UNIVERSIDAD AUTÓNOMA
METROPOLITANA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

Casa abierta al tiempo

SOBRE LA COMPLEJIDAD TEMPORAL DE LOS ALGORITMOS QUE BUSCAN CLANES

Tesis que presenta
Ismael Ariel Robles Martínez
Para obtener el grado de
Maestro en Ciencias (Matemáticas)

Asesor: Dr. Miguel Angel Pizaña

Jurado Calificador:

Presidente:	Dr. Miguel Angel Pizaña
Secretario:	Dr. Bernardo LLano Pérez
Vocal:	Dr. David Flores Peñaloza

México, D.F. Junio del 2016

UNIVERSIDAD AUTÓNOMA
METROPOLITANA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

SOBRE LA COMPLEJIDAD TEMPORAL DE LOS ALGORITMOS QUE BUSCAN CLANES

Tesis que presenta
Ismael Ariel Robles Martínez
Para obtener el grado de
Maestro en Ciencias (Matemáticas)

Asesor: Dr. Miguel Angel Pizaña

Jurado Calificador:

Presidente: Dr. Miguel Angel Pizaña

Secretario: Dr. Bernardo LLano Pérez

Vocal: Dr. David Flores Peñaloza

México, D.F. Junio del 2016

SOBRE LA COMPLEJIDAD TEMPORAL DE LOS ALGORITMOS QUE BUSCAN CLANES

Universidad Autónoma Metropolitana
Ismael Ariel Robles Martínez
Posgrado en Matemáticas

Para mi familia

Que a pesar de nuestras diferencias,
siempre nos hemos apoyado.

"Toda nuestra ciencia, comparada con la realidad,
es primitiva e infantil... y sin embargo
es lo más preciado que tenemos."

Albert Einstein (1879-1955)

AGRADECIMIENTOS

Quiero agradecer principalmente a mi madre, por tantos años de esfuerzo desinteresado para proveerme de una educación y por haberme guiado para perseguir una carrera científica. Gracias por haberme inculcado desde una temprana edad, el pensamiento critico y un profundo respeto por el quehacer científico.

A mis hermanos, por haber sido parte de mi formación como persona y por compartir muchos de mis intereses sobre sobre computación y Ciencia ficción.

A Daniela y toda su familia, por haberme apoyado en los momentos más difíciles de mi vida y por las charlas tan interesantes que se extendían hasta la madrugada sobre Ciencia, Política y Filosofía.

A mis compañeros de la maestría, por haber compartido esta etapa de nuestras vidas que antecede a la búsqueda de la verdad y del conocimiento.

A mi asesor de Tesis el Dr. Miguel Angel Pizaña, por todo su apoyo y dedicación para el desarrollo de este trabajo. También quiero agradecer sus múltiples consejos de desarrollo profesional que me han sido muy útiles en mi búsqueda por una carrera en la investigación de las Matemáticas.

A la coordinadora del posgrado la Dra. Patricia Saavedra Barrera y a la Mtra. Iseo Gonzáles Christen, por su atención a lo largo de la maestría.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT), por su apoyo y patrocinio para mis estudios de maestría, los cuales concluyen con la realización de este proyecto de tesis.

Y finalmente a todos los científicos que con su curiosidad y pasión por la verdad, contribuyen al desarrollo de la humanidad.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1

INTRODUCCIÓN

Un clan es una subgráfica completa maximal. El problema de decisión de encontrar un clan de tamaño k en una gráfica (problema Clique), es uno de los 21 problemas NP-completos clásicos de Karp [14]. El simple hecho de que este problema sea NP-completo le da una importancia esencial para el estudio de la complejidad temporal, ya que una de las preguntas abiertas más importantes en la teoría de la complejidad computacional, es la de encontrar un procedimiento de decisión que pueda resolver a un problema NP-completo en tiempo polinomial.

Existe la creencia bastante generalizada, de que probablemente no existe un problema NP-completo que puede ser resuelto por un procedimiento de decisión en tiempo polinomial. Sin embargo, aún cuando se descubriera que sí existe un problema con tales características, existen otros problemas de decisión relacionados con la búsqueda de clanes, que intuitivamente son más difíciles que el problema Clique; tal es el caso del problema MClique [14], el cual consiste en decidir si una gráfica tiene un clan máximo de tamaño k.

Además de los problemas de decisión antes mencionados, es bien conocido que el problema de buscar todos los clanes en una gráfica, en el peor de los casos, requiere tiempo exponencial. A este problema de búsqueda en este trabajo le hemos denominado LClique. Aunque el problema LClique no es un problema de decisión (razón por la cual formalmente no podemos compararlo con otros problemas de decisión), intuitivamente podemos pensar que es todavía más difícil que el problema MClique.

Existen múltiples aplicaciones para los problemas LClique, MClique y Clique, que van desde el estudio de redes sociales de millones de personas, hasta el estudio de estructuras comunes en moléculas de la Química básica [3, 8-10, 16, 26]. Es por ello que en este trabajo, nos hemos centrado en estudiar a fondo dichos problemas de búsqueda de clanes, así como las demostraciones clásicas de su pertenencia a las clases de complejidad NP-completo y NP-difícil. También hemos puesto mucho énfasis en estudiar uno de los algoritmos más utilizados para buscar clanes en una gráfica; el algoritmo de Bron-Kerbosch.

En la práctica, múltiples resultados experimentales muestran que en la mayoría de los casos, el algoritmo de Bron-Kerbosch es más rápido que cualquier otro algoritmo conocido [3, 9, 29]. Aunque se sabe que para una gráfica con n vértices, la complejidad del algoritmo de Bron-Kerbosch es $\mathcal{O}(3^{n/3})$ [29], no existe una cota de complejidad que esté en función de otros parámetros que permitan compararlo en casos de gráficas específicas. Esto hace particularmente difícil hacer una comparativa formal de la complejidad temporal del algoritmo de Bron-Kerbosch con otros algoritmos cuya complejidad temporal está en función de vértices y aristas, como es el caso del algoritmo TIAS [30]. Por esta razón, como parte de este trabajo se investigó una propuesta para la cota de complejidad temporal del algoritmo de Bron-Kerbosch, en función de vértices y aristas. Algunas conjeturas que tenemos al momento, se revisan en los capítulos 3 y 4 de este trabajo.

El contenido de esta tesis esta distribuido de la siguiente forma:

- En el capitulo 1 estudiamos definiciones y teoremas básicos de teoría de las gráficas y de complejidad computacional que son esenciales para entender el contenido del resto de los capítulos. En particular, se pone especial énfasis en el estudio de los modelos de la máquina de Turing, ya que son fundamentales para poder enunciar formalmente el concepto de función de complejidad temporal. También revisamos detalladamente los conceptos de transformación polinomial (reducción de Karp) y reducción de Turing (reducción de Cook). También estudiamos varios lemas y teoremas esenciales relacionados con las siguientes clases de complejidad: P, NP, co-NP, NP-completo y NP-difícil. Se revisan algunos de los 21 problemas clásicos de la lista de Karp y su respectiva prueba de que son NP-completos.

- En el capítulo 2 se revisan las definiciones de los problemas Clique, MClique y LClique. Se estudian a fondo las demostraciones de que Clique es NP-Completo y MClique es NP-difícil. Del problema LClique, primero estudiamos la demostración de que el mayor número de clanes que puede tener una gráfica es $3^{n/3}$ y posteriormente se demuestra que la gráfica de Moon-Moser alcanza esta cota.

- En el capitulo 3, se estudian las propiedades del algoritmo de Bron-Kerbosch en las versiones propuestas por J. Koch en [16]. También se estudia el algoritmo TTT que se propone en [29], así como la demostración de que dicho algoritmo es equivalente al algoritmo de Bron-Kerbosch y se demuestra que su complejidad temporal es $\mathcal{O}(3^{n/3})$. En las últimas secciones del capítulo 3, se demuestran cotas de complejidad para algunas familias de gráficas básicas que obtuvimos durante el desarrollo de este proyecto de maestría y se enuncian las

conjeturas que tenemos sobre la forma que puede tener una cota de complejidad temporal para el algoritmo de Bron-Kerbosch, en función de n vértices y m aristas.

- En el capítulo 4 se muestran varios resultados experimentales que confirman las cotas que se calcularon en el capítulo 3. También se resumen varios resultados experimentales que obtuvimos durante este trabajo y que sirven de evidencia para las conjeturas que presentamos en el capítulo 3.

2

PRELIMINARES

En este capítulo se resumen definiciones y teoremas básicos de teoría de las gráficas y de complejidad computacional. Se ha puesto especial atención en incluir únicamente aquellos temas que son necesarios para una mejor comprensión de este trabajo [1].

2.1 CONCEPTOS BÁSICOS DE TEORÍA DE LAS GRÁFICAS

gráfica

Una *gráfica* G es un par ordenado de conjuntos (V, E) donde E es una colección de subconjuntos de 2 elementos de V, esto es

$$E \subseteq \{\, \{u, v\} : u, v \in V \,\}.$$

vértices *aristas*

Para este trabajo se asumirá en todo momento que V y E son conjuntos finitos. A los elementos del conjunto V se les denomina *vértices* o nodos de la gráfica G y a los elementos del conjunto E se les denomina *aristas* de la gráfica G. En este trabajo usaremos las notaciones $V(G)$ y $E(G)$ para referirnos a los conjuntos de aristas y vértices de una gráfica G respectivamente.

orden *tamaño*

Si $\{u, v\}$ es una arista de $E(G)$, por simplicidad denotaremos a dicha arista como uv. Se dice que los vértices u y v son *adyacentes* si la arista uv está en $E(G)$. El *orden* de una gráfica G es el número de vértices (cardinalidad de $V(G)$) y el *tamaño* de una gráfica G es el total de aristas que tiene (cardinalidad de $E(G)$).

complemento

El *complemento* de la gráfica $G = (V, E)$, es la gráfica $\overline{G} = (V, E')$ tal que $e \in E'$ si y solo sí $e \notin E$. En este trabajo, en ocasiones, usaremos la notación G^c para referirnos a $\overline{G}$.

unión *suma de Zykov*

Si $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, la *unión* de las gráficas G_1 y G_2 es la gráfica $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. La *suma de Zykov* $G_1 + G_2$ es la gráfica que resulta de hacer la unión de G_1 y G_2 y añadir toda posible arista de los vértices de G_1 a los vértices de G_2.

[1] Si el lector desea profundizar en los tópicos de este capítulo, recomendamos revisar los libros [1, 2] para teoría de las gráficas y [11, 13, 28] para teoría de la complejidad computacional

Se dice que la gráfica $G' = (V', E')$ es una *subgráfica* de la gráfica $G = (V, E)$ si $V' \subseteq V$ y $E' \subseteq E$. Por otra parte, si para cada par de vértices u y v de V', se tiene que uv es una arista de G' si y solo si uv es una arista de G, entonces se dice que G' es una *subgráfica inducida* por V' y se denota por $G[V']$. En este trabajo usaremos la notación $G' \subseteq G$ para indicar que G' es una subgráfica de G.

subgráfica

Se define a la *vecindad abierta* del vértice v como el conjunto de todos los vértices adyacentes a v y se denota por $N_G(v)$. La *vecindad cerrada* se define como $N_G[v] = N_G(v) \cup \{v\}$. Cuando resulte claro de qué gráfica G se está hablando, omitiremos el subíndice y simplemente escribiremos $N(v)$ y $N[v]$ para referirnos a las vecindades abierta y cerrada respectivamente.

vecindad abierta
vecindad cerrada

Si v es un vértice de G, el *grado* $d_G(v)$ de v se define como $d_G(v) = |N_G(v)|$. Cuando resulte claro en el contexto, usaremos simplemente la notación $d(v)$ en lugar de $d_G(v)$. El *grado mínimo* de G se denota por $\delta(G)$ y el *grado máximo* de G se denota por $\Delta(G)$. Cuando resulte claro de que gráfica G se está hablando, usaremos δ y Δ para el grado mínimo y máximo respectivamente.

grado
grado mínimo
grado máximo

Un *camino* es una sucesión $v_0e_1v_1e_2v_2 \cdots v_{n-1}e_nv_n$ de vértices v_i y aristas e_i tal que para $1 \leqslant i \leqslant n$ se cumple que $e_i = \{v_{i-1}, v_i\}$. Normalmente para describir al camino se omiten las aristas y se deja solo la sucesión de vértices $v_0v_1v_2 \cdots v_{n-1}v_n$ o equivalentemente se usa la notación $v_0 - v_n$. La *longitud* de un camino es el total de aristas que tiene la sucesión.

camino

Una *trayectoria* es un camino $v_0 - v_n$ en el que todos los vértices de la sucesión son diferentes. Si solo los vértices v_0 y v_n son iguales, se dice que $v_0 - v_n$ es un *ciclo*. Usaremos las notaciones P_n y C_n para representar a las gráficas de n vértices que tienen una sola trayectoria y un solo ciclo respectivamente. Por lo general, cuando no cause ambigüedades, nos referiremos a dichas familias de gráficas como *trayectorias* y *ciclos*.

trayectoria
ciclo

Decimos que una gráfica es *conexa* si para cada par de vértices u y v existe una trayectoria de u a v. Un *árbol* es una gráfica conexa en la que existe una y solo una trayectoria entre cada par de vértices. Una *hoja* de un árbol es un vértice v con grado $d(v) = 1$.

gráfica conexa
árbol

Si T es un árbol, en ocasiones es conveniente elegir un vértice $r \in V(T)$ y denotar a dicho vértice como *nodo raíz*, en tal caso se dice que T *es un árbol con raíz*. La *profundidad de un vértice* $v \in T$ es la longitud de la trayectoria $r - v$. Por convención, el nodo raíz tiene profundidad cero. La *altura de un vértice* v es la mayor de las longitudes sobre todas las trayectorias que van del nodo v a una hoja y que incluyen únicamente vértices de una altura igual o mayor que v. La *altura de un árbol* es la altura del nodo raíz. El *vértice*

árbol con raíz
profundidad
altura
vértice padre

vértice hijo

padre de v, es un vértice que es adyacente a v y que tiene profundidad menor que v. Un *vértice hijo* de v, es un vértice que es adyacente a v y que tiene profundidad mayor que v.

En este trabajo haremos uso de los siguientes resultados para árboles con raíz con bastante frecuencia.

Lema 2.1.1. *Sea* T *un árbol con raíz* r *y altura* h *para el cual se cumple que todo vértice que no es una hoja tiene exactamente* k *hijos. Si* $d \leqslant h$*, entonces el número total de vértices con profundidad* d *es* k^d*.*

Demostración. La prueba se hará por inducción sobre la profundidad d. Si $d = 1$, los únicos nodos con profundidad 1 son los k hijos del nodo raíz r.

Suponga que la afirmación es cierta para $d < h$, mostraremos que la afirmación es cierta para $d + 1$.

Por la hipótesis de inducción hay k^d vértices con profundidad d y como cada uno tiene k hijos, el total de vértices con profundidad $d + 1$ es k^{d+1}. □

Teorema 2.1.1. *Sea* T *un árbol con raíz* r *tal que todo vértice que no es una hoja tiene exactamente* $k > 1$ *hijos. Sea* b *el total de hojas de* T*, entonces* $|V(T)| < 2b$

Demostración. Sea h la altura del árbol T. Por el lema 2.1.1, se tiene que $b = k^h$. La cardinalidad del conjunto $V(T)$ se obtiene sumando todos los vértices con profundidad $i \leqslant h$:

$$\begin{aligned} |V(T)| &= \sum_{i=0}^{h} k^i = \sum_{i=0}^{h-1} k^i + k^h = \frac{k^h - 1}{k-1} + b \\ &\leqslant k^h - 1 + b < k^h + b = 2b \end{aligned}$$

□

gráfica completa

Una *gráfica completa* $G = (V, E)$ es una gráfica en la que cada vértice $v \in V$ es adyacente a cada uno de los vértices de $V \setminus \{v\}$. Una gráfica completa de orden n se denota por K_n. Observe que una gráfica completa es una gráfica conexa y tiene un tamaño de $|E| = \frac{n(n-1)}{2}$.

clan

Un *clan* Q de una gráfica G es una subgráfica completa maximal. Denotaremos por $K(G)$ a el conjunto de todos los clanes de la gráfica G. De manera

similar, se dice que un subconjunto de vértices $I \subseteq V(G)$ es *independiente* si ningún par de vértices en I son adyacentes. Denotaremos por $MIS(G)$ a la colección de todos los conjuntos independientes maximales de G.

conjunto independiente

Una *gráfica bipartita* es una gráfica $G = (V, E)$ en la que su conjunto de vértices V puede ser dividido en dos subconjuntos disjuntos V_1 y V_2, tal que cada arista de G tiene un vértice en V_1 y V_2. Si $|V_1| = n$ y $|V_2| = m$ y cada vértice de V_1 es adyacente a todo vértice de V_2, se dice que G es una *gráfica bipartita completa* y se denota por $K_{n,m}$. Una *gráfica estrella* es una gráfica bipartita completa $K_{1,n}$ y se denota por S_n.

gráfica bipartita

gráfica bipartita completa

gráfica estrella

Puesto que en esta tesis se estudia principalmente la complejidad temporal de los algoritmos que listan todos los clanes de una gráfica G, en la siguiente sección se resumen conceptos básicos de complejidad computacional que se usarán frecuentemente en este trabajo.

2.2 MODELOS BÁSICOS DE LA MÁQUINA DE TURING

Por mucho tiempo, se daba por sentado que un algoritmo es simplemente un conjunto finito de pasos que deben de ser ejecutados en orden a fin de resolver un problema determinado. Fue hasta 1928 que a raíz de que David Hilbert formuló el *Entscheidungsproblem* [12, p.112-124] que resultó necesario formalizar el concepto de algoritmo. Dicho problema planteado por Hilbert, pregunta si es posible construir un algoritmo que decida si una formula del calculo de primer orden es universalmente válida.

Entscheidungsproblem

La respuesta negativa a la pregunta planteada por el Entscheidungsproblem fue demostrada en 1936 por *Alonzo Church* [6] y poco después de manera independiente por *Alan Turing* [31]. En sus trabajos se incluyen las primeras definiciones formales de lo que se entiende por algoritmo, aunque ambas son equivalentes, la definición más utilizada es la que propuso Alan Turing, la cual se desprende de su modelo de computación. En la actualidad a este modelo se le denomina máquina de Turing.

Alonzo Church

Alan Turing

Existen múltiples variantes de la máquina de Turing, todas equivalentes. En este trabajo centraremos nuestra atención en 4 modelos que son necesarios para desarrollar conceptos y teoremas básicos de la teoría de la complejidad computacional:

1. La máquina de Turing de 1 cinta.
2. La máquina de Turing de múltiples cintas.

modelos de la máquina de Turing

3. La máquina de Turing no determinista.
4. La máquina de Turing con oráculo.

alfabeto En la teoría de la computación, un *alfabeto* Σ es cualquier conjunto finito, *símbolos* a los elementos de Σ se les llama *símbolos* del alfabeto. Una sucesión fi- *cadena* nita de símbolos de Σ se denomina *cadena*. Si w es una cadena sobre el *longitud* alfabeto Σ, la *longitud* de w es el número de símbolos que tiene w y se denota por $|w|$. Sean $w_1 = x_1x_2\cdots x_m$ y $w_2 = y_1y_2\cdots y_n$ cadenas so- *concatenación* bre el alfabeto Σ, la *concatenación* de las cadenas w_1 y w_2 es la cadena $w_1w_2 = x_1x_2\cdots x_my_1y_2\cdots y_n$.

Σ^* Denotaremos por Σ^* al conjunto de todas las cadenas que se pueden for- *cadena vacía* ϵ mar sobre un alfabeto Σ, *incluyendo a la cadena vacía* ϵ (cadena sin nin- Σ^+ gún símbolo). Denotaremos por Σ^+ a el conjunto de todas las cadenas que se pueden formar sobre Σ, *excluyendo* a la cadena vacía, observe que *lenguaje* $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. Un *lenguaje* es cualquier subconjunto de Σ^*.

Los modelos de la máquina de Turing son fundamentales para el estudio formal de lenguajes. Cada modelo de computación tiene como objetivo principal, determinar si una cadena es parte de un determinado lenguaje de computación.

2.2.1 *Máquina de Turing*

Podemos visualizar de manera intuitiva a la máquina de Turing por me- *cinta* dio de la figura 1. La máquina consta de una *cinta* dividida en *casillas* que *casillas* se extienden a la izquierda y a la derecha infinitamente. La máquina tam- *unidad de control* bién tiene una *unidad de control*, que a su vez tiene un *cabezal* que en todo *cabezal* momento señala a una de las casillas de la cinta.

El cabezal puede leer y escribir (o sobrescribir) un *símbolo* en cada casilla, dicho símbolo se toma de un conjunto finito de símbolos previamente da- *alfabeto de cinta* do, el cual se denomina *alfabeto de cinta*. Después de que el cabezal a escrito o leído un símbolo, este debe moverse sobre la cinta en dirección izquierda o derecha.

estado En todo momento la unidad de control tiene asociado un *estado*, el cual se toma de un conjunto finito que contiene todos los estados posibles de la unidad de control. Con cada movimiento del cabezal, la unidad de control cambia de estado.

