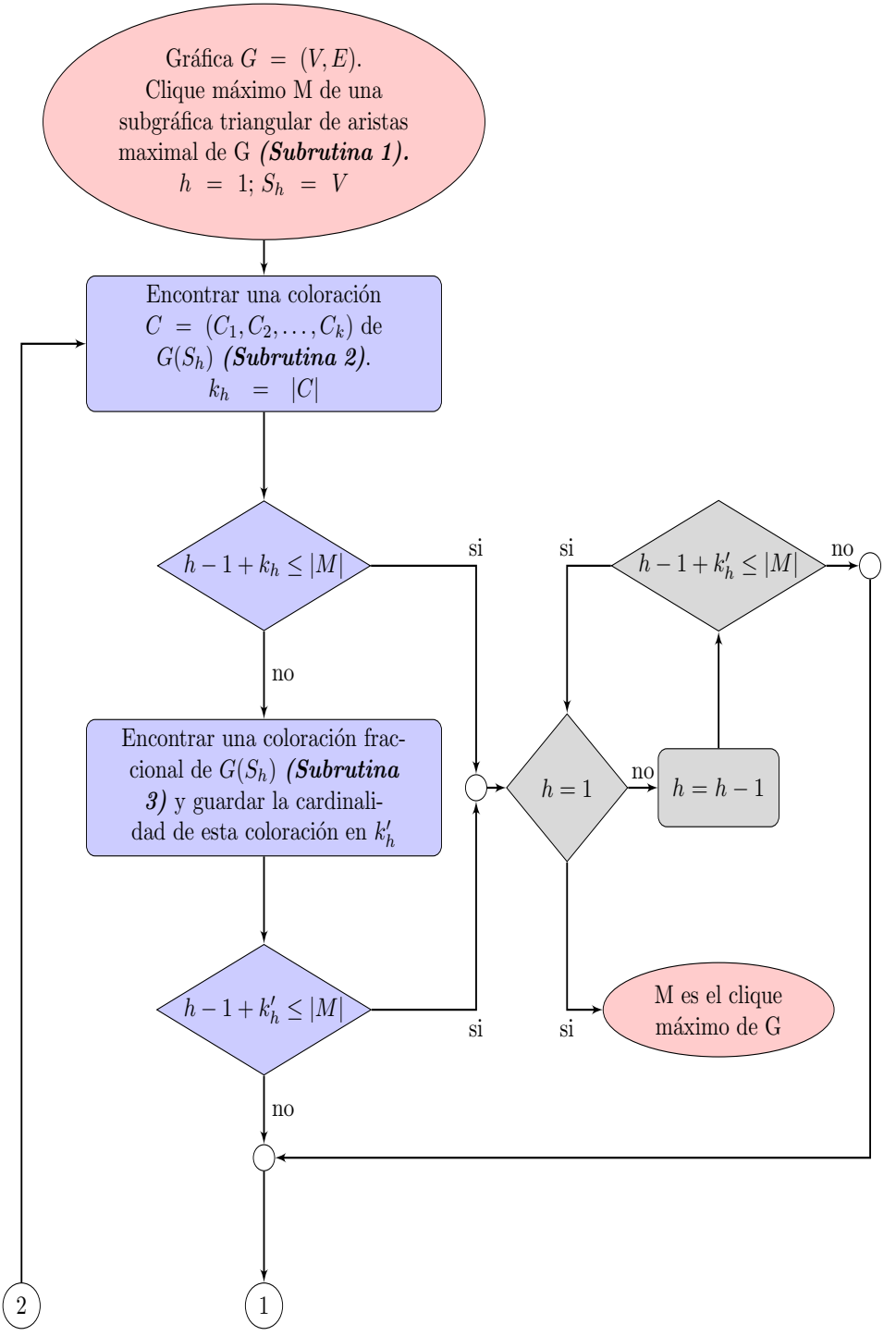


Figura 4.1: Diagrama de flujo del algoritmo del clique máximo (Parte1)

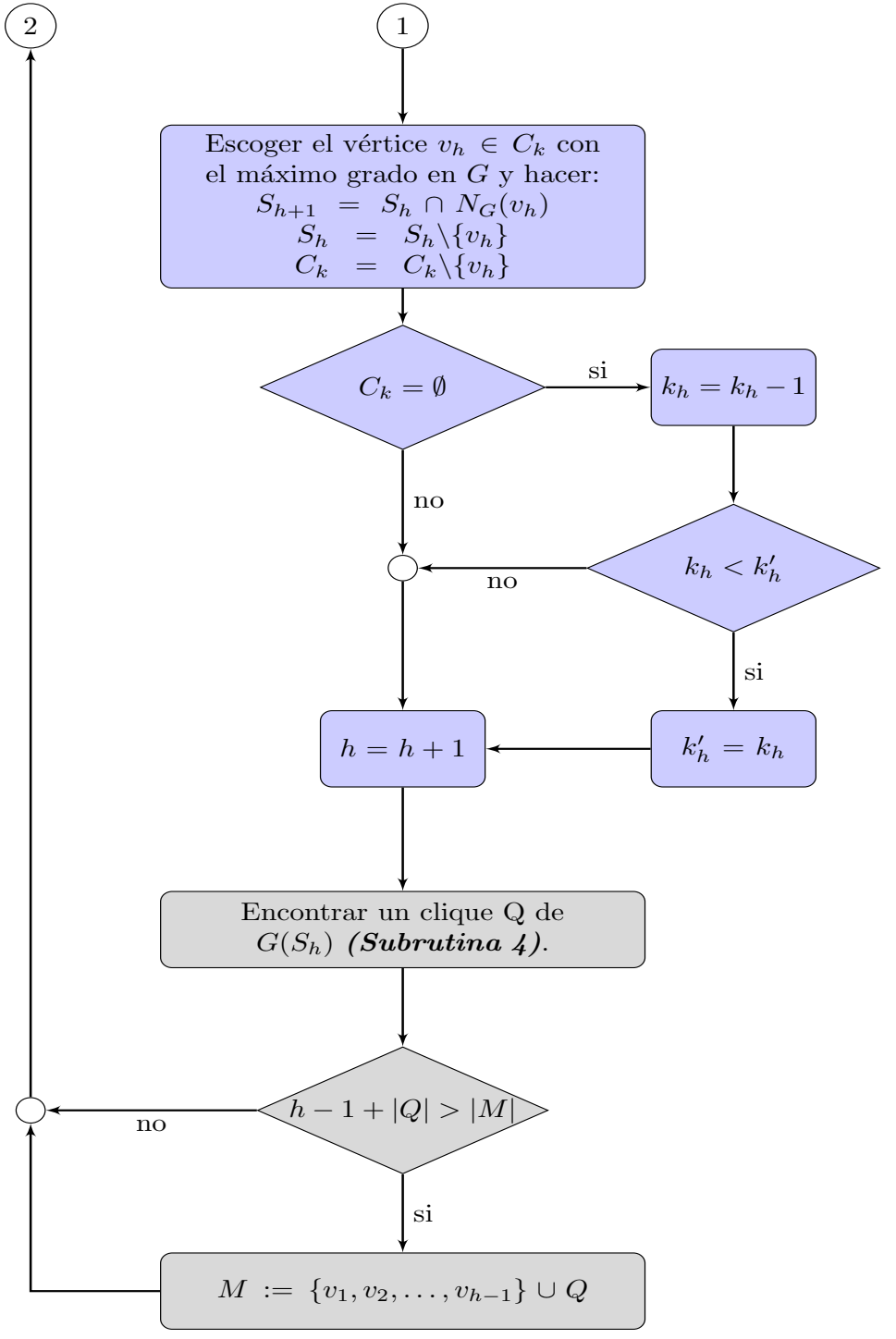


Figura 4.2: Diagrama de flujo del algoritmo del clique máximo (Parte2)

En el diagrama de flujo se puede ver que hay cuatro subrutinas, estas solo están indicadas y serán desarrolladas más adelante.

Descripción del algoritmo del clique máximo utilizando el diagrama de flujo

Inicialización. Como se puede ver en la Figura 4.1 el algoritmo comienza con un clique inicial M máximo de la subgráfica triangulada maximal (el cual es calculado con la subrutina 1 que se explicará más adelante) un conjunto $S_h = V$ y $h = 1$. Si esto comienza un árbol de búsqueda, estos serían los valores guardados en el nodo raíz. El conjunto S_h son los candidatos que pueden ser considerados en un clique de cardinalidad mayor, en este caso como $h = 1$ entonces $S_1 = V$; es decir, en el nodo inicial los posibles candidatos son todos los vértices de la gráfica G.

Calcular cotas superiores. El bloque de color azul en el diagrama de flujo parte 1 (Figura 4.1), es la parte que se encarga de calcular las cotas superiores. El primer paso es encontrar una coloración de la subgráfica inducida por el conjunto S_h; es decir, la gráfica $G(S_h)$, utilizando la subrutina 2. Las clases de coloración obtenidas por la subrutina 1 se guardan en $C = (C_1, C_2, \ldots, C_k)$ y la cardinalidad de la coloración se guarda en k_h.
Posteriormente se hace la comparación $h - 1 + k_h \leq |M|$, donde $|M|$ es la cardinalidad del clique encontrado hasta ese momento.
Si la desigualdad es verdadera y además $h = 1$; es decir, nos encontramos en el nodo raíz, entonces pasaríamos al bloque gris de la Figura 4.1 donde siguiendo el flujo se llega al símbolo final del diagrama, lo que nos indicaría que el clique máximo es el inicial.
Si nos encontramos en el caso en el que se cumple $h - 1 + k_h \leq |M|$ pero $h \neq 1$, entonces ya se ramificó al menos una vez; es decir, nos encontramos en un nodo diferente al raíz, en este punto el flujo del diagrama nos lleva al bloque gris de la Figura 4.1 que se encargará de realizar el retroceso en el árbol de búsqueda y que será descrito más adelante.
Ahora en caso de que $h - 1 + k_h > |M|$, se buscará una mejor cota superior utilizando la subrutina 3, que realizará una coloración fraccio-

nal de la gráfica $G(S_h)$. La cardinalidad de esta se guarda en k'_h, para después verificar si se cumple que $h-1+k'_h \leq |M|$, en caso de que no se cumpla, se pasa al bloque gris para hacer el retroceso.
En caso de que $h-1+k'_h > |M|$ entonces el flujo del diagrama nos lleva al bloque azul del diagrama de flujo parte dos (Figura 4.2), que corresponde a la ramificación y que será descrita a continuación.

Ramificación. El bloque que realiza la ramificación se encuentra en la Figura 4.2 en color azul. En el primer rectángulo, se escoge el vértice con el grado más alto en G de la última clase de coloración C_k. Esta debe existir ya que para llegar hasta aquí, es porque previamente se pasó por el bloque de las cotas superiores, donde por lo menos se realizó el primer proceso que consiste en encontrar una coloración. El vértice se guarda en v_h y se forma el nuevo conjunto de candidatos S_{h+1}, el cual es el anterior conjunto intersectado con los vecinos en G de v_h, mientras que el conjunto S_h también es modificado al eliminar del conjunto el vértice v_h. Después se puede ver que tenemos un símbolo de decisión, para que en el caso de que la clase C_k este vacía, entonces es claro que la cardinalidad de la coloración se va a decrementar en uno ($k_h = k_h - 1$). Además si la cardinalidad de la coloración es menor que la cardinalidad de la fraccional, entonces $k'_h = k_h$ ya que recordemos que $\chi_f(G) \leq \chi(G)$ para cualquier gráfica G. Por último se incrementa h. Con todo esto estamos generando un nuevo nodo del árbol de búsqueda.

Calcular la cota inferior. El bloque gris del diagrama de flujo parte 2 (Figura 4.2), muestra el cálculo de un clique Q de la subgráfica $G(S_h)$ utilizando la subrutina 4. En el símbolo de decisión tenemos que si la cardinalidad del clique generado sumando $h-1$, esto por que es el conjunto de candidatos para hacer más grande el clique, es mayor que $|M|$, entonces el nuevo clique es $M := \{\{v_1, v_2, \ldots, v_{h-1}\} \cup Q\}$.

Retroceso. Por último, el proceso de retroceso; es decir, si es que el problema ya no genera soluciones prometedoras, porque las cotas del bloque de cotas superiores se excedieron, entonces es necesario regresar un nodo o más atrás. Esto se puede observar en el bloque gris del diagrama parte 1 (Figura 4.1). Si $h = 1$ es claro que nos encontramos

en el nodo raíz y ya no es posible retroceder, por lo tanto, M es el clique máximo. En caso de que $h > 1$, entonces se decrementa h y se hace la comparación de la cota superior con la coloración fraccional, en caso de no cumplirla se procederá a ramificar, si no es así entonces retrocedemos un nodo más. Así hasta que $h-1+k'_h \leq |M|$ o bien $h = 1$.

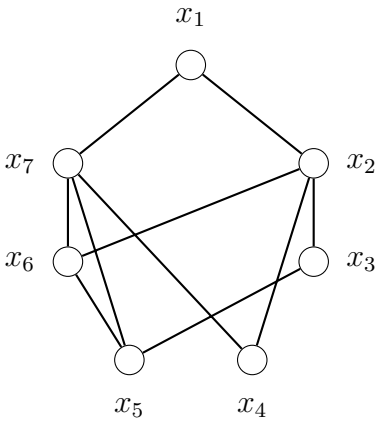


Figura 4.3: Gráfica del ejemplo del algoritmo general

Ejemplo 11. *Aplicaremos el algoritmo general a la gráfica de la Figura 4.3 para encontrar el máximo clique.*

Para ilustrar todos los pasos, los resultados de las subrutinas están dados de tal manera que obligamos al algoritmo a recorrer cada uno de los bloques. Además notemos que el clique máximo para la gráfica está formado por los vértices $\{x_5, x_6, x_7\}$, esto se puede probar por simple inspección.

Inicialización:

Gráfica $G = (V, E)$ como en la Figura 4.3.

$h = 1$

Cota inferior (Clique inicial utilizando la subrutina 1)

Con lo que obtenemos al clique $M = \{x_2, x_3\}$ y el conjunto de vértices candidatos es $S_1 = V = \{x_1, x_2, \ldots, x_7\}$.

Cota superior (Coloración utilizando la subrutina 2)

Encontramos una coloración de $G(S_1)$ que en este caso es igual a la gráfica G, obteniendo:
$C = \{C_1 = \{x_2, x_5\}, C_2 = \{x_7\}, C_3 = \{x_4\}, C_4 = \{x_6\}, C_1 = \{x_1, x_3\}\}$.

Por lo que, la cota superior es igual a cardinalidad de C, $k_1 = 5$.

Haciendo la prueba para k_1 y $|M|$ tenemos que:

$$5 = h - 1 + k_1 \not\leq |M| = 2.$$

Como no se cumple, continuamos encontrando una nueva cota superior.

Cota superior (Coloración fraccional utilizando la subrutina 3)

Ahora encontramos la cardinalidad de una coloración fraccional de $G(S_1)$ con lo que obtenemos:

$$k_1' = \lfloor 10/2 \rfloor = 5.$$

Haciendo la prueba para k_1' y $|M|$ tenemos que:

$$5 = h - 1 + k_1' \not\leq |M| = 2.$$

Debido a que no se cumple, continuamos con la ramificación.

Ramificación

Para ramificar elegimos un vértice con el grado máximo en la gráfica G, que pertenezca a la última clase de coloración y lo guardamos en v_1.

Ya que la última clase es $C_5 = \{x_1, x_3\}$ y los grados de estos vértices son $d(x_1) = d(x_3) = 2$, entonces podemos elegir a cualquiera de los dos, en este caso escogemos a x_1.

Entonces $v_1 = x_1$ y generamos el nuevo conjunto de vértices candidatos:

$$\begin{aligned} S_2 &= S_1 \cap N_G(v_1) \\ &= \{x_1, x_2, \ldots, x_7\} \cap \{x_2, x_7\} \\ &= \{x_2, x_7\}. \end{aligned}$$

Mientras que S_1 y C_5 quedan de la siguiente manera:

$$\begin{aligned} S_1 &= S_1 \setminus \{v_1\} = \{x_2, x_3, \ldots, x_7\} \\ C_5 &= C_5 \setminus \{v_1\} = \{x_3\}. \end{aligned}$$

Por último aumentamos en uno a h:

$$h = h + 1 = 2.$$

$h = 2$

Cota inferior (Clique utilizando la subrutina 4)

Encontramos a un clique de $G(S_2)$ utilizando la subrutina 4, obteniendo $Q = \{x_2\}$, con cardinalidad igual a uno.

Hacemos la prueba para ver si podemos mejorar al clique actual M:

$$2 = h - 1 + |Q| \not> |M| = 2.$$

No se cumple, por lo que continuamos con el algoritmo encontrando una nueva cota superior.

Cota superior (Coloración utilizando la subrutina 2)

Encontramos una coloración de $G(S_2)$ obteniendo:

$$C = \{C_1 = \{x_2\}, C_2 = \{x_7\}\}.$$

Por lo que la cota superior es igual a cardinalidad de C, $k_2 = 2$.

Haciendo la prueba para k_2 y $|M|$ tenemos que:

$$3 = h - 1 + k_2 \nleq |M| = 2.$$

Como no se cumple, continuamos encontrando una nueva cota superior.

Cota superior (Coloración fraccional utilizando la subrutina 3)

Ahora encontramos la cardinalidad de una coloración fraccional de $G(S_2)$ con lo que obtenemos:

$$k_2' = \lfloor 2 \rfloor = 2.$$

Haciendo la prueba para k_2' y $|M|$ tenemos que:

$$3 = h - 1 + k_2' \nleq |M| = 2.$$

Debido a que no se cumple, continuamos con la ramificación.

Ramificación

Para ramificar elegimos un vértice de grado máximo en la gráfica G, que pertenezca a la última clase de coloración y lo guardamos en v_2.

Ya que la última clase es $C_2 = \{x_7\}$ y debido a que solo tiene un vértice, entonces $v_2 = x_7$.

Ahora generamos el nuevo conjunto de vértices candidatos:

$$S_3 = S_2 \cap N_G(v_2) = \emptyset.$$

Mientras que S_2 y C_2 quedan de la siguiente manera:

$$S_2 = S_2 \setminus \{v_2\} = \{x_2\}$$
$$C_2 = C_2 \setminus \{v_2\} = \emptyset.$$

Será necesario actualizar k_2 debido a que $C_2 = \emptyset$, por lo que $k_2 = K_2 - 1 = 1$.

Y como $k_2 < k_2'$, entonces $k_2' = k_2' - 1 = 1$.

Por último aumentamos en uno a h:

$$h = h + 1 = 3.$$

$h = 3$

Cota inferior (Clique utilizando la subrutina 4)

Encontramos a un clique de $G(S_3)$, debido a que este es vacío, entonces el $|Q| = 0$.

Hacemos la prueba para ver si podemos mejorar al clique actual M:

$$2 = h - 1 + k_3 \leq |M| = 2.$$

Debido a que se cumple entonces es necesario hacer el paso del retroceso, esto quiere decir que no es prometedor este nodo del árbol de búsqueda.

Retroceso

Como $h \neq 1$, entonces disminuimos a h en una unidad obteniendo

$$h = h - 1 = 2.$$

$h = 2$

Hacemos la prueba para k_2' y $|M|$:

$$2 = h - 1 + k_2' \leq |M| = 2.$$

Ya que se cumple, pero $h \neq 1$ entonces disminuimos a h en una unidad

$$h = h - 1 = 1$$

.

$h = 1$

Hacemos la prueba para k_1' y $|M|$:

$$5 = h - 1 + k_1' \not\leq |M| = 2.$$

Como esta última prueba no se cumple, entonces es necesario ramificar.

$h = 1$

Ramificación

Para ramificar elegimos un vértice de grado máximo en la gráfica G, que pertenezca a la última clase de coloración y lo guardamos en v_1.

Ya que la última clase es $C_5 = \{x_3\}$ y debido a que solo tiene un vértice, entonces $v_1 = x_3$.

Ahora generamos el nuevo conjunto de vértices candidatos:

$$\begin{aligned} S_2 &= S_1 \cap N_G(v_1) \\ &= \{x_2, x_3, \ldots, x_7\} \cap \{x_2, x_5\} \\ &= \{x_2, x_5\}. \end{aligned}$$

Mientras que S_1 y C_5 quedan de la siguiente manera:

$$S_1 = S_1 \setminus \{v_1\} = \{x_2, x_4, x_5, x_6, x_7\}$$
$$C_5 = C_5 \setminus \{v_1\} = \emptyset.$$

Será necesario actualizar k_1 debido a que $C_5 = \emptyset$, por lo que $k_1 = K_1 - 1 = 4$.

Y como $k_1 < k_1'$, entonces $k_1' = k_1' - 1 = 4$.

Por último aumentamos en uno a h:

$$h = h + 1 = 2.$$

$h = 2$

Cota inferior (Clique utilizando la subrutina 4)

Encontramos a un clique de $G(S_2)$ utilizando la subrutina 4, obteniendo $Q = \{x_5\}$ de cardinalidad igual a uno.

Hacemos la prueba para ver si podemos mejorar al clique actual M:

$$2 = h - 1 + |Q| \not> |M| = 2.$$

No se cumple, por lo que continuamos con el algoritmo encontrando una nueva cota superior.

Cota superior (Coloración utilizando la subrutina 2)

Encontramos una coloración de $G(S_2)$ obteniendo:

$$C = \{C_1 = \{x_2, x_5\}\}.$$

Por lo que la cota superior es igual a cardinalidad de C, $k_2 = 1$.

Haciendo la prueba para k_2 y $|M|$ tenemos que:

$$2 = h - 1 + k_2 \leq |M| = 2.$$

Como se cumple entonces hacemos el retroceso.

Retroceso

Como $h \neq 1$, entonces disminuimos a h en una unidad obteniendo

$$h = h - 1 = 1.$$

$h = 1$

Y hacemos la prueba para k_1' y $|M|$:

$$4 = h - 1 + k_1' \nleq |M| = 2.$$

Como esta última prueba no se cumple, entonces es necesario ramificar.

h=1

Ramificación

Para ramificar elegimos un vértice de grado máximo en la gráfica G, que pertenezca a la última clase de coloración y lo guardamos en v_1.

Ya que la última clase es $C_4 = \{x_6\}$ y debido a que solo tiene un vértice, entonces $v_1 = x_6$.

Ahora generamos el nuevo conjunto de vértices candidatos:

$$\begin{aligned} S_2 &= S_1 \cap N_G(v_1) \\ &= \{x_2, x_4, x_5, x_6, x_7\} \cap \{x_2, x_5, x_7\} \\ &= \{x_2, x_5, x_7\}. \end{aligned}$$

Mientras que S_1 y C_4 quedan de la siguiente manera:

$$S_1 = S_1 \setminus \{v_1\} = \{x_2, x_4, x_5, x_7\}$$
$$C_4 = C_4 \setminus \{v_1\} = \emptyset.$$

Será necesario actualizar k_1 debido a que $C_4 = \emptyset$, por lo que $k_1 = K_1 - 1 = 3$.