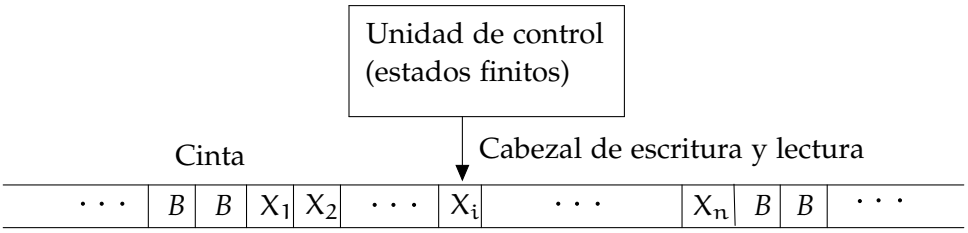


Figura 1: Representación de una máquina de Turing.

Antes de que inicie la máquina, se puede escribir en la cinta un sucesión finita de símbolos que se conocen como *cadena de entrada*, dichos símbolos se toman de un conjunto finito denominado *alfabeto de entrada*. Las casillas restantes de la cinta inicialmente almacenan un símbolo especial llamado *espacio en blanco*, que denotaremos por B. El espacio en blanco es un símbolo del alfabeto de cinta pero no es un símbolo del alfabeto de entrada.

cadena de entrada
alfabeto de entrada
espacio en blanco

Al arrancar la máquina, el cabezal señala a la casilla que contiene el símbolo más a la izquierda de la cadena de entrada. El comportamiento de la máquina está determinado por un conjunto finito de instrucciones que se le conoce como *programa*. Diferentes programas producirán una sucesión de *movimientos* diferentes en la máquina. Un *movimiento* de la máquina comprende las siguientes tareas:

programa
movimiento de la máquina de Turing

1. Dependiendo del estado en el que se encuentre la unidad de control y del símbolo que lee el cabezal, este último escribirá un símbolo del alfabeto de cinta en la casilla que señala, el cual remplaza al símbolo actual. Es posible que el símbolo a escribir sea el mismo símbolo que había previamente en la casilla.
2. La máquina moverá el cabezal *una* casilla hacia la izquierda o hacia la derecha.
3. La unidad de control cambiará de estado. Es posible que el siguiente estado sea el mismo que el estado actual.

La definición formal de la máquina de Turing puede variar ligeramente dependiendo del autor, en este trabajo usaremos la definición que se describe en [28, p. 139], en dicha definición se dice que la máquina se *detiene* o *termina* si entra en el estado q_A o q_R.

Si la máquina termina en el estado q_A, se dice que la cadena de entrada fue *aceptada* y si la máquina termina en el estado q_R, se dice que la cadena de entrada fue *rechazada*. Si la máquina nunca termina en el estado q_A o q_R, entonces la máquina nunca se detiene, en cuyo caso decimos que la máquina está en un *ciclo infinito* o *en un ciclo por tiempo indefinido*.

cadena aceptada
cadena rechazada
ciclo infinito

Teniendo en cuenta la descripción intuitiva de la máquina de Turing de los párrafos anteriores, procederemos a enunciar la definición formal de la misma.

Definición 2.2.1. *(Sipser [28, p. 139]) Una máquina de Turing* M *es una séptupla de la forma*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R),$$

Q, Σ *y* Γ *son conjuntos finitos. Los componentes de* M *tienen el siguiente significado:*

1. Q *son los* estados *en los que se puede encontrar la unidad de control.*
2. Σ *es el* alfabeto de entrada.
3. Γ *es el* alfabeto de cinta. *Se debe de cumplir que* $\Sigma \subsetneq \Gamma$, *además el espacio en blanco* B *está en* Γ *pero no pertenece a* Σ *(*B *no es un símbolo de entrada).*
4. δ *es la* función de transición, *la cual es de la forma:*

   $$\delta : (Q \setminus \{q_A, q_R\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

   Los argumentos de $\delta(q, X)$ *son un estado* q *y un símbolo de cinta* X. *El valor de* $\delta(q, X)$ *es la terna* (p, Y, D), *donde:*

   a) p *es el siguiente estado de la máquina.*
   b) Y *es símbolo del alfabeto de cinta* Γ *que se escribe en la casilla que señala el cabezal.*
   c) D *es la dirección en que el cabezal debe moverse* después *de haber escrito el símbolo* Y. *Solo puede moverse una casilla en dirección* izquierda *(L) o* derecha *(R).*
5. $q_0 \in Q$ *es el* estado inicial, *esto es, el estado en el que se encuentra la unidad de control al arrancar la máquina.*
6. $q_A \in Q$ *es el* estado de aceptación. *Cuando la unidad de control entra en este estado se dice que la máquina* aceptó *a la cadena de entrada.*
7. $q_R \in Q$ *es el* estado de rechazo. *Cuando la unidad de control entra en este estado se dice que la máquina* rechazó *a la cadena de entrada.*

Observación 2.2.1. *El* programa *de una máquina de Turing está determinado por la función de transición.*

ejemplo máquina de Turing

A manera de ejemplo la figura 2 describe los componentes de una máquina de Turing muy simple.

Para ilustrar los movimientos de la máquina de la figura 2, considere la cadena de entrada *10110*. La figura 3 muestra por cada renglón un movimiento de la máquina y el efecto que tiene en los símbolos en la cinta. En

$$\Gamma = \{0, 1, B\},\ \Sigma = \{0, 1\}$$

$$Q = \{q_0, q_1, q_A, q_R\}$$

$\delta(q, s)$			
Estados	0	1	B
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	(q_A, B, L)	(q_R, B, L)	(q_R, B, L)

Figura 2: Ejemplo de una máquina de Turing.

cada renglón el cabezal se representa con un triángulo encima de la cinta y el estado en el que se encuentra la unidad de control se muestra a la izquierda. El último renglón de la figura 3 muestra que la máquina termina aceptando a la cadena 10110.

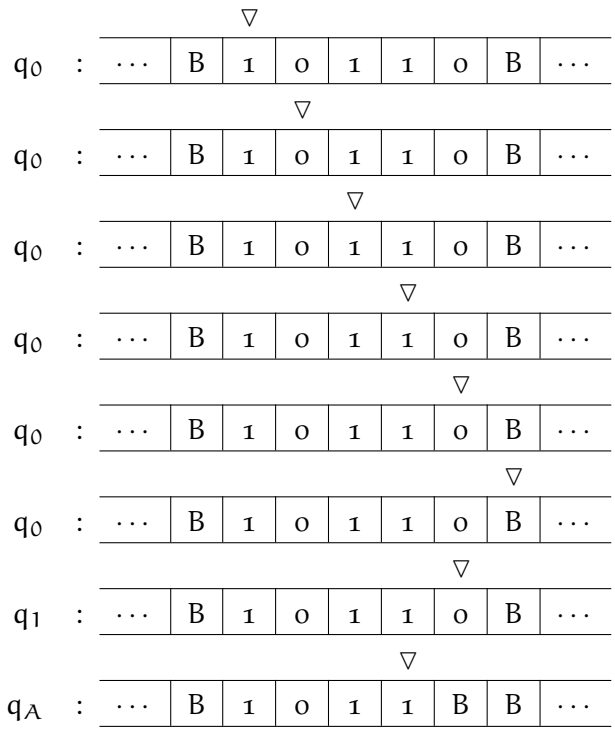


Figura 3: Movimientos de la máquina de Turing descrita en la figura 2, cuando la cadena de entrada es 10110.

La relación que hay entre lenguajes y máquinas de Turing resulta de la siguiente definición.

lenguaje reconocido por una máquina de Turing

Definición 2.2.2. *Sea* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$ *una máquina de Turing, el* lenguaje aceptado *o* reconocido *por* M *se denota por* L_M *y se define como*

$$L_M = \{w \in \Sigma^* : w \text{ es una cadena de entrada que es aceptada por } M\}.$$

No es difícil demostrar que la máquina de la figura 2 acepta aquellas cadenas cuyo símbolo más a la derecha es cero, por lo tanto la cadena *10110* está en lenguaje reconocido por la máquina de la figura 2.

Puesto que no necesariamente una máquina de Turing se detiene con cada cadena de entrada, es conveniente hacer una distinción entre aquellas máquinas que siempre se detienen para cualquier cadena de entrada y las que no.

máquina de Turing total

Definición 2.2.3 (Kozen [17, p. 213])**.** *Se dice que una máquina de Turing es* total *si se detiene para cualquier cadena de entrada.*

También es importante distinguir entre los lenguajes que son reconocidos por una máquina de Turing y aquellos para los que existe una *máquina de Turing total* que los reconoce, esto nos lleva al siguiente concepto.

lenguaje Turing decidible

Definición 2.2.4. *Sea* $L \subseteq \Sigma^*$ *un lenguaje con alfabeto* Σ*. Se dice que* L *es un* lenguaje Turing decidible *si existe una máquina de Turing total* M *tal que* $L_M = L$.

algoritmo

Por medio de la definición 2.2.3 podemos enunciar de una manera más formal el concepto de *algoritmo*. Se tiene una gran cantidad de evidencia empírica que muestra que nuestra noción intuitiva de algoritmo, se corresponde con la de un programa que es ejecutado por una máquina de Turing total. A esta tesis se le conoce como *tesis de Church-Turing*.

Tesis de Church-Turing

Una idea fundamental para el estudio de la clase NP-completo (esta clase se estudia en la sección 2.3.3) es asociar una función a una máquina de Turing total, esto se formaliza en la siguientes definiciones.

función asociada a una máquina de Turing total

Definición 2.2.5. *Sea* M *una máquina de Turing total con alfabeto de entrada* Σ *y sea* ϵ *la cadena vacía. Se define la función* $f_M : \Sigma^* \to \Sigma^*$ *como:*

- $f(w) = \epsilon$ *si al detenerse la máquina la cinta contiene únicamente símbolos en blanco* B.

- $f(w) = w'$, *donde* $w' \in \Sigma^+$ *es la cadena no vacía con la longitud más grande que se puede encontrar al leer la cinta de izquierda a derecha, una vez que se ha detenido la máquina.*

Definición 2.2.6. *Se dice que una función* f *es* computable por una máquina de Turing total *si existe una máquina de Turing total* M *tal que* $f = f_M$.

función computable por una máquina de Turing total

Una técnica muy utilizada para definir el programa de una máquina de Turing es la de *guardar información en los estados de la máquina*. Por ejemplo, si q es el estado actual en el que se encuentra la máquina y queremos guardar el símbolo x, podemos definir un estado con etiqueta (q, x) para guardar dicha información. Esto no significa que estamos creando un nuevo estado *después* de que ha iniciado la ejecución de la máquina, sino que *previamente* al definir la función de transición, hemos considerado el caso en el cual la máquina guarda el símbolo x por medio del estado (q, x).

guardar información en los estados de una máquina

La técnica de guardar información en los estados de una máquina, también se puede utilizar para programar cualquiera de los modelos de computación que estudiaremos en las siguientes secciones.

2.2.2 *Máquina de Turing con múltiples cintas*

Como su nombre lo indica, la máquina de Turing con múltiples cintas (o multicinta) puede escribir y leer múltiples cintas. En este modelo, la unidad de control tiene *múltiples cabezales*. Cada cabezal apunta a una cinta en particular, tal como se muestra en la figura 4.

cinta, cabezales y unidad de control de una máquina multicinta

Al igual que en la máquina de Turing de 1 sola cinta, cada cabezal puede moverse en direcciones izquierda (L) y derecha (R); sin embargo, a diferencia de la máquina de Turing en la máquina multicinta cualquier cabezal tiene permitido quedarse en la misma casilla, en este caso se dice que el cabezal tiene dirección estática (S).

direcciones de una máquina de Turing con múltiples cintas

Como es de esperarse, la definición formal de la máquina de Turing multicinta es muy similar a la máquina de Turing con 1 sola cinta.

Definición 2.2.7. *Una* máquina de Turing M con k cintas (multicinta) *es una séptupla de la forma*

máquina de Turing multicinta

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R),$$

donde

- Q, Σ, Γ, q_0, q_A *y* q_R *se definen igual que en la definición 2.2.1.*
- δ *es la* función de transición *y es de la forma:*

$$\delta : (Q \setminus \{q_A, q_R\}) \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k.$$

ejemplo sobre la función de transición δ

A fin de ejemplificar como opera la función de transición δ, sea M una máquina de Turing con k cintas y con componentes

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R).$$

Sean q_i y q_j estados de Q. Sean $(a_1, \ldots, a_k)$ y $(b_1, \ldots, b_k)$ k-tuplas de Γ^k y sea $(D_1, \ldots, D_k)$ una k-tupla de $\{L, R, S\}^k$. Entonces la expresión

$$\delta(q_i, (a_1, \ldots, a_k)) = (q_j, (b_1, \ldots, b_k), (D_1, \ldots, D_k)),$$

indica que si la unidad de control está en el estado q_i y los cabezales de las cintas 1 a la k están leyendo los símbolos desde a_1 hasta a_k, entonces los cabezales deben de reemplazar dichos símbolos por b_1 hasta b_k y moverse en las direcciones D_1 a D_k respectivamente. Hecho esto la unidad de control pasa al estado q_j.

movimiento de una máquina multicinta

Es importante notar que un *movimiento de la máquina multicinta* consta de las siguiente tareas:

1. Actualizar cada una de las casillas que señalan las cabeceras.
2. Mover todas las cabeceras en la dirección que indique la función de transición.
3. Actualizar el estado de la unidad de control.

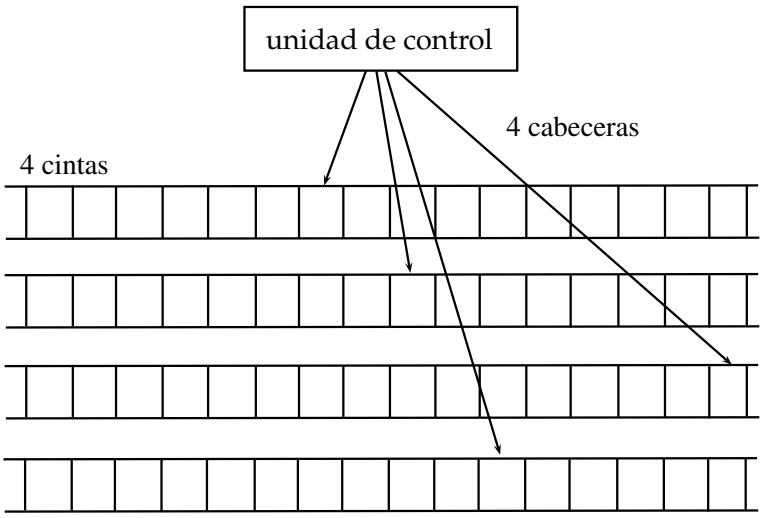


Figura 4: Representación de una máquina de Turing con múltiples cintas.

Puesto que la máquina multicinta puede actualizar múltiples casillas en un solo movimiento, intuitivamente esto nos lleva a pensar que la máquina multicinta es más "rápida" que la máquina de Turing de 1 cinta. Más adelante cuando desarrollemos los conceptos de complejidad temporal regresaremos a estudiar esta idea.

Al arrancar la máquina de Turing multicinta, la *cadena de entrada* se coloca en la primera cinta y el resto de las cintas se dejan en blanco. La máquina se *detiene* cuando la unidad de control entra en el estado q_A o q_R. Si una cadena de entrada hace que la unidad de control termine en el estado q_A se dice que la *cadena fue aceptada*, si la unidad de control termina en el estado q_R se dice que la *cadena fue rechazada*.

cadena aceptada/rechazada por una máquina multicinta

Definición 2.2.8. *Sea* M *una máquina de Turing con múltiples cintas. El* lenguaje aceptado *por* M *se denota por* L_M *y se define como*

$$L_M = \{w \in \Sigma^* : w \text{ es una cadena de entrada que es aceptada por } M\}.$$

lenguaje reconocido por una máquina de Turing multicinta

De la definición 2.2.8, resulta evidente que cualquier lenguaje que es aceptado por una máquina de Turing puede ser aceptado por una máquina de Turing con múltiples cintas. El siguiente teorema muestra que el recíproco también es cierto.

Teorema 2.2.1. *Sea* S *una máquina de Turing con* k *cintas que acepta al lenguaje* L_S*, entonces existe una máquina de Turing* M *tal que* $L_M = L_S$*.*

Equivalencia entre máquina de Turing y máquina de Turing multicinta

Demostración. Construiremos una máquina de 1 sola cinta M que puede simular los movimientos de la máquina multicinta S. El programa de la máquina M está descrito por las siguientes reglas:

1. Sea $w = a_1 \cdots a_n$ la cadena de entrada que se coloca en la primera cinta de S. Para simular a S con la máquina M colocamos la cadena w en la cinta de entrada de la máquina M. Lo primero que hace la máquina M es reemplazar la cadena w por:

$$w' = I \# \dot{a}_1 \cdots a_n \# \dot{B} \# \dot{B} \# \cdots \# \dot{B} \# F$$

   El reemplazo anterior tiene como finalidad representar las k cintas de S, para esto cada cinta de S y su contenido están representados en M entre 2 símbolos #. Los símbolos I y F representan el inicio de la primera cinta y el final de la última cinta respectivamente. La posición de cada cabezal de S se representa por medio de un símbolo que resulta de colocar un punto sobre el símbolo original de S. Podemos pensar que cada uno de estos puntos son *cabezales virtuales* en M que representan a los cabezales originales de S.

representación de las cintas de S

cabezas virtuales

simulación de un movimiento de S

2. Para simular un movimiento de la función de transición de S, el cabezal de M se coloca en el símbolo I. A continuación el cabezal se mueve en dirección derecha buscando los símbolos que tengan un punto encima, esta información se almacena en los estados de M. El cabezal detiene su movimiento hacia la derecha cuando llega al símbolo F. A continuación la máquina vuelve a mover el cabezal hacia la izquierda (*sin alterar los símbolos de la cinta*) hasta llegar al símbolo I. A toda esta ejecución le llamaremos *primer recorrido*.

   Terminado el primer recorrido, la máquina M vuelve a mover el cabezal en dirección derecha, la información que se obtuvo sobre los símbolos con puntos en el primer recorrido, se usa para actualizar cada símbolo en la cinta de acuerdo a las reglas de la función de transición de S. El cabezal detiene su movimiento hacia la derecha cuando llega al símbolo F, acto seguido la máquina reanuda el movimiento del cabezal hacia la izquierda (*sin alterar los símbolos de la cinta*) hasta llegar al símbolo I. A toda esta ejecución le llamaremos *segundo recorrido*.

   Un solo movimiento de S se simula en M ejecutando el primer y segundo recurrido. La máquina M entra en el estado q_A solo si la simulación de S entra en estado de aceptación. La máquina M entra en el estado q_R solo si la simulación de S entra en estado de rechazo.
3. Al simular un movimiento de S puede ocurrir que al mover algún cabezal virtual en dirección derecha se sustituya alguno de los símbolos {#, I, F} por $\dot{\#}$, esto significa que en la máquina S uno de los cabezales de la cinta se movió a una casilla que no se había leído. Si esto ocurre, la simulación del movimiento de S se pausa y a continuación se ejecuta una subrutina que desplaza una casilla a la derecha todos los símbolos adelante de $\dot{\#}$, acto seguido escribe un símbolo # en la siguiente casilla y sustituye $\dot{\#}$ por $\dot{B}$. Al finalizar esta subrutina la máquina M reanuda la simulación del movimiento de S.

   Una subrutina similar aplica si al mover un cabezal virtual a la izquierda se sustituye alguno de los símbolos {#, I, F} por $\dot{\#}$.

   Puesto que la máquina M termina en un estado de aceptación (o rechazo) si y solo si la máquina S termina en un estado de aceptación (o rechazo), concluimos que $L_M = L_S$

□

Observación 2.2.2. *Sea S una máquina de Turing de múltiples cintas que se detiene para cualquier cadena de entrada. Sea M la máquina de Turing que simula a la máquina S de acuerdo a las reglas descritas en el teorema 2.2.1. Entonces M*

también se detiene para cualquier cadena de entrada, esto es, M *es una máquina de Turing total.*

Más adelante mostraremos cual es el costo en *tiempo*[1] de simular una máquina de Turing multicinta, con la máquina M propuesta en el teorema 2.2.1.

2.2.3 *Máquina de Turing no determinista*

determinismo

Hasta este momento, hemos definido modelos de máquinas de Turing en los que cada paso está *determinado* de manera *única* por el paso anterior. En estos modelos la máquina no puede estar en diferentes estados al mismo tiempo. Es por ello que denominamos a estos modelos como *modelos deterministas*. En un modelo *no determinista*, la máquina *sí* puede estar estar en múltiples estados al mismo tiempo. Esta idea es la base para la máquina de Turing no determinista.

no determinismo

cinta, cabezal, unidad de control y direcciones de una máquina no determinista

Podemos representar visualmente a la máquina no determinista de la misma forma que se hizo para la máquina de Turing de 1 cinta (figura 1), esto es, la máquina cuenta con una *cinta*, una *unidad de control* y un *cabezal de escritura/lectura* que se mueve en *direcciones* izquierda (L) o derecha (R).

La diferencia que tiene la máquina no determinista con respecto al modelo estudiado en la sección 2.2.1, es que por cada movimiento la máquina no determinista puede existir en múltiples estados, cada uno de los cuales resulta de una decisión diferente que tomó la máquina al leer un determinado símbolo de la cinta. El funcionamiento de la máquina resultará más claro después de estudiar la siguiente definición.

máquina de Turing no determinista

Definición 2.2.9. *Una* máquina de Turing no determinista M *es una tupla de la forma*

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

donde

- Q, Σ, Γ, q_0 *y* q_A *se definen igual que en la definición 2.2.1.*

[1] Por ahora de manera intuitiva, pensaremos que el tiempo es el número de movimientos del modelo de computación, más adelante en la sección 2.3 damos la definición formal de este concepto.

- δ *es la* función de transición *y es de la forma*

$$\delta : (Q \setminus \{q_A\}) \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}),$$

donde $\mathcal{P}(Q \times \Gamma \times \{L, R\})$ *es el conjunto potencia de* $Q \times \Gamma \times \{L, R\}$.

A fin de ilustrar el uso de la función de transición delta δ en la definición 2.2.9, sean $D_1, \ldots, D_n$ direcciones en $\{R, L\}$. Suponga que la máquina no determinista N se encuentra en el estado q y que el cabezal lee el símbolo x. Si δ tiene el valor

$$\delta(q, x) = \{ (q_1, x_1, D_1), (q_2, x_2, D_2), \ldots, (q_m, x_m, D_m) \},$$

movimiento de una máquina de Turing no determinista

entonces un *movimiento de la máquina* N hará que la máquina se encuentre al *mismo tiempo* en m posibles estados diferentes:

- En una de estas posibilidades la máquina sustituye el estado q y el símbolo x por el estado q_1 y el símbolo x_1 respectivamente y termina moviendo el cabezal en dirección D_1.
- En otra posibilidad la máquina sustituye el estado q y el símbolo x por el estado q_2 y el símbolo x_2 respectivamente y termina moviendo el cabezal en dirección D_2.
- Así sucesivamente llegamos a la posibilidad en la que la máquina sustituye el estado q y el símbolo x por el estado q_m y el símbolo x_m respectivamente y termina moviendo el cabezal en dirección D_m.

En el siguiente movimiento, cada uno de los m posibles estados en los que se encuentra la máquina termina produciendo otras posibilidades para la máquina N.

árbol de ejecución de una máquina no determinista

Dada una cadena de entrada w, se puede visualizar la ejecución de una máquina de Turing no determinista por medio de un *árbol de ejecución*. En dicho árbol, cada nodo representa una posible elección que puede hacer la máquina al ir procesando la cadena w. Los nodos que están a una profundidad n, representan todos los posibles estados en los que se puede encontrar la máquina después de realizar n movimientos sobre la cadena w.

La figura 5 muestra una comparativa entre los árboles de ejecución de una máquina determinista y una máquina no determinista. Observe que en la máquina determinista cada estado está determinado de manera única por el estado anterior.

cadena aceptada por una máquina no determinista

Sea N una máquina de Turing no determinista. Formalmente decimos que w es una *cadena aceptada* por N, si existe una sucesión de movimientos

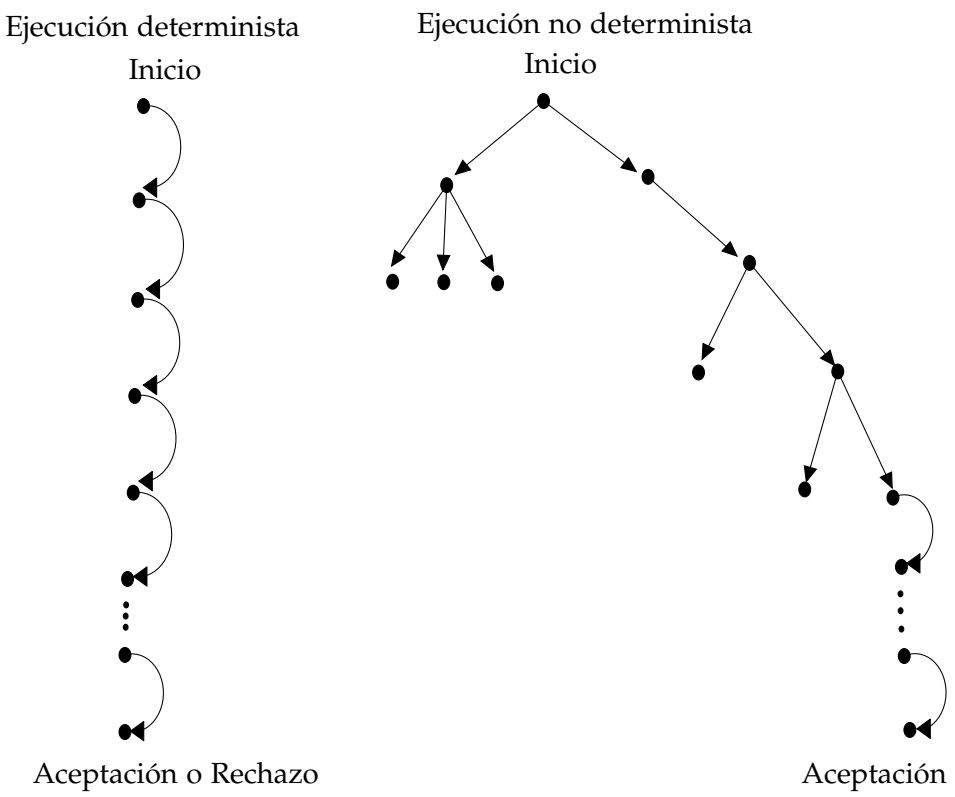


Figura 5: Representación de la ejecuciones de una máquina determinista y una máquina no determinista.

(definidos por la función de transición) que terminan en el estado q_A, en caso contrario se dice que w es una *cadena rechazada* por N. Intuitivamente esto significa que si w es aceptada por la máquina N, entonces existe al menos una hoja en el árbol de ejecución de N que tiene asociado el estado q_A.

cadena rechazada por una máquina no determinista

Observación 2.2.3. *En ocasiones se utiliza la palabra* "adivinar"*, para indicar que en una máquina no determinista, existe una sucesión de movimientos que hacen que la máquina acepte una determinada cadena.*

adivinar una cadena

El lenguaje reconocido por una máquina no determinista se define, como era de esperar, de forma similar al modelo determinista.

Definición 2.2.10. *Sea* N *una máquina de Turing no determinista,* el lenguaje reconocido *por* N *se denota por* L_N *y se define como:*

$$L_N = \{w \in \Sigma^* : w \text{ es una cadena de entrada que es aceptada por } N\}.$$

lenguaje reconocido por una máquina no determinista

De la definición 2.2.9 es evidente que una máquina de Turing no determinista puede ejecutar los mismos programas que la máquina de Turing

determinista. Por lo tanto, cualquier lenguaje que es aceptado por una máquina de Turing determinista, también es aceptado por alguna máquina de Turing no determinista. Esto nos lleva a preguntarnos si existirá algún lenguaje que pueda ser aceptado por una máquina no determinista, para el que no existe una máquina determinista que pueda reconocerlo. El siguiente teorema muestra que este no es el caso.

equivalencia entre máquina de Turing no determinista y máquina de Turing determinista

Teorema 2.2.2. *Sea* N *una máquina de Turing no determinista que acepta al lenguaje* L_N*, entonces existe una máquina de Turing* M *tal que* $L_N = L_M$.

Demostración. Se construirá una máquina de Turing de múltiples cintas S que se usará para simular a la máquina no determinista N. Recordemos que si w es una cadena de entrada, podemos representar cada paso que realiza la máquina N por medio de un árbol de ejecución, tal como se mostró en la figura 5.

búsqueda en anchura

La idea intuitiva de la simulación es que la máquina S debe recorrer el árbol de ejecución de N haciendo una *búsqueda en anchura*, esto es, primero se comienza visitando todos los nodos adyacentes al vértice raíz, después se visitan todos los vecinos de los vecinos del nodo raíz y así sucesivamente.

Sea δ_N la función de transición de la máquina N, sea w una cadena de entrada y sea x un símbolo de la cadena w. Observe que si para el símbolo x y el estado q, el valor de δ_N es de la forma

$$\delta_N(q,x) = \{\ (q_1,x_1,D_1),\ (q_2,x_2,D_2),\ \ldots\ ,\ (q_r,x_r,D_r)\ \},$$

entonces podemos enumerar de izquierda a derecha a las configuraciones, de tal forma que el número 0 corresponde a la configuración (q_1,x_1,D_1), el número 1 corresponde a (q_2,x_2,D_2) y así sucesivamente hasta llegar al número $r-1$, que corresponde a (q_r,x_r,D_r). Observe que en el árbol de ejecución de N, el nodo que corresponde al estado q debe tener exactamente r hijos (uno por cada configuración posible).

Puesto que en el árbol de ejecución de N, cada uno de los nodos hijos de cada vértice representa uno de los *siguientes estados posibles*. El número máximo de nodos hijos que puede tener cualquier vértice está acotado por

$$m = \max\{\ |\delta_N(q,x)|\ :\ x \in \Gamma,\ q \in Q \setminus \{q_A\}\ \}.$$

Por lo tanto, podemos representar la posición de cada nodo en el árbol de ejecución, por medio de un número en base m de la siguiente manera; el número $a_1a_2\cdots a_p$ representa p movimientos que realiza la máquina N con la cadena de entrada w, de tal forma que en el primer movimiento la máquina N selecciona la configuración con número a_1 de δ_N, en el

segundo movimiento selecciona la configuración con número a_2 y así sucesivamente hasta a_p.

La máquina S que simula a N tiene 3 cintas que se usan para las siguientes tareas:

- La *primera cinta* se usa para almacenar a la cadena de entrada w. La máquina nunca modifica al contenido de la primera cinta.
- La *segunda cinta* se usa para almacenar un número en base m que se incrementa gradualmente, dicho número representa uno de los posibles nodos del árbol de ejecución de la máquina N.
- La *tercera cinta* se usa para simular la ejecución de la máquina N y determinar cual es el estado que tendrá la unidad de control, cuando se recorre el árbol de ejecución de N hasta llegar al nodo representado por el número en la segunda cinta.

Al arrancar la máquina S, la primera cinta contiene la cadena de entrada w y el resto de las cintas están vacías. La descripción de la máquina S está determinada por las siguientes reglas, que se ejecutan en estricto orden:

1. Copiar la cadena w en la tercera cinta.
2. Tomar el número en base m que está en la segunda cinta e incrementarlo en 1 unidad. Si la cinta está vacía escribir el número 0.
3. Si en la segunda cinta está el número $a_1 \dots a_n$, usar la tercera cinta para simular la ejecución de la máquina N cuando selecciona las configuraciones $a_1, \dots, a_n$. Si al llegar a la configuración a_n, la simulación de N termina en el estado de aceptación de la máquina N, entonces S acepta a la cadena w, de lo contrario la máquina S ejecuta el paso 4.

   Observe que el número en la segunda cinta, representa a un nodo de un árbol en el que todo vértice que no sea una hoja tiene k hijos. Puesto que el árbol de ejecución de N no necesariamente es de esta forma, algunos de los números que aparezcan en la cinta 2 son configuraciones que no son válidas en N, cuando este sea el caso la máquina S detiene la simulación en la cinta 3 y salta al paso 4.
4. Limpiar la tercera cinta (reemplazar la cadena actual por símbolos en blanco B) y saltar al paso 1.

Puesto que la máquina S acepta a una cadena w si y solo si w es aceptada por la máquina N, se tiene que $L_N = L_S$.

Finalmente, por el teorema 2.2.1, la máquina S puede ser simulada por una máquina M de 1 cinta tal que $L_M = L_S$.

□

Observación 2.2.4. *Sean* N *y* M *las máquinas no determinista y determinista descritas en el teorema 2.2.2. Si* w *es una cadena que no está en el conjunto* L_N*, entonces la máquina* M *se queda en un ciclo por tiempo indefinido.*

Observación 2.2.5. *Sea* m *la cota que se definió en la demostración del teorema 2.2.2. Si la máquina no determinista* N *se detiene para cualquier cadena de entrada y se desea que la máquina de Turing multicinta* S *tenga el mismo comportamiento (y en consecuencia, la máquina de Turing* M *que simula a* S *será una máquina de Turing total), se puede incluir una* cuarta cinta *que sirve para almacenar un contador.*

Cada que se incrementa un dígito en el contador[1] *de la* segunda cinta*, el contador de la* cuarta cinta *se reinicia con el valor* 0*. A continuación, el contador de la cuarta cinta se incrementa en una unidad, únicamente cuando el número del contador en la segunda cinta, representa un nodo válido del árbol de ejecución de la máquina* N*.*

La máquina S *que simula a la máquina* N*, termina en el estado de rechazo* q_R*, si al incrementar un dígito del contador de la segunda cinta se tiene que el contador de la cuarta cinta vale* 0 *(antes de reiniciar el contador). Observe que esta condición se cumple cuando ya no quedan más nodos por explorar del árbol de ejecución de* N*.*

búsqueda en profundidad

Es importante mencionar que en la prueba del teorema 2.2.2, una estrategia errónea para recorrer el árbol de ejecución de N, sería haciendo una *búsqueda en profundidad*, es decir, primero se recorre una rama del árbol hasta llegar a una hoja, posteriormente se recorre una rama diferente hasta llegar a una hoja diferente y así sucesivamente. El problema de esta estrategia, es que la máquina N para la misma cadena de entrada w puede tener una sucesión de configuraciones que la llevan a un estado de aceptación y también puede tener una sucesión de configuraciones que hagan que la máquina quede inmersa en un ciclo infinito. En este caso, una de las ramas del árbol de ejecución tendría longitud finita (la de aceptación) y otra rama tendría longitud infinita. Si la búsqueda en profundidad elige esta última rama, la simulación que ejecuta S quedaría inmersa en un ciclo infinito y nunca reportaría el estado de aceptación.

[1] Por ejemplo, en base m esto pasa cuando se suma una unidad al número aa con $a = m - 1$. En este caso el siguiente número es 100, este número tiene un dígito más que el número previo aa.

Mas adelante mostraremos cual es el costo en *tiempo* de simular una máquina de Turing no determinista, por medio de una variación de la máquina de Turing M propuesta en el teorema 2.2.2. Por ahora, podemos pensar intuitivamente que dado que una máquina no determinista en 1 movimiento puede "probar" múltiples opciones de manera simultanea, dicha máquina es más "rápida" que cualquiera de los modelos que estudiamos en las secciones anteriores.

2.2.4 *Máquina de Turing con oráculo*

Por los teoremas 2.2.2 y 2.2.1, podemos concluir que todas las variantes de la máquina de Turing que se estudiaron en las secciones anteriores son equivalentes, es decir, los lenguajes que reconoce una variante de la máquina de Turing pueden ser reconocidos por las otras variantes de la máquina de Turing.

Aunque está fuera de los objetivos de esta tesis, se puede demostrar que existen lenguajes para los que *no existe* una máquina de Turing que pueda reconocerlos [1]. Por otra parte, también existen lenguajes indecidibles y lenguajes para los que *no se conoce* un programa de una máquina de Turing que pueda reconocer las cadenas de dicho lenguaje en tiempo eficiente (este problema se explora a detalle en la sección 2.3 de esta tesis).

Estas cuestiones nos llevan a formular de manera intuitiva lo que se conoce como oráculo. Podemos pensar que un *oráculo* es un programa que bajo cierto modelo de computación (quizás desconocido), puede reconocer cualquier cadena de un determinado lenguaje L en 1 solo movimiento de dicho modelo de computación. Esto también nos lleva a preguntarnos sobre el tipo de problemas que podría resolver una máquina de Turing, que tiene un programa que puede invocar tantas veces como sea necesario a un oráculo que reconoce al lenguaje L. Una máquina con tales características se le conoce como *máquina de Turing con oráculo* o simplemente *máquina con oráculo*.

oráculo

máquina de Turing con oráculo

Podemos visualizar a una máquina de Turing con oráculo como una máquina con 2 cintas, cada una asociada con un cabezal de escritura/lectura. A la primera cinta se le denomina *cinta primaria* y a su cabezal asociado se le llama *cabezal primario*. A la segunda cinta se le conoce como *cinta oráculo* y a su cabezal asociado se le llama *cabezal oráculo*. La máquina con oráculo

cinta primaria

cabezal primario

cinta oráculo

cabezal oráculo

[1] Un ejemplo de un lenguaje para el que no existe una máquina de Turing que pueda reconocerlo se estudia en [28, p. 181-182].

unidad de control

también tiene un *estado* asociado a la *unidad de control*. Una representación de la máquina de Turing con oráculo se muestra en la figura 6.

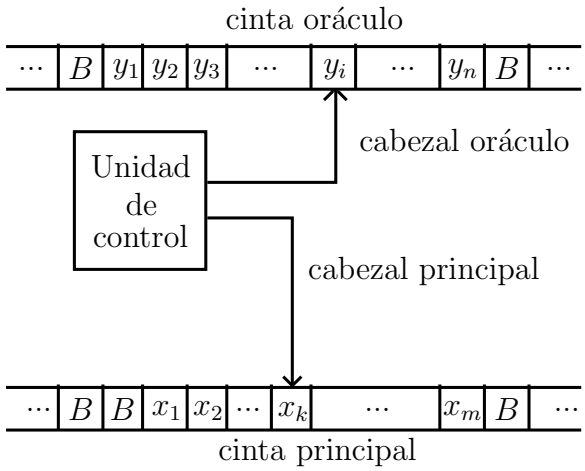


Figura 6: Representación de una máquina de Turing con oráculo.

La definición formal de la máquina de Turing con oráculo se muestra a continuación.

máquina de Turing con oráculo

Definición 2.2.11. *Una* máquina de Turing con oráculo *es una tupla* $M = (Q, \Sigma, \Gamma, \delta, g, q_0, q_C, q_A, q_F)$*, cuyos componentes tienen el siguiente significado:*

- Q *es el* conjunto de estados.
- $q_0 \in Q$ *es el* estado inicial.
- $q_C \in Q$ *es el* estado de consulta.
- $q_T \in Q$ *es el* estado de consulta terminada.
- $q_F \in Q$ *es el* estado final.
- Σ *es el* alfabeto de entrada.
- Γ *es el* alfabeto de cinta. *El* símbolo blanco B *es parte de* Γ *y se debe cumplir que* $\Gamma \supsetneq \Sigma$.
- δ *es la* función de transición *y es de la forma*

$$\delta : (Q \setminus \{q_C, q_F\}) \times \Gamma \times \Gamma \to Q \times \Gamma \times \Gamma \times \{R,\ L\} \times \{R, L\},$$

  donde R *es la dirección derecha y* L *es la dirección izquierda.*
- g *es la* función oráculo *y es de la forma* $g : \Sigma^* \to \Sigma^*$.

Antes de arrancar la máquina, la cinta oráculo solo contiene símbolos en blanco B y la cadena de entrada se coloca en la cinta primaria.

movimiento de la máquina con oráculo

Sea $M = (Q, \Sigma, \Gamma, \delta, g, q_0, q_C, q_A, q_F)$ una máquina con oráculo y sea q el estado actual en la unidad de control, un *movimiento* consiste en las

siguientes tareas:

- Si $q \in Q \setminus \{q_C, q_F\}$, entonces la máquina lee los símbolos x y y de la cinta primaria y la cinta oráculo respectivamente. Sean D_1 y D_2 direcciones en $\{R, L\}$, si se tiene que

$$\delta(q, x, y) = (q', x', y', D_1, D_2),$$

  entonces la máquina sustituye el símbolo x en la cinta primaria por x' y el símbolo y en la cinta oráculo por y'. A continuación mueve el cabezal primario en dirección D_1 y el cabezal oráculo en dirección D_2. Finalmente, el estado de la unidad de control se cambia a q'.
- Si $q = q_C$, entonces leyendo de izquierda a derecha, el cabezal oráculo toma la cadena S que se encuentra delimitada por un símbolo blanco B a la izquierda y a la derecha, es decir; BSB. A continuación, la cadena $g(S)$ sustituye a la cadena S en la cinta oráculo. Si se tiene que $|g(S)| < |S|$, entonces el resto de los símbolos de S se cambian por el símbolo blanco B. Finalmente, la máquina pasa al estado q_T.
- Si $q = q_F$, entonces la máquina detiene su ejecución.

Bajo ciertas condiciones, podemos asociar una función a una máquina con oráculo, de una manera similar como se hizo con la máquina de Turing. Como se verá mas adelante, esta idea es fundamental para el estudio de la clase NP-difícil (sección 2.3.4).

función asociada a una máquina de Turing con oráculo

Definición 2.2.12. *Sea M una máquina con oráculo con alfabeto de entrada Σ y sea ϵ la cadena vacía. Si para toda cadena de entrada w, la máquina M termina en el estado q_F, entonces se define la función $f_M : \Sigma^* \to \Sigma^*$ como:*

- $f(w) = \epsilon$, *si al entrar en el estado q_F, la cinta primaria contiene únicamente símbolos en blanco B.*
- $f(w) = w'$, *si la cinta primaria contiene al menos un símbolo del alfabeto Σ, diferente de B. w' es la cadena no vacía de mayor longitud, que resulta de leer de izquierda a derecha la cinta primaria, una vez que la máquina ha entrado en el estado q_F.*

función computable por una máquina con oráculo

Definición 2.2.13. *Se dice que una función f es* computable por una máquina con oráculo, *si existe una máquina de Turing con oráculo M tal que $f = f_M$.*

La máquina de Turing con oráculo es muy útil para estudiar relaciones de cadenas, que a su vez son muy importantes para poder formalizar el estudio de problemas de búsqueda.

relación de cadenas

Definición 2.2.14. *Sea Σ^+ el conjunto de cadenas no vacías sobre el alfabeto Σ. Una* relación de cadenas *sobre el alfabeto Σ, es un subconjunto $R \subseteq \Sigma^+ \times \Sigma^+$.*

relación de cadenas identificada con un lenguaje

Observación 2.2.6. *Sea s un símbolo fijo del alfabeto Σ. Es posible* identificar *un lenguaje L con la relación de cadenas*

$$R_L = \{ (x, s) \mid x \in L \cap \Sigma^+ \}.$$

función que realiza una relación de cadenas

Definición 2.2.15. *Sea ϵ la cadena vacía, se dice que una función $f : \Sigma^* \to \Sigma^*$* realiza *la relación de cadenas R, si cumple con las siguientes condiciones:*

- $f(x) = \epsilon$, *si y solo si x es un símbolo en Σ^+, para el que* no existe *un símbolo y en Σ^+ tal que $(x, y) \in R$.*
- $f(x) = y$, *si y solo si x es un símbolo en Σ^+ para el que* existe *un símbolo y en Σ^+ tal que $(x, y) \in R$.*

relación de cadenas que realiza una máquina con oráculo

Definición 2.2.16. *Se dice que una máquina de Turing con oráculo M,* realiza *a la relación de cadenas R, si la funciónf_M realiza a R_L.*

Por medio de una relación de cadenas, se puede definir el lenguaje que acepta una máquina de Turing con oráculo de la siguiente manera.

lenguaje reconocido por una máquina de Turing con oráculo

Definición 2.2.17. *Sea R_L la relación de cadenas que se identifica con el lenguaje L (observación 2.2.6). Sea M una máquina de Turing con oráculo. Se dice que M* reconoce *al lenguaje L si f_M realiza a R.*

La máquina con oráculo es capaz de reconocer lenguajes que ninguna máquina de Turing puede reconocer, por lo tanto, estos modelos no son equivalentes. Sin embargo, aún con el poder computo del oráculo, se puede demostrar que existen lenguajes que ninguna máquina de Turing con oráculo puede reconocer [27]. Para los propósitos de este trabajo, es suficiente conocer los modelos de computación estudiados en estas secciones.