Y como $k_1 < k_1'$, entonces $k_1' = k_1' - 1 = 3$.

Por último aumentamos en uno a h:

$$h = h + 1 = 2.$$

$h = 2$

Cota inferior (Clique utilizando la subrutina 4)

Encontramos a un clique de $G(S_2)$ utilizando la subrutina 4, obteniendo $Q = \{x_5, x_7\}$, con cardinalidad igual a dos.

Hacemos la prueba para ver si podemos mejorar al clique actual M.

$$3 = h - 1 + |Q| > |M| = 2.$$

Al cumplirse la desigualdad, quiere decir que encontramos un clique de mayor cardinalidad, el cual es:

$$M = v_1 \cup Q = \{x_5, x_6, x_7\}.$$

Cota superior (Coloración utilizando la subrutina 2)

Encontramos una coloración de $G(S_2)$ obteniendo:

$$C = \{C_1 = \{x_7, x_2\}, C_2 = \{x_5\}\}.$$

Por lo que la cota superior es igual a cardinalidad de C, $k_2 = 2$.

Haciendo la prueba para k_2 y $|M|$ tenemos que:

$$3 = h - 1 + k_2 \leq |M| = 3.$$

Como se cumple entonces hacemos retroceso.

Retroceso

Como $h \neq 1$, entonces disminuimos a h en una unidad obteniendo

$$h = h - 1 = 1.$$

$h = 1$

Hacemos la prueba para k_1' y $|M|$:

$$3 = h - 1 + k_1' \leq |M| = 3.$$

Como se cumple y además $h = 1$, entonces $M = \{x_5, x_6, x_7\}$ es el clique máximo.

En las siguientes secciones se describirá a detalle cada una de las subrutinas que utiliza el algoritmo del clique máximo.

4.2. Algoritmos para las cotas inferiores

A continuación se describirá las subrutinas uno y cuatro, que se encargan de calcular cotas inferiores para $\omega(G)$. La primera subrutina solo será utilizada una vez por el algoritmo del clique máximo, se encargará de encontrar una subgráfica triangular de aristas maximal en el nodo raíz, y ocupará al algoritmo del clique máximo para calcular un clique máximo de la subgráfica generada, que será la cota inferior inicial en el nodo raíz del árbol de búsqueda. La subrutina cuatro calculará las cotas inferiores en los otros nodos.

4.2.1. Algoritmo para encontrar una subgráfica triangular de aristas maximal

El primer algoritmo que se utiliza para ayudar a determinar una cota inferior en el nodo raíz del árbol de búsqueda, es el algoritmo de Balas [5] para encontrar una subgráfica triangulada de aristas maximal, de ésta se obtendrá el clique máximo utilizando el algoritmo del máximo clique que será la cota inferior inicial, todo este procedimiento en el algoritmo general se presenta como la subrutina 1.

Antes de explicar el algoritmo, recordemos que una gráfica G es triangulada si todo ciclo C_n, con $n > 3$, posee una cuerda, esto es, una arista adyacente a dos vértices no consecutivos en un ciclo. Algunas de sus propiedades son: que si una gráfica es triangulada, entonces cada una de sus subgráficas inducidas también lo son. Un vértice es llamado simplicial si todos sus vecinos son adyacentes entre sí. Cada gráfica triangulada tiene un vértice simplicial; estó nos conduce a decir que una gráfica triangulada tiene al menos tantos cliques como vértices. Una ordenación $\sigma = (v_1, v_2, \ldots, v_n)$, donde $n = |V|$ los vértices de una gráfica $G = (V, E)$, es llamada de eliminación perfecta si para $i = 1, \ldots, n$, v_i es simplicial en $G(\{v_i, v_i + 1, \ldots, v_n\})$. Una gráfica es triangulada si y solo si existe una ordenación de eliminación perfecta de sus vértces. Para alguna ordenación $\sigma = (v_1, \ldots, v_n)$ de los vértices de G, decimos que v_j es un sucesor de v_i, si v_j y v_i son adyacentes con $j > i$. Si además, $k > j$ para todo sucesor v_k de v_i, entonces v_j es el

primer sucesor de v_i.

Además, se usan tres proposiciones establecidos por Rose, Tarjan y Lucker [70].

Proposición 1: Una gráfica es triangulada si y solo si el siguiente algoritmo produce una ordenación perfecta de sus vértices.

Algoritmo LEX P.: Asignar a cada vértice la etiqueta $\emptyset$. Para $i = n, n-1, \ldots, 1$, escoger lexicográficamente un vértice v no numerado con la etiqueta más grande (lexicográficamente), asignar a v el número i, y agregar i a la etiqueta de cada vértice no numerado adyacente a v.

Proposición 2: Una ordenación $\sigma = (v_1, \ldots, v_n)$ de vértices de una gráfica es de eliminación perfecta si y solo si para $i = 1, \ldots, n$, el primer sucesor de v_i es adyacente a todo sucesor de v_i.

Proposición 3: Dada una gráfica $G = (V, E)$ y un subconjunto $F \subset E$ tal que $G[F]$ [1] es triangulada, $G[F]$ es una subgráfica triangulada de aristas maximales si y solo sí no existe $e \in E \backslash F$ tal que $G[F \cup \{e\}]$ es triangulada.

Descripción del algoritmo para encontrar una subgráfica triangulada de aristas maximal

Para describir la manera en la que opera el algoritmo utilizaremos el digrama de flujo de la Figura 4.4.

Inicializaremos el algoritmo con una gráfica $G = (V, E)$ con $V = n$; un conjunto vacío F que guardará las aristas que formarán la subgráfica triangular de aristas maximal; σ que guardará la ordenación de eliminación perfecta de vértices, comienza vacío, en la primera vuelta del algoritmo va agregando un vértice que ocupará la posición n en la ordenación, para la segunda vuelta el siguiente vértice seleccionado

[1] En este caso con $G[F]$ nos referimos a las subgráfica donde el conjunto de vértices es un subconjuto de V y F un subconjunto de E.

ocupara la posición $n-1$ y asi sucesivamente hasta que todos esten ordenados. En este algoritmo cuando se diga que un vértice ocupa la posición j con $j \in \{1, 2, \ldots, n\}$ será equivalente a decir que el vértice está numerado y le corresponde el número j, por lo que los vértices que aún no tengan un lugar en σ serán los vértices no numerados. La variable de control es i y la inicializaremos con la cardinalidad de V, por lo que $i = n$, esta variable se disminuirá en una unidad al final de cada iteración del bucle. Todos los vértices estarán no marcados, esto quiere decir que en algún momento los vértices podrán tener una marca que puede ser cualquier elemento del conjunto $\{P, T, *\}$. Si un vértice tiene una marca igual a P, quiere decir que el vértice tiene asignado un número de forma permanente, esto quiere decir que la posición asignada al vértice en la ordenación perfecta ya no podrá ser modificada; si la marca es igual a T nos indica que el vértice tiene asignado un número solo durante esa iteración, antes de pasar a otra esta marca pasará a ser permanente (P) o bien quedará marcado con $*$; por último un vértice puede tener una marca igual a $*$, cuando esto sucede quiere decir que el vértice no está numerado en ese momento, pero en algún momento estuvo numerado de manera temporal. Todos los vértices tendrán una etiqueta que al inicio estará vacía, mientras el vértice permanezca no numerado, se guardará en esta etiqueta el número del vértice númerado adyacente. Denotaremos con [ ] a la etiqueta, dentro se guardarán los números de los vértices.

La primer condición del algoritmo es que i sea diferente de cero como se puede ver en la Figura 4.4, sino se cumple entonces el algoritmo devuelve F y σ, esto es el conjunto de aristas que forman la subgráfica de aristas maximal y la ordenación perfecta de vértices asociada a estos vértices. Si la condición se cumple, entonces se selecciona un vértice con la etiqueta más grande lexicográficamente.

Vamos a hacer un pequeño paréntesis en la explicación del algoritmo para aclarar el orden lexicográfico para cadenas de caracteres.

Un alfabeto es un conjunto de símbolos o caracteres finito y no vacío que denotaremos con Σ . Una cadena es una sucesión finita de símbolos de Σ, la cadena vacía ϵ es la cadena con cero ocurrencias de

símbolos del alfabeto. Σ^* es el conjunto de todas las cadenas de un alfabeto Σ.

La secuencia más corta a considerar será la cadena vacía ϵ; es decir, $\forall b \in \Sigma^*$ tal que $\epsilon \leq b$. Entonces la definición de ordenación lexicográfico en forma recursiva queda de la siguiente manera:

$$\forall [a_1 \dots a_n], [b_1 \dots b_m] \in \Sigma^* \setminus \{\epsilon\} \text{ tal que}$$

$$[a_1 \dots a_n] \leq [b_1 \dots b_m] \Leftrightarrow a_1 < b_1 \vee \Big(a_1 = b_1 \wedge [a_2 \dots a_n] \leq [b_2 \dots b_m]\Big).$$

Por ejemplo, si $a = [26]$ y $b = [219]$ tenemos que $b < a$ en orden lexicográfico, porque $a_1 = b_1 = 2$ y $b_2 = 1 < a_2 = 6$. Los diccionarios son uno de los ejemplos más conocidos de este orden. ■

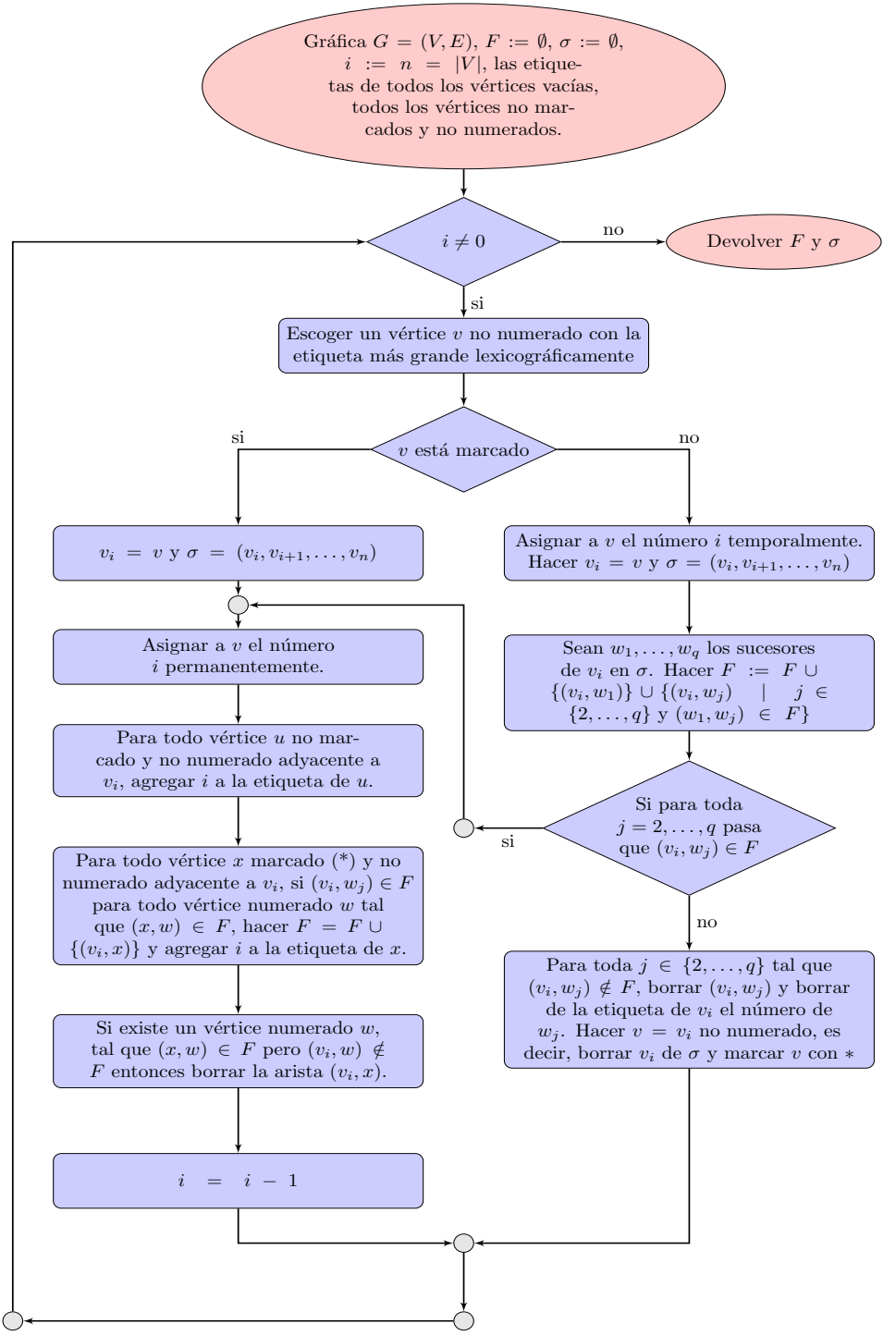


Figura 4.4: Diagrama de flujo del algoritmo para encontrar una subgráfica triangulada de aristas maximal.

Regresando con la explicación del algoritmo, suponiendo $i \leq 0$ y que tenemos asignados los números $n, n-1, \ldots, i+1$ en la iteración i el algoritmo escoge un vértice no numerado v con la etiqueta más grande de acuerdo al orden lexicográfico, considerando como alfabeto los números del 1 al $n = |V|$.

Si el vértice no está marcado entonces asignamos el número i de forma temporal y lo agregamos a la ordenación con lo que tenemos $\sigma = (v_i, v_{i+1}, \ldots, v_n)$. Consideremos que $w_1, \ldots, w_q$ son los sucesores de v_i en σ. Si el primer sucesor de v es adyacente a todos los sucesores, entonces a v se le asigna el número i permanentemente e i es agregado a la etiqueta de todo vértice adyacente no numerado a v_i. De otra forma v es borrado y la arista (v_i, w_j) tal que $(w_1, w_j) \notin F$, actualizamos la etiqueta de v_i borrando el número de w_j, marcamos v_i como no numerado, lo marcamos con $*$ y seleccionamos nuevamente un vértice. Todo esto para asegurar que cuando escojamos el vértice por segunda vez (ahora como un vértice marcado) se le asignará el número i de forma permanente, borramos las aristas incidentes a v_i no numerados, pero si marcados $(*)$ que tengan un sucesor no adyacente a v_i en $G[F]$. Por último disminuimos a i en una unidad.

Ejemplo 12. *Aplicaremos el algoritmo para encontrar una subgráfica de aristas maximal de Egon Balas a la gráfica de la Figura 4.5*

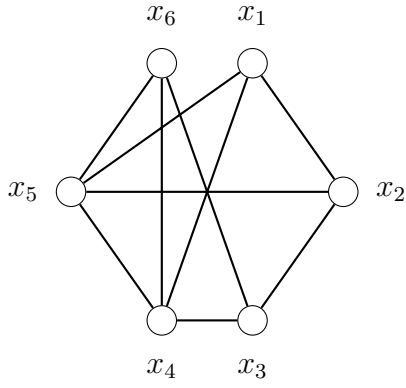


Figura 4.5: Gráfica del ejemplo 12.

Inicialización:

Gráfica $G = (V, E)$ como en la Figura 4.5. $F = \emptyset$, $\sigma = \emptyset$, todas las etiquetas de los vértices vacías y todos los vértices no marcados.

$i = 6$

Escogemos el vértice x_1 ya que no está marcado y le asignamos el número 6 temporalmente, entonces $v_6 = x_1$ y $\sigma = (x_1)$.

Como F está vacío, entonces el número v_6 se vuelve permanente.

Los vértices no marcados y no numerados adyacentes a v_6 son: x_2, x_4, x_5. Agregamos a las etiquetas de estos vértices el número i.

Toda esta información la podemos resumir en la siguiente tabla:

Marca	Vértice	Etiqueta
P	x_1	[ ]
	x_2	[6]
	x_3	[ ]
	x_4	[6]
	x_5	[6]
	x_6	[ ]

Por último disminuimos en una unidad a i:

$$i = i - 1 = 5.$$

$i = 5$

Escogemos el vértice x_2 que es no numerado y es uno de los vértices con la etiqueta más grande lexicográficamente. Como no está marcado, entonces le asignamos el número 5 temporalmente, por lo que $v_5 = x_2$ y $\sigma = (x_2, x_1)$.

El único sucesor de v_5 en σ es x_1, entonces $F = \{(x_2, x_1)\}$ y hacemos v_5 permanente.

Los vértices no marcados y no numerados adyacentes a v_5 son: x_3, x_5. Agregamos a las etiquetas de estos vértices el número i.

Toda esta información la podemos resumir en la siguiente tabla:

Marca	Vértice	Etiqueta
P	x_1	[ ]
P	x_2	[6]
	x_3	[5]
	x_4	[6]
	x_5	[6, 5]
	x_6	[ ]

Por último disminuimos en una unidad a i:

$$i = i - 1 = 4.$$

$i = 4$

Escogemos el vértice x_5 que es no numerado y es el vértice con la etiqueta más grande lexicográficamente. Como no está marcado, entonces le asignamos el número 4 temporalmente, por lo que $v_4 = x_5$ y $\sigma = (x_5, x_2, x_1)$.

Los sucesores de v_4 en σ son: x_2, x_1.

Entonces $F = \{(x_5, x_2), (x_5, x_1), (x_2, x_1)\}$ y hacemos v_4 permanente.

Los vértices no marcados y no numerados adyacentes a v_4 son: x_4, x_6. Agregamos a las etiquetas de estos vértices el número i.

Toda esta información la podemos resumir en la siguiente tabla:

Marca	Vértice	Etiqueta
P	x_1	[ ]
P	x_2	[6]
	x_3	[5]
	x_4	[6, 4]
P	x_5	[6, 5]
	x_6	[4]

Por último disminuimos en una unidad a i:

$$i = i - 1 = 3.$$

$i = 3$

Escogemos el vértice x_4 que es no numerado y es el vértice con la etiqueta más grande lexicográficamente. Como no está marcado, entonces le asignamos el número 3 temporalmente, por lo que $v_3 = x_4$ y $\sigma = (x_4, x_5, x_2, x_1)$.

Los sucesores de v_3 en σ son: x_5, x_1.

Entonces $F = \{(x_4, x_5), (x_4, x_1), (x_5, x_2), (x_5, x_1), (x_2, x_1)\}$ y hacemos v_3 permanente.

Los vértices no marcados y no numerados adyacentes a v_3 son: x_3, x_6. Agregamos a las etiquetas de estos vértices el número i.

Toda esta información la podemos resumir en la siguiente tabla:

Marca	Vértice	Etiqueta
P	x_1	[ ]
P	x_2	[6]
	x_3	[5, 3]
P	x_4	[6, 4]
P	x_5	[6, 5]
	x_6	[4, 3]

Por último disminuimos en una unidad a i:

$$i = i - 1 = 2.$$

$i = 2$

Escogemos el vértice x_3 que es no numerado y es el vértice con la etiqueta más grande lexicográficamente. Como no está marcado, entonces le asignamos el número 2 temporalmente, por lo que $v_2 = x_3$ y $\sigma = (x_3, x_4, x_5, x_2, x_1)$.

Los sucesores de v_2 en σ son: x_4, x_2.

Entonces $F = \{(x_3, x_4), (x_4, x_5), (x_4, x_1), (x_5, x_2), (x_5, x_1), (x_2, x_1)\}$.

Debido a que $(x_4, x_2) \notin F$ entonces no se pudo incluir (x_3, x_2) en F; es decir, $(x_3, x_2) \notin F$, ya que esto pasa entonces es necesario borrar (x_2, x_3) de las aristas de la gráfica.

Borramos de la etiqueta de $v_2 = x_3$ el número 5 que le corresponde a x_2, recordemos que $v_5 = x_2$.

Además borramos a v_2 de σ, por lo que x_3 ahora es no numerado, pero se queda marcado, para indicarlo utilizaremos *. Como se puede ver en la siguiente tabla:

Marca	Vértice	Etiqueta
P	x_1	[ ]
P	x_2	[6]
*	x_3	[3]
P	x_4	[6, 4]
P	x_5	[6, 5]
	x_6	[4, 3]

Y $\sigma = (x_4, x_5, x_2, x_1)$, mientras que $i = 2$ nuevamente.

$i = 2$

Escogemos el vértice x_6 que es no numerado y es el vértice con la etiqueta más grande lexicográficamente. Como no está marcado, entonces le asignamos el número 2 temporalmente, por lo que $v_2 = x_6$ y $\sigma = (x_6, x_4, x_5, x_2, x_1)$.

Los sucesores de v_2 en σ son: x_4, x_5.

Entonces $F = \{(x_6, x_4), (x_6, x_5), (x_3, x_4), (x_4, x_5), (x_4, x_1), (x_5, x_2), (x_5, x_1), (x_2,$

Marcamos a v_2 permanentemente.

Como no hay vértices no marcados y no numerados, entonces buscamos los vértices marcados, pero no numerados adyacentes a v_2, que este caso el único que cumple con las condiciones es x_3.

Como $(x_3, x_4) \in F$, x_4 está numerado y (x_6, x_4) está en F, entonces

$$\begin{aligned} F &= F \cup \{(x_6, x_3)\} \\ &= \{(x_6, x_3), (x_6, x_4), (x_6, x_5), (x_3, x_4), (x_4, x_5), (x_4, x_1), (x_5, x_2), (x_5, x_1), \\ &\quad (x_2, x_1)\}. \end{aligned}$$

Agregamos 2 a la etiqueta de x_3, como se puede ver en la tabla.

Marca	Vértice	Etiqueta
P	x_1	[ ]
P	x_2	[6]
*	x_3	[3, 2]
P	x_4	[6, 4]
P	x_5	[6, 5]
P	x_6	[4, 3]

Por último disminuimos en una unidad a i:

$$i = i - 1 = 1.$$

$i = 1$

Escogemos el vértices x_3, como está marcado le asignamos el número i permanentemente. Por lo que $v_1 = x_3$ y $\sigma = (x_3, x_6, x_4, x_5, x_2, x_1)$. Con esto terminamos el algoritmo.