2.3 CONCEPTOS BÁSICOS DE COMPLEJIDAD COMPUTACIONAL

2.3.1 *Complejidad temporal*

Nos gustaría definir una medida para cuantificar el "tiempo" de un programa en los modelos de computación previamente estudiados, para esto,

consideraremos que una *unidad de tiempo* en un determinado modelo de computación, corresponde a *un movimiento* de dicho modelo de computación.

tiempo de ejecución de una máquina de Turing/ multicinta/ con oráculo

Definición 2.3.1. *Sea* M *una máquina de Turing (o una máquina de Turing multicinta o una máquina de Turing con oráculo) y sea* w *una cadena de entrada. Si la máquina* M *se* detiene *después de realizar* m *movimientos, se dice que* M *requiere un* tiempo de ejecución m *con la cadena* w*.*

Observación 2.3.1. *Si* M *es una máquina de Turing y* M' *es una máquina de Turing con múltiples cintas (o con oráculo) tal que* $L_M = L_{M'}$*, no necesariamente se cumple que los tiempos de ejecución de* M *y* M' *son iguales cuando reciben la misma cadena de entrada* w*.*

Puesto que en una máquina de Turing no determinista en cada movimiento puede estar en múltiples configuraciones al mismo tiempo, es necesario dar una definición diferente de 2.3.1 para el tiempo de ejecución de máquinas no deterministas.

tiempo de ejecución de una máquina de Turing no determinista

Definición 2.3.2. *Sea* M *una máquina de Turing no determinista que* acepta *a la cadena de entrada* w*. El* tiempo de ejecución *de* M *con la cadena* w *es el* mínimo *número de movimientos sobre todas las secuencias de movimientos que* aceptan *a la cadena* w*.*

Por lo general, para estudiar la complejidad temporal del programa de un modelo de computación, resulta más útil considerar el máximo tiempo de ejecución sobre todas las cadenas de una longitud n. Esto nos lleva a la siguientes definiciones.

función de complejidad temporal para una máquina de Turing/ multicinta/ con oráculo

Definición 2.3.3 (Garey [11, p. 26])**.** *Sea* M *una máquina de Turing (o una máquina con múltiples cintas o una máquina con oráculo) que* siempre termina *con* cualquier *cadena de entrada. La* función de complejidad temporal $T_M : \mathbb{N} \to \mathbb{N}$ *de la máquina* M*, se define como*

$$T_M(n) = \max \left\{ m \in \mathbb{N} : \begin{array}{c} \text{Existe una cadena } w \in \Sigma^* \text{ con } |w| = n \\ \text{tal que } M \text{ termina con la cadena } w \\ \text{en tiempo } m. \end{array} \right\}.$$

función de complejidad temporal no determinista

Definición 2.3.4 (Garey [11, p. 31]). *Sea* N *una máquina de Turing no determinista. La* función de complejidad temporal $T_N : \mathbb{N} \to \mathbb{N}$ *de la máquina* N*, se define como*

$$T_N(n) = \max\left\{\{1\} \cup \left\{ m \in \mathbb{N} : \begin{array}{l} \text{Existe una cadena } w \in L_N \text{ con } |w| = n \\ \text{tal que } N \text{ acepta a la cadena } w \\ \text{en tiempo } m. \end{array} \right\}\right\}.$$

Observación 2.3.2. *Observe que la función de complejidad temporal* $T_N(n)$ *para un algoritmo no determinista, considera únicamente las cadenas de longitud* n *que son* aceptadas*, si no existe ninguna cadena con dichas características la función* $T_N(n)$ *tiene valor 1.*

Es importante notar que si M es una máquina de Turing y M' es una máquina no determinista (o de múltiples cintas o con oráculo) tal que $L_M = L_{M'}$, no necesariamente se cumple que $T_M(n) = T_{M'}(n)$.

En general, nos interesa saber cómo se comporta la función de complejidad temporal $T_M(n)$ a medida que n crece de manera arbitraria. Más aún, si M_1 y M_2 son ambas máquinas de Turing, es posible construir una máquina M_3 que tiene a M_1 y M_2 como subrutinas en su ejecución, en tal caso nos interesa saber como es la complejidad temporal de M_3, al depender de M_1 y M_2. La siguiente notación, propuesta originalmente por Edmund Landau [18, págs. 59-63] y posteriormente popularizada en la teoría de la computación por Donald Knuth [15], nos provee de las herramientas para tratar estas cuestiones.

notación $\mathcal{O}$ grande

Definición 2.3.5 (Landau [18, págs. 59-63], Knuth [15]). *Sean* $f(n)$ *y* $g(n)$ *funciones definidas sobre los enteros. Se dice que* $f(n)$ *está en* $\mathcal{O}(g(n))$ *si existe una constante* $c > 0$ *y un entero* n_0 *tal que para todo entero* $n \geqslant n_0$ *se cumple que:*

$$|f(n)| \leqslant c|g(n)|$$

Observación 2.3.3. *A la definición 2.3.5 también se le conoce como notación* $\mathcal{O}$ *grande o simplemente notación* $\mathcal{O}$*.*

Observación 2.3.4. *Es común utilizar la notación* $f(n) = \mathcal{O}(g(n))$ *o simplemente* $f = \mathcal{O}(g)$ *para indicar que* $f(n)$ *está en* $\mathcal{O}(g(n))$ *. Es importante notar* $f = \mathcal{O}(g)$ *solo es una notación y de ninguna manera representa una igualdad en sentido estricto.*

La definición 2.3.5 se puede generalizar para funciones sobre los reales, para nuestros propósitos las funciones definidas sobre los enteros serán suficientes. La notación $\mathcal{O}$ es muy útil para denotar funciones que están

acotadas superiormente. Además de la notación O grande, resulta conveniente tener una notación para las funciones $f(n)$ que están acotadas *inferiormente* por una función $g(n)$. En estos casos usamos la siguiente notación propuesta por Donald Knuth.

notación Ω

Definición 2.3.6 (Knuth [15])**.** *Sean $f(n)$ y $g(n)$ funciones definidas sobre los enteros. Se dice que $f(n) = \Omega(g(n))$ si y solo si $g(n) = \mathcal{O}(f(n))$.*

Observación 2.3.5. *A la definición 2.3.5 también se le conoce como notación Omega grande o simplemente notación Ω.*

Si M es una máquina de Turing (o una máquina no determinista o una máquina multicinta o una máquina con oráculo) con función de complejidad temporal $T_M(n)$, se dice que la máquina M tiene complejidad $g(n)$, si $T_M(n)$ y $g(n)$ cumplen con la definición 2.3.5, esto es, si $T_M(n) = \mathcal{O}(g(n))$. A continuación se listan algunas propiedades básicas de la notación $\mathcal{O}$.

propiedades de la notación O grande

Teorema 2.3.1. *Sean f, g, h y s funciones definidas sobre los enteros. Sea c una constante positiva. La notación $\mathcal{O}$ tiene las siguientes propiedades:*

(i) Escalamiento: cf está en $\mathcal{O}(f)$

(ii) Transitividad: Si $h = \mathcal{O}(g)$ y $g = \mathcal{O}(f)$, entonces $h = \mathcal{O}(f)$

(iii) Regla de la suma: Si $f = \mathcal{O}(h)$ y $g = \mathcal{O}(s)$, entonces $f + g = \mathcal{O}(\max\{h, s\})$

(iv) Regla del producto: Si $f = \mathcal{O}(h)$ y $g = \mathcal{O}(s)$, entonces $fg = \mathcal{O}(hs)$

Demostración.
(i) Como $c|f(n)| < (c+1)|f(n)|$ para todo n, tome a $(c+1)$ como constante de la definición 2.3.5.

(ii) Por definición, existen constantes positivas c_1 y c_2 y enteros n_1 y n_2 tales que

$$|h(n)| \leqslant c_1|g(n)| \qquad \forall n \geqslant n_1,$$
$$|g(n)| \leqslant c_2|f(n)| \qquad \forall n \geqslant n_2.$$

Luego

$$h(n) \leqslant c_1|g(n)| \leqslant c_1c_2|f(n)| \qquad \forall n \geqslant \max\{n_1, n_2\}.$$

(iii) y (iv) Por definición, existen constantes positivas c_1 y c_2 y enteros n_1 y n_2 tales que

$$|f(n)| \leqslant c_1|h(n)| \qquad \forall n \geqslant n_1,$$
$$|g(n)| \leqslant c_2|s(n)| \qquad \forall n \geqslant n_2.$$

Para $n \geqslant max\{n_1, n_2\}$ y por la desigualdad del triángulo se tiene que

$$\begin{aligned} |f(n) + g(n)| &\leqslant |f(n)| + |g(n)| \leqslant c_1|h(n)| + c_2|s(n)| \\ &\leqslant c_1|max\{h(n), s(n)\}| + c_2|max\{h(n), s(n)\}| \\ &\leqslant (c_1 + c_2)|max\{h(n), s(n)\}|. \end{aligned}$$

Por otra parte, para el inciso (iv)

$$\begin{aligned} |f(n)g(n)| &\leqslant |f(n)||g(n)| \leqslant (c_1c_2)|h(n)||s(n)| \\ &\leqslant (c_1c_2)|h(n)s(n)|. \end{aligned}$$

□

El teorema 2.3.1, provee de herramientas muy útiles para *acotar superiormente* a una función de complejidad temporal. En particular, si $f(n) \leqslant g(n)$ y si c_1 y c_2 son constantes positivas, entonces por los incisos (ii) y (iii) del teorema 2.3.1 se tiene que $c_1 f(n) + c_2 g(n) = \mathcal{O}(g(n))$.

Si w es una cadena sobre un alfabeto Σ, en ocasiones es posible representar a dicha cadena usando una convención que permita usar menos símbolos. Por ejemplo, la cadena $w = 10000$ se puede representar como 1#5, donde el número 5 es el número de ceros que hay después del número 1. En este caso se dice que la cadena 10000 es una *representación explícita* y la cadena 1#5 es una *representación implícita* de la cadena 10000.

representación explícita

representación implícita

Observe que para dar una representación implícita, es necesario especificar un algoritmo que permite pasar de la representación implícita a la representación explícita. Podría ocurrir que bajo dicho algoritmo, existen múltiples representaciones implícitas para la misma cadena. Esto nos lleva a preguntarnos cual es la representación implícita más pequeña que se puede tener para una determinada cadena con dicho algoritmo, la siguiente definición nos permite formalizar esta cuestión.

complejidad de Kolmogorov con respecto a una función computable

Definición 2.3.7 (Vitányi [21, p. 94]). *Sea* $\phi : \Sigma^* \to \Sigma^*$ *una función computable por una máquina de Turing con alfabeto de entrada* Σ. *Sea* $y \in \Sigma^*$ *una cadena en el rango de* ϕ. *La* complejidad de Kolmogorov[1] K_ϕ *de la cadena* y *con respecto a* ϕ *es*

$$K_\phi(y) = min\{|x| \ : \ x \in \Sigma^* \quad y \quad \phi(x) = y\}.$$

La definición 2.3.7 se puede extender para hacer uso de la *máquina de Turing Universal*[2], para los fines de este trabajo la definición 2.3.7 es suficiente.

[1] En honor al matemático ruso Andrey Nikolaevich Kolmogorov (1903- 1987).

[2] En [21, p. 94-99] se da una explicación detallada de como hacer dicha extensión.

Teorema 2.3.2. *Sea* $\Phi : \Sigma^* \to \Sigma^*$ *una función computable. Sea* $h : \mathbb{N} \to \mathbb{N}$ *una función sobre los enteros y sea* $f : \Sigma^* \to \Sigma^*$ *una función computable por la máquina de Turing* M*. Si para todo entero* $n \geqslant 0$*, existe una cadena* $x \in \Sigma^*$ *tal que* $|x| = n$ *y* $K_\Phi(f(x)) \geqslant h(|x|)$*, entonces la complejidad temporal* T_M *de la máquina* M *cumple que* $T_M(n) = \Omega(h(n))$*.*

Demostración. Puesto que para la cadena de entrada x, la máquina M imprime en su cinta $K_\Phi(f(x))$ símbolos, la máquina requiere al menos $K_\Phi(f(x))$ movimientos y en consecuencia

$$T_M(n) \geqslant K_\Phi(f(x)) \geqslant h(|x|)$$

Por lo tanto, $T_M(n) = \Omega(h(n))$. □

El teorema 2.3.2 será de mucha utilidad para acotar inferiormente a la complejidad temporal del problema Lclique que se estudia en la sección 3.3.

El siguiente teorema muestra cual es la complejidad temporal de simular una máquina multicinta con la máquina de Turing del teorema 2.2.1.

complejidad temporal de simular una máquina multicinta con una máquina de Turing

Teorema 2.3.3. *Sea* S *una* máquina de Turing multicinta *con* $T_S(n) = f(n) \geqslant n$*. Existe una* máquina de Turing M *tal que* $L_M = L_S$ *y* $T_M(n) = \mathcal{O}(f^2(n))$

Demostración. Sea k el número de cintas de la máquina S. Defina una máquina de Turing M que simula a S descrita por las mismas reglas del teorema 2.2.1. Observe que en la regla 1, la máquina S lee toda la cadena de entrada w para colocar los símbolos # que representan las k cintas, como $|w| = n$, ejecutar la regla 1 lleva tiempo $\mathcal{O}(n)$.

Puesto que S acepta o rechaza la cadena w en no más de $f(n)$ movimientos, cada una de las cintas de S a lo más requieren $f(n)$ casillas para ejecutar el programa de S. Por lo tanto, la máquina M usa un total de casillas proporcional a $kf(n)$ para poder simular a S. Por la regla 2, la máquina M realiza 2 recorridos de la cinta a fin de simular un movimiento de S, esta operación requiere tiempo $\mathcal{O}(2kf(n)) = \mathcal{O}(f(n))$.

Dado que la máquina S realiza a lo más $f(n)$ movimientos y por cada movimiento la máquina M realiza los pasos descritos en la regla 2, la máquina M requiere un tiempo $\mathcal{O}(f^2(n))$ para simular a S, *sin contar* el tiempo que requiere para ejecutar la regla 1. Por lo tanto, $T_M(n) = \mathcal{O}(f^2(n)) + \mathcal{O}(n)$.

Por hipótesis $f(n) > n$, luego $T_M(n) = \mathcal{O}(f^2(n))$.

□

contador

Para las siguientes demostraciones, necesitamos definir formalmente el concepto de *contador*. Un contador es una máquina de Turing que toma como entrada a un número n (en base $k \geqslant 2$) y calcula la función $f(n) = n+1$ (en base k). Con frecuencia haremos uso del siguiente resultado.

complejidad temporal de un contador

Lema 2.3.1. *Sea w una cadena que representa a un número de n dígitos en base k. Un contador M tiene complejidad temporal $T_M(n) = \mathcal{O}(n)$.*

Demostración. Construiremos una máquina M que al recibir el número w imprime en su cinta el resultado de la suma $w+1$ en base k. La máquina realiza las siguientes reglas en estricto orden:

1. Mover el cabezal en dirección derecha (sin modificar ningún símbolo) hasta llegar al primer símbolo en blanco B.
2. Mover el cabezal en dirección izquierda, al encontrar el primer símbolo diferente de B (es decir, el último dígito del número w) sumar una unidad a dicho dígito. Para determinar el resultado de esta operación, previamente se ha definido un estado por cada posible par de números $(x, y) \in K \times K$, cada uno de estos estados tiene el resultado de sumar $x + y$.
3. Si la suma resulta en un número con 2 dígitos; $b_1 b_2$. Reemplazar el símbolo actual por b_2 y guardar en el estado de la máquina el símbolo b_1 (a este dígito le llamaremos acarreo). Si la suma resulta en un número con 1 dígito; b_1. Reemplazar al símbolo actual por b_1 y guardar el número 0 como acarreo en el estado de la máquina.
4. Mover el cabezal a la izquierda, si el símbolo que se lee es B y el acarreo que se guardo previamente en el estado es 0, terminar la ejecución de la máquina, de lo contrario sumar el acarreo al símbolo actual y repetir desde la regla 3.

Para determinar la complejidad de la máquina, observe que la regla 1 recorre toda la cadena de entrada w en dirección derecha, el resto de las reglas terminan recorriendo otra vez la cadena w pero en dirección izquierda. Por lo tanto, existe una constante C_1 tal que

$$T_M(|w|) = T_M(n) \leqslant 2C_1 n = \mathcal{O}(n).$$

□

Si w es un número en base k, denotaremos por $(w)_{10}$ al resultado de convertir el número w a base decimal.

Los siguientes resultados muestran cual es la complejidad temporal de realizar operaciones aritméticas básicas sobre los enteros en una máquina de *Turing multicinta*. Para algunas de estas operaciones es posible construir algoritmos más eficientes de los que se describen en este trabajo, sin embargo, las cotas que se presentan a continuación, son suficientes para los teoremas que se revisarán en las siguientes secciones.

Lema 2.3.2. *Sea* k *un entero positivo. Sea* $f_k : \mathbb{N} \to \mathbb{N}$ *la función definida como*

$$f_k(n) = nk^n.$$

Sea x *el número más grande en base* k*, que se puede construir usando* n *dígitos. Entonces, la función* f_k *tiene la siguiente propiedad:*

$$(x)_{10} < f_k(n)$$

Demostración. El valor del número x en decimal está dado por :

$$(x)_{10} = \sum_{i=0}^{n-1} k^i < \sum_{i=0}^{n-1} k^n = nk^n = f_k(n)$$

□

complejidad temporal de operaciones aritméticas básicas

Teorema 2.3.4. *Sean* w_1 *y* w_2 *números enteros en base* $k \geqslant 2$*, con* $|w_1| = n$ *y* $|w_2| = m$*. Si* $m \leqslant n$*, existe una máquina de* Turing multicinta M *para cada una de las siguientes operaciones tal que:*

(a) Determinar si $w_1 = w_2$ *se puede hacer en tiempo* $T_M(n) = \mathcal{O}(n)$*.*

(b) Sumar w_1 *y* w_2 *(en base* k*) se puede hacer en tiempo* $T_M(n) = \mathcal{O}(n)$*.*

(c) Multiplicar w_1 *por* w_2 *(en base* k*) se puede hacer en tiempo* $T_M(n) = \mathcal{O}(n^2)$*.*

(d) Elevar w_1 *a una constante entera* p *(en base* k*) se puede hacer en tiempo* $T_M(n) = \mathcal{O}(n^2)$*.*

Demostración.

(a) Defina una máquina multicinta M con 2 cintas que recibe como entrada la cadena $w_1\#w_2$. Al inicio, la máquina M copia la cadena w_1 a la segunda cinta y borra la cadena $w_1\#$ de la primera cinta. Posteriormente los cabezales en la primera y segunda cinta se colocan al final de las cadenas w_1 y w_2. Observe que para realizar estas operaciones, existe una constante C_1 tal que en el peor caso, se requieren $C_1(n+m) \leqslant 2C_1n$ movimientos.

A continuación ambos cabezales se mueven *al mismo tiempo* en dirección izquierda, comparando los símbolos de ambas cadenas; si se encuentra un par que es diferente la cadena se rechaza, si se lee el símbolo blanco en ambas cintas entonces la cadena se acepta. Para realizar estas operaciones, existe una constante C_2 tal que en el peor caso, se requieren $C_2 n$ movimientos.

Por lo tanto, la complejidad temporal de la máquina M es

$$T_M(n) \leqslant 2C_1 n + C_2 n = \mathcal{O}(n).$$

(b) Defina una máquina multicinta M con 2 cintas que recibe como entrada una cadena de la forma $w_1\#w_2$. La *primera y segunda cinta* se usarán para almacenar las cadenas w_1 y w_2 respectivamente. El resultado de la suma se sobrescribe en la *segunda cinta*.

Al inicio, la máquina copia la cadena w_1 a la segunda cinta y borra la cadena $w_1\#$ de la primera cinta. A continuación los cabezales en la primera y la segunda cinta se colocan al final de las cadenas w_1 y w_2 respectivamente. Por lo tanto, existe una constante C_1 tal que en el peor caso, estas operaciones requieren $C_1(m+n+1) \leqslant C_1(2n+1)$ movimientos.