La subgráfica de aristas maximal tiene las aristas $\{(x_6, x_3), (x_6, x_4), (x_6, x_5), (x_3, x_4), (x_4, x_5), (x_4, x_1), (x_5, x_2), (x_5, x_1), (x_2, x_1)\}$ como se puede ver en la Figura 4.6 y la ordenación de eliminación perfecta de vértices
$\sigma = (x_3, x_6, x_4, x_5, x_2, x_1)$. ■

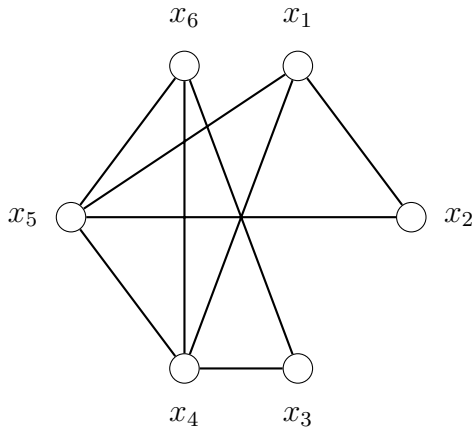


Figura 4.6: Subgráfica de aristas maximal $G[F]$

Pseudocódigo 1: Algoritmo para encontrar una subgráfica triangulada de aristas maximal

Paso *1*:

$G = (V, E)$. Sean $F \leftarrow \emptyset$, $\sigma \leftarrow \emptyset$, todos los vértices no marcados, no numerados y no etiquetados.

$i \leftarrow n = |V|$.

Ir al paso 2.

Paso *2*:

Si $i = 0$ **entonces**

parar.

Escoger un vértice v no numerado con la etiqueta más grande lexicográficamente.

Si v está marcado (no marcado) asignar a v permanentemente (temporalmente) el número i; es decir, $v_i \leftarrow v$, $\sigma \leftarrow (v_i, v_i + 1, \ldots, v_n)$.

Ir al paso 3.

Paso *3*:

Sean $w_1, \ldots, w_q$ los sucesores de v_i en σ. Entonces $F \leftarrow F \cup \{(v_i, w_1)\} \cup \{(v_i, w_j) | j \in \{2, \ldots, q\}$ y $(w_1, w_j) \in F\}$

Si *para todo* $j = 2, \ldots, q$ $(v_i, w_j) \in F$ **entonces**

hacer el número de v_i permanente e ir a paso 4.

de lo contrario

para toda $j \in \{2, \ldots, q\}$ tal que $(v_i, w_j) \notin F$, borrar (v_i, w_j) y borrar de la etiqueta de v_i el número de w_j. Hacer $v = v_i$ como no numerada; es decir, borrar v_i de σ. Marcar v e ir al paso 1.

Paso *4*:

Para todo *vértice no marcado y no numerado u adyacente a* v_i **hacer**

agregar i a la etiqueta de u.

Para todo *vértice no marcado y no numerado x adyacente a* v_i **hacer**

Si $(v_i, w) \in F$ *para todo vértice no numerado w tal que* $(x, w) \in F$ **entonces**

$F \leftarrow F \cup \{(v_i, x)\}$ y agregar i a la etiqueta de x.

Si *existe un vértice no numerado w tal que* $(x, w) \in F$ *y* $(v_i, w) \notin F$ **entonces**

borrar la arista (v_i, x).

Ir al paso 2.

Como se dijo al inicio de esta sección y como se demuestra en el artículo de Balas [5], este algoritmo genera una subgráfica triangulada de aristas maximal, además su complejidad es $O(\Delta|E|)$. Para presentar las demostraciones utilizaremos el pseudocódigo 1.

Proposición 37. *La subgráfica inducida por el conjunto de aristas F, $G[F]$, generada por el algoritmo para encontrar una subgráfica de aristas maximal es triangulada.*

Demostración. Veamos que todo vértice no numerado permanentemente v_i es simplicial en la subgráfica de $G[F]$ inducida por $v_i, v_{i+1}, \ldots, v_n$. Esto es verdadero para v_n.
Supongamos que esto es verdadero para $v_n, v_{n-1}, \ldots, v_{i+1}$ y tomemos el vértice v_i. Si el número de v_i fue hecho permanente en el paso 2, entonces todo sucesor w_j de v_i, con $j = 2, \ldots, q$, es adyacente en $G[F]$ a w_1, el primer sucesor de v_i; por el la proposición 2 y la hipótesis de inducción, v_i es simplicial en la subgráfica inducida por $\{v_i, \ldots, v_n\}$. Si v_i fue numerada permenentemente en el paso 2, entonces v_i fue marcado. Sus sucesores cuando el vértice fue marcado fueron encontrados en el paso 3 por ser adyacentes entre si en $G[F]$. Los vértices que llegaron a ser sucesores de v_i esto cuando ya se encontraba marcado se encontraron en el paso 4 que son adyacentes en $G[F]$ a todos los otros sucesores de v_i. En este caso tenemos que también v_i es simplicial en la subgráfica de $G[F]$ inducida por $v_i, \ldots, v_n$. □

Parar mostrar que una subgráfica triangulada $G[F]$ es de aristas maximal, es suficiente con tomar la proposición 3 para probar que para todo $e \in E \backslash F$, la gráfica $G[F \cup \{e\}]$ no es triangulada.

Proposición 38. *Para algún $e \in E \backslash F$, $G[F \cup \{e\}]$ no es triangulada.*

Demostración. Dado la proposición 1, $G[F \cup \{e\}]$ es triangulada si y solo si el algoritmo LEX P. produce una ordenación perfecta de V en $G[F \cup \{e\}]$. Aplicando LEX P a $G[F]$ produce la misma ordenación de V como el algoritmo para encontrar una subgráfica de aristas maximal a G, excepto para posibles diferencias al romper empates cuando escogemos entre etiquetas iguales. Suponiendo el uso de la misma regla para romper empates, independientemente del grado del vértice, las

dos ordenaciones son obviamente la misma. Ahora sea $e = (u, v)$ con el número en σ (la ordenación perfecta) de u más grande que v. Entonces si i fue el número asignado a v cuando este fue seleccionado antes (como un vértice no marcado), la sucesión $(v_{i+1}, \ldots, v_n)$ de σ es el misma que le corresponde a la ordenación producida por LEX P en $G[F]$, por lo tanto, en $G[F \cup \{e\}]$. Ya que $e = (u, v)$ fue borrado por el algoritmo, este segmento contiene a u y a algún vértice w adyacente a $v = v_i$, tal que $(u, w) \notin F$. Por lo que v_i no es simplicial en la subgráfica de $G[F \cup \{e\}]$ inducida por $(v_i, \ldots, v_n)$; es decir, la ordenación producida por LEX P en $G[F \cup \{e\}]$ no es perfecta. □

Por último se demostrará que el algoritmo para encontrar una subgráfica triangular de aristas maximal tiene una complejidad de $O(\Delta|E|)$.

Proposición 39. *El algoritmo tiene una complejidad de orden* $O(\Delta|E|)$.

Demostración. Notemos que en lugar de encontrar la etiqueta más grande cada vez que estamos en el paso 2, es más eficiente tener a los vértices no numerados ordenados lexicográficamente de acuerdo a sus etiquetas y actualizar esta ordenación cada vez que una etiqueta es cambiada. Por lo tanto, el paso 2 puede ser ejecutado en tiempo constante.

Mantener una ordenación lexicográfica de los vértices no numerados requiere dos tipos de actualizaciones: después de agregar un número al final de una etiqueta y después de borrar uno o varios números de una etiqueta. El primer tipo de actualización toma a lo más $d(v)$ comparaciones de entradas de la etiqueta, mientras el segundo tipo toma a lo más $\Delta(G)d(v)$ comparaciones de entradas de la etiqueta, para un vértice v.

Ya que un vértice es escogido en el paso 2 a lo más dos veces, el algoritmo itera $O(|V|)$ veces. Por lo tanto, el paso 2 requiere $O(|V|)$ veces a lo largo del algoritmo. El paso 3 requiere $O(d(v))$ veces para verificar si el primer sucesor de v es adyacente a todos los otros, y $O(\Delta(G)d(v))$ veces para ordenar lexicográficamente, simpre y cuando

alguna de las aristas haya sido borrada. Por lo tanto, el paso 3 toma $O(\Delta(G)\sum(d(v) : v \in V)) = O(\Delta(G)|E|)$ veces. Finalmente el paso 4 toma $O(\Delta(G)d(v))$ veces para revisar todos los vértices no marcados x, adyacentes a v, si $(u, w) \in F$ para todo w tal que $(x, w) \in F$, y de nuevo $O(\Delta(G)d(v))$ pasos para actualizar las etiquetas de todos los vecinos no numerados de v y actualizar la ordenación lexicográfica. Por lo tanto, el paso 4 toma $O(\Delta(G)|E|)$ veces. □

Con este algoritmo tenemos una parte de la subrutina uno, aún hace falta encontrar el clique máximo para la subgráfica encontrada, la cual será una cota inferior para $\omega(G)$ y clique inicial para el algoritmo.

4.2.2. Heurístico CLIQUE (Subrutina 4)

Para obtener cotas inferiores en otros nodos del árbol de búsqueda diferentes al nodo raíz, usaremos el siguiente heurístico simple para determinar un clique Q de una gráfica $G = (V, E)$. En el algoritmo del máximo clique es presentado como la subrutina cuatro.

El diagrama de flujo lo podemos ver en la Figura 4.7, este heurístico es de los conocidos como glotones o voraces, estos se caracterizan por elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. En este caso selecciona a los vértices con el grado más alto en cada paso. Nos referiremos al heurístico como CLIQUE, por ser como se nombra en la literatura, o bien como subrutina cuatro.

Descripción del heurístico:

Inicializa con una gráfica $G = (V, E)$; un conjunto S que guardará todos los vértices candidatos para formar el clique, al inicio $S = V$; otro conjunto Q que será donde se guardarán los vértices que formarán el clique, este conjunto estará vacío al inicio del algoritmo.

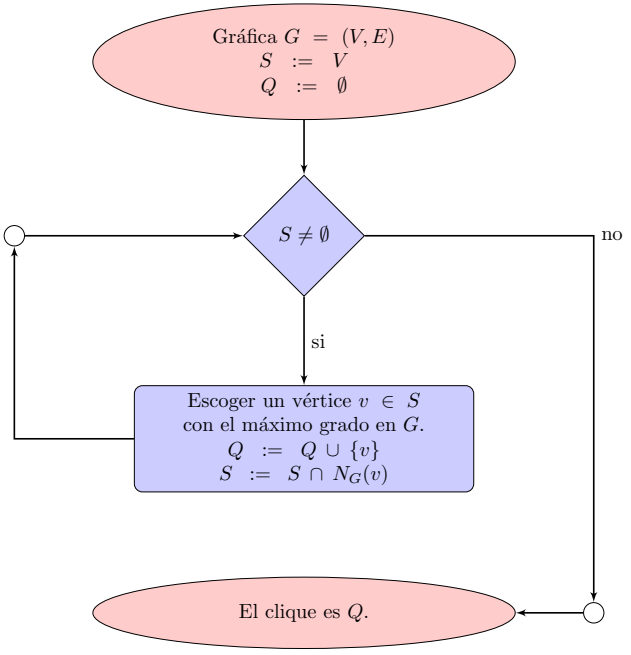


Figura 4.7: Diagrama de flujo del heurístico glotón CLIQUE

Mientras el conjunto S sea diferente del vacío, se seleccionará el vértice con el grado más alto en la gráfica y se incluirá en Q. El conjunto de candidatos S se actualizará, haciendo la intersección de S con los vecinos del vértice seleccionado. Con esto aseguramos que cuándo volvamos a seleccionar un vértice nuevo para estar en Q, éste será vecino de los vértices que estén en Q. El heurístico termina cuando $S = \emptyset$; es decir, cuando ya no tenemos más vértices que puedan ser vecinos a todos los elementos del clique.

Pseudocódigo 2: Heurístico CLIQUE (Subrutina cuatro)

Datos: $G = (V, E)$, $S := V$ y $Q := \emptyset$.
Resultado: Q.
Mientras $S \neq \emptyset$ **hacer**
    Escoger un vértice $v \in S$ con máximo grado en G.
    Hacer $Q := Q \cup \{v\}$ y $S := S \cap N_G(v)$.
Devolver Q

Ilustraremos el diagrama de flujo con el siguiente ejemplo.

Ejemplo 13. *Aplicaremos el algoritmo para encontrar un clique a la gráfica de la Figura 4.8.*

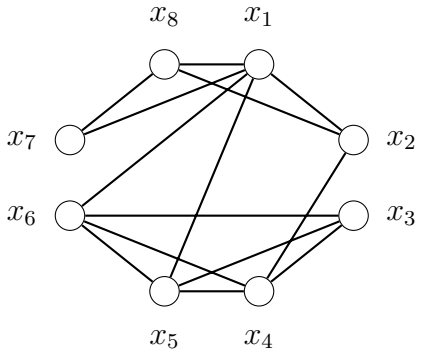


Figura 4.8: Gráfica del ejemplo para el algoritmo CLIQUE

Inicialización:

Gráfica $G = (V, E)$ como en la Figura 4.8. $S =$ Vértices candidatos $= V$, $Q = \emptyset$.

Ya que $S \neq \emptyset$ entonces hacemos la iteración 1.

Iteración 1

Escogemos al vértice x_1 ya que está en S y es uno de los que tiene el grado más grande en G. Hacemos $v = x_1$.

Y hacemos:

$$Q = Q \cup \{v\} = \{x_1\}$$
$$S = S \cap N_G(v) = \{x_2, x_5, x_6, x_7, x_8\}.$$

Ya que $S \neq \emptyset$ entonces hacemos la iteración 2.

Iteración 2

Escogemos al vértice x_5 ya que está en S y es uno de los que tiene el grado más grande en G. Hacemos $v = x_5$.

Hacemos:

$$Q = Q \cup \{v\} = \{x_1, x_5\}$$
$$S = S \cap N_G(v) = \{x_6\}.$$

Ya que $S \neq \emptyset$ entonces hacemos la iteración 3.

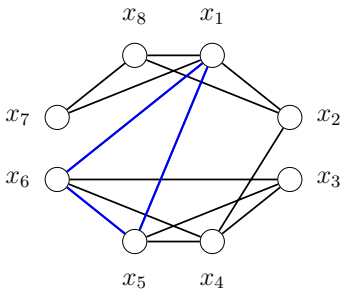
Iteración 3

Escogemos al vértice x_6 ya que es el único vértice en S. Hacemos $v = x_6$.

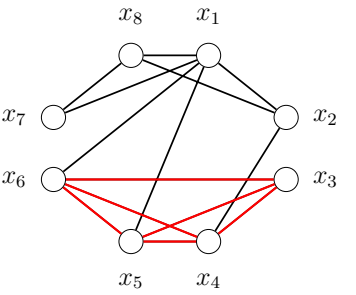
Y hacemos:

$$Q = Q \cup \{v\} = \{x_1, x_5, x_6\}$$
$$S = S \cap N_G(v) = \emptyset$$

Como S es vacío entonces termina el algoritmo. Un clique para la gráfica 4.8 está formado por los vértices $\{x_1, x_5, x_6\}$, sin embargo como podemos ver en la Figura 4.9 éste no es el óptimo, el clique máximo está formado por los vértices $\{x_3, x_4, x_5, x_6\}$. ■



(a) Clique encontrado por el heurístico en azul



(b) Clique máximo en rojo

Figura 4.9: Comparación del resultado del algoritmo heurístico y el óptimo

4.2.3. Subrutina 1

Ahora que ya tenemos todos los elementos para describir a la subrutina uno del algoritmo general. El pseudocódigo es el siguiente:

Pseudocódigo 3: Subrutina uno

Datos: $G = (V, E)$
Resultado: M un clique de G
Paso *1:*
  Utilizar el algoritmo para encontrar una subgráfica triangulada de aristas maximal, el cual nos devolverá un subconjuto de aristas F
Paso *2:*
  Con el conjunto de aristas F, formamos la subgráfica $SG = (V, F)$ donde V es el conjunto de aristas de la gráfica original y F es el conjunto de aristas que se obtuvo en el paso anterior.
Paso *3:*
  Aplicar el heurístico CLIQUE (Subrutina cuatro) a la subgráfica SG y el clique obtenido lo guardamos en Q.
Paso *4:*
  Aplicar el algoritmo del clique máximo (ACM) con Q como clique inicial. Al clique máximo encontrado lo guardamos en M.
Devolver M

En este caso se manda llamar al algoritmo del clique máximo (ACM) para encontrar un clique máximo en la subgráfica triangulada de aristas maximal. Recordemos que el máximo clique en una subgráfica G' de una gráfica G, es un clique de G. Además al ser triangulada es también una gráfica perfecta y por lo tanto, como se demostró en el capítulo anterior, la cardinalidad del clique máximo es igual al número cromático de G'.

Ejemplo 14. *Aplicaremos la subrutina uno a la gráfica de la Figura 4.5 del ejemplo 12.*

Paso 1: Se realizó en el ejemplo 12, donde al aplicar el algoritmo para encontrar una subgráfica triangular de aristas maximal nos devuelve un conjunto de aristas $F = \{(x_6, x_3), (x_6, x_4), (x_6, x_5), (x_3, x_4),$

$(x_4, x_5), (x_4, x_1), (x_5, x_2), (x_5, x_1), (x_2, x_1)\}$.

Paso 2: Ahora generamos la subgráfica $SG = (V, F)$ como se puede ver en la Figura 4.10.

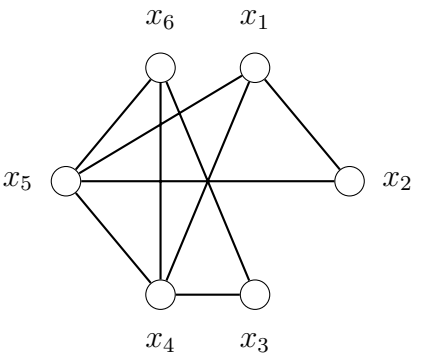


Figura 4.10: Subgráfica de aristas maximal SG.

Paso 3: A la subgráfica SG se le aplica el algoritmo CLIQUE (Subrutina cuatro) con lo que obtenemos $Q = \{x_1, x_4, x_5\}$.

Paso 3: Ahora mandamos llamar al algoritmo del clique máximo (ACM), al que le vamos a pasar como clique inicial a Q. No vamos a ejemplificar cada uno de los pasos porque aún no conocemos los detalles de todas las subrutinas, sin embargo es fácil verificar por inspección, que el algoritmo nos devolverá como clique máximo de la gráfica SG a $M = Q$.

En la Figura 4.11 se puede ver la gráfica original y el clique máximo para la subgráfica triangulada de aristas maximal.

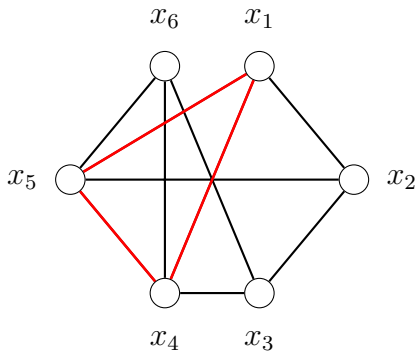


Figura 4.11: Gráfica original G y el clique M en rojo que será la cota inferior inicial.

Para el algoritmo general el clique M será la cota inferior inicial.

4.3. Heurísticos para las cotas superiores

Para encontrar una cota superior para el algoritmo, se utilizará el hecho de que la coloración de vértices en una gráfica es una cota para el tamaño del clique máximo. En el algoritmo del clique máximo se utilizarán dos heurísticos: el primero propuesto por Biggs [12] y el otro por Brelaz [16].

4.3.1. Heurístico COLOR (Subrutina 2)

El siguiente heurístico de coloración de vértices descrito en Biggs [12] como un método glotón o voraz *(greedy)* y nombrado como heurístico COLOR, asigna el primer color a todo vértice disponible, utilizando como regla de decisión del grado más alto; repite para el segundo color y así sucesivamente hasta que todos los vértices esten coloreados. En el algoritmo general este procedimiento corresponde a la subrutina dos, sin embargo tendremos dos opciones de heurísticos para hacerla, uno de ellos es COLOR y el otro es DSATUR que será presentado en la

siguiente sección.

En la Figura 4.12 se presenta el diagrama de flujo para el heurístico COLOR.

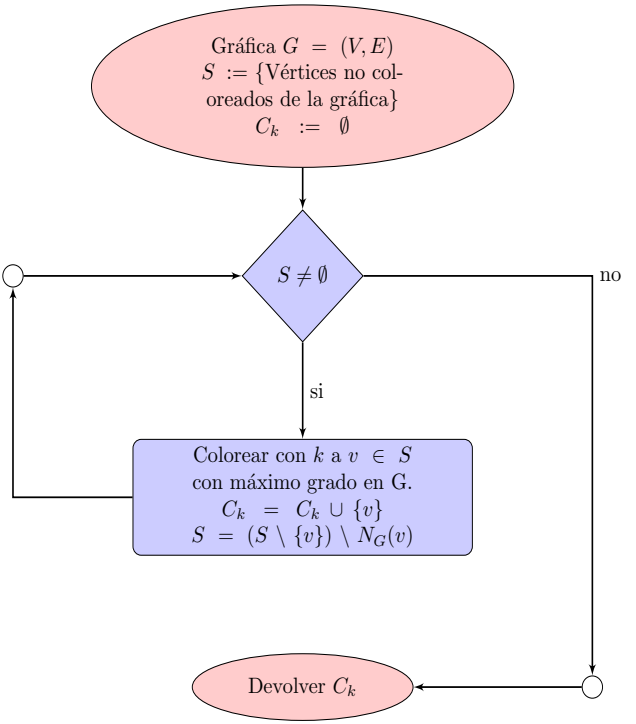


Figura 4.12: Diagrama de flujo del heurístico COLOR (Subrutina 2)

Descrpción del heurístico COLOR (Subrutina 2)

El heurístico comienza con una gráfica $G = (V, E)$, un conjunto S que guardará todos los vértices que no tengan asignado un color; una clase de color vacía C_k, en ella se irán guardando los vértices a los que se les asigne el color k.

Mientras S sea diferente del vacío, se seleccionará al vértice con el grado más alto en G y se agregará a la clase C_k. El conjunto de candidatos se actualizará eliminando el vértice seleccionado y todos

los vecinos a éste. De esta forma se asegura que para la siguiente iteración, los candidatos no son vecinos de ningún vértice en C_k, por lo que pueden ser seleccionados para formar parte de la clase y obtener una coloración propia. Una vez que S sea igual al vacío el algoritmo termina y devuelve la clase k-ésima; es decir,los vértices que fueron coloreados con el color k.

Como se dijo anteriormente y como se puede observar en el diagrama de flujo de la Figura 4.12, el proceso se hace para una clase de color C_k, si queremos colorear de manera propia una gráfica comenzaremos con $k = 1$ con lo que generamos la primer clase de color C_1. Una vez finalizado, actualizamos al conjunto S agregando todos los vértices de V que no están incluidos en C_1 y volvemos a seguir el algoritmo para crear C_2. Nuevamente se actualiza S, incluyendo todos los vértices que no estan en C_1 o C_2 y así sucesivamente hasta que todos los vértices estén en alguna clase de coloración.

Pseudocódigo 4: Heurístico COLOR

Datos: $G = (V, E), C_k = \emptyset, S := \{v :$
v es un vértice no coloreado$\}$.
Resultado: C_k
Mientras $S \neq \emptyset$ **hacer**
  Asignar el color k al vértice $v \in S$ con máximo grado en G.
  Hacer $C_k := C_k \cup \{v\}$ y $S := (S \setminus \{v\}) \setminus N_G(v)$.
Devolver C_k

Ejemplo 15. *Ahora aplicaremos el heurístico para encontrar una coloración de la gráfica de la Figura 4.13.*

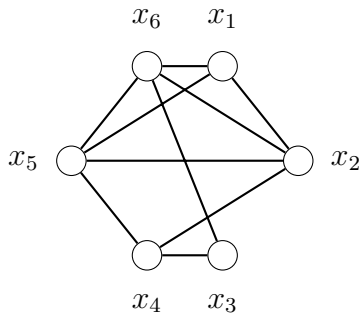


Figura 4.13: Gráfica del ejemplo para el algoritmo COLOR

Inicialización:

Sea la $G = (V, E)$ como en la Figura 4.8, con $V = \{x_1, x_2, \ldots, x_6\}$ y $E = \{(x_1, x_2), (x_1, x_5), (x_1, x_6), (x_2, x_4), (x_2, x_5), (x_2, x_6), (x_3, x_4), (x_3, x_6), (x_4, x_5), (x_5, x_6)\}$.

$S =$ Vértices no coloreados $= V$.

Como $S \neq \emptyset$, entonces $k = 1$ y $C_k = C_1 = \emptyset$.

$k = 1$

Iteración 1

Escogemos al vértice x_2 ya que está en S y es uno de los que tiene el grado más alto en G. Así que $v = x_2$.

Y hacemos:

$$\begin{aligned} C_1 &= C_1 \cup \{v\} = \{x_2\} \\ S &= (S \setminus \{v\}) \setminus N_G(v) \\ &= \{x_1, x_3, x_4, x_5, x_6\} \setminus \{x_4, x_5, x_6, x_1\} \\ &= \{x_3\} \end{aligned}$$

Como $S \neq \emptyset$ entonces hacemos otra iteración.

Iteración 2

Escogemos al vértice x_3 ya que es el único vértice de S. Así que $v = x_3$.

Y hacemos:

$$\begin{aligned} C_1 &= C_1 \cup \{v\} = \{x_2, x_3\} \\ S &= (S \setminus \{v\}) \setminus N_G(v) \\ &= \emptyset. \end{aligned}$$

Como S es un conjunto vacío. Entonces volvemos a calcular S, con los vértices que aún no están en ninguna clase de coloración. Así que:

$$S = V \setminus C_1 = \{x_1, x_4, x_5, x_6\}.$$

Como $S \neq \emptyset$ entonces:

$$\begin{aligned} k &= k + 1 = 2 \\ C_k &= C_2 = \emptyset. \end{aligned}$$

k=2

Iteración 1

Escogemos al vértice x_5 ya que está en S y tiene el grado más alto en G. Así que $v = x_5$.

Hacemos:

$$\begin{aligned} C_2 &= C_2 \cup \{v\} = \{x_5\} \\ S &= (S \setminus \{v\}) \setminus N_G(v) \\ &= \{x_1, x_4, x_6\} \setminus \{x_1, x_2, x_4, x_6\} \\ &= \emptyset. \end{aligned}$$

Como S es un conjunto vacío. Entonces volvemos a calcular S, con los vértices que aún no están en ninguna clase de coloración. Así que:

$$S = V \setminus (C_1 \cup C_2) = \{x_1, x_4, x_6\}.$$

Como $S \neq \emptyset$ entonces:

$$\begin{aligned} k &= k + 1 = 3 \\ C_k &= C_3 = \emptyset. \end{aligned}$$

$k = 3$

Iteración 1

Escogemos al vértice x_6 ya que está en S y tiene el grado más alto en G. Así que $v = x_6$.

Y hacemos:

$$\begin{aligned} C_3 &= C_3 \cup \{v\} = \{x_6\} \\ S &= (S \setminus \{v\}) \setminus N_G(v) \\ &= \{x_1, x_4\} \setminus \{x_1, x_2, x_3, x_5\} \\ &= \{x_4\}. \end{aligned}$$

Como $S \neq \emptyset$ entonces hacemos otra iteración.

Iteración 2

Escogemos al vértice x_4 ya que es el único vértice que tenemos en S. Así que $v = x_4$.

Y hacemos:

$$\begin{aligned} C_3 &= C_3 \cup \{v\} = \{x_4, x_6\} \\ S &= (S \setminus \{v\}) \setminus N_G(v) \\ &= \emptyset. \end{aligned}$$

Como S es un conjunto vacío. Entonces volvemos a calcular S, con los vértices que aún no están en ninguna clase de coloración. Así que:

$$S = V \setminus (C_1 \cup C_2 \cup C_3) = \{x_1\}.$$

Como $S \neq \emptyset$ entonces:

$$\begin{aligned} k &= k + 1 = 4 \\ C_k &= C_3 = \emptyset. \end{aligned}$$

$k = 4$

Iteración 1

Escogemos al vértice x_1 ya que es el único vértice que tenemos en S. Así que $v = x_1$.

Hacemos:

$$\begin{aligned} C_4 &= C_4 \cup \{v\} = \{x_1\} \\ S &= (S \setminus \{v\}) \setminus N_G(v) \\ &= \emptyset. \end{aligned}$$

Como S es igual al conjunto vacío y $S = V \setminus (C_1 \cup C_2 \cup C_3 \cup C_4) = \emptyset$; es decir, ya todos los vértices están en alguna clase de color. Entonces el algoritmo termina.

Y la coloración es igual a $C = \{C_1 = \{x_2, x_3\}, C_2 = \{x_5\}, C_3 = \{x_4, x_6\}, C_4 = \{x_1\}\}$. Si a la clase C_1 le asignamos el color azul, a C_2 el rojo, a C_3 el amarillo y a C_4 el verde, obtenemos la gráfica G coloreada propiamente como se puede ver en la Figura 4.14.

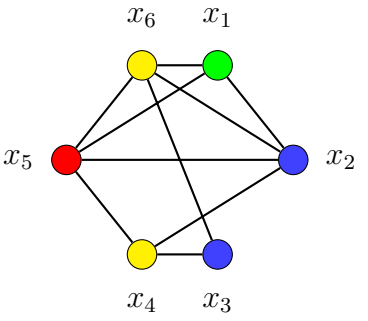


Figura 4.14: Gráfica coloreada con el heurístico COLOR

4.3.2. Heurístico DSATUR (Subrutina 2)

Otro algoritmo que también será usado para colorear una gráfica es el heurístico de Brelaz [16], este algoritmo se conoce como DSATUR y utiliza el grado de saturación de los vértices de una gráfica. El grado de saturación de un vértice $v$es el número de colores diferentes usados en los vecinos de v.

Una de las aplicaciones de la coloración de vértices es resolver el problema de asignación, para el que DSATUR ha demostrado ser exacto para colorear gráficas bipartitas [16]. Para los objetivos del algortimo del máximo clique lo utilizaremos para obtener una cota superior para $\chi(G)$, o lo que es lo mismo, una cota superior para $\omega(G)$.

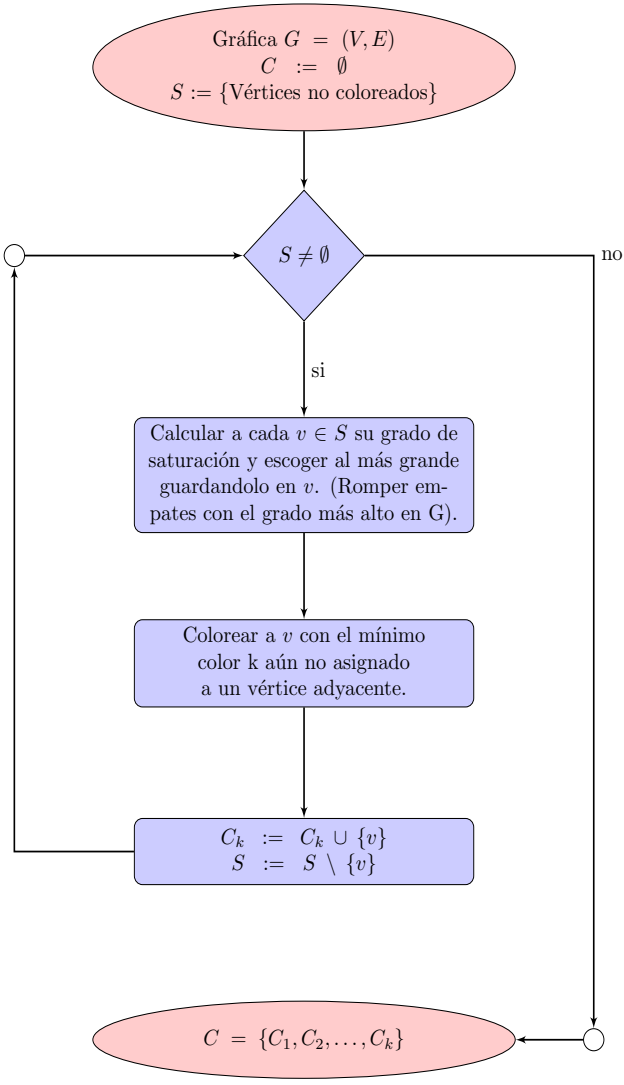


Figura 4.15: Diagrama de flujo del heurístico DSATUR

Descripción del heurístico DSATUR

Se inicializa con una gráfica $G = (V, E)$, un conjunto C vacío que guardará cada clase de coloración generada y un conjunto S que contendrá los vértices no coloreados al comienzo será igual a V.

Mientras $S \neq \emptyset$ seleccionar el vértice con el grado de saturación

más grande; es decir, el vértice que tenga más vecinos coloreados con diferente color. Los empates se rompen con el grado más alto en G. A este vértice se le asigna el color k más pequeño aún no asignado a ningún vértice adyacente. Por último se actualiza el conjunto S eliminando al vértice de éste. Y así sucesivamente hasta que todos los vértices de la gráfica estén coloreados.

El pseudocódigo del heurístico DSATUR se presenta acontinuación.

Pseudocódigo 5: Heurístico DSATUR

Datos: $G = (V, E), C = \emptyset, S := \{v :$
v es un vértice no coloreado$\}$.
Resultado: $C = (C_1, C_2, \ldots, C_k)$
Mientras $S \neq \emptyset$ **hacer**
  Escoger un vértice $v \in S$ con máximo grado de saturación en G rompiendo empates con el grado más alto.
  Colorear a v con el mínimo color i no asignado a un vértice adyacente; es decir, $C_i := C_i \cup \{v\}$.
  $S := S \setminus \{v\}$.
Devolver C

Veamos que este algoritmo colorea las componentes conexas de G en turno, esto debido a que escogemos entre los vértices que ya tienen vecinos coloreados, pero al seleccionar el que tenga el grado de saturación más grande, obtenemos que los vértices elegidos formen un clique. Por lo tanto, obtenemos una cota inferior para el número cromático ($\chi(G)$) y nos da también una cota inferior para $\omega(G)$, si es que somos capaces de obtener la información del tamaño de los cliques formados. Para el algoritmo del clique máximo se utilizara la cota inferior de $\chi(G)$ proporcionada por el héurístico, como una cota superior para $\omega(G)$.

Ejemplo 16. *Ahora aplicaremos el algoritmo para encontrar una coloración en la gráfica de la Figura 4.13.*

Inicialización:

Sea la gráfica $G = (V, E)$ como en la Figura 4.13, con $V = \{x_1, x_2, \ldots, x_6\}$ y $E = \{(x_1, x_2), (x_1, x_5), (x_1, x_6), (x_2, x_4), (x_2, x_5), (x_2, x_6), (x_3, x_4), (x_3, x_6), (x_4, x_5), (x_5, x_6)\}$.

Se denotará con $sd(v)$ el grado de saturación del vértice v, $sd(x_i) = 0$ para $i \in \{1, 2, 3, 4, 5, 6\}$.

$S =$ Vértices no coloreados $= V$.

$C_1 = \emptyset$.

Como $S \neq \emptyset$, entonces $k = 1$ y $C_k = C_1 = \emptyset$.
Iteración 1

Como el grado de saturación de todos los vértices en S es igual a cero, entonces elegimos al que tenga el grado más alto en G, en este caso es x_2, así que $v = x_2$.

Seleccionamos la primer clase de coloración ya que está vacía y por lo tanto, no contiene vecinos de v, hacemos:

$$C_1 = C_1 \cup \{v\} = \{x_2\}$$
$$S = (S \setminus \{v\}) = \{x_1, x_3, x_4, x_5, x_6\}.$$

Ahora a los vecinos de v aumentamos en uno su grado de saturación.

$$sd(x_1) = sd(x_4) = sd(x_5) = sd(x_6) = 1.$$

Ya que $S \neq \emptyset$ hacemos otra iteración.
Iteración 2

Veamos que x_1, x_4, x_5, x_6 en S tienen los grados de saturación más altos, por lo que romperemos empates con el grado más alto entre ellos. Como $d(x_1) = 3$, $d(x_4) = 3$, $d(x_5) = 4$ y $d(x_6) = 4$, como x_5 y x_6 tienen

el mismo grado y es el más alto, entonces podemos elegir a cualquiera de los dos, en este caso tomamos a x_5, por lo que $v = x_5$.

Ahora como la única clase de coloración que existe es C_1, pero esta contiene un vecino de v, entonces creamos una nueva clase que la contenga.

$$C_2 = \{x_5\}.$$

Hacemos

$$S = S \setminus \{v\} = \{x_1, x_3, x_4, x_6\}.$$

Y actualizamos los grados de saturación de los vecinos de v.

$sd(x_1) = 2$, $sd(x_2) = 1$, $sd(x_4) = 2$ y $sd(x_6) = 2$.

Ya que $S \neq \emptyset$ hacemos otra iteración.

Iteración 3

x_1, x_4 y x_6 en S tienen el grado de saturación más alto, como tenemos empate entonces seleccionamos al que tenga el grado más alto, en este caso es x_6. Así que $v = x_6$.

Como C_1 y C_2 tienen vecinos de v, entonces creamos una nueva clase de coloración que contenga a v.

$$C_3 = \{x_6\}.$$

Hacemos:

$$S = S \setminus \{v\} = \{x_1, x_3, x_4\}.$$

A los vecinos de v aumentamos en uno su grado de saturación.

$sd(x_1) = 3$, $sd(x_2) = 2$, $sd(x_3) = 1$ y $sd(x_5) = 2$.

Ya que $S \neq \emptyset$ hacemos otra iteración.

Iteración 4

x_1 en S tienen el grado de saturación más alto, entonces $v = x_1$.

Como C_1, C_2 y C_3 tienen vecinos de v, entonces creamos una nueva clase de coloración que contenga a v.

$$C_4 = \{x_1\}.$$

Y hacemos:

$$S = S \setminus \{v\} = \{x_3, x_4\}.$$

A los vecinos de v aumentamos en uno su grado de saturación.

$sd(x_2) = 3$, $sd(x_5) = 3$ y $sd(x_6) = 3$.

Ya que $S \neq \emptyset$ hacemos otra iteración.

Iteración 5

x_4 en S tienen el grado de saturación más alto, entonces $v = x_4$.

La clase C_3 es la primera clase en la que podemos incluir a v, ya que no contiene vecinos de este. Entonces:

$$C_3 = C_3 \cup \{v\} = \{x_4, x_6\}.$$

Y hacemos:

$$S = S \setminus \{v\} = \{x_3\}.$$

A los vecinos de v aumentamos en uno su grado de saturación.

$sd(x_2) = 4$, $sd(x_3) = 2$ y $sd(x_5) = 4$.

Ya que $S \neq \emptyset$ hacemos otra iteración.

Iteración 6

x_3 es el único vértice en S, entonces $v = x_3$.

Como C_1 no tienen vecinos de v, entonces:

$$C_1 = C_1 \cup \{v\} = \{x_2, x_3\}.$$

Y hacemos:

$$S = S \setminus \{v\} = \emptyset.$$

Como S es igual al conjunto vacío, entonces termina el algoritmo. La coloración de vértices es igual a

$$C = \{C_1 = \{x_2, x_3\}, C_2 = \{x_5\}, C_3 = \{x_4, x_6\}, C_4 = \{x_1\}, \}.$$

Si a la clase C_1 le asignamos el color azul, al C_2 el rojo, al C_3 el amarillo y al C_4 el verde, la gráfica queda como en la Figura 4.16.

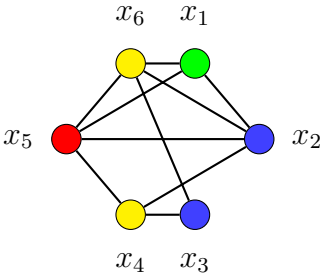


Figura 4.16: Gráfica coloreada con el algoritmo DSATUR

4.3.3. Heurístico para encontrar una coloración fraccional FCP (Subrutina 3)

Ahora presentaremos el siguiente heurístico para el problema de coloración fraccional, llamado procedimiento de coloración fraccional (FCP) de Egon Balas y Jue Xue [6], el cual determina una cota superior para el problema del clique máximo. En el algoritmo general se muestra como subrutina tres.

Recordemos que una coloración fraccional de una gráfica $G = (V, E)$ hace una asignación de un conjunto de colores a cada vértice, de tal manera que vértices adyacentes tienen asignados conjuntos ajenos. A cada color se le asignará un peso mayor que cero, de tal manera que la suma de ellos en cada vértice sea al menos uno. El peso de la coloración fraccional es la suma de los pesos de los colores. El número cromático fraccional de una gráfica, $\chi_f(G)$, es el mínimo peso posible tal que exista una coloración fraccional propia. Sin embargo el problema de hallar $\chi_f(G)$, al igual que el del clique máximo y el de la coloración mínima, son problemas dentro de la clase NP-Díficil.

El algoritmo FCP colorea cada vértice i veces y a cada clase de color se le asigna un peso de $t_i = |C_i|/i$, por lo que el heurístico está haciendo una coloración fraccional regular de vértices; es decir, ya que la suma de los pesos de cada vértice es igual a uno.

Descripción de FCP

El heurístico comienza con una gráfica $G = (V, E)$; un conjunto C vacío que guardará las clases de coloración; $i = 1$ variable para controlar las iteraciones; t_i el peso de la coloración obtenida en la iteración i; t_0 que es igual al infinito sirve para tener con que comparar en la iteración uno; U el conjunto de vértices no incluidos en ninguna clase de coloración.

En la iteración i para cada vértice en V, incluiriemos a v en la primer clase de coloración $C_j \in C$ tal que $C_j \cup \{v\}$ es un conjunto independiente, en caso de que no exista tal clase entonces lo agregaremos

en U. Después encontraremos una coloración de vértices $C_1, C_2, \ldots, C_k$ de $G(U)$ utilizando COLOR o DSATUR y $C = C \cup \{C_1, C_2, \ldots, C_k\}$. El peso de la coloración fraccional encontrada será igual a $t_i = |C|/i$.

El procedimiento continuará hasta que $t_i > t_{i-1}$ o $|C| > |V|$ y regresará $\lfloor \text{mín}(t_i, t_{i-1}) \rfloor$.

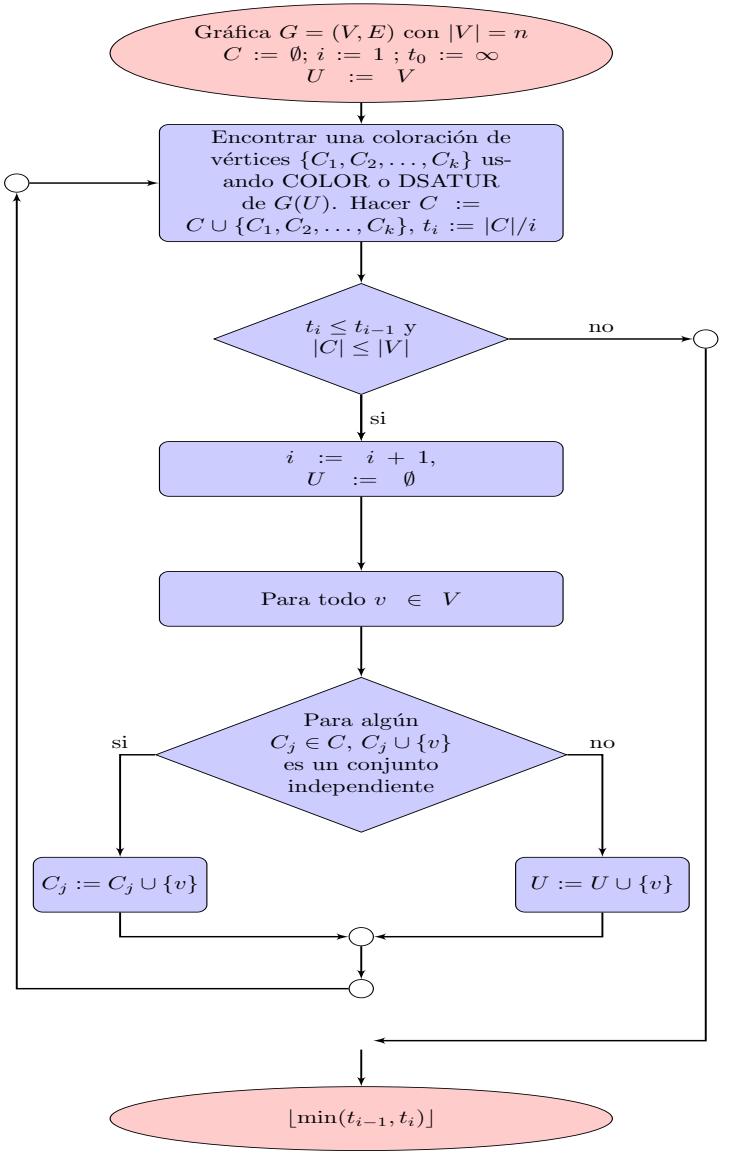


Figura 4.17: Diagrama de flujo del procedimiento para coloración fraccional FCP (Subrutina 3)

Pseudocódigo 6: Procedimiento de coloración fraccional (FCP)

Datos: $G = (V, E)$, $C := \emptyset$, $i := 1$ y $t_0 := \infty$.
Resultado: Una coloración fraccional igual a $\lfloor \text{mín}(t_{i-1}, t_i) \rfloor$.
Mientras $t_i \leq t_{i-1}$ *o* $|C| \leq |V|$ **hacer**
  $U := \emptyset$.
  Para todo $v \in V$ **hacer**
    Si *Para algún* $C_j \in C$, $C_j \cup \{v\}$ *es un conjunto independiente* **entonces**
      $C_j := C_j \cup \{v\}$
    de lo contrario
      Hacer $U := U \cup \{v\}$, que es el conjunto de vértices no incluidos en alguna clase de coloración.
  Encontrar una coloración de vértices $(C_1, C_2, \ldots, C_k)$de $G(U)$ usando COLOR o DSATUR. Hacer $C := C \cup \{C_1, C_2, \ldots, C_k\}$ y $t_i := |C|/i$.
Devolver $\lfloor \text{mín}(t_{i-1}, t_i) \rfloor$.