Posteriormente la máquina mueve los cabezales en la primera y segunda cinta en dirección izquierda *al mismo tiempo*. En cada movimiento, la máquina suma los dígitos que estén leyendo el primer y segundo cabezal (la suma se hace en base k), si ocurre un acarreo este se suma en el siguiente movimiento. La máquina se detiene si lee el símbolo blanco en la primera y en la segunda cinta y si no hay ningún acarreo guardado en su estado. El resultado de la suma se muestra en la segunda cinta. Por lo tanto, existe una constante C_2 tal que en el peor caso, estas operaciones requieren $C_2(n+1)$ movimientos.

Por lo tanto, la complejidad de la máquina M es

$$T_M(n) \leqslant C_1(2n+1) + C_2(n+1) = \mathcal{O}(n).$$

(c) Defina una máquina con 4 cintas que recibe como entrada una cadena de la forma $w_1\#w_2$. La *primera* y *segunda cinta* se usan para almacenar las cadenas w_1 y w_2 respectivamente. La *tercera cinta* contendrá el resultado de la multiplicación. La *cuarta cinta* se usa para almacenar un *contador* en base k.

El funcionamiento de la máquina se basa en el hecho de que multiplicar w_1 por w_2 en base k, es equivalente a realizar la suma

$$w_1 w_2 = \sum_{i=1}^{w_1} w_2.$$

La máquina ejecuta las siguientes reglas en estricto orden:

1 Copiar la cadena w_1 a la segunda cinta. En la primera cinta borrar la cadena $w_1\#$. Escribir en la tercera y cuarta cinta el número 0.
2 Revisar que $w_1 \neq 0$ y $w_2 \neq 0$, para ello, la máquina compara w_1 y w_2 con el número en la tercera cinta, si alguno es igual a 0, la máquina termina su ejecución, en caso contrario la máquina continua con la regla 3.
3 Sumar los números en la segunda y tercera cinta, sobrescribir el resultado en la tercera cinta.
4 Incrementar el contador en la cuarta cinta en una unidad, comparar el valor del contador con el número en la primera cinta. Si son iguales la máquina termina su ejecución y el resultado de la multiplicación es el número en la tercera cinta, en caso contrario la máquina vuelve a ejecutar las reglas 3 y 4.

Puesto que la regla 1 recorre la cadena $w_1\#$, se requieren $n+1$ movimientos para esta regla. Por el inciso *(a)*, existe una constante C_1 tal que la regla 2 en el peor caso requiere $C_1 n$ movimientos.

Sea x el número en base k que resulta de multiplicar w_1 por w_2 (en base k). Por el inciso *(b)*, para poder ejecutar la regla 3, la máquina requiere un número de movimientos proporcional al número de dígitos que tiene el número x. Por el lema 2.3.2, el valor máximo del número x esta acotado por

$$(x)_{10} < f_k(|w_1|) f_k(|w_2|) = f_k(n) f_k(m) \leqslant (f_k(n))^2 = n^2 k^{2n}.$$

Por lo tanto, el número de dígitos que se requieren para representar a x en base k, está acotado por el número de dígitos que se requieren para representar a $n^2 k^{2n}$ en base k, como

$$\lceil \log_k(n^2 k^{2n}) \rceil = \lceil 2n + 2\log_k(n) \rceil \leqslant 2n + 2\log_k(n) + 1 < 4n + 1,$$

el número x en base k no puede tener más de $4n+1$ dígitos y por el inciso *(b)*, existe una constante C_2 tal que en el peor caso, la regla 3 requiere $C_2(4n+1)$ movimientos.

Por el inciso *(a)* y por el lema 2.3.1, existe una constante C_3 tal que en el peor caso, la regla 4 requiere $C_3 n$ movimientos.

Finalmente, como las reglas 3 y 4 se ejecutan n veces, se tiene que

$$\begin{aligned} T_M(n) &< (n+1) + C_1 n + n\,(C_2(4n+1) + C_3 n) \\ &= (4C_2 + C_3)n^2 + (C_1 + C_2 + 1)n + 1 = \mathcal{O}(n^2). \end{aligned}$$

(d) Si $p = 0$, entonces $w_1^p = 1$. Por lo tanto, es suficiente definir una máquina de 1 cinta que para cualquier cadena de entrada w_1, la máquina reemplaza a dicha cadena por el número 1.

Si $p = 1$, entonces $w_1^p = w_1$,. Por lo tanto, es suficiente definir una máquina de 1 cinta que al recibir una cadena de entrada w_1, inmediatamente acepta a w_1.

Si $p > 1$, entonces el funcionamiento de la máquina se basa en notar que elevar w_1 a una potencia p en base k, es equivalente a

$$w_1^p = \prod_{i=1}^{p} w_1.$$

Defina una máquina M con 4 cintas que recibe como cadena de entrada a la cadena w_1. Puesto que p es una constante, podemos definir un contador c cuyo valor se *guarda en el estado de la unidad de control* (no es necesario usar una cinta). La máquina M funciona ejecutando las siguientes reglas en estricto orden:

1 Agregar el símbolo # al final de la cadena w_1 y asignar al contador el valor $c = 1$.
2 Copiar la cadena w_1 a la tercera cinta.
3 Copiar la cadena en la tercera cinta al final del símbolo # en la primera cinta.
4 Limpiar la segunda, tercera y cuarta cinta (poner en todas las casillas el símbolo blanco B).
5 Usar las 4 cintas para hacer la multiplicación de las cadenas que se encuentran en la primera cinta, usando las mismas reglas que se usaron para la máquina que multiplica del inciso *(c)*.
6 Incrementar el contador c en una unidad. Si $c = p$, entonces la máquina termina su ejecución y el resultado de la multiplicación es el número en la tercera cinta. Si $c \neq p$, entonces la máquina vuelve a ejecutar las reglas desde la número 3 en adelante.

Puesto que las reglas 1 y 2 recorren la cadena w_1, existe una constante C_1 tal que estas operaciones en el peor caso, requieren $C_1 n$ movimientos.

Sea x el número que resulta de elevar w_1 a la potencia p (en base k). Denotaremos por r a la cantidad de dígitos que tiene p, es decir $r = |p|$. Por el lema 2.3.2, el valor de x está acotado superiormente por

$$(x)_{10} = ((w_1)_{10})^{(p)_{10}} < f_k(|w_1|)^{f_k(|p|)} = (nk^n)^{rk^r}.$$

Por lo tanto, el número de dígitos que requiere x en base k está acotado superiormente por

$$\left\lceil \log_k \left((nk^n)^{rk^r} \right) \right\rceil = \lceil rk^r \left(\log_k(n) + n \right) \rceil < 2rk^r n + 1.$$

Por el inciso *(c)*, existe una constante C_2 tal que en el peor caso, la regla 5 requiere $C_2|x|^2 < C_2 \left(2rk^r n + 1\right)^2$ movimientos.

Las reglas 3 y 4 recorren todas las cadenas que hay en las cintas, la longitud máxima de dichas cadenas no puede ser mayor a $|x|$. Por lo tanto, existe una constante C_3 tal que en el peor caso, la máquina realiza $C_3|x| < C_3(2rk^r n + 1)$ movimientos.

La regla 6 requiere solo un movimiento. Las reglas 3, 4, 5 y 6 se ejecutan p veces. Como r y k son constantes, la complejidad temporal de la máquina M es

$$\begin{aligned} T_M(n) &< C_1 n + p \left(C_3(2rk^r n + 1) + C_2 \left(2rk^r n + 1\right)^2 + 1 \right) \\ &< C_1 n + p \left(C_3 + C_2 + 1\right) \left(4rk^r n\right)^2 \\ &= C_1 n + p \left(C_3 + C_2 + 1\right) \left(16r^2 k^{2r}\right) n^2 \\ &= \mathcal{O}(n^2). \end{aligned}$$

□

2.3.2 *Clases P, NP y co-NP*

Independientemente de cual sea el modelo de la máquina M, se dice que M tiene *complejidad temporal polinomial*, si existe un polinomio $p(n)$ tal que $T_M(n) = \mathcal{O}(p(n))$. Esto nos lleva a distinguir 2 tipos de clases de lenguajes que son fundamentales en teoría de la complejidad computacional.

complejidad temporal polinomial

Definición 2.3.8. *La* clase de complejidad P*, es la clase de lenguajes* L *para los que existe una máquina de Turing* M *tal que* $L_M = L$ *y* M *tiene complejidad temporal polinomial.*

clase P

clase NP

Definición 2.3.9. *La* clase de complejidad NP*, es la clase de lenguajes* L *para los que existe una máquina de Turing no determinista* N *tal que* $L_N = L$ *y* N *tiene complejidad temporal polinomial.*

Usando la notación $\mathcal{O}$ grande podemos decir que un lenguaje L está en la clase de complejidad P, si existe una máquina de Turing M y un polinomio $p(n)$ tal que $L_M = L$ y $T_M(n) = \mathcal{O}(p(n))$. Una definición similar se puede hacer para la clase NP.

Una de las preguntas abiertas más importantes en la teoría de la complejidad computacional es determinar si la clase P y NP son iguales. Algunos de los resultados que se han desarrollado para responder a esta cuestión y algunas de las implicaciones que tiene la respuesta afirmativa o negativa a este problema, se estudian en el resto de este capitulo.

verificador de un lenguaje

Definición 2.3.10. *Sea* L *un lenguaje sobre el alfabeto* Σ_1 *y sea* $\#$ *un símbolo que* no *está en* Σ_1*. Sea* V *una* máquina de Turing total *con alfabeto de entrada* $\Sigma_2 \supseteq \Sigma_1 \cup \{\#\}$*. Se dice que la máquina* V *es un* verificador del lenguaje L *si se cumple que*

$$L = \{\, w \in \Sigma_1^* \mid V \text{ acepta a } w\#s \text{ para alguna cadena } s \in \Sigma_2^* \setminus \{\#\}\,\}.$$

certificado

Observación 2.3.6. *A la cadena* s *de la definición 2.3.10 se le llama* certificado *de pertenencia al lenguaje* L.

verificador con tiempo polinomial

Observación 2.3.7. *Se dice que el verificador* V *de la definición 2.3.10 es un* verificador con tiempo polinomial*, si existe un polinomio* p *tal que para toda cadena* $w\#s$ *la máquina* V *tiene complejidad temporal* $T_V(|w\#s|) = \mathcal{O}(p(|w|))$.

Lema 2.3.3. *Sea* V *un verificador con tiempo polinomial* $\mathcal{O}(p(n))$ *del lenguaje* L*. Para toda cadena* $w \in L$ *existe una cadena* s *y una constante* c *tal que* $|s| \leqslant cp(n)$ *y* $w\#s$ *es una cadena que acepta* V.

Demostración. Suponga que la afirmación del lema es falsa. Sea w una cadena del lenguaje L con $|w| = n$. Sea s la cadena con la menor longitud que hace que la cadena $w\#s$ sea aceptada por V. Puesto que V tiene tiempo polinomial, existe una constante c tal que la máquina solo puede leer a lo más $cp(n)$ símbolos de la cadena s. Por lo tanto, la cadena s se puede escribir como $s = s_1 s_2$, donde $s1$ es una cadena con $|s_1| \leqslant Cp(n)$ y s_2 es una cadena con $|s_2| = |s| - |s_1|$.

Como la máquina no lee ninguno de los símbolos de la cadena s_2, entonces la cadena $w\#s_1$ es aceptada por la máquina V, contradiciendo la hipótesis de que s es la cadena más pequeña que cumple dicha condición. □

El siguiente teorema muestra que la definición de la clase de complejidad NP se puede hacer en términos de verificadores con tiempo polinomial.

Teorema 2.3.5. L *es un lenguaje en la clase* NP *si y solo si existe un verificador* V *del lenguaje* L *con tiempo polinomial.*

Demostración.
($\Rightarrow$) La idea intuitiva de la demostración es construir un verificador del lenguaje L que toma como cadenas testigo a una sucesión de movimientos de la máquina N.

Sea L un lenguaje sobre el alfabeto Σ_1 que está en la clase NP, existe una máquina de Turing no determinista N y un polinomio $p(n)$ tal que $T_M(n) \leqslant p(n)$ y $L_N = L$. Por lo tanto, existen constantes C_1 y r tal que para todo $n \geqslant 1$ se cumple que $p(n) \leqslant C_1 n^r$.

Sea δ_N la función de transición de la máquina N. Por las mismas observaciones que se hicieron en la demostración del teorema 2.2.2, existe un entero k tal que $|\delta_N(q, x)| \leqslant k$, para todo estado q y símbolo x de la máquina N. Además, es posible asignar a cada configuración de $\delta(q, x)$ un número entre el 0 y $k-1$.

Sea w una cadena en Σ_1^* con $|w| = n$ y sea s una cadena en $\{0, 1, \ldots, k-1\}^+$ (es decir, s representa un número en base k). Defina una máquina de Turing M con 7 cintas que tiene alfabeto de entrada $\Sigma_2 = \Sigma_1 \cup \{\#\} \cup \{0, 1, \ldots, m-1\}$. La máquina M recibe cadenas de entrada de la forma $w\#s$ (esta cadena se coloca en la primera cinta).

La *primera cinta* de la máquina M se usa para almacenar el número s, la *segunda cinta* se usa para simular a la máquina N cuando tiene entrada w, la *tercera cinta* se usa para guardar un contador, el *resto de las cintas* se usan para hacer multiplicaciones y calcular potencias.

La máquina M ejecuta las siguientes reglas en estricto orden:

1. Copiar la cadena w a la segunda cinta y en la primera cinta borrar la cadena $w\#$. Inicializar el contador en la tercera cinta a 0. Colocar los cabezales en la primera y segunda cinta al principio de las cadenas s y w respectivamente.
2. Usar las cintas de la 4 a la 7 para calcular el valor de $C_1|w|^p = C_1 n^r$.
3. Sea q_0 el estado inicial de la máquina N, guardar dicho estado en la unidad de control de la máquina M.

4. Sea q el estado que guarda actualmente la unidad de control de la máquina M. Sea x el símbolo que lee la máquina en la *segunda cinta* y sea y el símbolo que lee en la *primera cinta*.

   Si y es el símbolo blanco B, la máquina M termina en estado de rechazo, de lo contrario sea (q', x', D) la configuración con número y de $\delta_N(q, x)$ y sean q_A y q_R los estados de rechazo y aceptación de la máquina N. La máquina M simula el movimiento que corresponde a la configuración (q', x', D) en la segunda cinta y guarda el estado q' en la unidad de control de la máquina M. Si $q' = q_A$ la máquina M termina en estado de aceptación, si $q' = q_R$ la máquina M termina en estado de rechazo.
5. Mover el cabezal en la primera cinta una casilla a la derecha.
6. Incrementar el contador c en la tercera cinta en una unidad. Si $c = C_1 n^r + 1$, la máquina termina en estado de rechazo, de lo contrario volver a ejecutar las reglas 4 en adelante.

Por la regla 6 se tiene la garantía de que la máquina M se detiene con toda cadena de entrada. Si w es una cadena del lenguaje L entonces existe una sucesión de movimientos s con $|s| \leqslant C_1 n^r$, que hacen que la máquina N acepte a w, por lo cual la cadena $w\#s$ es aceptada por la máquina M. De manera inversa, si $w\#s$ es una cadena que acepta M, entonces $|s| \leqslant C_1 n^r$ y s es una sucesión de movimientos que hacen a la máquina N aceptar a la cadena w. De estas observaciones y del teorema 2.2.1, una máquina de Turing V que simula a la máquina multicinta M es un verificador del lenguaje L.

Para determinar la complejidad temporal del verificador V, primero calculamos la complejidad temporal de la máquina M. Puesto que la regla 1 requiere un número de movimientos proporcional a la cadena w, existe una constante C_2 tal que en el peor caso la regla 1 requiere $C_2|w| = C_2 n$ movimientos.

La regla 3 solo requiere un movimiento y por el teorema 2.3.4 incisos *(c)* y *(d)*, existe una constante C_3 tal que en el peor caso las reglas 2 y 3 requieren $C_3 n^2$ movimientos.

Las reglas 4 y 5 se pueden hacer en 1 solo movimiento. Por el lema 2.3.1 y por el teorema 2.3.4 inciso *(a)*, existe una constante C_4 tal que en el peor caso las regla 6 requiere $C_4 \lceil \log_{10}(C_1 n^r) \rceil < C_1 C_4 n^r + C_4$ movimientos.

Puesto que las reglas 4, 5 y 6 se repiten a lo más $C_1 n^p + 1$ veces. la complejidad temporal de la máquina M está acotada superiormente por

$$\begin{aligned} T_M(|w\#s|) &= T_M(n + |s| + 1) \\ &< C_2 n + C_3 n^2 + (C_1 n^r + 1)(C_1 C_4 n^r + C_4) \\ &= C_4 + C_2 n + C_3 n^2 + 2C_1 C_4 n^r + C_1^2 C_4 n^{2r}. \end{aligned}$$

Como r es una constante, la máquina M tiene complejidad temporal polinomial y por el teorema 2.3.3, el verificador V tiene complejidad $T_V(n) = \mathcal{O}\left((T_M(n))^2\right)$. Por lo tanto, V es un verificador con tiempo polinomial.

($\Leftarrow$)

Sea V un verificador con tiempo polinomial del lenguaje $L \subset \Sigma_1^*$. Sea $\Sigma_2 \supseteq \Sigma_1 \cup \{\#\}$ el alfabeto de entrada de V. Defina una máquina de Turing no determinista N, con alfabeto de entrada Σ_1. La máquina N para cadenas $w \in \Sigma_1$ opera de acuerdo a las siguientes reglas:

1. Escribir el símbolo # al final de w (el símbolo # es parte del alfabeto de cinta de N).
2. Adivinar una cadena $s \in \Sigma_2^* \setminus \{\#\}$ y escribir dicha cadena al final de $w\#$.
3. Colocar el cabezal al principio de la cadena $w\#s$.
4. Ejecutar el verificador V con la cadena $w' = w\#s$. Si V acepta a w', entonces N termina en estado de aceptación.

Por la regla 4, si w es una cadena que acepta N, entonces existe una cadena s tal que el verificador acepta a $w\#s$. De manera inversa, si la cadena $w\#s$ es aceptada por el verificador V, entonces por las reglas 2 y 4 la máquina N acepta a la cadena w. Por lo tanto $L_N = L$.

La regla 1 requiere $|w| + 1$ movimientos. La regla 2 en una máquina no determinista requiere $|s|$ movimientos. La regla 3 requiere $|s| + |w| + 2$ movimientos.

Puesto que V es un verificador con tiempo polinomial, existen constantes C_1 y r tales que la regla 3 requiere a lo más, $C_1|w|^r$ movimientos.

Por lo tanto, el número de movimientos m que requiere la máquina N para aceptar o rechazar una cadena de entrada w es

$$\begin{aligned} m &\leqslant (|w| + 1) + |s| + (|s| + |w| + 2) + C_1|w|^r \\ &= 3 + 2|s| + 2|w| + C_1|w|^r. \end{aligned} \tag{1}$$

Si $w \in L$, por el lema 2.3.3 existe una constante C_2 y una cadena s tal que $|s| \leqslant C_2|w|^r$. Por lo tanto, si m_w es el mínimo número de movimientos que requiere la máquina N para aceptar a w, de la desigualdad (1) se tiene que

$$m_w \leqslant 3 + 2|w| + (2C_2 + C_1)|w|^r.$$

Por lo tanto, la complejidad temporal de la máquina N con $|w| = n$ es

$$T_N(n) = \max\{0, m_w\} \leqslant \max\{0, 3 + 2n + (2C_2 + C_1)n^r\} = \mathcal{O}(n^r).$$

Como N tiene complejidad temporal polinomial, concluimos que L está en la clase NP. □

El siguiente teorema nos permitirán demostrar que un lenguaje en la clase NP, puede ser reconocido por una máquina de Turing en *tiempo exponencial*.

Teorema 2.3.6. *Sea* N *una máquina de Turing no determinista con* $T_N(n) \leqslant f(n)$. *Existe una máquina de Turing determinista* M *tal que* $L_M = L_N$ *y* $T_M(n) = \mathcal{O}\left(2^{f(n)}\right)$

Demostración. Sea δ_N la función de transición de N y sea k el número máximo de configuraciones posibles de δ_N (cardinalidad máxima de los conjuntos en el contradominio). Si $k = 1$ la máquina es esencialmente una máquina de Turing determinista y la afirmación del teorema es trivial. Suponga que $k > 1$.

Puesto que la máquina N acepta o rechaza una cadena $w \in L_N$ en no más de $T_N(n) = f(n)$ movimientos, el árbol de ejecución de la máquina N tiene una altura menor que $f(n)$. Por el lema 2.1.1, el árbol de ejecución de la máquina N tiene a lo más $k^{f(n)}$ hojas y por el teorema 2.1.1, el total de vértices no puede ser mayor que $2k^{f(n)}$.

Construya una máquina de Turing con 3 cintas S que simula a la máquina N usando las mismas reglas que se usaron para construir la máquina de 3 cintas del teorema 2.2.2; con la diferencia de que si el contador de la regla 2 llega a $2k^{f(n)}$, la máquina S se detiene en el estado de rechazo. Esta condición extra en el contador de la cinta 2, garantiza que la máquina S acepta (o rechaza) la cadena w si y solo si la máquina N acepta (o rechaza) la cadena w.