Ejemplo 17. *Ahora aplicaremos el heurístico FCP para encontrar la cardinalidad de una coloración fraccional en la gráfica de la Figura 4.18.*

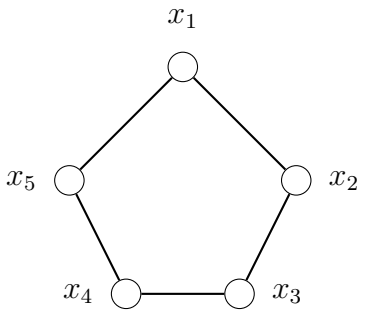


Figura 4.18: Gráfica para el ejemplo del algoritmo FCP

Inicialización:

Sea la gráfica $G = (V, E)$ como en la Figura 4.18, con $V = \{x_1, x_2, \ldots, x_5\}$ y $E = \{(x_1, x_2), (x_1, x_5), (x_2, x_3), (x_3, x_4), (x_4, x_5)\}$.

$C = \emptyset$.

$t_0 = \infty$.

$U =$ Vértices no coloreados $= V$.

$i = 1$.

$i = 1$

Encontramos una coloración de $G(U)$ utilizando COLOR o DSATUR [2], obteniendo la siguiente coloración:

$$\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\}$$

Y hacemos:

$$C = C \cup \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\} = \{C_1, C_2, \ldots, C_5\}$$
$$t_1 = \frac{|C|}{i} = 5.$$

Como $t_1 < t_0$ entonces

$$i = i + 1 = 2$$
$$U = \emptyset.$$

[2]Las coloraciones propias a lo largo de este ejemplo, no fueron obtenidas con COLOR o DSATUR sino fueron realizados de tal manera que FCP pase por todos sus pasos.

$i = 2$

Para cada vértice en V veamos si puede ser incluido en alguna clase de coloración.

Para x_1

Se puede incluir en la clase C_3 ya que $C_3 \cup \{x_1\}$ es un conjunto independiente.

Entonces

$$C_3 = \{x_1, x_3\}$$

Para x_2

Se puede incluir en la clase C_4 ya que $C_4 \cup \{x_2\}$ es un conjunto independiente.

Entonces

$$C_4 = \{x_2, x_4\}.$$

Para x_3

Se puede incluir en la clase C_1 ya que $C_1 \cup \{x_3\}$ es un conjunto independiente.

Entonces

$$C_1 = \{x_1, x_3\}.$$

Para x_4

Se puede incluir en la clase C_2 ya que $C_2 \cup \{x_4\}$ es un conjunto independiente.

Entonces

$$C_2 = \{x_2, x_4\}.$$

Para x_5

No se puede incluir en ninguna clase de coloración.

Entonces

$$U = U \cup \{x_5\} = \{x_5\}.$$

Encontramos una coloración de $G(U)$ utilizando COLOR o DSATUR, obteniendo la siguiente coloración:

$$\{\{x_5\}\}.$$

Y hacemos:

$$\begin{aligned} C &= C \cup \{\{x_5\}\} \\ &= \{C_1, C_2, \ldots, C_6\} \\ t_2 &= \frac{|C|}{i} = 6/2 = 3. \end{aligned}$$

Como $t_2 < t_1$ y $|C| \leq V$ entonces

$$\begin{aligned} i &= i + 1 = 3 \\ U &= \emptyset. \end{aligned}$$

$i = 3$

Para cada vértice en V veamos si puede ser incluido en alguna clase de coloración.

Para x_1

No se puede incluir en ninguna clase de coloración.

Entonces

$$U = U \cup \{x_1\} = \{x_1\}.$$

Para x_2

Se puede incluir en la clase C_5 ya que $C_5 \cup \{x_2\}$ es un conjunto independiente.

Entonces

$$C_5 = \{x_2, x_5\}.$$

Para x_3

Se puede incluir en la clase C_6 ya que $C_6 \cup \{x_3\}$ es un conjunto independiente.

Entonces

$$C_6 = \{x_3, x_5\}.$$

Para x_4

No se puede incluir en ninguna clase de coloración.

Entonces

$$U = U \cup \{x_4\} = \{x_1, x_4\}.$$

Para x_5

No se puede incluir en ninguna clase de coloración.

Entonces

$$U = U \cup \{x_5\} = \{x_1, x_4, x_5\}.$$

Encontramos una coloración de $G(U)$ utilizando COLOR o DSATUR, obteniendo la siguiente coloración:

$$\{\{x_5\}, \{x_1, x_4\}\}.$$

Y hacemos:

$$\begin{aligned} C &= C \cup \{\{x_5\}, \{x_1, x_4\}\} \\ &= \{\{x_1, x_3\}, \{x_2, x_4\}, \{x_1, x_3\}, \{x_2, x_4\}, \\ &\quad \{x_2, x_5\}, \{x_3, x_5\}, \{x_5\}, \{x_1, x_4\}\} \\ &= \{C_1, C_2, \ldots C_8\} \\ t_3 &= \frac{|C|}{i} = 8/3. \end{aligned}$$

Tenemos que $t_3 < t_2$ pero $|C| \leq V$ entonces

$$\lfloor 8/3 \rfloor = 2.$$

Cada clase de color tiene un peso de $1/i = 1/3$ y debido a que cada vértice es coloreado $i = 3$ veces, entonces la coloración es regular.

Por lo tanto, una cota superior para $\omega(G)$ es 2.

Esta coloración fraccional puede ser representada como se puede ver en la Figura 4.19, si a la clase C_1 le assignamos el color amarillo, a C_2 el rojo, a C_3 el azul, a C_4 el verde, a C_5 el morado, a C_6 el naranja, a C_7 el gris y por último a C_8 el rosa.

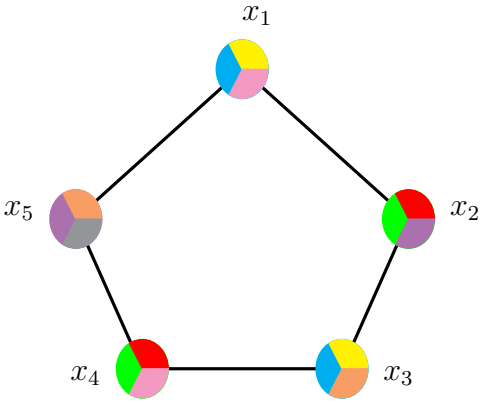


Figura 4.19

Se utilizará FCP_C y FCP_D para indicar que el procedimiento de coloración fraccional utilizará COLOR o DSATUR respectivamente.

4.4. Algoritmo del clique máximo

Ahora presentaremos el algoritmo del clique máximo (AMC) que utiliza el método ramificación y acotamiento de David R. Wood [78]. Como mencionamos al inicio de este capítulo el algoritmo mantiene las siguientes condiciones:

- Si h es la profundidad del árbol de búsqueda, el conjunto $\{v_1, v_2, \ldots, v_{h-1}\}$ consiste de vértices adyacentes entre sí.
- M es el clique más grande encontrado por el algoritmo y su cardinalidad es la cota inferior para $\omega(G)$; $1 \leq |M| \leq \omega(G)$.
- Para $1 \leq i \leq h$, el conjunto de vértices $S_i \subseteq \bigcap_{j=1}^{i-1} N_G(v_j)$ contiene candidatos para hacer más grande $\{v_1, v_2, \ldots, v_{i-1}\}$.
- Para $1 \leq i \leq h$, $(C_1^i, C_2^i, \ldots, C_k^i)$ es una coloración de vértices de $G(S_i)$, con cardinalidad k_i. Mientras que la cardinalidad de la coloración fraccional de $G(S_i)$ es k_i'. Ambas son cotas superiores para $\omega(G(S_i))$, con $k_i' \leq k_i$.
- Un nodo activo del árbol de búsqueda corresponde al subproblema de encontrar un clique más grande que M de la subgráfica:

$$G_i = G(\{v_1, v_2, \ldots, v_{i-1}\} \cup S_i),$$

  para $1 \leq i \leq h$. Claramente, $\omega(G_i) \leq i - 1 + k_i' \leq i - 1 + k_i$.
- En cada estado de ramificación se activa exactamente un nodo en el árbol de búsqueda.

El pseudocódigo 7 presenta el algoritmo como aparece en el artículo de Wood [78], el cual es equivalente al digrama de flujo del algoritmo general que fue expuesto al inicio del capítulo.

Veamos que cada uno de los pasos de este pseudocódigo corresponden a cada uno de los bloques del diagrama. El paso 2 y 4 se encuentran en la Figura 4.1, el bloque en azul del diagrama de flujo es el paso 2 que corresponde a calcular las cotas superiores para la cardinalidad del clique máximo utilizando coloraciones, mientras que el paso 4 está en el bloque gris que se encarga de hacer el retroceso en el árbol de búsqueda. Los pasos restantes (1 y 3) los hallamos en la parte dos del diagrama de flujo (Figura: 4.1). En este caso el bloque azul pertenece al paso 3 que hace la ramificación; es decir, se encarga de dividir en problema en dos subproblemas. Y el bloque gris (paso 1) consiste en calcular un clique de la subgráfica $G(S_h)$.

Notemos que en el pseudocódigo 7 en la línea dos del paso 3, el problema de encontar un máximo clique de G_h se divide en dos subproblemas. Si v_h es un vértice de $G(S_h)$ entonces un clique Q de G_h estará contenido en otro:
$G_{h+1} = G(\{v_1, v_2, \ldots, v_h\} \cup (S_h \cap N_G(v_h)))$ Si $v_h \in Q$
o
$G_h = G(\{v_1, v_2, \ldots, v_{h-1} \cup (S_h \backslash v_h))$ si $v_h \notin Q$.

Además es importante notar que el algortimo selecciona a v_h de la clase de color $C^h_{k_h}$ generadas por COLOR y por DSATUR, se hace de esta manera porque son las clases que tienden a ser las más pequeñas en tamaño, por lo que la cota superior k_h se reduce más rapidamente que si un vértice arbitrario de S_h fuera escogido. Por último tenemos que $|M| \geq h - 1$ y $h - 1 + k_h > |M|$ cuando el algoritmo va al paso 3, teniendo $k_h \geq 1$ por lo tanto, en este paso simpre va a a existir $C^h_{k_h}$.

Con todo lo aquí desarrollado se implementará el programa computacional y se mostrarán los resultados en el siguiente capítulo.

Pseudocódigo 7: Algoritmo del clique máximo (AMC)

Datos: $G = (V, E)$
Resultado: El máximo clique M de G.
Paso *0* *(Inicialización):*
  Encontrar un clique máximo de una subgráfica triangulada de aristas maximal de G.
  Sea $h := 1$ y $S_h := V$.
  Ir al paso 2.
Paso *1* *(Calcular la cota inferior):*
  Encontrar un clique Q de $G(S_h)$.
  Si $h - 1 + |Q| > |M|$ **entonces**
    $M := \{v_1, v_2, \ldots, v_{h-1}\} \cup Q$.
  Ir al paso 2.
Paso *2* *(Calcular cota superior):*
  Encontrar una coloración de vértices $(C_1^h, C_2^h, \ldots, C_k^h)$, de $G(S_h)$.
  Si $h - 1 + k_h \leq |M|$ **entonces**
    ir al paso 4
  Aplicar FCP a $G(S_h)$ para obtener una mejor cota superior $k'_h \geq \omega(G(S_h))$.
  Si $h - 1 + k'_h \leq |M|$ **entonces**
    ir al paso 4
  Ir al paso 3.
Paso *3* *(Ramificar):*
  Escoger un vértice $v_h \in C_{k_h}^h$ con máximo grado en G.
  Hacer $S_{h+1} := S_h \cap N_G(v_h)$, $S_h := S_h \setminus \{v_h\}$, $C_{k_h}^h := C_{k_h}^h \setminus \{v_h\}$.
  Si $C_{k_h}^h = \emptyset$ **entonces**
    $k_h = k_{h-1}$.
  Si $k_h < k'_h$ **entonces**
    $k'_h = k_h$.
  Incrementar h e ir al paso 1.
Paso *4* *(Retroceso):*
  Si $h = 1$ **entonces**
    Parar M es el clique máximo de G.
  Disminuir h.
  Si $h - 1 + k'_h \leq |M|$ **entonces**
    ir al paso 4.
  Ir al paso 3.

Capítulo 5

Implementación computacional

El algoritmo se implementó bajo el lenguaje de programación Ruby, el código se presenta en el Apendice A.

Para explicar los resultados que nos arroja el programa utilizaremos como ejemplo la gráfica de la Figura 5.1, que cuenta con 20 vértices y 50 aristas; ya se encuentra previamente cargada en la aplicación dentro del archivo *Grafica_prueba_1.clq*, las instrucciones para manejarlo se encuentran en el manual de usuario en el apéndice B.

Para calcular el clique inicial M el programa da la opción de utilizar la subrutina uno, que como recordaremos consiste en encontrar un clique máximo de una subgráfica de aristas maximal, o bien utilizar unicamente la subrutina 4 para encontrarlo. Así como también se tiene la alternativa de utilizar los heurísticos COLOR o DSATUR para la subrutina 2 y 3.

Comenzaremos seleccionando la opción Sí, a la pregunta ¿Utilizar triangular?, para inicializar con un clique de una subgráfica triangular de aristas maximal; en la parte de procedimiento de coloración fraccional seleccionaremos DSATUR, con esto le indicamos al programa que utilize el heurístico DSATUR para la subrutina 2 y 3.

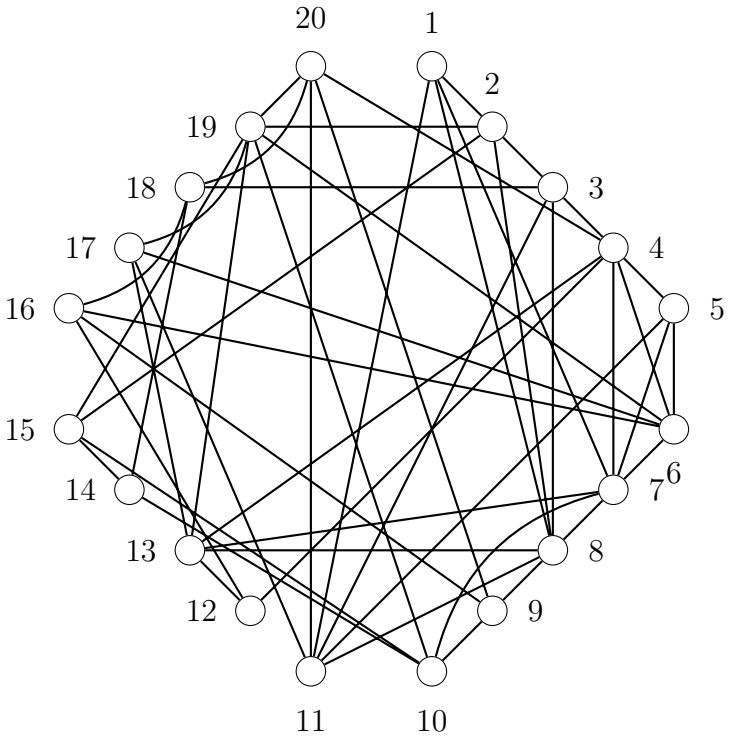


Figura 5.1: Gráfica para el ejemplo de prueba

El resultado de la ejecución del programa se puede ver en la Figura 5.2 a continuación se explicará cada uno de los resultados. Lo primero que nos muestra el programa es la subgráfica triangular de aristas maximal dado por el conjunto:

$$f = \{(1,2),(2,8),(1,8),(7,8),(1,7),(8,11),(1,11),(3,11),(3,8),(2,15),(15,19),(2,19),(13,19),(8,9),(5,11),(4,5),(9,10),(4,6),(5,6),(6,17),(4,20),(18,20),(14,18),(4,12),(12,16)\}.$$

La subgráfica generada por f, $G[f]$, se puede ver en la Figura 5.3. La ordenación de eliminación perfecta de vértices asociada a $G[f]$, está dado por:

$$sigma = \sigma = (16,12,14,18,17,20,6,10,4,5,9,13,19,15,3,11,7,8,2,1).$$

```
ruby  graphs.rb Grafica_prueba_1.clq
Gráfica para prueba
El archivo se cargo exitosamente
Calculando paso inicial...


La subgráfica de aristas maximal tiene las aristas: f= [[1, 2], [2, 8]
    , [1, 8], [7, 8], [1, 7], [8, 11], [1, 11], [3, 11], [3, 8], [2,
    15], [15, 19], [2, 19], [13, 19], [8, 9], [5, 11], [4, 5], [9, 10]
    , [4, 6], [5, 6], [6, 17], [4, 20], [18, 20], [14, 18], [4, 12], [
    12, 16]]
La ordenación perfecta de vértices es: sigma = [16, 12, 14, 18, 17,
    20, 6, 10, 4, 5, 9, 13, 19, 15, 3, 11, 7, 8, 2, 1]

Clique inicial: [8, 1, 2]

Calculando Máximo Clique...

h = 1
        s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 5, 6, 12, 13,
            20, 16, 17, 10, 9, 14]
        Cota superior (Subrutina 2)
                c[h = 1] = [[8, 4, 19, 18], [7, 17, 3, 20, 15, 16], [
                    13, 6, 11, 2, 10], [5, 1, 9, 12, 14]]
                k[h = 1] = 4
        Cota superior (Subrutina 3)
                k'[h = 1] = 4
        Ramificación
                v[h = 1] = [5]
                s[h+1 = 2] = [7, 11, 4, 6]
                s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 6, 12,
                    13, 20, 16, 17, 10, 9, 14]
                k[h = 1] = 4
                k'[h = 1] = 4
h = 2
        Cota inferior (Subrutina 4)
                m = [5, 7, 4, 6]
        Cota superior (Subrutina 2)
                c[h = 2] = [[7, 11], [4], [6]]
                k[h = 2] = 3
        Retroceso
h = 1

Máximo clique: [5, 7, 4, 6] -> w=4

Verificando que sea clique...
En realidad es clique?: true
```

Figura 5.2: Salida computacional utilizando la subrutina uno y el heurístico DSATUR

Una vez obtenida la subgráfica $G[f]$ el algoritmo calcula un clique máximo utilizando el algoritmo ACM, el cual utilizará como clique ini-

cial al calculado con la subrutina 4 a $G[f]$, todo este procediemiento se conoce como la subrutina 1 en el algoritmo general. Una vez hecho este procedimiento obtenemos un clique inicial $M = \{8, 1, 2\}$ el cual está representado con las aristas en azul en la Figura 5.3, el cual será la cota inferior en el nodo raíz del árbol de búsqueda.

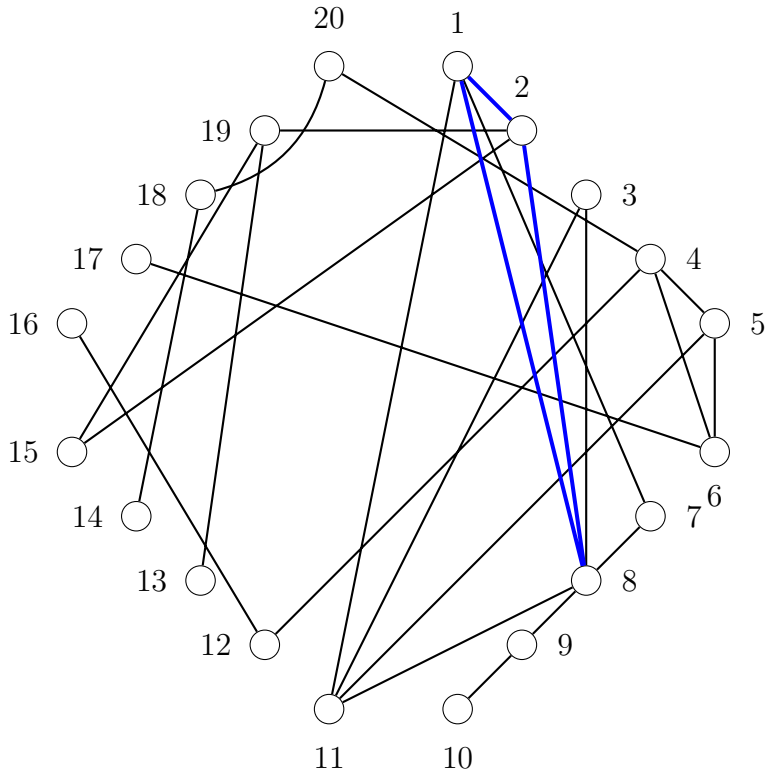


Figura 5.3: Subgráfica de aristas maximal y en azul el clique inicial

Después el programa nos muestra el conjunto de candidatos S_h cuando $h = 1$, que en este caso como va comenzando es igual a todos los vértices. En la siguiente línea nos indica que está calculando la cota superior utilizando la subrutina 2; es decir, el heurístico DSATUR, con lo que se obtiene:

$c[h = 1] =[[8, 4, 19, 18], [7, 17, 3, 20, 15, 16], [13, 6, 11, 2, 10], [5, 1, 9, 12, 14]].$

Esta parte nos indica las clases de coloración obtenidas para el conjunto $S_{h=1}$, como se puede observar obtenemos cuatro clases de coloración, por lo que $k_{h=1} = 4$, que es lo que aparece en la siguiente línea.

A continuación se calcula otra cota superior, esto debido a que $h-1+k_{h=1} > |M| = 4$, utilizando la subrutina 3 (heurístico del procedimiento de coloración fraccional), proporcionando $k'[h = 1] = 4$, con lo que $h - 1 + k'_{h=1} > |M| = 4$.

El programa continua ahora con el paso de ramificación, que como recordaremos comienza seleccionando el vértice con el grado más alto de la última clase de coloración. En este caso la última clase de coloración es $[5, 1, 9, 12, 14]$, los vértices con el grado más alto son 5, 1 y 9, el programa rompe empates con el primer vértice que encuentre en su ordenación, por lo tanto, selecciona el vértice 5, lo podemos observar en la Figura 5.2 como $v[h = 1] = [5]$. Después calcula el nuevo conjunto de vértices candidatos para $h = 2$, que serán los vecinos del vértice 5 intersección S_h, quedando como $s[h + 1 = 2] = [7, 11, 4, 6]$. Se modifica a $S_{h=1}$ eliminando el vértice 5, obteniendo: $s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 6, 12, 13, 20, 16, 17, 10, 9, 14]$. El programa muestra también a las cotas superiores ya que estas pueden ser actualizadas en el proceso de ramificación, en este caso quedan igual $k[h = 1] = 4$ y $k'[h = 1] = 4$. Por último incrementamos en una unidad a h, generando con esto una nueva rama del árbol de búsqueda.