Para calcular la complejidad temporal de la máquina S, recuerde que la máquina S realiza una simulación de N en la tercera cinta (por medio la configuración que indica el contador de la cinta 2), por lo tanto, la complejidad temporal de cada simulación en la cinta 3 está en $\mathcal{O}(f(n))$. Como la

máquina S a lo más tendrá que realizar $2k^{f(n)}$ simulaciones (valor máximo del contador en la cinta 2), se tiene que

$$T_S(n) = \mathcal{O}\left(2f(n)k^{f(n)}\right) = \mathcal{O}\left(f(n)k^{f(n)}\right).$$

Como

$$k^{f(n)} = 2^{\log_2(k^{f(n)})} = 2^{\log_2(k)f(n)} = 2^{\log_2(k)}2^{f(n)} = \mathcal{O}(2^{f(n)}),$$
$$f(n) = 2^{\log_2(f(n))} < 2^{f(n)} = \mathcal{O}\left(2^{f(n)}\right).$$

La complejidad temporal de la máquina S es

$$T_S(n) = \mathcal{O}\left(f(n)k^{f(n)}\right) = \mathcal{O}\left(2^{2f(n)}\right) = \mathcal{O}\left(2^{f(n)}\right).$$

Finalmente, defina una máquina de Turing M que simula a N usando la construcción propuesta en el teorema 2.2.1. Por el teorema 2.3.3, la complejidad temporal de la máquina M es

$$\begin{aligned} T_M(n) &= \mathcal{O}\left((T_S(n))^2\right) = \mathcal{O}\left(\left(2^{f(n)}\right)^2\right) \\ &= \mathcal{O}\left(2^{2f(n)}\right) = \mathcal{O}\left(2^{f(n)}\right). \end{aligned}$$

□

Corolario 2.3.1. *Sea* L *un lenguaje en la clase NP. Existe un polinomio* $p(n)$ *y una máquina de Turing total* M *tal que* $L = L_M$ *y* $T_M(n) = \mathcal{O}(2^{p(n)})$

Demostración. Como L es un lenguaje en la clase NP, existe una máquina de Turing no determinista N tal que $L = L_N$ y $T_N(n) \leqslant p(n)$. Por el teorema 2.3.6, existe una máquina de Turing total M con $T_M(n) = \mathcal{O}(2^{p(n)})$ □

A la fecha no se sabe si existe una mejor estrategia que permita reducir el tiempo del corolario 2.3.1. Observe que si la cota de complejidad de la simulación propuesta en el teorema 2.3.6, pudiera reducirse a un orden polinomial, se tendría inmediatamente que P = NP.

Si L es un lenguaje sobre el alfabeto Σ, definimos al *complemento del lenguaje* L como el conjunto $L^c = \Sigma^* \setminus L$. Esto nos lleva a definir la siguiente clase de complejidad.

complemento del lenguaje

Definición 2.3.11. *La clase de complejidad* co-NP *es la clase de lenguajes* L *para los que su complemento* L^c *está en la clase* NP.

No se sabe si *NP=co-NP*. Algunos de los teoremas que se estudian en la siguiente sección, tienen implicaciones que podrían ayudar a resolver este problema abierto.

2.3.3 *Clase NP-completo*

Para fines prácticos, un algoritmo que requiere tiempo exponencial deja de ser útil a medida que aumenta la longitud de la cadena de entrada, ya que por la naturaleza de las funciones exponenciales, puede ocurrir que el número de pasos que requiere el algoritmo excede el número de microsegundos que tiene el universo. Es por esta cuestión que resulta muy importante tener algoritmos que puedan trabajar en tiempo polinomial.

En ocasiones puede ocurrir que no se conozca un algoritmo para resolver un problema, sin embargo sería posible resolver dicho problema si pudiéramos "transformarlo" a otro problema equivalente para el cual sí tenemos un algoritmo. Si además dicha transformación podemos hacerla en "poco tiempo", la solución dependería principalmente del tiempo del algoritmo que ya conocemos. La siguiente definición permite formalizar esta idea.

reducción de Karp

Definición 2.3.12. *Sean* $L_1 \subseteq \Sigma_1^*$ *y* $L_2 \subseteq \Sigma_2^*$ *lenguajes sobre los alfabetos* Σ_1 *y* Σ_2 *respectivamente. Una* transformación polinomial *o* reducción de Karp[1] *es una función* $f : \Sigma_1^* \longrightarrow \Sigma_2^*$ *que cumple las siguientes condiciones:*

1. *Existe una máquina de Turing* M *con complejidad temporal polinomial tal que* f *es una función computable por* M *(ver definición 2.2.6).*
2. *Toda cadena* $w \in \Sigma_1^*$ *está en* L_1 *si y solo si la cadena* $f(w)$ *está en* L_2.

Observación 2.3.8. *Usaremos la notación* $L_1 \propto L_2$ *para indicar que hay una transformación polinomial de* L_1 *a* L_2. *Por simplicidad diremos que* L_1 *se transforma* L_2 *para referirnos a* $L_1 \propto L_2$.

La siguiente clase de lenguajes resulta fundamental para investigar la cuestión de si P es igual a NP. Intuitivamente esta clase abarca a los problemas en NP que son los más "difíciles" (algunos de estos problemas se estudian en la sección 2.3.5).

NP-completo

Definición 2.3.13. *Un lenguaje* L *es NP-completo si cumple con las 2 condiciones siguientes:*

1. $L \in NP$

[1] En honor del matemático estadounidense Richard M. Karp.

2. $\forall L' \in NP,\ L' \propto L$

La importancia de la clase NP-completo se deriva del siguiente teorema.

Teorema 2.3.7. *Sean* $L_1 \subseteq \Sigma^*$ *y* $L_2 \subseteq \Sigma^*$ *lenguajes con alfabetos* Σ_1 *y* Σ_2 *respectivamente. Suponga el lenguaje* L_1 *se transforma al lenguaje* L_2*. Entonces se cumple lo siguiente:*

$$L_2 \textit{ está en la clase } P \Longrightarrow L_1 \textit{ está en la clase } P.$$

Demostración. Sea $f(n)$ la función que cumple las condiciones de la definición 2.3.12 para $L_1 \propto L_2$. Existe una máquina de Turing M_f y un polinomio $p_f(n)$ tal que M_f calcula a la función $f(n)$ y $T_{M_f}(n) \leqslant p_f(n)$.

Como L_2 está en la clase P, existe una máquina de Turing M_2 y un polinomio $p_2(n)$ tal que $T_{M_2}(n) \leqslant p_2(n)$ y $L_{M_2} = L_2$

Construimos una máquina de Turing total M_1 que reconoce a L_1 por medio de las siguientes reglas:

1. Al leer la cadena de entrada $w \in \Sigma_1^*$, la máquina calcula $f(w) \in \Sigma_2^*$ por medio de las reglas de la máquina M_f.
2. Posteriormente la máquina usa las reglas de la máquina M_2 para aceptar o rechazar a la cadena $f(w)$.

Por la definición de transformación polinomial, tenemos que $w \in L_1$ si y solo si $f(w) \in L_2$. Por lo tanto, el lenguaje que acepta la máquina M_1 es L_1, es decir $L_{M_1} = L_1$.

Para acotar a la función de complejidad temporal $T_{M_1}(n)$ de la máquina M_1, observe que con $|w| = n$, la complejidad temporal de la regla 1 es proporcional a $p_f(n)$. Debido a que $|f(w)| \leqslant p_f(|w|)$, la complejidad de la regla 2 es proporcional a $p_2(p_f(n))$. Por lo tanto, la complejidad temporal de M_1 es

$$T_{M_1}(n) = \mathcal{O}\left(p_f(n) + p_2(p_f(n))\right).$$

Concluimos que M_1 tiene complejidad temporal polinomial y por lo tanto L_1 está en la clase P. □

El teorema 2.3.7 tiene una consecuencia inmediata muy interesante: si existe un lenguaje NP-completo que puede ser reconocido por una máquina de

Turing en tiempo polinomial, entonces se debe de cumplir que $P = NP$. Por lo tanto, si se puede demostrar que un lenguaje es NP-completo, entonces se tiene la certeza de que no se conoce un algoritmo que puede reconocer a dicho lenguaje en tiempo polinomial.

Los siguientes resultados son de gran utilidad para demostrar que un lenguaje es NP-completo.

transitividad de $\propto$

Lema 2.3.4. *Si* $L_1 \propto L_2$ *y* $L_2 \propto L_3$ *entonces* $L_1 \propto L_3$.

Demostración. Sean Σ_1^*, Σ_2^* y Σ_3^* los alfabetos de los lenguajes L_1, L_2 y L_3 respectivamente. Como $L_1 \propto L_2$, existe una función $f_1 : L_1 \to L_2$ y una máquina M_1 que calcula la función f_1 en tiempo polinomial $p_1(n)$. Como $L_2 \propto L_3$, existe una función $f_2 : L_2 \to L_3$ y una máquina M_2 que calcula a la función f_2 en tiempo polinomial $p_1(n)$.

La función $f_3 : L_1 \to L_3$ definida como $f_3 = (f_2(f_1(n)))$ cumple que $w \in L_1$ si y solo si $f_3(w) \in L_3$. Construya una máquina M_3 que calcula la función f_3 de acuerdo a las siguientes reglas:

1. Con la cadena de entrada w la máquina M_3 calcula a $f_1(w)$ usando las reglas de la máquina M_1.
2. Posteriormente, con la cadena $z = f_1(w) \in L_2$, la máquina M_3 calcula $f_2(z)$ usando las reglas de la máquina M_2.

Con $|w| = n$, el tiempo requerido para la regla 1 es $\mathcal{O}(p_1(n))$. Como $|f_1(w)| \leqslant p_1(|w|)$, el tiempo de la regla 2 es $\mathcal{O}(p_2(p_1(n)))$. Por lo tanto, el tiempo total de M_3 es $\mathcal{O}(p_1(n)) + \mathcal{O}(p_2(p_1(n)))$, el cual está acotado por un polinomio. Luego $L_1 \propto L_3$. □

Teorema 2.3.8. *Sean* L_1 *y* L_2 *lenguajes en la clase* NP. *Si* L_1 *es NP-completo y* $L_1 \propto L_2$ *entonces* L_2 *es NP-completo.*

Demostración. Puesto que L_2 está en la clase NP, solo hace falta probar que cumple la condición 2 de la definición 2.3.13. Sea L' un lenguaje en NP. Como L_1 es NP-completo se tiene que $L' \propto L_1$. Por hipótesis $L_1 \propto L_2$ y por el lema 2.3.4, se tiene que $L' \propto L_2$. □

Como ya se mencionó, determinar si NP=co-NP es un problema abierto. Esta cuestión junto con el siguiente lema, son de mucha utilidad para justificar que un determinado lenguaje *posiblemente* no está en la clase NP[1].

Lema 2.3.5. *Sea* L *un lenguaje NP-completo tal que su complemento* L^c *está en la clase NP. Entonces NP=co-Np.*

Demostración. Sea L' un lenguaje en la clase NP. Como L es NP-completo, existe una reducción de Karp f tal que $w \in L'$ si y solo si $f(w) \in L$, en consecuencia $w' \in (L')^c$ si y solo si $f(w') \in L^c$. Por hipótesis, L^c está en la clase NP y en consecuencia existe una máquina no determinista N con complejidad temporal polinomial tal que $L_N = L^c$.

Para reconocer al lenguaje $(L')^c$, defina una máquina N' que al recibir la cadena de entrada w calcula $f(w)$ y a continuación ejecuta la máquina N con la cadena de entrada $f(w)$, la máquina N' acepta a w si la máquina N acepta a $f(w)$. Puesto que calcular f y ejecutar N requiere tiempo polinomial, la máquina N' está en la clase NP. □

2.3.4 *Clase NP-difícil*

Intuitivamente, los lenguajes que están en la clase NP-completo son más difíciles de reconocer que los lenguajes en la clase P. Sin embargo, existen lenguajes que intuitivamente parecen ser todavía más difíciles de reconocer que los lenguajes que están en la clase NP-completo. Para estudiar a dichos lenguajes necesitamos la siguiente definición [2].

Definición 2.3.14 (Garey [11, p. 113]). *Sean* R_1 *y* R_2 *relaciones de cadenas sobre los alfabetos* Σ_1 *y* Σ_2 *respectivamente. Sea* M *una máquina de Turing con oráculo que tiene alfabeto de entrada* Σ_1 *y que tiene complejidad temporal polinomial.*
Si para toda función $g : \Sigma_2^* \to \Sigma_2^*$ *que realiza a* R_2*, la máquina* M *al usar a* g *como oráculo cumple que su función asociada* f_M *realiza a* R_1*, entonces se dice que* M *es una* reducción de Cook[3] *o* reducción polinomial de Turing *de* R_1 *a* R_2*.* *reducción de Cook/Turing*

Observación 2.3.9. *Usaremos la notación* $R_1 \propto_T R_2$ *para indicar que existe una reducción de Turing/Cook de la relación de cadenas* R_1 *a la relación de cadenas* R_2*.* *notación* $\propto_T$

[1] Un problema que posiblemente no está en la clase NP se estudia en el siguiente capítulo en la sección 3.1.

[2] El lector puede repasar los conceptos de *relación de cadenas*, *función que realiza a una relación de cadenas* y *función asociada con la máquina oráculo* en las secciones 2.2.14, 2.2.15 y 2.2.12 respectivamente.

[3] En honor al matemático estadounidense Stephen Arthur Cook.

De manera intuitiva, una reducción de Cook formaliza la idea de que es posible tener un algoritmo A_1, que resuelve un problema de manera "eficiente" usando como parte de su programa otro algoritmo A_2, este último no necesariamente es un algoritmo conocido. Mediante la reducción de Cook es posible formular la siguiente clase de complejidad computacional.

relación de cadenas NP-difícil

Definición 2.3.15 (Garey [11, p. 113]). *Sea* L *un lenguaje NP-completo y sea* R_L *la relación de cadenas identificada con el lenguaje* L*, tal como se definió en 2.2.6. Se dice que una relación de cadenas* R *es NP-difícil si* $R_L \propto_T R$.

clase NP-difícil

Definición 2.3.16. *La* clase NP-difícil *son todos los lenguajes* L *para los que* R_L *es NP-difícil.*

Aunque la definición 2.3.15 permite extender la clase NP-difícil a problemas de búsqueda, para los fines de este trabajo la definición 2.3.16 es suficiente.

Los siguientes resultados son de mucha utilidad para demostrar que un lenguaje es NP-difícil, usando una reducción de Karp en lugar de una reducción de Cook.

Teorema 2.3.9. *Sean* L_1 *y* L_2 *lenguajes sobre los alfabetos* Σ_1 *y* Σ_2 *respectivamente. Sean* R_{L_1} *y* R_{L_2} *las relaciones de cadenas identificadas con* L_1 *y* L_2 *respectivamente. Si existe una reducción de Karp tal que* $L_1 \propto L_2$ *entonces existe una reducción de Cook tal que* $R_{L_1} \propto_T R_{L_2}$.

Demostración. Como $L_1 \propto L_2$, existe una máquina de Turing M con complejidad temporal polinomial tal que $w \in L_1$ si y solo si $f_M(w) \in L_2$. Sea s un símbolo fijo en Σ_1 y sea $g : \Sigma_2^* \to \Sigma_2^*$ una función que realiza a la relación de cadenas R_{L_2}. Defina una máquina de Turing con oráculo M' que tiene alfabeto de entrada Σ_1 y que para toda cadena de entrada $w \in L_1$ opera de acuerdo a las siguientes reglas:

1. Calcular $f_M(w)$ usando la máquina de Turing M.
2. calcular $w' = g(f_M(w))$. Si $w' = \epsilon$, limpiar la cinta primaria de la máquina (poner en cada casillas el símbolo blanco B). Si $w' \neq \epsilon$ imprimir en la cinta primaria el símbolo s.

Por la regla 2, la máquina M' realiza a la relación de cadenas R_{L_1} y como las reglas 1 y 2 requieren tiempo polinomial, la máquina M' es una reducción de Cook de R_{L_1} a R_{L_2}. □

Corolario 2.3.2. *Sea* L *un lenguaje tal que para todo lenguaje* L' *en la clase* NP*, se cumple que* $L' \propto L$*. Entonces* L *está en la clase NP-difícil.*

Demostración. Puesto que todo lenguaje L' en la clase NP cumple que $L' \propto L$, en particular un lenguaje L_c que sea NP-completo cumple esta condición. Por el teorema 2.3.9, se tiene que $R_{L_c} \propto_T R_L$. Por lo tanto, L está en la clase NP-difícil. □

Del corolario 2.3.2, los lenguajes en la clase NP-completo también están en la clase NP-difícil. Sin embargo, existen lenguajes que están en la clase NP-difícil que no se sabe si están en la clase NP[1].

2.3.5 *Problemas de decisión*

problema de decisión

Un *problema de decisión* es una pregunta que solo admite como respuesta *"sí"* o *"no"*. La pregunta se plantea de tal forma que es necesario especificar ciertos parámetros de entrada a fin de que la pregunta tenga sentido. El siguiente es un ejemplo de un problema de decisión:

> *Parámetros de entrada*: Una gráfica $G = (V, E)$ y un entero positivo $k \leqslant |V(G)|$.
>
> *Pregunta*: ¿Tiene la gráfica G un clan de tamaño mayor o igual a k?

instancia de un problema de decisión

Cuando en el problema de decisión se especifican valores para los parámetros de entrada, se dice que se tiene una *instancia* del problema de decisión.

conjunto D_Π
conjunto Y_Π
conjunto N_Π

Si Π es un problema de decisión, denotaremos por D_Π al conjunto de instancias de Π. Denotaremos por Y_Π al conjunto de instancias del problema Π que tienen como respuesta un *"si"* y denotaremos por N_Π al conjunto de instancias del problema Π que tienen como respuesta un *"no"*. Observe que $N_\Pi = D_\Pi \setminus Y_\Pi$.

Problema complementario Π^c

A partir del problema Π podemos formar un nuevo problema de decisión Π^c que consta de todas las instancias de Π con las respuestas invertidas, esto es, $Y_{\Pi^c} = N_\Pi$ y $N_{\Pi^c} = Y_\Pi$. Al problema Π^c se le conoce como *problema de decisión complementario* del problema Π.

Con la codificación adecuada podemos representar a cada instancia del problema de decisión Π, como una cadena w sobre un alfabeto Σ. De esta

[1] Un problema con tales características se estudia en la sección 3.2.

forma el problema Π induce una partición en Σ^* que consta de los siguientes subconjuntos:

lenguaje asociado a un problema de decisión

- El subconjunto de cadenas $w \in \Sigma^*$ que representan instancias de Y_Π. A este subconjunto lo denotaremos por $L[\Pi]$ y se le denomina *lenguaje asociado* al problema de decisión Π.
- El subconjunto de cadenas $w \in \Sigma^*$ que representan instancias en N_Π.
- El subconjunto de cadenas $w \in \Sigma^*$ que no representan a ninguna instancia de D_Π.

En ocasiones, por simple conveniencia, usaremos los conceptos de problema de decisión Π y su lenguaje asociado $L[\Pi]$ como si fueran sinónimos, aunque estrictamente hablando en todo momento estaremos trabajando con $L[\Pi]$.

procedimiento de decisión

Si existe una máquina de Turing total M tal que $L_M = L[\Pi]$, se dice que la máquina M *resuelve* al problema de decisión Π, también se dice que Π tiene un *procedimiento de decisión*.

Se dice que un problema de decisión Π está en las clase P, si su lenguaje asociado $L[\Pi]$ está en la clase P. Definiciones similares se hacen para las clases de complejidad que se definieron en las secciones anteriores.

Observe que el complemento del lenguaje asociado con un problema de decisión Π no es lo mismo que el lenguaje asociado al problema complementario Π^c, es decir $L[\Pi]^c \neq L[\Pi^c]$. Esto se debe a que $L[\Pi]^c$ contiene cadenas que no representa a ninguna instancia de Π. Observe que sí se cumple que $L[\Pi^c] \subsetneq L[\Pi]^c$. El siguiente lema muestra que esta cuestión, realmente no es relevante para la clase NP.

Lema 2.3.6. *Sea Π un problema de decisión y sea Π^c su problema complementario. Si $L[\Pi^c]$ está en la clase NP entonces $L[\Pi]^c$ también está en la clase NP.*

Demostración. Puesto que $L[\Pi^c]$ está en la clase NP, existe una máquina de Turing no determinista N con tiempo polinomial tal que N reconoce al lenguaje $L[\Pi^c]$. Defina una máquina no determinista N' que reconoce al lenguaje $L[\Pi]^c$ usando las siguientes reglas:

1. Para toda cadena de entrada w, verificar de manera determinista que w es una codificación de una instancia de Π. Si w no representa una instancia aceptar la cadena, de lo contrario, continuar con la regla 2.
2. Ejecutar la máquina N con la cadena de entrada w, si la máquina N acepta a w entonces N' también acepta a w.

La regla 1 se puede hacer en tiempo proporcional a la longitud de la cadena de entrada y la regla 2 requiere tiempo polinomial, por lo tanto N' tiene complejidad temporal polinomial y en consecuencia $L[\Pi]^c$ está en la clase NP. □

Una estrategia muy poderosa que surge del teorema 2.3.8 para demostrar que un problema Π_2 es NP-completo a partir de otro problema Π_1 que se sabe que es NP-completo, consiste en demostrar que Π_2 está en NP y que existe una reducción de Karp de Π_1 a Π_2. Más adelante usaremos esta estrategia para demostrar que el problema de decisión CLIQUE es NP-completo.

El primer problema de decisión que se demostró que es NP-completo es el *problema de satisfabilidad booleana*. La terminología formal para dicho problema se describe a continuación.

Sea $B = \{x_1, x_2, \ldots, x_n\}$ un conjunto que denominaremos como conjunto de *variables Booleanas*. Una *función de verdad* es una función de la forma $f : B \to \{V, F\}$. Si la variable booleana $x \in B$ cumple que $f(x) = V$, se dice que x es *verdadero* sobre f. Por otra parte, si $f(x) = F$ se dice que x es *falso*. El literal $\overline{x}$ es verdadero sobre f si y solo si x es falso sobre f. En general, si x es una variable booleana diremos que $\overline{x}$ y x son *literales sobre B*.

función de verdad

literales

Una *cláusula* sobre B es un conjunto de literales sobre B, por ejemplo, $U = \{x_1, x_2, \overline{x_3}\}$ representa la *disyunción lógica* de dichos literales, esto es, U representa a $x_1 \vee x_2 \vee \overline{x_3}$. Se dice que una cláusula se *satisface* por una función de verdad f si y solo si al menos uno de sus miembros es verdadero sobre f; por ejemplo, la cláusula U se satisface a menos que $f(x_1) = F$, $f(x_2) = F$ y $f(x_3) = V$.

cláusula

Una colección de cláusulas C sobre B se *satisface* si y solo si existe una función de verdad f que satisface simultáneamente todas las cláusulas de C; en tal caso se dice que *f es una función que satisface a C*. La colección de cláusulas representa una *conjunción lógica* de las cláusulas[1] de C. Esto es, si $C = \{\{x_1, \overline{x_2}\}, \{x_1, x_2\}, \{x_3\}\}$, entonces C representa a $(x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (x_3)$. Con esta terminología se define el problema de la satisfabilidad booleana de la siguiente manera.

colección de cláusulas

satisfabilidad

Definición 2.3.17. *Instancia: Colección de cláusulas C sobre el conjunto de variables booleanas B. Pregunta: ¿Existe una función de verdad f que satisface a C?*

problema de la satisfabilidad booleana

[1] Las expresiones booleanas que representan a C se dice que están en forma normal conjuntiva (FNC)

Observación 2.3.10. *Por lo general, se usan las siglas SAT para referirse al problema de la satisfabilidad booleana.*

3-SAT

Si en la definición 2.3.17 se pone la restricción de que cada cláusula debe tener exactamente 3 literales, se dice que es un problema de *3-satisfabilidad* (3-SAT).

El descubrimiento de que SAT es NP-completo, fue hecho de manera independiente primero por el matemático estadounidense Stephen Cook [7] y poco después por el matemático ruso Leonid Levin [20]. La idea intuitiva de la demostración de dicho teorema es que para cada lenguaje L en NP, sabemos que existe una máquina no determinista N que reconoce a L en tiempo polinomial. La idea entonces es dar una reducción de L a SAT tal que para toda cadena $w \in L$, la reducción produce una formula booleana que simula a la máquina N con la cadena de entrada w.

Enunciamos el teorema de Cook-Levin sin su demostración ya que la prueba es demasiado extensa y nos alejaríamos de los propósitos principales de este trabajo[1].

Teorema de Cook-Levin

Teorema 2.3.10 (Cook [7], Levin [20])**.** *SAT es NP-completo*

Posteriormente Richard Karp, usó el resultado de Cook para demostrar que 3-SAT es NP-completo [14]. Este resultado se demuestra en el siguiente teorema.

Teorema 2.3.11 (Karp [14])**.** *3-SAT es NP-completo*

Demostración. Sea I una cadena que representa una instancia de 3-SAT y sea S una cadena que representa un grupo de valores posibles para la función de verdad de I. Construya un verificador que recibe como entrada cadenas de la forma I#S, el verificador evalúa a la Instancia I usando los valores de S para la función de verdad. Puesto que esta operación puede hacerse en tiempo polinomial (con respecto a |I|) se tiene que 3-SAT está en NP.

Sea $B = \{x_1, x_2, \ldots, x_n\}$ un conjunto de variables Booleanas y sea $C = \{c_1, c_2, \ldots, c_m\}$ una colección de cláusulas sobre B. Se mostrará que es posible transformar C a una colección de cláusulas C' sobre el conjunto de variables Booleanas B' tal que cada cláusula de C', tiene 3 literales y C se satisface si y solo si C' se satisface.

[1] El lector puede consultar las demostraciones de estos teoremas en [11, p. 38-50]

Sea $c_i = \{z_1, z_2, \ldots, z_p\}$ una de las cláusulas de la colección C (cada z_j es un literal sobre B), defina la colección de cláusulas con 3 literales C'_i y el conjunto de variables Booleanas B'_i de acuerdo a los siguientes casos:

- Caso 1. $p = 1$.

$$B'_i = \{y_i^1, y_i^2\}$$
$$C'_i = \left\{\{z_1, y_i^1, y_i^2\}, \{z_1, y_i^1, \overline{y}_i^2\}, \{z_1, \overline{y}_i^1, y_i^2\}, \{z_1, \overline{y}_i^1, \overline{y}_i^2\}\right\}$$

- Caso 2. $p = 2$.

$$B'_i = \{y_i^1\}$$
$$C'_i = \left\{\{z_1, z_2, y_i^1\}, \{z_1, z_2, \overline{y}_i^1\}\right\}$$

- Caso 3. $p = 3$.

$$B'_i = \emptyset$$
$$C'_i = \{c_i\}$$

- Caso 4. $p > 3$.

$$B'_i = \{y_i^1, y_i^2, \ldots, y_i^{p-3}\}$$
$$C'_i = \left\{\{z_1, z_2, y_i^1\}\right\} \bigcup_{j=1}^{p-4} \left\{\{\overline{y}_i^j, z_{j+2}, y_i^{j+1}\}\right\} \bigcup \left\{\{\overline{y}_i^{p-3}, z_{p-1}, z_p\}\right\}$$

Defina la colección de cláusulas C' sobre el conjunto de variables Booleanas como

$$B' = B \cup \left\{\bigcup_{i=1}^{m} B'_i\right\} \qquad C' = \bigcup_{i=1}^{m} C'_i.$$

Suponga que C se satisface por una función de verdad $f : B \to \{V, F\}$, mostraremos que esto implica que C también se satisface. Defina la función de verdad $f' : B' \to \{V, F\}$ como $f'(x) = f(x)$ si $x \in B$. Si $x \in B' \setminus B = \bigcup_{i=1}^{m} B'_i$, entonces x está en alguno de los conjuntos B'_i, se tienen los siguientes casos:

- B'_i es un conjunto que se construyo usando el caso 1 o 2. Entonces hacer $f'(x) = V$, para todo $x \in B'_i$
- $B'_i = \{y_i^1, y_i^2, \ldots, y_i^{p-3}\}$ es un conjunto que se construyo usando el caso 4 con la cláusula $c_i = \{z_1, z2, \ldots, z_p\} \in C$. Por hipótesis, la cláusula c_i se satisface, por lo tanto, existe un índice $k \leqslant p$ tal que $z_k = V$. Consideramos los siguientes casos:

  - Si $k \leqslant 2$, definir $f'(x) = F$ para todo $x \in B'_i$.

- Si $3 \leqslant k \leqslant p-2$, definir $f'(y_i^j) = V$ para $1 \leqslant j \leqslant k-2$ y $f'(y_i^j) = F$ para $k-1 \leqslant j \leqslant p-3$.
- Si $k = p-1$ o si $k = p$, definir $f'(x) = V$ para todo $x \in B_i'$.

Es fácil comprobar que si la función f satisface a C entonces la función de verdad f' satisface a C'. Suponga ahora que C' se satisface, se mostrará por contradicción que para toda cláusula $c_i = \{z_1, z_2, \ldots, z_p\} \in C$, existe un índice k tal que $z_k = V$. Suponga entonces que esta afirmación es falsa, tenemos los siguientes casos:

- Si $p \neq 3$, entonces $y_i^1 \in B_i'$ cumple que $y_i^1 = V$ y $y_i^1 = F$, una contradicción.
- Si $p = 3$, entonces la única cláusula de C_i' no se satisface, contradiciendo la hipótesis de que la colección de cláusulas C' se satisface.

Por lo tanto, si C' se satisface entonces C se satisface. Finalmente, para ver que esta transformación requiere tiempo polinomial, observe que el número de cláusulas de 3 literales en C' está acotado por mn y en consecuencia, el tamaño de una cadena que representa a la instancia de 3-SAT, está acotado por un polinomio en función de la longitud de la cadena que representa a la instancia SAT. Puesto que las operaciones de unión requieren un tiempo polinomial que depende de la longitud de la instancia SAT, esta transformación requiere tiempo polinomial y por el teorema 2.3.10 y el lema 2.3.8 se concluye que 3-SAT es NP-completo.

□

En el siguiente capítulo estudiaremos el problema CLIQUE y por medio de los teoremas desarrollados en este capítulo, se demostrará que dicho problema es NP-completo.

3

COMPLEJIDAD DE LA BÚSQUEDA CLANES

En este capítulo estudiamos 3 problemas fundamentales relacionados con la búsqueda de clanes:

- *Problema* Clique: Consiste en determinar si una gráfica tiene *un clan* con al menos k vértices.
- *Problema* MClique: Consiste en determinar si una gráfica tiene un *clan máximo* de k vértices.
- *Problema* LClique: Consiste en listar *todos* los clanes de una gráfica.

problemas sobre búsqueda clanes

En particular, nos interesa revisar en qué clase de complejidad se encuentra cada uno de los problemas antes mencionados. Este enfoque permite estudiar cuáles pueden ser las restricciones en tiempo que tienen los algoritmos que resuelven problemas de búsqueda de clanes.

3.1 EL PROBLEMA CLIQUE

El problema Clique es uno de los 21 problemas clásicos NP-completos de Karp[14]. Podemos enunciar el problema de decisión Clique de la siguiente forma:

> Instancia: Una gráfica $G = (V, E)$ y un entero positivo $k \leqslant |V|$.
> Pregunta: ¿Contiene G un clan de orden k o mayor?

problema Clique

Para demostrar que Clique es NP-completo, usaremos la estrategia que surgió a raíz de estudiar el lema 2.3.8. Esto es, se demostrará que Clique es NP y que toda instancia del problema NP-completo 3-SAT se puede transformar a una instancia del problema Clique.

Comenzamos con el siguiente lema que demuestra que 3-SAT se transforma a Clique.

Lema 3.1.1 (Karp [14]). *Existe una transformación polinomial de 3-SAT a* CLIQUE.

Demostración. Sea $B = \{x_1, x_2, \ldots, x_r\}$ un conjunto de variables booleanas y sea $C = \{C_1, C_2, \ldots, C_m\}$ una colección de cláusulas sobre B. Construiremos una gráfica $G = (V, E)$ y un entero positivo $K \leqslant |V|$ tal que G tiene un clan de tamaño K o mayor si y solo si se satisface la colección de cláusulas C.

Observe que C tiene m cláusulas y cada cláusula tiene exactamente 3 literales, por lo tanto C tiene en total $3m$ literales.

La reglas para construir la gráfica G son las siguientes:

1. Por cada cláusula $C_i = \{c_{i_1}, c_{i_2}, c_{i_3}\}$ de C, construya el conjunto de vértices $V_i = \{c_{i_1}, c_{i_2}, c_{i_3}\}$. Observe que los vértices en V_i se etiquetan usando los mismos literales de C_i.
2. Defina los vértices de G como $V = \bigcup_{i=1}^{k} V_i$. Por la regla 1, existe una correspondencia uno a uno entre los vértices en V y los literales en C.
3. Cada par de vértices v_i y v_j de G son adyacentes *excepto* en los siguientes casos:

   a) El par de vértices v_i y v_j son parte del mismo conjunto $V_l \subseteq V$. Por lo tanto cada conjunto V_l es independiente.

   b) El par de vértices v_i y v_j tienen etiquetas de la forma $v_i = x_i$ y $v_j = \overline{v_i} = \overline{x_i}$. Es decir, ningún vértice con etiqueta x es adyacente a un vértice con etiqueta $\overline{x}$.

Defina $K = m$. Mostraremos que la colección de cláusulas C se satisface si y solo si la gráfica G tiene un clan de orden igual o mayor que K.

Suponga que C se satisface, entonces al menos un literal de cada cláusula es verdadero. Defina el conjunto de literales S seleccionando de cada cláusula el literal que es verdadero, en caso de que exista más de un literal verdadero en la cláusula, tome uno de los literales de manera arbitraria. Observe que $|S| = K = m$.

Formamos el subconjunto $Q \subset V$ seleccionando todos los vértices que tienen como etiquetas a los literales en S. Por la manera en que se definió la gráfica G, se tiene que $|Q| = |S| = K$. Como cada literal de S se seleccionó de cláusulas diferentes, ningún *par* de vértices v_i y v_j en Q están ambos en el mismo conjunto $V_p \subset V$, por lo tanto, el *caso 3 inciso (a)* no se cumple.

Puesto que cada par de literales x_i y x_j en S es verdadero en las cláusulas de C, no es posible que $v_i = x_i$ y $v_j = \overline{v_i} = \overline{x_i}$ y por lo tanto tampoco se cumple el caso 3 *inciso (b)*. Concluimos que cada par de vértices en Q son adyacentes y en consecuencia Q tiene una subgráfica completa con K vértices y G tiene un clan de orden igual o mayor a K.

Por otra parte, suponga que G tiene un clan Q de K vértices. Por la forma en que se definió G, ningún par de vértices $v_i = x_i$ y $v_j = x_j$ en Q pueden estar ambos en el mismo conjunto $V_p \subset V$, por lo tanto, cada etiqueta x_i en el conjunto Q está en una cláusula diferente $C_p \in C$.

Si $x \in V_p \subset V$, asignamos el valor verdadero para el literal x en la cláusula $C_p \in C$. Como la arista $\{x, \overline{x}\}$ no existe en $E(G)$, no es posible que esta asignación haga a x y $\overline{x}$ verdaderas. Por lo tanto, cada cláusula $C_p \in C$ tiene un literal verdadero y por ende la colección de cláusulas C se satisface.

Finalmente, dado que una codificación para una instancia de 3-SAT puede hacerse en tiempo proporcional a $n = |C| + |B| = r + m$, la construcción de los vértices de la gráfica G puede hacerse en tiempo $\mathcal{O}(n)$. La construcción de las aristas de G siguiendo las reglas 2 y 3 puede hacerse en tiempo $\mathcal{O}(n^2)$. Concluimos que esta transformación tiene complejidad polinomial. □

A manera de ejemplo, considere la instancia de 3-SAT que tiene variables Booleanas $B = \{x_1, x_2, x_3\}$ y colección de cláusulas

$$C = \{\{x_1, \overline{x_2}, \overline{x_3}\}, \{\overline{x_1}, x_2, x_3\}, \{\overline{x_1}, x_2, \overline{x_3}\}\}$$

La gráfica que resulta de aplicar la transformación descrita en el lema 3.1.1 a la colección de cláusulas C se muestra en la figura 7.

Gracias al lema 3.1.1, estamos listos para demostrar el teorema más importante de esta sección.

Teorema 3.1.1 (Karp [14]). *El problema* CLIQUE *es NP-completo.*

Demostración. Sea $G = (V, E)$ una gráfica con conjuntos de vértices V y aristas E. Sea $K \leqslant |V| = n$ un entero positivo. Primero demostraremos que el problema CLIQUE está en NP, para ello construimos una máquina no determinista N que recibe una cadena de entrada de la forma <*V,E*>*K* y que opera siguiendo las siguientes reglas:

1. La máquina selecciona de manera no determinista un posible conjunto solución $Q = \{v_1, \ldots, v_K\} \subseteq V$.

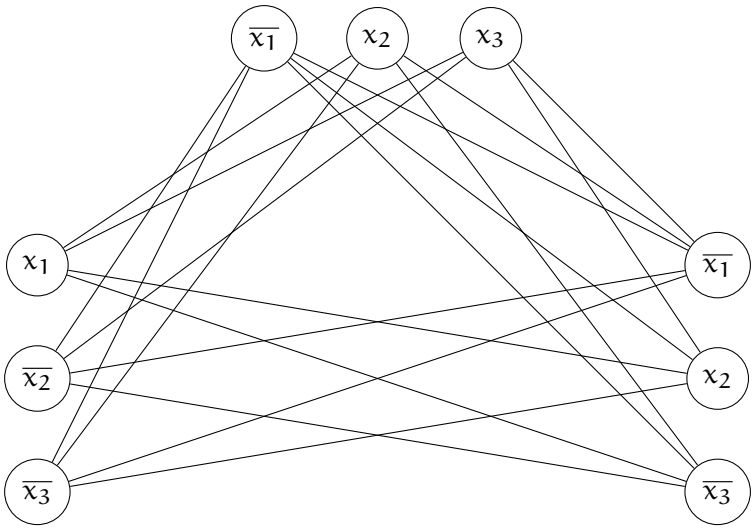


Figura 7: La gráfica que resulta de la transformación de la instancia 3-SAT con cláusulas $C = \{\{x_1, \overline{x_2}, \overline{x_3}\}, \{\overline{x_1}, x_2, x_3\}, \{\overline{x_1}, x_2, \overline{x_3}\}\}$.

2. Por cada par de vértices v_i y v_j en Q, la máquina comprueba que la arista $v_i v_j$ está en E.
3. Si todos los pares de vértices del paso 2 están en el conjunto E, la máquina N termina en estado de aceptación, en caso contrario la máquina termina en estado de rechazo.

La máquina N por ser no determinista puede ejecutar la regla 1 en un tiempo $\mathcal{O}(K)$, las reglas 2 y 3 se pueden hacer en un tiempo $\mathcal{O}(K^2)$. Como $K \leqslant n$ la máquina N tiene complejidad temporal polinomial y en consecuencia el problema *Clique* es NP.

Por el teorema 2.3.11, 3-SAT es NP-completo y por el lema 3.1.1, existe una transformación polinomial de 3-SAT al problema CLIQUE. Finalmente, por el teorema 2.3.8, concluimos que CLIQUE es NP-completo.

□

Corolario 3.1.1. *Sea* Π *el problema* CLIQUE *y sea* Π^c *su problema complementario. Si* $L[\Pi^c]$ *está en la clase* NP*, entonces* NP=*co*-NP.

Demostración. Por el lema 2.3.6, $L[\Pi]^c$ está en la clase NP. Por el teorema 3.1.1, $L[\Pi]$ está en la clase NP-completo y por el lema 2.3.5, se tiene que NP=co-NP. □

El teorema 3.1.1 tiene consecuencias practicas sumamente importantes; puesto que CLIQUE es NP-completo, no se conoce a la fecha un procedimiento

de decisión para el problema CLIQUE con complejidad temporal polinomial. Más aún, por el teorema 2.3.6 el mejor tiempo que se conoce para dicho procedimiento de decisión es de orden exponencial.

Podemos usar el teorema 3.1.1 para probar que otros problemas similares son NP-completos. Uno de estos problemas es el problema del conjunto independiente maximal (INDEPENDENT SET), que podemos enunciar de la siguiente manera:

problema INDEPENDENT SET

> Instancia: Una gráfica $G = (V, E)$ y un entero positivo $k \leqslant |V|$.
>
> Pregunta: ¿Contiene G un conjunto independiente maximal de orden k o mayor?

Para demostrar que INDEPENDENT SET es NP-completo, primero demostraremos el siguiente teorema, que aunque es muy simple, muestra una relación importante entre los clanes de una gráfica y los conjuntos independientes maximales de su complemento.

Teorema 3.1.2. *Q es una subgráfica completa de G si y solo si Q es un conjunto independiente de* $\overline{G}$.

Demostración. ($\Rightarrow$) Sea Q una subgráfica completa de la gráfica G. Por definición, para cada par de vértices v y u en Q, existe una arista $e = \{v, u\} \in E(G)$. Por lo tanto, los vértices v y u no son adyacentes en $\overline{G}$ y en consecuencia Q es un conjunto independiente de $\overline{G}$.

($\Leftarrow$) Si Q es un conjunto independiente de $\overline{G}$, cada par de vértices u y v en Q son adyacentes en G, por lo que Q es una subgráfica completa de G. □

Corolario 3.1.2. *Q es un clan de la gráfica G si y solo si Q es un conjunto independiente maximal de* $\overline{G}$

Demostración. Inmediato de la demostración del teorema 3.1.2 □

Corolario 3.1.3. *El número de clanes de una gráfica G es igual a el número de conjuntos independientes maximales de su complemento* $\overline{G}$*, es decir:*

$$|K(G)| = |MIS(\overline{G})|$$

Demostración. Inmediato de la demostración del teorema 3.1.2 □

Por medio del teorema 3.1.2, podemos visualizar a un problema de clanes como un problema equivalente de búsqueda de conjuntos independientes. Esta dualidad será de mucha importancia para el resto de los capítulos de este trabajo.

Teorema 3.1.3. INDEPENDENT SET *es NP-completo*

Demostración. Primero demostraremos que INDEPENDENT SET es NP. Para esto basta construir un algoritmo S *no determinista* para INDEPENDENT SET con complejidad polinomial. Dicho algoritmo se puede construir de manera similar al algoritmo del teorema 3.1.1, con la diferencia de que en S se revisa que todos los vértices del conjunto solución no sean adyacentes. De esto se concluye que INDEPENDENT SET es NP.