El programa ahora nos muestra la iteración con $h = 2$, comienza encontrando una cota inferior utilizando la subrutina 4 (heurístico CLIQUE) para encontrar un clique para $S_{h=2}$, si este mejora al clique obtenido en $h = 1$ entonces este será el nuevo clique M, en este caso se mejora por lo que, $M = [5, 7, 4, 6]$ de tamaño 4.

A continuación el programa vuelve a buscar una cota superior utilizando la subrutina 2 (heurístico DSATUR) pero ahora para $S_{h=2}$, con lo que encuentra $c[h = 2] = [[7, 11], [4], [6]]$ con $k[h = 2] = 3$, en este caso $h - 1 + k_{h=2} = |M| = 4$, por lo que el algoritmo continua con la parte del retroceso.

En la parte del retroceso, disminuye a h en una unidad ya que $h \leq 1$, debido a que $h - 1 + k'_{h=1} = |M| = 4$ y además $h = 1$, entonces M es el clique máximo.

Adicionalmente el programa hace una prueba para mostrar que efectivamente el clique generado por el algoritmo es un clique, en caso de que pase la prueba muestra *true* como se puede observar en la Figura 5.2. El clique máximo generado por el algoritmo se muestra en la Figura 5.4 en rojo.

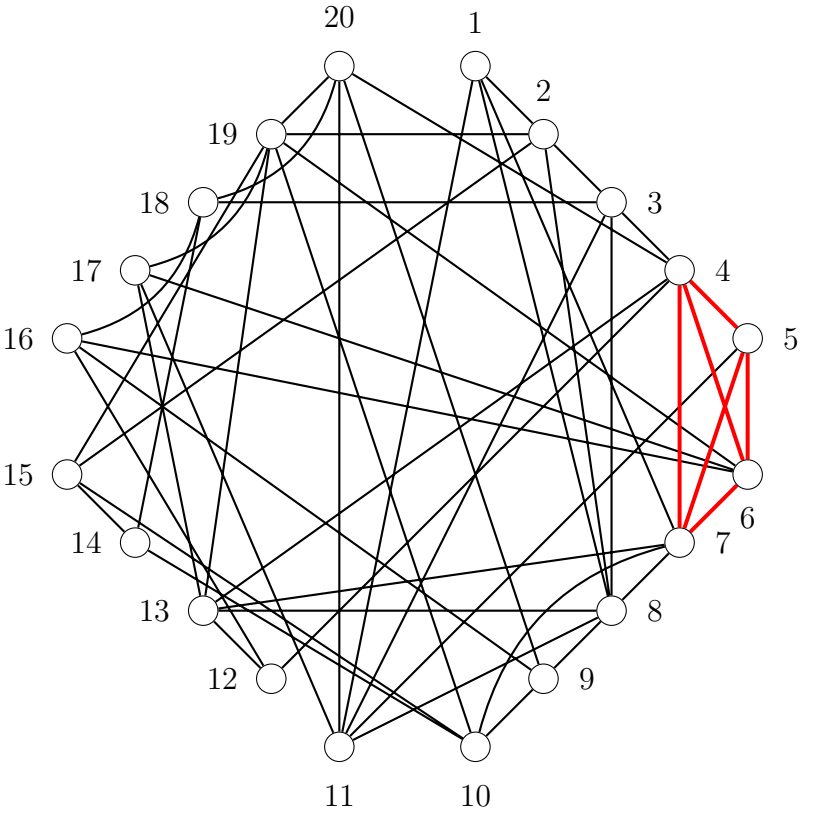


Figura 5.4: Clique máximo para el ejemplo de prueba

Ahora se muestra en la Figura 5.5 el resultado computacional de la implementación en la misma gráfica, pero calculando el clique inicial

con la subrutina 4. Como clique inicial se tiene a los vértices 8, 7 y 13, los siguientes pasos son iguales al anterior ya que también se utiliza DSATUR.

```
ruby  graphs.rb Grafica_prueba_1.clq
El archivo se cargo exitosamente

Calculando paso inicial...

Clique inicial: [8, 7, 13]

Calculando Máximo Clique...

h = 1
          s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 5, 6, 12, 13,
              20, 16, 17, 10, 9, 14]
          Cota superior (Subrutina 2)
                  c[h = 1] = [[8, 4, 19, 18], [7, 17, 3, 20, 15, 16], [
                      13, 6, 11, 2, 10], [5, 1, 9, 12, 14]]
                  k[h = 1] = 4
          Cota superior (Subrutina 3)
                  k'[h = 1] = 4
          Ramificación
                  v[h = 1] = [5]
                  s[h+1 = 2] = [7, 11, 4, 6]
                  s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 6, 12,
                      13, 20, 16, 17, 10, 9, 14]
                  k[h = 1] = 4
                  k'[h = 1] = 4
h = 2
          Cota inferior (Subrutina 4)
                 M = [5, 7, 4, 6]
          Cota superior (Subrutina 2)
                  c[h = 2] = [[7, 11], [4], [6]]
                  k[h = 2] = 3
          Retroceso
h = 1

Máximo clique: [5, 7, 4, 6] -> w=4

Verificando que sea clique...
En realidad es clique?: true
```

Figura 5.5: Salida computacional utilizando el heurístico CLIQUE y DSATUR

Ahora presentaremos los resultados, pero ahora utilizando el heurístico COLOR para la subrutina 2 y 3. El resultado del programa cuando se encuentra un clique máximo en una subgráfica triangular de aristas

maximal se puede ver en la Figura 5.6, mientras que cuando no se hace uso de este método, está en la Figura 5.7. Los dos resultados son similares al anterior, solo que en este caso se generan más ramas para el árbol de búsqueda comparado cuando solo se hace uso de COLOR. El clique máximo es el mismo en cualquier caso, ya que recordemos que el algoritmo es exacto por lo que siempre obtendremos la misma cardinalidad para el clique máximo.

```
El archivo se cargo exitosamente

Calculando paso inicial...

La subgráfica de aristas maximal tiene las aristas: f= [[1, 2], [2, 8]
    , [1, 8], [7, 8], [1, 7], [8, 11], [1, 11], [3, 11], [3, 8], [2,
    15], [15, 19], [2, 19], [13, 19], [8, 9], [5, 11], [4, 5], [9, 10]
    , [4, 6], [5, 6], [6, 17], [4, 20], [18, 20], [14, 18], [4, 12], [
    12, 16]]
La ordenación perfecta de vértices es: sigma = [16, 12, 14, 18, 17,
    20, 6, 10, 4, 5, 9, 13, 19, 15, 3, 11, 7, 8, 2, 1]

Clique inicial: [8, 1, 2]

Calculando Máximo Clique...

h = 1
        s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 5, 6, 12, 13,
            20, 16, 17, 10, 9, 14]
        Cota superior (Subrutina 2)
                c[h = 1] = [[8, 19, 4, 18], [7, 11, 2, 16, 14], [6,
                    13, 3, 20, 10, 1], [15, 5, 17, 9, 12]]
                k[h = 1] = 4
        Cota superior (Subrutina 3)
                k'[h = 1] = 4
        Ramificación
                v[h = 1] = [15]
                s[h+1 = 2] = [2, 19, 10, 14]
                s[h = 1] = [1, 2, 8, 7, 11, 3, 19, 4, 18, 5, 6, 12,
                    13, 20, 16, 17, 10, 9, 14]
                k[h = 1] = 4
                k'[h = 1] = 4
h = 2
        Cota inferior (Subrutina 4)
                M = [8, 1, 2]
        Cota superior (Subrutina 2)
                c[h = 2] = [[19, 14], [10, 2]]
                k[h = 2] = 2
        Retroceso
h = 1
        Ramificación
                v[h = 1] = [5]
                s[h+1 = 2] = [7, 11, 4, 6]
                s[h = 1] = [1, 2, 8, 7, 11, 3, 19, 4, 18, 6, 12, 13,
                    20, 16, 17, 10, 9, 14]
                k[h = 1] = 4
                k'[h = 1] = 4
h = 2
        Cota inferior (Subrutina 4)
                m = [5, 7, 4, 6]
        Cota superior (Subrutina 2)
                c[h = 2] = [[7, 11], [4], [6]]
                k[h = 2] = 3
        Retroceso
h = 1
Máximo clique: [5, 7, 4, 6] -> w=4
```

Figura 5.6: Salida computacional utilizando la subrutina uno y el heurístico COLOR

```
ruby  graphs.rb Grafica_prueba_1.clq
El archivo se cargo exitosamente

Calculando paso inicial...

Clique inicial: [8, 7, 13]

Calculando Máximo Clique...
h = 1
        s[h = 1] = [1, 2, 8, 7, 11, 3, 15, 19, 4, 18, 5, 6, 12,
            13, 20, 16, 17, 10, 9, 14]
        Cota superior (Subrutina 2)
                c[h = 1] = [[8, 19, 4, 18], [7, 11, 2, 16, 14], [
                    6, 13, 3, 20, 10, 1], [15, 5, 17, 9, 12]]
                k[h = 1] = 4
        Cota superior (Subrutina 3)
                k'[h = 1] = 4
        Ramificación
                v[h = 1] = [15]
                s[h+1 = 2] = [2, 19, 10, 14]
                s[h = 1] = [1, 2, 8, 7, 11, 3, 19, 4, 18, 5, 6,
                    12, 13, 20, 16, 17, 10, 9, 14]
                k[h = 1] = 4
                k'[h = 1] = 4
h = 2
        Cota inferior (Subrutina 4)
                M = [8, 7, 13]
        Cota superior (Subrutina 2)
                c[h = 2] = [[19, 14], [10, 2]]
                k[h = 2] = 2
        Retroceso
h = 1
        Ramificación
                v[h = 1] = [5]
                s[h+1 = 2] = [7, 11, 4, 6]
                s[h = 1] = [1, 2, 8, 7, 11, 3, 19, 4, 18, 6, 12,
                    13, 20, 16, 17, 10, 9, 14]
                k[h = 1] = 4
                k'[h = 1] = 4
h = 2
        Cota inferior (Subrutina 4)
                m = [5, 7, 4, 6]
        Cota superior (Subrutina 2)
                c[h = 2] = [[7, 11], [4], [6]]
                k[h = 2] = 3
        Retroceso
h = 1

Máximo clique: [5, 7, 4, 6] -> w=4

Verificando que sea clique...
En realidad es clique?: true
```

Figura 5.7: Salida computacional utilizando el heurístico CLIQUE y COLOR.

5.1. Resultados computacionales

Para mostrar que el programa es correcto se utilizo como prueba algunas gráficas del *Second DIMACS Implementation Challenge (1992-1993)*, donde es reportado el tamaño del clique máximo, las cuales pueden ser encontradas en el siguiente enlace: http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark.
Además se generaron quince gráficas aleatorias, nueve con 100 vértices y el resto con 200 vértices, el código con el que se producen se encuentra en el apéndice A.

El algoritmo se ejecutó en una computadora Dell Inspiron 5420, con 8GB de memoria RAM, procesador Intel Core i7-3612QM a 2.10GHz bajo un sistema operativo Arch GNU/Linux de 64bits. La versión 2.4.11 del interprete de Ruby fue utilizado para ejecutar el programa.

El Cuadro 5.1 muestra los resultados en segundos del tiempo CPU que tardó en ejecutarse el programa. La primer columna muestra el nombre del archivo de cada gráfica, cada una de ellas se encuentran previamente cargadas en la aplicación del programa; la brock202_2 y p_hat300_1 son del conjunto DIMACS, mientras que el resto son aleatorias. La segunda columna indica el número de vértices $n = |V|$, la tercera el número de aristas, la cuarta la densidad o probabilidad de aristas ($d = 2|E|/n(n-1)$); todas estas columnas dan información de la gráfica con la que estamos trabajando. Las siguientes muestran los resultados del programa, la columna $w(G)$ nos da la cardinalidad del clique máximo encontrado, mientras que las últimas cuatro columnas nos dan los tiempos en segundos que tarda el programa en calcular el clique máximo. Se ha utilizado MC_C para indicar que el algoritmo está utilizando la subrutina 1 para generar un clique máximo de una subgráfica triangulada de aristas maximal, que será la cota inferior inicial del algoritmo y el heurístico COLOR para las subrutinas 2 y 3, en la columna correspondiente nos muestra el tiempo en segundos para hacer todo el procedimiento. Mientras que MC_D, al igual que MC_C, utiliza la subrutina 1, pero para calcular las cotas superiores utiliza el heurístico DSATUR. También se evaluó el tiempo en segundos cuando

no se utiliza la subrutina 1, sino lo que se hace es encontrar un clique inicial en el nodo raíz del árbol de búsqueda con el heurístico CLIQUE, estos tiempos se encuentran en las columnas $MC0_C$ y $MC0_D$, donde en la primera se utiliza COLOR y en la segunda DSATUR como métodos de coloración.

Se puede observar que en general los tiempos obtenidos son menores cuando se utilizaba COLOR para obtener las cotas superiores en comparación con DSATUR, esto independientemente de si se utilizaba la subrutina 1. Si comparamos los tiempos que tarda el programa cuando se utiliza la subrutina uno y cuando no, los mejores tiempos siempre se obtuvieron cuando no se realizaba la subrutina uno; es decir, cuando se usa el heurístico CLIQUE para calcular la cota inferior inicial.

Nombre	n	$\|e\|$	d	$\omega(G)$	MC_C	MC_D	$MC0_C$	$MC0_D$
ga-n100-d0.1	100	495	0.1	4	4.349	6.555	0.489	1.006
ga-n100-d0.2	100	990	0.2	5	14.815	12.202	1.847	4.709
ga-n100-d0.3	100	1485	0.3	6	11.595	8.919	6.716	2.31
ga-n100-d0.4	100	1980	0.4	8	43.294	125.698	30.166	91.368
ga-n100-d0.5	100	2475	0.5	9	238.793	194.259	166.285	107.028
ga-n100-d0.6	100	2970	0.6	11	272.246	503.274	176.774	413.95
ga-n100-d0.7	100	3465	0.7	15	2372.125	2513.68	702.597	970.999
ga-n100-d0.8	100	3960	0.8	20	4606.851	4739.787	1174.63	1349.393
ga-n100-d0.9	100	4455	0.9	32	4289.693	4425.319	677.43	843.973
ga-n200-d0.1	200	1990	0.1	4	43.407	76.837	8.304	10.984
ga-n200-d0.2	200	3980	0.2	6	254.935	278.234	62.199	85.113
ga-n200-d0.3	200	5970	0.3	7	1304.345	2177.135	991.482	1854.932
ga-n200-d0.4	200	7960	0.4	9	3387.345	5840.217	2419.103	5102.709
ga-n200-d0.5	200	9950	0.5	11	12848.503	8817.918	10792.946	7384.432
ga-n200-d0.6	200	11940	0.6	14	46199.104	46841.822	24170.094	25087.369
brock200_2	200	7960	0.5	12	3908.133	6334.268	2861.355	5325.854
p_hat300-1	300	8970	0.24	8	2190.037	5981.886	1672.651	5582.503

Cuadro 5.1: Resultados obtenidos de la implementación computacional.

Fuente: Elaboración propia.

Capítulo 6

Conclusiones

Se demostró que el problema del clique máximo, en su versión de decisión es NP-Completo, por lo que al menos que se demuestre que $P = NP$ no encontraremos ningún algoritmo que resuelva en tiempo polinomial cualquier instancia; la versión de optimización es NP-Duro, la cual es al menos tan complicado como el de decisión. A pesar de la inevitable intratabilidad del problema, muchas veces es preciso dar una solución exacta aunque esto implique un costo exponencial en el tiempo.

El algoritmo aquí tratado fue exacto, utiliza la técnica de ramificación y acotamiento; la ventaja de este procedimiento es que considera solo una porción de todas las soluciones factibles, descartando las que se consideran no prometedoras. En este caso todos los cliques de la gráfica son soluciones factibles al problema; la manera en la que descarta es encontrando cotas inferiores y superiores para la cardinalidad del clique máximo $\omega(G)$, entre mejores sean estas cotas el número de posibilidades a analizar será menor.

En el capítulo 3 se demostró que el número cromático ($\chi(G)$) y el número cromático fraccional ($\chi_f(G)$) son mayores o iguales a la cardinalidad del clique máximo ($\omega(G)$), es por ello que el algoritmo utiliza a los dos primeros como cotas superiores. Encontrar estas cotas de forma exacta es también exponencial debido a que los problemas de mínima

coloración y el de coloración fraccional de una gráfica son NP-Duros, y debido a esto se utilizan heurísticos para aproximarlos. En particular los algoritmos COLOR y DSATUR para la coloración y el procedimiento de coloración fraccional FCP para la coloración fraccional.

Por otro lado, para determinar una cota inferior se utiliza el hecho de que cualquier clique de una subgráfica de G es menor o igual al clique máximo de la gráfica; siguiendo esta idea se utilizó un heuristico glotón (CLIQUE) para generarlos.

Otra de las ventajas del algoritmo del clique máximo (ACM) es que en el nodo raíz utiliza como cota inicial a la cardinalidad de un clique máximo de una subgráfica triangulada de aristas maximal, en el capítulo 3 se demuestra que estas gráficas son también gráficas perfectas y cumplen con $\omega(G_A) = \chi(G_A)$ para toda subgráfica G_A; se aprovecha esta propiedad para obtener una cota inferior.

Para ramificar; es decir, dividir el problema en subproblemas más pequeños, se elige un vértice v con el grado más alto de la última clase de coloración y se generan dos subproblemas. El primero genera un nuevo nodo de búsqueda que corresponderá a encontrar un clique máximo de la subgráfica generada por los vecinos de v y su intersección con los vértices candidatos, los cliques generados a partir de este nodo simpre incluirán a v. El otro subproblema consiste en encontrar un clique máximo de la subgráfica generada por los vértices candidatos sin el vértice v, por lo que todos los cliques generados a partir de este momento no incluirán a éste.

Se mostró que el programa realizado es correcto con la implementación de una función, que revisa si el clique generado por el programa efectivamente lo es. Además se probaron algunas gráficas del *Second DIMACS Implementation Challenge (1992-1993)*, que es un conjunto de gráficas con su clique máximo reportado, para las cuales se obtuvo el resultado correcto.

Se compararon los tiempos de ejecución del algoritmo utilizando la generación del clique máximo de una subgráfica triangular de aristas

maximal para la determinación de la cota inicial y utilizando CLIQUE para obtenerla; se obtuvieron mejores tiempos con CLIQUE, independientemente del método de coloración usado. Mientras que COLOR comparado con DSATUR en la mayoría de las gráficas reportó mejores tiempos, sin importar si se utilizaba CLIQUE o no en el nodo raíz. Todos los tiempos reportados fueron muy grandes para la mayoría de gráficas contrastados con los reportados en el artículo [78], lo cual se puede atribuir al lenguaje usado. Sin embargo, el fin de este trabajo era poder implementarlo para mostrar que es correcto, además aprovechar la facilidad del método de ramificación y acotamiento para generar el clique máximo de la gráfica.

Apéndice A

Código en Ruby

```ruby
def mc(g1,m)
  # Algoritmo del clique máximo MC, propuesto por
  # David Wood.
  # Devuelve M, el clique máximo de una gráfica.

  h = 1
  sh = []
  sh << g1.vertices

  # Método para colorear: color o dsatur
  metodo = "color"
  c = []
  u = []
  kh = []
  khf = []
  fin = false

  while(1)
     c[h-1] = mc_paso2(g1,sh,metodo,h)
     kh[h-1] = c[h-1].size
     if h-1+kh[h-1] <= m.size
        h,fin = mc_paso4(h,m,kh)
         return m if fin
```

```
            c,sh,kh,khf,u = mc_paso3(g1,c,sh,h,kh,khf,
               u)
            h += 1
            m = mc_paso1(g1,sh,h,m,u)
            next
         end
         khf[h-1] = fcp(g1,metodo)
         if h-1+khf[h-1] <= m.size
            h,fin = mc_paso4(h,m,khf)
             return m if fin
            c,sh,kh,khf,u = mc_paso3(g1,c,sh,h,kh,khf,
               u)
            h += 1
            m = mc_paso1(g1,sh,h,m,u)
            next
         end
         c,sh,kh,khf,u = mc_paso3(g1,c,sh,h,kh,khf,u)
         h +=1
         m = mc_paso1(g1,sh,h,m,u)
      end
   end
```

```
   def mc_paso0(g)
      # Implementación del paso 0 para el
      # algoritmo MC.

      g1 = emts(g)
      m = clique(g1)
      m1 = mc(g1,m)
       return m1
   end
```

```
   def mc_paso1(grafica,sh,h,m,u)
      # Implementación del paso 1 para el
      # algoritmo MC.

      g = grafica.clone
      a = g.vertices - sh[h-1]
```

```
7
8       a.each do  |v |
9          g.remove_vertex(v) #grafica de G(sh)
10      end
11
12      q = clique(g)
13
14      if h-1+(q.size) > m.size
15         if u.size == 0  | | u.size == 1
16            m = q
17         else
18            n = h-2
19            m = u[0..n]+q
20            m.uniq!
21         end
22      end
23
24       return m
25  end
```

```
1   def mc_paso2(grafica,sh,metodo,h)
2      # Implementación del paso 2 para el
3      # algoritmo MC.
4
5      g = grafica.clone
6      c = []
7      a = g.vertices - sh[h-1]
8      a.each do  |v |
9         g.remove_vertex(v)
10     end
11     if metodo == "color"
12        c = color(g)
13     elsif metodo == "dsatur"
14        c = dsatur(g)
15     end
16
17      return c
18  end
```

```
def mc_paso3(grafica,c,sh,h,kh,khf,u)
  # Implementación del paso 3 para el
  # algoritmo MC.

  v = 0
  sh1 = []
  maxg = 0
  c[h-1].last.each do  |a|
     grado = grafica.out_degree(a)
     if grado > maxg
        maxg = grado
        v = a
     end
  end
  sh1 = sh[h-1] &(grafica.adjacent_vertices(v))
  sh[h-1].delete(v)
  sh[h] = sh1
  u[h-1] = v
  c[h-1].last.delete(v)
  if c[h-1].last.empty?
     kh[h-1] -= 1
     c[h-1] = c[h-1][0..-2]
  end
  khf[h-1] = kh[h-1] if kh[h-1] < khf[h-1]

   return c,sh,kh,khf,u
end
```

```
def mc_paso4(h,m,khf)
  # Implementación del paso 4 para el
  # algoritmo MC.

  fin = false
  if h==1
     @maxc = m
      return h,true
  end

```

```
11      h -= 1
12
13      if h-1+khf[h-1] <= m.size
14         h,fin = mc_paso4(h,m,khf)
15      end
16
17       return h,fin
18  end
```

Implementación de los algoritmos utilizados por AMC.

```
1   def emts(grafica)
2     # Implementa el algoritmo para encontrar la
3     # subgráfica triangular de aristas maximal
4     # EMTS, propuesto por Balas.
5     # Devuelve a F el  conjunto de aristas que
6     # forman la subgráfica triangular
7     # de aristas maximal y a sigma la ordenación
8     # de eliminación perfecta de vértices.
9
10    g = grafica.clone
11    n = g.num_vertices
12    f = []
13    sigma = []
14    e = {}
15    m = []
16    i = n
17
18    n.times{ |i | e[i+1] = []}
19
20    while(i>0)
21       s = g.vertices - sigma #vertices no numerados
22       v = s.first
23       e = e.sort{ |x,y |
24          xn = x[1].sort.reverse.join
25          yn = y[1].sort.reverse.join
26          r = -(xn <=> yn)
27          if r == 0
```

```
28              r = (x[0] <=> y[0])
29           end
30           r
31        }.to_h
32        e.each do |ei|
33           u = s &ei
34           unless u.empty?
35              v = u.first
36              break
37           end
38        end
39        if m.include?(v)
40           m.delete(v)
41           sigma.unshift(v)
42           e,f,g = emts_paso3(g,v,m,sigma,e,f,i)
43           i-=1
44        else
45           break if v.nil?
46           sucesores = sigma & g.adjacent_vertices(v)
47           sigma.unshift(v)
48           if sucesores.empty?
49              e,f,g = emts_paso3(g,v,m,sigma,e,f,i)
50              i -= 1
51              next
52           end
53           y = sucesores.shift #Primer sucesor de v
54           vy = [v,y].sort
55           f << vy
56           contador = 0
57           sucesores.each do |w|
58              yw = [y,w].sort
59              if f.include?(yw)
60                 vw = [v,w].sort
61                 f << vw
62                 contador += 1
63              end
64           end
65           if (contador == sucesores.size &&
             sucesores.size != 0)
```

```
66                | | sucesores.empty?
67                e,f,g = emts_paso3(g,v,m,sigma,e,f,i)
68                i -= 1
69             else
70                sucesores.each do  |w |
71                   vw = [v,w].sort
72                   if !f.include?(vw)
73                      g.remove_edge(v,w)
74                      e[v].delete(w)
75                   end
76                end
77                sigma.delete(v)
78                m << v
79             end
80          end
81       end
82       g1 = RGL::AdjacencyGraph.new
83       f.each{ |e | g1.add_edge(e[0],e[1])}
84        return g1
85    end
```

```
1    def emts_paso3(g, v, m, sigma, e, f, i)
2       # Método utilizado por el algoritmo para
3       # encontrar la subgráfica triangular de
4       # aristas maximal, implementa el paso 3
5       # de dicho algoritmo.
6
7       n = g.adjacent_vertices(v)
8       n.each do  |x |
9          if( !m.include?(x)  && !sigma.include?(x) )
10            e[x] << i
11         elsif( m.include?(x) && !sigma.include?(x) )
12            sigma.each do  |w |
13               vw = [v,w].sort
14               xw = [x,w].sort
15               if( f.include?(vw) && f.include?(xw) )
16                  vx = [v,x].sort
17                  f << vx
18                  e[x] << i
```

```
19              elsif( !f.include?(vw) && f.include?(xw
                   ) )
20                 g.remove_edge(v,x)
21              end
22           end
23        end
24     end
25      return [e,f,g]
26  end
```

```
1   def clique(grafica)
2     # Devuelve un clique de una gráfica dada.
3
4     g = grafica.clone
5     s = ordena_grado(g)
6     q = []
7
8     while s.size > 0
9        v = s.first
10       q << v
11       n = g.adjacent_vertices(v)
12       s  &= n
13    end
14
15     return q
16  end
```

```
1   def color(grafica)
2     # Devuelve las clases de coloración de una
3     # gráfica utilizando el heurístico  COLOR.
4
5     g = grafica.clone
6     clases = []
7     s = ordena_grado(g)
8     i = 0
9
10    while s.size > 0
11       clases[i] = []
```

```
12          s -= clases[i-1]
13          sw = s.clone
14          while sw.size > 0
15             tmp = sw.delete_at(0)
16             clases[i] << tmp
17             g.adjacent_vertices(tmp).each do |v|
18                sw.delete(v)
19             end
20          end
21          i += 1
22       end
23
24       return clases[0..-2]
25    end
```

```
1     def dsatur(grafica)
2        # Implementa el algoritmo heurístico de
3        # coloración de vértices DSATUR, propuesto
4        # por Brelaz.
5        # Devuelve las clases de coloración de una grá
            fica.
6
7        g = grafica.clone
8        clases = []
9        s = crea_saturacion(g)
10       i = 0
11
12       while s.size > 0
13          v = ordena_saturacion(s).first.first
14          n = g.adjacent_vertices(v)
15          clases = agrega_vertice_saturacion(clases, v,
                n)
16          s = aumenta_saturacion(s,n)
17          s.delete(v)
18          i += 1
19       end
20
21       return clases
22    end
```

```
def fcp(g, metodo)
  # Implementa el heurístico de coloración
  # fraccional FCP, de Balas y Xue.
  # Usa COLOR o DSATUR para encontrar una
  # coloración de vértices.
  # Devuelve una cota superior para un
  # clique de una gráfica.

  clases = []
  t0 = 9999999
  t1 = 0
  i = 1

  while(1)
     s = []
     g.each_vertex do  |v |
        n = g.adjacent_vertices(v)
        n << v
        c = clases.clone
        c.each_with_index do  |clase, j |
           if (clase  & n).size == 0
              clases[j] << v
              s << v
              break
           end
        end
     end
     g2 = g.clone
     s.each do  |v |
        g2.remove_vertex(v)
     end
     clases += dsatur(g2) if metodo == "dsatur"
     clases += color(g2) if metodo == "color"
     t1 = clases.size*(1.0/i)
     if (t1 < t0)
        t0 = t1
        i = i+1
```

```
38          else
39              return t0.floor
40          end
41      end
42
43  end
```

Métodos utilizados por los algoritmos.

```
1   def crea_saturacion(g)
2     # Genera un hash, que asocia a cada
3     # vértice con su grado de saturación
4     # y su vecindad.
5
6     s = {}
7     g.each_vertex do  |v |
8        s[v] = [0,g.out_degree(v)]
9     end
10
11     return s
12  end
```

```
1   def ordena_saturacion(s)
2     # Ordena de mayor a menor, al hash s,
3     # de acuerdo a su grado de saturación.
4
5     s = s.sort do  |x,y |
6        r = 0
7        sx = x[1].first
8        sy = y[1].first
9        dx = x[1][1]
10       dy = y[1][1]
11
12       if sx == sy
13          r = -1*(dx <=> dy)
14       else
15          r = -1*(sx <=> sy)
16       end
```

```
17
18          r
19        end
20
21         return s.to_h
22      end
```

```
1     def aumenta_saturacion(s,n)
2       # Aumenta el grado de saturación de
3       # los vértices que están en el arreglo n.
4       # Devuelve s con los nuevos grados de
5       # saturación.
6
7       n.each do  |v |
8          next if s[v].nil?
9          s[v][0] += 1
10      end
11
12       return s
13    end
```

```
1     def ordena_grado(g)
2       # Ordena los vértices de una gráfica
3       # de mayor a menor y los devuelve en
4       # un arreglo.
5
6       vertices = {}
7       g.each_vertex do  |v |
8          vertices[v] = g.out_degree(v)
9       end
10
11      vertices = vertices.sort do  |x,y |
12         -1*(x[1] <=> y[1])
13      end
14
15      vs = []
16      vertices.each do  |v,d |
17         vs << v
```

```
    end

     return vs
end
```

```
 def agrega_vertice_saturacion(c, v, n)
    # Agrega el vértice v a una clase de coloración,
    # si está no existe crea una.

    clases = c.clone

    c.each_with_index do  |clase,i |
       if (clase  & n).size == 0
          clase << v
          clases[i] = clase

           return clases
       end
    end

    clases << [v]

     return clases
end
```

Función que verifica que un subconjunto de vértices realmente sea un clique.

```
puts "Verificando que sea clique..."
is_clique = true
maxc.each do  |v |
    n = @graph.adjacent_vertices(v)
    r = maxc-[v] == (maxc-[v]) &n
    is_clique = is_clique && r
end
```

Función que genera y guarda en un archivo una gráfica aleatoria, con n vértices y una densidad d.

```
# vertices
n = ARGV[0].to_i
# densidad
d = ARGV[1].to_f
# aristas
mmax = ((n * (n-1) * d) / 2).floor

 def ordena(x, y)
    return [x,y] if x < y
    return [y,x]
end

puts "Generando gráfica aleatoria con densidad #{d}
    y #{n} vértices..."

ms = []
while ms.size < mmax
   x = rand(n) + 1
   y = rand(n) + 1

   next if x == y

   ms << ordena(x,y)
   ms.uniq!
end

fname = "ga-n#{n}-m#{mmax}-d#{d}.clq"
puts "Gráfica generada, guardando archivo: #{fname
   }..."

open(fname, 'w+') do |f|
   f.puts 'c Gráfica aleatoria'
   f.puts "c Densidad: #{d}"
```

```
32      f.puts "c Aristas: #{mmax}"
33      f.puts "c Vértices: #{n}"
34      f.puts 'c'
35      f.puts "p edge #{n} #{mmax}"
36      f.puts 'c'
37
38      ms.each do  |e |
39          f.puts "e #{e[0]} #{e[1]}"
40      end
41  end
```

Función para cargar el archivo.

```
1   require 'rgl/adjacency'
2
3   @maxc = []
4
5   @graph = RGL::AdjacencyGraph.new
6   @total_vertices = 0
7   @total_edges = 0
8
9   @color_classes = []
10
11  unless Array.respond_to?(:to_h)
12      class Array
13          def to_h
14            h = {}
15            self.each{ |e | h[e.first] = e[1]}
16
17             return h
18         end
19      end
20  end
21
22   def load_graph(fn)
23      f = open(fn, 'r')
24      lines = f.readlines
25      f.close
```

```
26
27      lines.each do  |line |
28          if line.index('e') == 0
29              elems = line.split(' ')
30              @graph.add_edge(elems[1].to_i, elems[2].
                    to_i)
31              next
32          end
33
34          if line.index('c') == 0
35              puts "#{line[1..-1].strip}"
36              next
37          end
38
39          if line.index('p') == 0
40              elems = line.split(' ')
41              @total_vertices = elems[2].to_i
42              @total_edges = elems[3].to_i
43              next
44          end
45      end
46
47      if @graph.num_vertices != @total_vertices
48          puts "Error, el archivo reporta #{
                @total_vertices} vertices pero solo se
                cargaron #{@graph.num_vertices} vertices"
49          exit
50      end
51
52      if @graph.num_edges != @total_edges
53          puts "Error, el archivo reporta #{
                @total_edges} aristas pero solo se cargaron
                 #{@graph.num_edges} aristas"
54          exit
55      end
56
57      puts 'El archivo se cargo exitosamente'
58      @graph.freeze
59  end
```

Apéndice B

Manual de usuario

La implementación computacional del Algoritmo del Clique Máximo se realizó en Ruby, un lenguaje de programación de scripts (interpretado) relativamente moderno (1995) completamente orientado a objetos y desarrollado por Yukihiro *Matz* Matzumoto.

Ruby fue elegido para la implementación debido a que se trata de un lenguaje muy expresivo, es decir, se pueden lograr una gran cantidad de cosas con pocas instrucciones. Además, desde su creación, Ruby pretende ser un lenguaje que minimice el trabajo de programación, por lo que resulta idóneo para codificar prototipos de manera rápida y simple.

La implementación del algoritmo utiliza **RGL** o *Ruby Graph Library* como base para las estructuras de datos. **RGL** es una biblioteca cuyo diseño ha sido influenciado por **BGL** (*Boost Graph Library*), una biblioteca de uso común con C++. Si bien **RGL** posee algoritmos implementados para gráficas ninguno de estos ha sido utilizado.

B.1. Interfaz gráfica

Se diseñó una interfaz gráfica de usuario (*GUI*) con el fin de facilitar el uso de la implementación computacional del algoritmo.

Existe una implementación de Ruby (**JRuby**) que se ejecuta sobre la máquina virtual de Java (*JVM*) y que permite la interacción directa con cualquier biblioteca de Java. Utilizando esta flexibilidad se utilizó **Swing**, una biblioteca para la construcción de interfaces gráficas sobre Java muy popular, para construír una interfaz que pudiera ser ejecutada en diferentes plataformas (Windows y MacOSX).

Para reducir al mínimo la complicación en el uso del programa y buscando que el usuario no tenga que realizar instalaciones ni configuraciones, se han proporcionado todos los archivos necesarios para la ejecución y prueba de la implementación del Algoritmo del Clique Máximo.

Dentro del directorio *jre* se incluyen los archivos de la *JVM* para Windows y MacOSX. La versión 9.2.0.0 de *JRuby* se encuentra en el directorio *rb* empaquetado en un archivo *jar* propio de Java. La biblioteca *RGL* se incluye dentro de *rb/lib*. Los archivos *clq* que describen las gráficas utilizadas se pueden encontrar en el directorio *clq*. Mientras que la interfaz gráfica de usuario y su código fuente se encuentran en *gui* y en *gui/src* respectivamente; la interfaz se encuentra también empaquetada como un archivo *jar*. Además, dentro de *scripts* y *src* se encuentrar archivos auxiliares para la correcta ejecución del programa en los diferentes sistemas operativos.

La implementación del ACM se encuentra en *rb/acm.rb* y cualquier usuario con conocimientos técnicos puede utilizarla sin problemas acompañado de los archivos *clq* correspondientes o construyendo los propios.

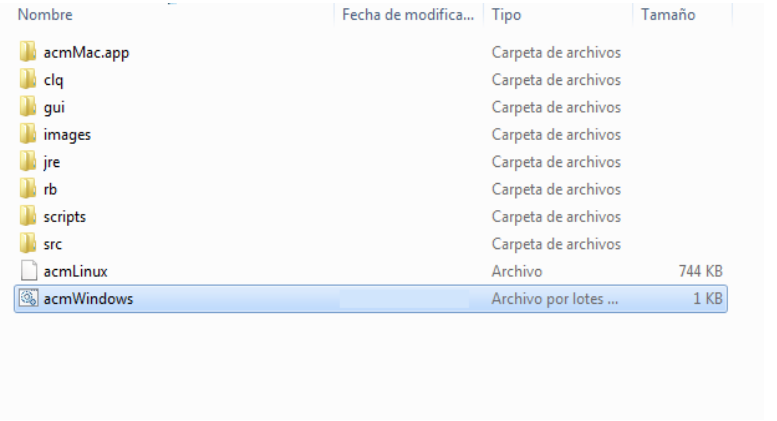
B.2. Ejecución y uso de la interfaz

En esta sección se describe de manera simultanea el uso de la interfaz gráfica de usuario para sistemas operativos Windows y MacOSX.

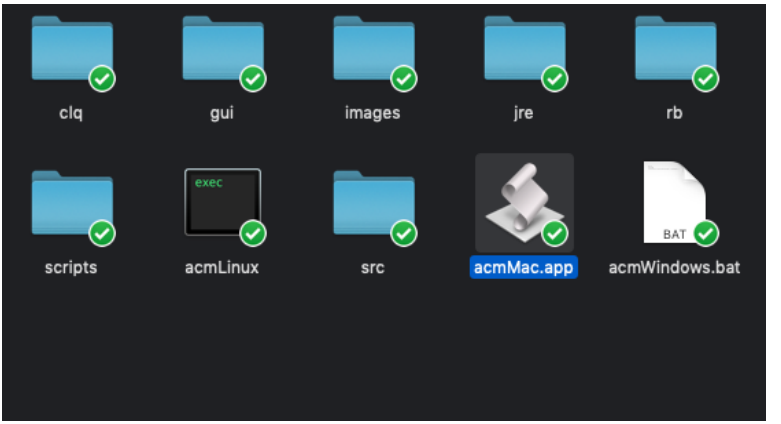
A continuación se muestran los pasos a seguir y se incluyen captu-

ras de pantalla para ambos sistemas operativos.

Al abrir el directorio donde se encuentran los archivos correspondientes deberán localizarse los ejecutables que corresponden a cada sistema operativo. Para Windows, el archivo adecuado es *acmWindows.bat* y para MacOSX el correspondiente es *acmMac.app* como se muestra en las Figura (B.1).



(a) Archivo ejecutable en Windows.

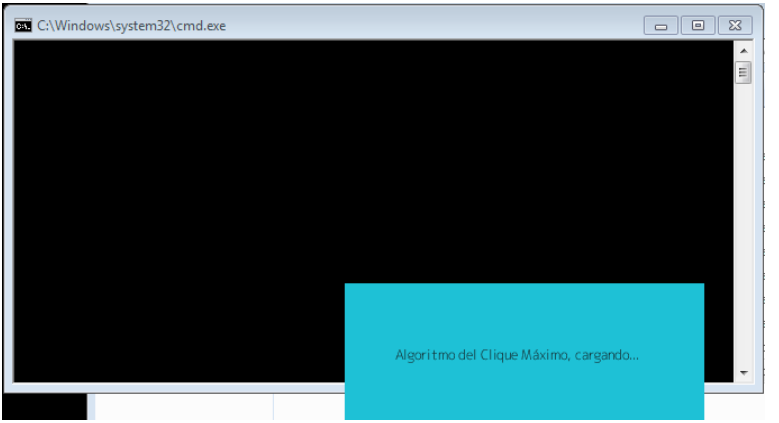


(b) Archivo ejecutable en MacOSX.

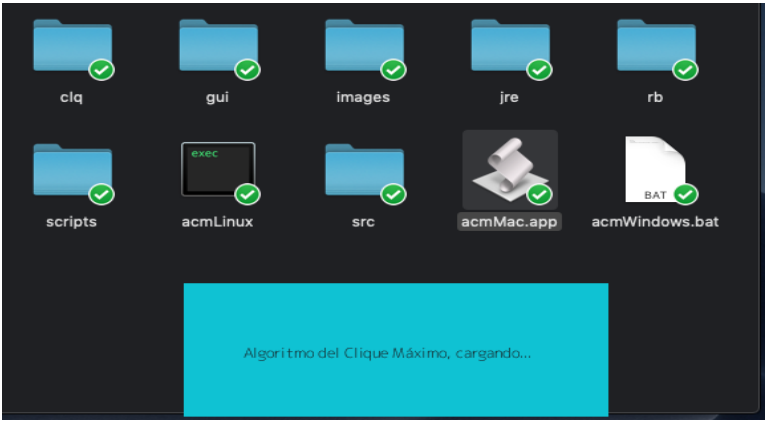
Figura B.1

Al dar doble click en el ejecutable correspondiente aparece un men-

saje que indica que el programa (interfaz gráfica) se encuentra cargando como se muestra en las Figura (B.2). El tiempo de carga dependerá de los recursos de la computadora del usuario.



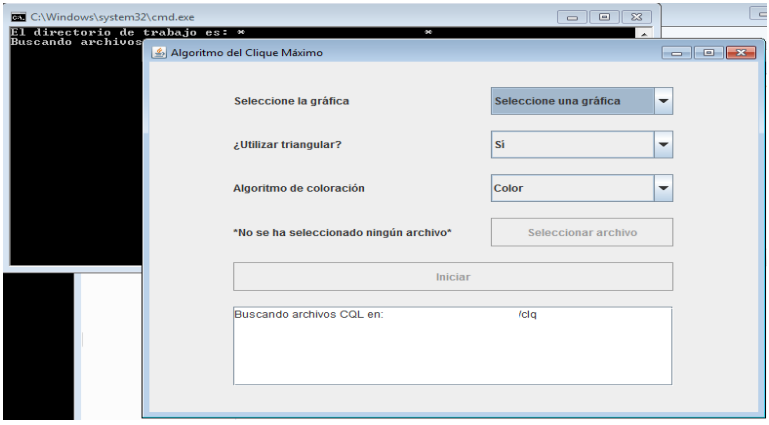
(a) Ejecución de la *GUI* en Windows.



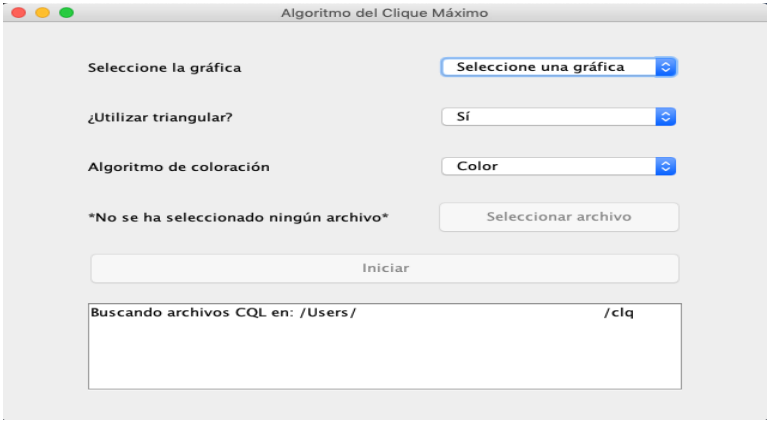
(b) Ejecución de la *GUI* en MacOSX.

Figura B.2

Cuando ha terminado la carga de la interfaz gráfica de usuario se muestra el estado inicial de la misma con las diferentes opciones a elegir, esto se muestra en las Figura (B.3)



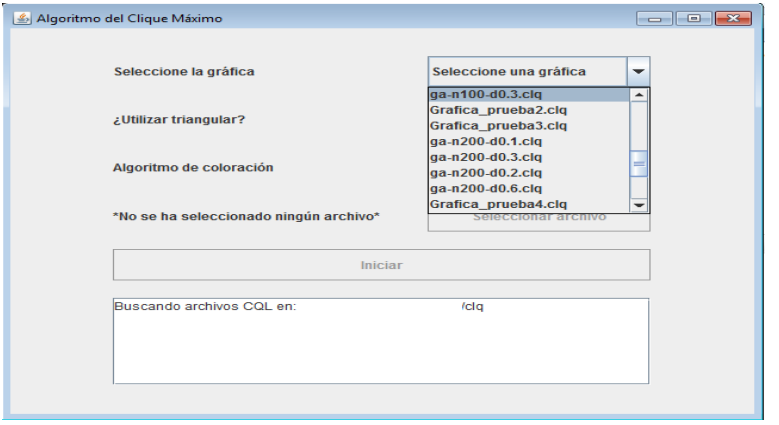
(a) Estado inicial de la *GUI* en Windows.



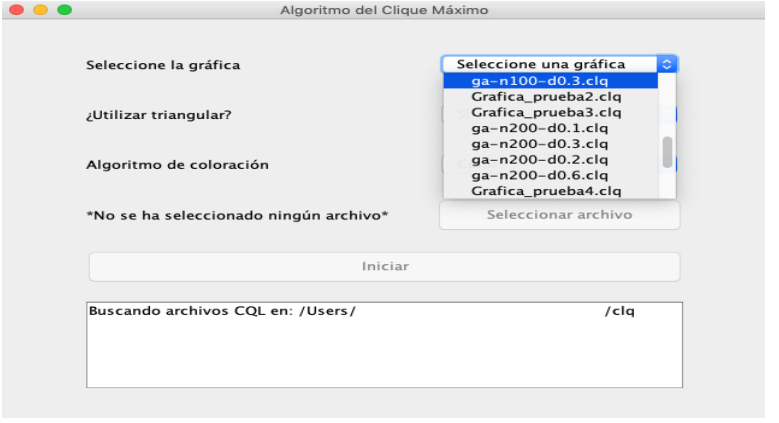
(b) Estado inicial de la *GUI* en MacOSX.

Figura B.3

La primera opción mostrada es la elección de la gráfica con la cual trabajará el programa. La interfaz cargará por defecto los archivos que se encuentren en el directorio *clq* como se muestra en las Figura (B.4)



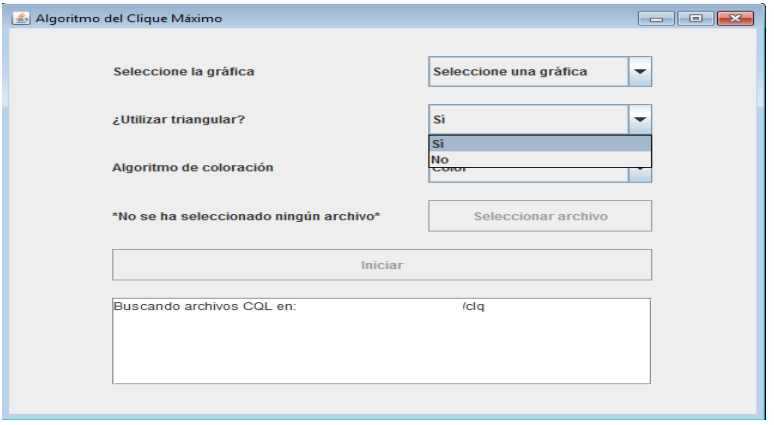
(a) Opciones de selección de gráfica en Windows.



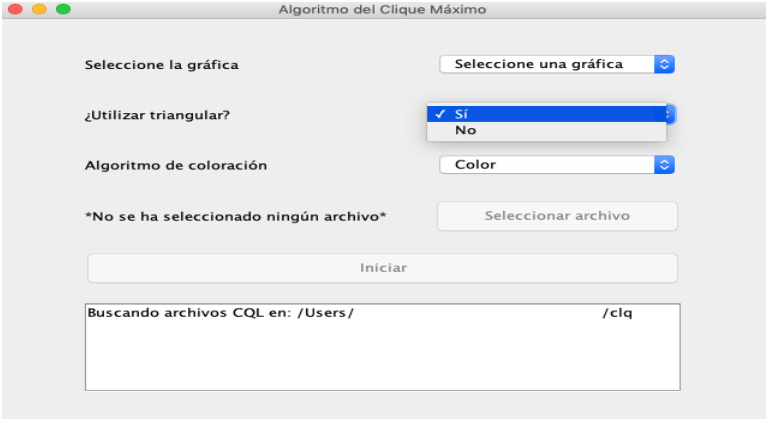
(b) Opciones de selección de gráfica en MacOSX.

Figura B.4

A continuación, en las Figura (B.5) se muestra si el programa debe utilizar triangular, esto nos indica si ejecutará el algoritmo para encontrar una subgráfica triangular de aristas maximal y después CLIQUE para encontrar el clique inicial.



(a) Selección del uso de la subrutina TRIANGULAR en Windows.

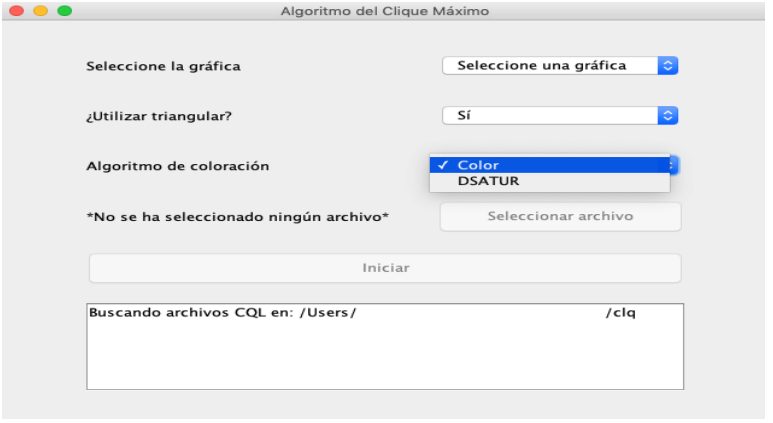


(b) Selección del uso de la subrutina TRIANGULAR en MacOSX.

Figura B.5

La siguiente opción (Figura (B.6)) es la elección del heurístico que utilizará la subrutina 2 y el procedimiento de coloración fraccional, ya sea *color* o *DSATUR*

(a) Elección del algoritmo de coloración fraccional en Windows.

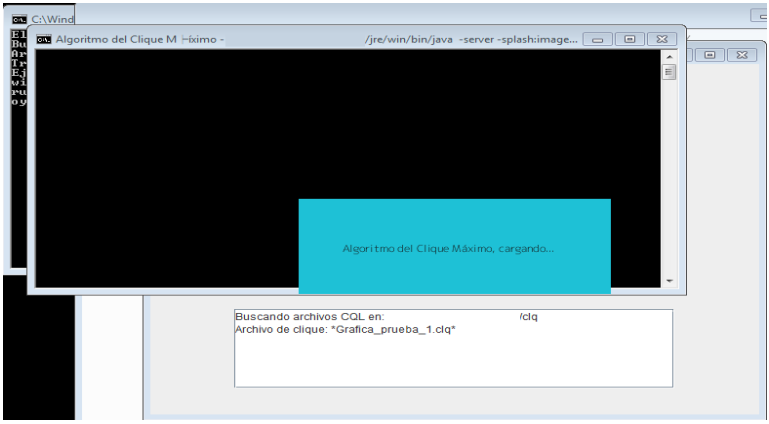


(b) Elección del algoritmo de coloración fraccional en MacOSX.

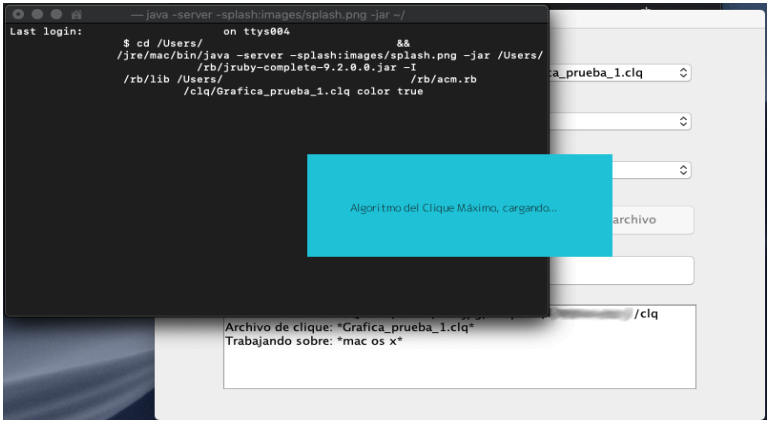
Figura B.6

Si se ha seleccionado un archivo correctamente, el botón *Iniciar* se activará automáticamente (inicialmente está desactivado) y ahora será posible comenzar la ejecución del algoritmo del clique máximo. Al dar click en *Iniciar* se desplegará nuevamente un aviso en color azul que indica la carga del programa correspondiente con los parámetros elegidos. El tiempo de espera puede ser variado dependiendo del equipo de cómputo en el que se ejecute. Nuevas ventanas de ejecución de

comandos pueden ser abiertas en este punto, no deben cerrarse. Esto se muestra en las Figura (B.7).



(a) Ejecución del ACM con los parámetros elegidos en Windows.

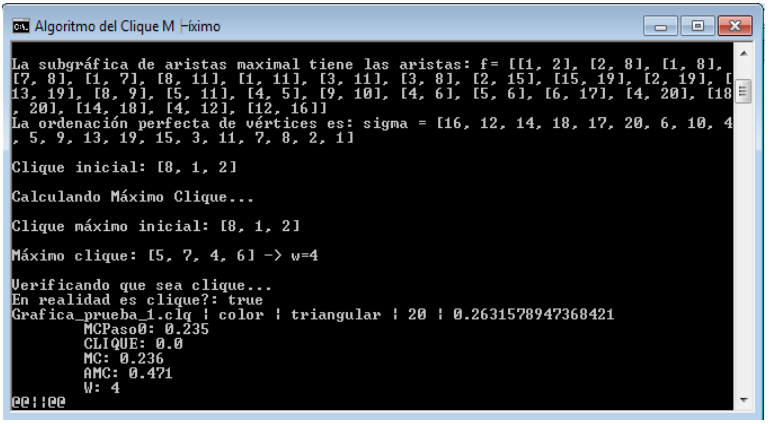


(b) Ejecución del ACM con los parámetros elegidos en MacOSX.

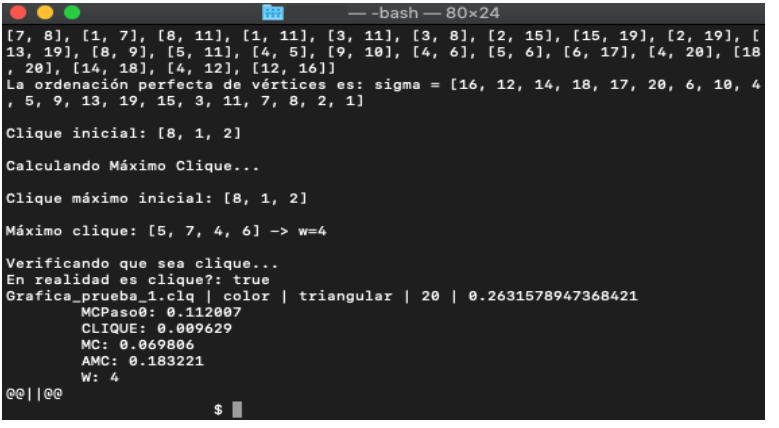
Figura B.7

En las ventanas de ejecución de comandos abiertas se mostrará el proceso del algoritmo, dependiendo de la gráfica y los parámetros que se hayan seleccionado será el tiempo de ejecución necesario. Los resultados son mostrados en la misma ventana donde se ha ejecutado

la implementación del algoritmo, un ejemplo puede verse en las Figura (B.8).



(a) Resultados de ejecución del ACM en Windows.



(b) Resultados de ejecución del ACM en MacOSX.

Figura B.8

Referencias

[1] E. A. Akkoyunlu. The Enumeration of Maximal Cliques of Large Graphs. *SIAM Journal on Computing*, 2:1–6, 1973.

[2] J. G. Auguston and J. Minker. An Analysis of Some Graph Theoritical Cluster Techniques. *J. Assoc. Comput. Mach.*, 17:571–588, 1970.

[3] L. Babel. Finding Maximum Cliques in Arbitrary and in Special Graphs. *Computing*, 46:321–341, 1991.

[4] L. Babel and G. Tinhofer. A Branch and Bound Algorithm for the Maximum Clique Problem. *ZORN-Methods and Models of Operations Research*, 34:207–217, 1990.

[5] E. Balas. A Fast Algorithm for Finding an Edge-Maximal Subgraph with a TR-formative Coloring. *Discrete Applied Mathematics*, 15:123–134, 1986.

[6] E. Balas and J. Xue. Weighted and Unweighted Maximum Clique Algorithms with Upper Bounds from Fractional Coloring. *Algorithmica*, 15(5):397–412, 1996.

[7] E. Balas and C. S. Yu. Finding a Maximum Clique in an Arbitrary Graph. *SIAM Journal on Computing*, 14(4):1054–1068, 1986.

[8] R. Barr, B. Richard, A. Golden, J. Kelly, et al. Designing and Reporting on Computational Experiments with Heuristic Methods. *Journal of Heuristic*, 1:9–32, 1995.

[9] A. R. Bednarek and O. Taulbee. On Maximal Chains. *Roum. Math. Pres et Appl.*, (11):23–25, 1966.

[10] C. Berge. Les problèmes de colorations en thèorie des graphs. *Publ. Inst. Statist. Univ. Paris*, 9:123–160, 1960.

[11] C. Berge. *Graphs and Hypergraphs*, volume 6 of *North-Holland mathematical library*. North-Holland Pub. Co., 1973.

[12] N. Biggs. Some Heuristics for Graph Coloring. In R. Nelson and R. Wilson, editors, *Graph Colourings*, pages 87–96. Longman, New York, 1990.

[13] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[14] V. Blum and G. Minty. How Good is the Simplex Algorithm? *Inequalities III*, pages 159–175, 1972.

[15] R. E. Bonner. On Some Clustering Techniques. *IBM J. Res. Develop.*, 8(1):22–32, 1964.

[16] D. Brelaz. News Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(4):251–256, 1979.

[17] C. Bron and J. Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Comm. of ACM*, 16:575–577, 1973.

[18] R. Carraghan and P. M. Pardalos. An Exact Algorithm for the Maximum Clique Problem. *Operations Research Letters*, 9:375–382, 1990.

[19] N. Chiba and T. Nishizeki. Arboricity and Subgraph Listing Algorithm for the Maximum Independent Set Problem on Planar. *SIAM Journal on Computing*, 14:663–675, 1985.

[20] M. Chudnovsky, N. Robertson, P. Seymour, and R. Thomas. The strong perfect graph theorem. *Annals of Mathematics*, 164:51–229, 2006.

[21] V. Chávatal. Determining the Stability Number of a Graph. *SIAM Journal on Computing*, 6:643–662, 1977.

[22] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[23] G. B. Dantzig. On the significance of solving linear programming problems with some integer variables. *Econometrica*, 28(1):pp. 30–44, 1960.

[24] J. F. Desler and S. L. Hakimi. On Finding a Maximum Internally Stable Set on a Graph. *Proc. of Fourth Annual Princeton Conference on Information Sciences and Systems*, 4:459–462, 1970.

[25] R. Diestel. *Graph Theory*. Springer Graduate Texts in Mathematics 173. Springer Verlag, 2000.

[26] J. Edmonds. Covers and packings in a family of sets. *Bulletin of the American Mathematical Society*, 68(5):494–499, 09 1962.

[27] T. A. Feo, M. G. Resende, and S. H. Smith. A Greedy Ramdomized Adaptive Search Procedure for Maximum Independent Set. *Operations Research*, 1994.

[28] C. Friden, A. Hertz, and D. de Werra. Stabulus: A Technique for Finding Stable Sets in Large Graphs with Tabu Search. *Computing*, 42:4–35, 1989.

[29] C. Friden, A. Hertz, and D. de Werra. TABARIS: An Exact Algorithm Based on Tabu Search For Finding a Maximum Independent Set in a Graph. *Computers and Operations Research*, 17(5):437–445, 1989.

[30] D. R. Fulkerson. The perfect graph conjecture and pluperfect graph theorem. In *Proceedings of the Second Chapel Hill Conference on Combinatorial Mathematics and its Applications*, pages 171–175, 1969.

[31] D. R. Fulkerson. Blocking and anti-blocking pairs of polyhedra. *Mathematical Programming*, 1:168–194, 1971.

[32] D. R. Fulkerson. Anti-blocking polyhedra. *Journal of Combinatorial Theory, Series B*, 12:50–71, 1972.

[33] D. R. Fulkerson. *On the perfect graph theorem*, pages 68–76. Academic Press, New York, 1973.

[34] M. R. Garey and D. S. Johson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, USA, 1979.

[35] M. d. l. L. Gasca Soto. *Introducción a los Problemas NP-Completos*. Number 19 in Notas de clase. UNAM, Facultad de Ciencias, 2003.

[36] M. Gendreau, J. C. Picard, and L. Zubieta. An Efficient Implicit Enumeration Algorithm for the Maximum Clique Problem. In A. K. et ál., editor, *Lecture Notes in Economics and Mathematical Systems*, volume 304, pages 70–91. Springer-Verlag, 1988.

[37] J. Gil Mendieta and S. Schmidt, editors. *Análisis de Redes. Aplicaciones en Ciencias Sociales*. Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas de la Universidad Nacional Autónoma de México, 2002.

[38] M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, 1980.

[39] A. Hajnal and J. Suranyi. Über die auflösung von graphen in vollständige teilgraphen. *Annales Universitatis Sci. Budapestiensis*, 1:113–121, 1958.

[40] F. Harary and I. C. Ross. A Procedure for Clique Detection Using the Group Matrix. *Sociometry*, 20:205–215, 1957.

[41] M. d. C. Hernández Ayuso. *Introducción a la programación lineal*. Prensas de ciencias. UNAM, Facultad de Ciencias, 2007.

[42] D. J. Houck. *On the Vertex Packing Problem, Ph.D. dissertation.* PhD thesis, The Johns Hopkins University, Baltimore, 1974.

[43] D. J. Houck and R. R. Vemuganti. An Algorithm for the Vertex Packing Problem. *Operations Research*, 25:773–787, 1977.

[44] A. Jagota. Efficiently Aproximating Max-Clique in a Hopfield-style Network. *In Inter. Joint. Conf. on Neural Networks*, 2:256–278, 1992.

[45] A. Jagota and K. W. Regan. Performance of Max-Clique Approximation Heuristics under Description-Length Weighted Distributions. Technical report, Department of Computer Science, State University of New York at Buffalo, 1992.

[46] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On Generating All Maximal Independent Sets. *Information Processing Letters*, 27:119–123, 1988.

[47] R. M. Karp. Reducibility Among Combinatorial Problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.

[48] P. Kikusts. Another Algorithm Determining the Independence Number of a Graph. *Elektron. Inf. Verarb. Kybern.*, ELK 22:157–166, 1986.

[49] R. Kopf and G. Ruhe. A Computational Study of the Weighted Independent Set Problem for General Graphs. *Fundations of Control Engineering*, 12:167–180, 1987.

[50] A. H. Land and A. G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrika*, 28:497–520, 1960.

[51] E. Loukakis and C. Tsouros. A Depth First Search Algorithm to Generate the Familiy of Maximal Independent Sets of a Graph Lexicographically. *Computing*, 27:249–266, 1981.

[52] L. Lovász. A characterization of perfect graphs. *Journal of Combinatorial Theory, Series B*, 13(2):95–98, 1972.

[53] L. Lovász. Normal hypergraphs and the perfect graph conjecture. *Discrete Mathematics*, 2(3):253–267, 1972.

[54] K. Maghout. Sur la Determination des Nombres de Stabilité et du Nombre Chromatique d un Graphe. *C.R. Acad.Sci. Paris*, 248:2522–2523, 1959.

[55] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Publishing Company Inc., USA, 1989.

[56] P. M. Marcus. Derivation of Maximal Compatibles Using Boolean Algebra. *IBM J. Res. Develop.*, 8:537–538, 1964.

[57] K. G. Murty. *Operations Research: Deterministic Optimization Models*. Prentice Hall PTR, 1995.

[58] K. C. y. L. J. D. C. Murty K. G. *The traveling salesman problem: Solution by a method of ranking assignments*. Cleveland: Case Institute of Technology, 1962.

[59] H. Müller-Merbach. Heuristic and their Design: A Survey. *European Journal of Operation Research*, 8:1–23, 1981.

[60] T. Nicholson. *Optimization in Industry*, volume 11. Longman Press, 1971.

[61] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Dover Publications, 1998.

[62] P. M. Pardalos and A. T. Phillips. A Global Optimization Approach for Solving the Maximum Clique Problem. *International Journal of Computer Mathematics*, 33:209–216, 1990.

[63] P. M. Pardalos and G. P. Rodgers. A Branch and Bound Algorithm for the Maximum Clique Problem. *Computers and Operations Research*, 19(5):363–375, 1992.

[64] P. M. Pardalos and J. Xue. The Maximum Clique Problem. *Journal of Global Optimization*, 4:301–328, 1994.

[65] M. Paull and S. H. Unger. Minimizing the Number of States in Incompletely Specified Sequential Switching Functions. *IRE TRANS. Electronic Computers*, EC-8:356–367, 1959.

[66] G. Pólya, editor. *How to solve it: a new aspect of mathematical method*. Princeton University Press, 1945.

[67] J. Ramanujam and Sadayappanv. Optimization by Neural Networks. volume II, pages 325–332, San Diego, 1988.

[68] E. Rich and K. Knight. *Inteligencia Artificial*. McGraw Hill, 1994.

[69] J. M. Robson. Algorithms for Maximum Independent Sets. *Journal of Algorithms*, 7:425–440, 1986.

[70] D. Rose, R. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

[71] S. S. Skiena. *The Algorithm Design Manual*. Telos, Santa Clara, California, 1997.

[72] R. Tarjan. Finding a Maximum Clique, 1972.

[73] R. E. Tarjan and A. E. Trojanowski. Finding a Maximum Independent Set. *SIAM Journal on Computing*, 6:537–546, 1977.

[74] E. Tomita, Y. Kohata, and H. Takahashi. A Simple Algorithm for Finding a Maximum Clique. Technical report uec-tr-c5, Dept. of Communications and Systems Engineering, Univ. of Electro communications, 1988.

[75] E. Tomita, A. Tanaka, and H. Takahashi. The Worst-Case Time Complexity for Finding All the Cliques. *Technical Report UEC-TR-C5*, 1988.

[76] S. Tsukiyama, M. Ide, H. Aviyoshi, and I. Shirakawa. A New Algorithm for Generating All the Maximum Independent Sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[77] D. B. West. *Introduction to graph theory*. Prentice Hall, 2001.

[78] D. R. Wood. An Algorithm for Finding Maximum Clique in a Graph. *Operations Research Letters*, 21:211–217, 1997.

[79] J. Xue. *Fast Algorithms for Vertex Packing and Related Problems.* Ph. d. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[80] H. Zanakis, J. Stelios, and A. Vazacopoulos. Heuristic Optimization: Why, When and How to Use It. *Interfaces*, 11(5):84–91, 1981.