Sea <*V,E*>*K* la cadena que representa una instancia de CLIQUE con $G = (V, E)$ y entero K. Para transformar CLIQUE a INDEPENDENT SET construimos el complemento $\overline{G}$ de la gráfica G. Esta transformación puede hacerse en tiempo $\mathcal{O}(|< V, E > K|^2)$ y por el corolario 3.1.2, se tiene que Q es un clan en G si y solo si Q es un conjunto independiente maximal en $\overline{G}$. Por el teorema 2.3.8, concluimos que INDEPENDENT SET es NP-completo. □

Las implicaciones que tiene el teorema 3.1.3 en los procedimientos de decisión que buscan conjuntos independientes, son las mismas que se discutieron para el teorema 3.1.1.

3.2 EL PROBLEMA MCLIQUE

Un problema de búsqueda de clanes que está muy relacionado con el problema CLIQUE, es el problema MCLIQUE el cual se enuncia a continuación:

problema MCLIQUE

Instancia: Una gráfica $G = (V, E)$ y un entero positivo $k \leqslant |V|$.
Pregunta: ¿Contiene G un clan *máximo* de orden k?

Observe que a diferencia del problema CLIQUE, el problema MCLIQUE no busca cualquier clan con orden k o mayor, sino que busca el *clan máximo* (el clan de mayor número de vértices) cuyo orden es exactamente k. Esta pequeña diferencia hace que el problema MCLIQUE esté en la clase de complejidad NP-difícil.

Teorema 3.2.1 (Karp [14]). MCLIQUE *es NP-difícil*

Demostración. Se mostrará que existe una transformación polinomial de 3-SAT a MCLIQUE.

Sea I una instancia de 3-SAT con la colección de cláusulas $C = \{C_1, \ldots, C_k\}$ y sea f la transformación polinomial descrita en el lema 3.1.1. Recuerde que f transforma a la instancia I en una gráfica G de acuerdo a las siguientes reglas:

1. Por cada literal x de cada cláusula $C_i \in C$, la gráfica G tiene un vértice (C_i, x).
2. Los vértices (C_i, x) y (C_j, y) en G son adyacentes si y solo si $C_i \neq C_j$ y $x \neq \overline{y}$.

Por el lema 3.1.1, I se satisface si y solo si G contiene un clan Q tal que $|Q| \geqslant k = \frac{|V|}{3}$.

Suponga que G tiene un clan Q tal que $|Q| > k$. Por el principio del palomar existe una cláusula $C_i \in C$ con literales x y y tales que los vértices (C_i, x) y (C_i, y) son adyacentes en G, pero esto contradice la regla 2. Por lo tanto $|Q| \leqslant k$ y en consecuencia Q es clan máximo de G. Por lo tanto, I se satisface sí y solo sí G tiene un clan máximo de orden k, en consecuencia, f es una trasformación polinomial de 3-SAT a MCLIQUE.

Por el teorema 2.3.11, todo problema de decisión Π en NP se transforma a 3-SAT. Como 3-SAT se transforma a MCLIQUE, por el lema 2.3.4, existe una transformación polinomial de Π a MCLIQUE y por el corolario 2.3.2, MCLIQUE es NP-difícil.

□

El significado intuitivo del teorema 3.2.1 es que el problema MCLIQUE es tanto o más complicado que el problema CLIQUE. No se sabe si el problema MCLIQUE está en NP, sin embargo se puede demostrar que de ser así entonces NP=co-NP [19, Teorema 2.2], omitimos la demostración de este resultado ya que es demasiado extensa y requiere terminología que nos alejaría de los objetivos de este trabajo.

3.3 EL PROBLEMA LCLIQUE

problema LCLIQUE

El problema LCLIQUE consiste en listar todos los clanes de una gráfica. Hasta este momento hemos estudiado problemas de decisión relacionados con la búsqueda de clanes, sin embargo, el problema LCLIQUE *no es un problema de decisión*. Por esta razón no podemos usar las poderosas herramientas que se desarrollaron para la teoría de la NP-Completez, sin embargo, sí podemos mostrar que el problema LCLIQUE, en el peor de los casos, requiere un tiempo exponencial. Con este fin enunciamos los siguientes lemas y observaciones.

Observación 3.3.1. *Sea* I *un conjunto independiente maximal de la gráfica* G *y sea* v *un vértice de* $V(G)$ *entonces:*

$$I \cap N[v] \neq \emptyset.$$

Lema 3.3.1. *Sea* I *un conjunto independiente maximal de la gráfica* $G = (V, E)$ *y sea* w *un vértice del conjunto* I*. Entonces* $I \setminus N[w]$ *es un conjunto independiente maximal de* $G[V \setminus N[w]]$*.*

Demostración. Si G es una gráfica sin aristas, la demostración es trivial. Considere una gráfica G con al menos una arista.

La prueba se hará por contradicción. Suponga que la afirmación del lema no se cumple para algún vértice $w \in I$.

Sea $V' = V \setminus N[w]$, existe un vértice $x \in V'$ tal que $x \notin I$ y $I' = \{x\} \cup (I \setminus N[w])$ es un conjunto independiente (no necesariamente maximal) de la gráfica $G[V']$.

Por la observación 3.3.1, existe un vértice $y \in V$ tal que $y \in I \cap N[x]$. Como y es un vértice del conjunto I, no es posible que $y = x$, ya que esto implicaría que $x \in I$, una contradicción. Por lo tanto, $y \neq x$.

Observe que $y \neq w$, de lo contrario x no podría pertenecer al conjunto V'. Por otra parte, si $y \in N(w)$ entonces los vértices y y w están en el conjunto I y son adyacentes, otra contradicción. Por lo tanto, $y \notin N[w]$ y en consecuencia $y \in V'$.

Como los vértices x y y están en el conjunto I y la arista $\{x, y\}$ es parte de la gráfica $G[V']$, concluimos que I' no es un conjunto independiente, una contradicción.

□

Lema 3.3.2. *Sean* $f(x) = 3^{x/3}$ *y* $g(x) = x$*, funciones definidas sobre los reales, entonces:*

$$f(x) \geqslant g(x) \qquad \forall x \in [3, \infty).$$

Demostración. Sea h la función sobre los reales definida como

$$h(x) = f(x) - g(x).$$

Sea b un número real en el intervalo $(3, \infty)$. Por el teorema del valor medio, existe una constante $c \in (3, b)$ tal que

$$h'(c) = \frac{h(b) - h(3)}{b - 3} = \frac{3^{b/3} - b}{b - 3}.$$

Como $c \geqslant 3$, se tiene que

$$h'(c) = \frac{3^{c/3}}{3} - 1 \geqslant 0$$

Luego,

$$\frac{3^{b/3} - b}{b - 3} \geqslant 0$$

Por lo tanto

$$3^{b/3} \geqslant b$$

Como b es un número arbitrario en $(3, \infty)$ y como $g(3) = f(3)$ se concluye que

$$3^{x/3} \geqslant x \qquad \forall x \geqslant 3$$

□

El siguiente teorema fue demostrado de manera independiente primero por Miller y Muller en 1960 [24] y posteriormente por Moon y Moser en 1965 [25]. Sin embargo, una prueba más simple fue publicada por David Wood en 2011 [33] y es la que se describe a continuación.

Teorema 3.3.1 (Miller y Muller [24, Teorema 4], Moon y Moser[Teorema 1] [25], Wood [24])**.** *Sea G una gráfica con n vértices y sea MIS(G) la colección de todos los conjuntos independientes maximales de la gráfica G. Entonces* *Teorema de Wood*

$$|\mathrm{MIS}(G)| \leqslant g(n),$$

donde la función $g(n)$ *esta definida como:*

$$g(n) = \begin{cases} 3^{n/3}, & \text{si } n \equiv 0 \pmod 3 \\ 4 \cdot 3^{(n-4)/3}, & \text{si } n \equiv 1 \pmod 3 \\ 2 \cdot 3^{(n-2)/3}, & \text{si } n \equiv 2 \pmod 3 \end{cases}$$

Demostración. La prueba se hará por inducción sobre n.

Si $n = 1$, entonces $g(1) = \frac{4}{3}$, el cual es un valor mayor que el número de conjuntos independientes maximales que se pueden formar con 1 vértice.

Si $n = 2$, entonces $g(2) = 2$, el cual es un valor igual al mayor número de conjuntos independientes maximales que se pueden formar con 2 vértices.

Suponga que la afirmación es válida para $n \leqslant N$. Mostraremos que también es válida para $n = N + 1$.

Sea δ el grado mínimo de G y sea v un vértice de G de grado δ. Sea $N[v]$ la vecindad cerrada de v. Por la observación 3.3.1, existe un vértice $w \in I \cap N[v]$ y por el lema 3.3.1, se tiene que $I \setminus N[w] \in MIS(G \setminus N[w])$. Por lo tanto, para algún vértice $w \in N[v]$, cada conjunto independiente maximal $I \in MIS(G)$ tiene asociado al menos un conjunto independiente maximal $I' \in MIS(G \setminus N[w])$. Por lo tanto

$$|MIS(G)| \leqslant \sum_{w \in N_G[v]} |MIS(G \setminus N_G[w])|.$$

Puesto que $d_G(w) \geqslant \delta$ se tiene que $|G \setminus N_G[w]| \leqslant n - \delta - 1$ y por la hipótesis de inducción se tiene que

$$|MIS(G)| \leqslant \sum_{w \in N_G[v]} g(|G \setminus N_G[w]|).$$

Como $g(n)$ es una función creciente, en la desigualdad anterior se obtiene que

$$|MIS(G)| \leqslant (\delta + 1) \cdot g(n - \delta - 1). \tag{2}$$

Observe que

$$4 \cdot 3^{(n-4)/3} \leqslant g(n) \leqslant 3^{n/3}.$$

A fin de obtener una cota superior para (2), considere los siguientes casos:

- Si $\delta \geqslant 3$, entonces

$$|MIS(G)| \leqslant (\delta + 1) \cdot 3^{(n-\delta-1)/3} = 3 \left(\frac{\delta + 1}{3^{\delta/3}} \right) 3^{(n-4)/3}.$$

  Por el lema 3.3.2 se tiene

$$\frac{\delta + 1}{3^{\delta/3}} = \frac{\delta}{3^{\delta/3}} + \frac{1}{3^{\delta/3}} \leqslant 4/3$$

  Por lo tanto en (2) queda

$$|MIS(G)| \leqslant 4 \cdot 3^{(n-4)/3} \leqslant g(n)$$

- Si $\delta = 2$, entonces

$$|MIS(G)| \leqslant 3 \cdot g(n-3) = g(n)$$

- Si $\delta = 1$ y $n \equiv 1 \pmod 3$, como $n-2 \equiv 2 \pmod 3$ se tiene que

$$|MISG(G)| \leqslant 2 \cdot g(n-2) \leqslant 2 \cdot 2 \cdot 3^{(n-2-2)/3} = 4 \cdot 3^{(n-4)/3} = g(n)$$

- Si $\delta = 1$ y $n \equiv 0 \pmod 3$, como $n-2 \equiv 1 \pmod 3$ se tiene que

$$|MIS(G)| \leqslant 2 \cdot g(n-2) \leqslant 2 \cdot 4 \cdot 3^{(n-2-4)/3} \leqslant 3^{n/3} = g(n)$$

- Si $\delta = 1$ y $n \equiv 2 \pmod 3$, como $n-2 \equiv 0 \pmod 3$ se tiene que

$$|MIS(G)| \leqslant 2 \cdot g(n-2) \leqslant 2 \cdot 3^{(n-2)/3} = g(n)$$

Esto prueba que $|MIS(G)| \leqslant g(n)$.

□

Una pregunta que surge inmediatamente después de revisar el teorema 3.3.1, es si la cota superior que se propone para el número de conjuntos independientes maximales de una gráfica, efectivamente se alcanza. El complemento de la siguiente gráfica propuesta por Moon y Moser [25] cumple esta condición.

Definición 3.3.1 (Moon-Moser [25]). *La gráfica de Moon-Moser* G *de* n *vértices, es el* complemento *de la gráfica* H*, la cual se define de acuerdo a los siguientes casos:*

- *Si* $n \equiv 0 \pmod 3$

$$H = \bigcup_{i=1}^{n/3} K_3$$

- *Si* $n \equiv 1 \pmod 3$

$$H = \left(\bigcup_{i=1}^{(n-4)/3} K_3 \right) \cup K_4$$

- *Si* $n \equiv 2 \pmod 3$

$$H = \left(\bigcup_{i=1}^{(n-2)/3} K_3 \right) \cup K_2$$

Para demostrar que el complemento de la gráfica de Moon-Moser alcanza la cota del teorema 3.3.1 necesitamos el siguiente lema.

Lema 3.3.3. *Sea* G *una gráfica arbitraria y sea* K_1 *la gráfica de 1 solo vértice, entonces:*

$$|K(G)| = |K(G + K_1)|.$$

Demostración. Construiremos una función biyectiva que permita relacionar clanes en $K(G)$ con clanes de $K(G + K_1)$. Para ello, observe que si Q es un clan de G, entonces $Q + K_1$ es un clan de $K(G + K_1)$.

Defina la función $f : K(G) \to K(G + K_1)$ como

$$f(Q) = Q + K_1 \qquad \forall Q \in K(G).$$

Sea x el único vértice de la gráfica K_1. Como el vértice x adyacente a cada uno de los vértices de G, todo clan de $G + K_1$ debe contener a x. Si Q' es un clan de $K(G + K_1)$, entonces $S = Q' \setminus \{x\}$ es una subgráfica completa de G. Si S no es clan un clan de G, existe un vértice $y \in V(G) \setminus S$ tal que los vértices $S \cup \{y\}$ forman una completa, pero esto implica que $S \cup \{x, y\} = Q' \cup \{x\}$ es un completa de $G + K_1$, una contradicción. Por lo tanto f es sobreyectiva.

Sean Q_1 y Q_2 clanes de G tales que $f(Q_1) = f(Q_2)$. Entonces $Q_1 + K_1 = Q_2 + K_1$ y por lo tanto $Q_1 = Q_2$. De esto concluimos que la función f es inyectiva y en consecuencia

$$|K(G)| = |K(G + K_1)|.$$

□

El último teorema de esta sección demuestra que el total de clanes de una gráfica de Moon-Moser se corresponden con los valores de la función del teorema 3.3.1.

Teorema 3.3.2 (Moon y Moser [25])**.** *Sea* $g(n)$ *la función que se definió en el teorema 3.3.1. Sea* M_n *la gráfica de Moon-Moser de* n *vértices. Entonces para* $n \geqslant 3$

$$|K(M_n)| = g(n).$$

Demostración. La prueba se hará por inducción sobre n. Para la base de la inducción consideramos los siguientes casos:

- Si $n = 3$, entonces $M_3 = \overline{K_3}$ y $|K(M_3)| = g(3) = 3$.
- Si $n = 4$, entonces $M_4 = \overline{K_4}$ y $|K(M_4)| = g(4) = 4$.
- Si $n = 5$, entonces $M_5 = (K_3 \cup K_2)^c$ y $|K(M_5)| = g(5) = 6$.

Suponga que el teorema es verdadero para $n \leqslant N$ vértices. Mostraremos que el teorema es válido para $n = N + 1$.

Sea $\overline{K_n}$ la gráfica discreta (sin aristas) de n vértices. Para $|K(M_n)|$ tenemos los siguientes casos:

- Si $n \equiv 0 \pmod 3$, observe que

$$M_n = \left(\bigcup_{i=1}^{n/3} K_3 \right)^c = \left(\bigcup_{i=1}^{(n-3)/3} K_3 \right)^c + \overline{K_3} = M_{n-3} + \overline{K_3}$$

  Si x es un vértice de $\overline{K_3}$, por el lema 3.3.3 se cumple que

$$|K(M_{n-3} + \{x\})| = |K(M_{n-3})|$$

  Como $n - 3 \equiv 0 (\mod 3)$, por la hipótesis de inducción $|K(M_{n-3})| = 3^{(n-3)/3}$, por lo tanto:

$$|K(M_n)| = 3|K(M_{n-3})| = g(n) = 3^{n/3}$$

- Si $n \equiv 1 \pmod 3$, observe que

$$M_n = \left(\left(\bigcup_{i=1}^{(n-4)/3} K_3 \right) \cup K_4 \right)^c = \left(\bigcup_{i=1}^{(n-4)/3} K_3 \right)^c + \overline{K_4} = M_{n-4} + \overline{K_4}$$

  Como $n - 4 \equiv 0 (\mod 3)$, por la hipótesis de inducción $|K(M_{n-4})| = 3^{(n-4)/3}$ y por el lema 3.3.3 se tiene que:

$$|K(M_n)| = 4|K(M_{n-4})| = g(n) = 4 \cdot 3^{(n-4)/3}$$

- Si $n \equiv 2 \pmod 3$, observe que

$$M_n = \left(\left(\bigcup_{i=1}^{(n-2)/3} K_3 \right) \cup K_2 \right)^c = \left(\bigcup_{i=1}^{(n-2)/3} K_3 \right)^c + \overline{K_2} = M_{n-2} + \overline{K_4}$$

  Como $n - 2 \equiv 0 (\mod 3)$, por la hipótesis de inducción $|K(M_{n-2})| = 3^{(n-2)/3}$ y por el lema 3.3.3 se tiene que:

$$|K(M_n)| = 2|K(M_{n-2})| = g(n) = 2 \cdot 3^{(n-2)/3}$$

□

Corolario 3.3.1. *Sea* $\overline{M_n}$ *el complemento de la gráfica de Moon-Moser de* n *vértices, entonces:*

$$|\mathrm{MIS}(\overline{M_n})| = g(n)$$

Demostración. Aplicar el lemma 3.3.3 al teorema 3.3.2 □

Corolario 3.3.2. *Sea* $g(n)$ *la función que se definió en el teorema 3.3.1. Un algoritmo que imprime al menos un símbolo por cada clan de una gráfica* G *con* n *vértices, tiene complejidad temporal* $\Omega(g(n))$.

Demostración. Inmediato del teorema 2.3.2 y del teorema 3.3.2. □

Del teorema 3.3.2, se tiene que la mejor cota posible para el total de clanes (o conjuntos independientes) de una gráfica G es la función $g(n) \leqslant \mathcal{O}(3^{n/3})$ del teorema 3.3.1. Por el corolario 3.3.2, un algoritmo que resuelve el problema LCLIQUE puede llegar a usar *al menos* $\mathcal{O}(3^{n/3})$ movimientos. En el siguiente capítulo estudiaremos un algoritmo sumamente eficiente que permite encontrar todos los clanes de una gráfica en tiempo $\mathcal{O}(3^{n/3})$.

4

EL ALGORITMO DE BRON-KERBOSCH

El algoritmo de Bron-Kerbosch es uno de los algoritmos más utilizados para listar todos los clanes de una gráfica (problema LClique). Este algoritmo fue presentado por primera vez en 1977 en [4]. Es curioso que en la publicación original no se incluye ninguna demostración de que el algoritmo efectivamente encuentra todos los clanes de una gráfica, tampoco se incluye un análisis de la complejidad temporal.

El determinar cual es la complejidad temporal del algoritmo de Bron-Kerbosch fue una pregunta abierta por más de 30 años. Sin embargo, en el año 2006 los japoneses Tomita, Tanaka y Takahashi [29] publicaron un algoritmo que resuelve el problema LClique usando una estrategia muy similar al del algoritmo de Bron-Kerbosch. Dicho algoritmo para un gráfica de n vértices tiene complejidad temporal $\mathcal{O}(3^{n/3})$. Dos años después, Cazals y Karande [5] demostraron que el algoritmo propuesto por Tomita et al. es esencialmente el algoritmo de Bron-Kerbosch, con esto finalmente quedaba resuelta la cuestión de cual es la complejidad temporal del algoritmo de Bron-Kerbosch.

Existe mucha evidencia empírica que muestra que para gráficas de n vértices y m aristas, el algoritmo de Bron-Kerbosch tiene un tiempo de ejecución menor que cualquier otro algoritmo que resuelve el problema LClique [3, 9, 29]. A la fecha no existe un estudio de la complejidad temporal de este algoritmo en función de estos parámetros (o en función otros parámetros relevantes). Esta cuestión es un problema muy importante, ya que nos limita a usar el número de vértices como un medio parar comparar el tiempo de ejecución entre algoritmos, mientras que otras propiedades importantes como son el número de aristas y el número de independencia simplemente se ignoran. Es por esta razón que uno de los problemas que se investigaron como parte de este trabajo, es la complejidad temporal del algoritmo de Bron-Kerbosch en función del número de vértices y aristas de una gráfica. Al final de este capítulo se muestran algunos resultados parciales y conjeturas relacionadas con este problema.

4.1 VERSIÓN 1 DEL ALGORITMO DE BRON-KERBOSCH

Para estudiar el algoritmo de Bron-Kerbosch, usaremos el enfoque utilizado en [16], esto es, partiremos de una versión muy simple (versión 1) e iremos agregando mejoras hasta llegar a la versión 3, que corresponde a la versión original que se publicó en [4].

El pseudocódigo de la primera versión del algoritmo de Bron-Kerbosch se describe a continuación:

Algoritmo 1: Versión 1 del algoritmo de Bron-Kerbosch.

llamada inicial: BK1($\emptyset$, $V(G)$, $\emptyset$)

```
1:  procedure BK1(C, P, N)
2:      Sea P = {v_1, ··· , v_k}
3:      if P = ∅ y N = ∅ then
4:          Reportar C como clan
5:      else
6:          for i ← 1 to k do
7:              P = P \ {v_i}
8:              C_new = C ∪ {v_i}
9:              P_new = P ∩ N(v_i)
10:             N_new = N ∩ N(v_i)
11:             BK1(C_new, P_new, N_new)
12:             N = N ∪ {v_i}
13:         end for
14:     end if
15: end procedure
```

El algoritmo 1 usa principalmente 3 conjuntos: C, P y N. La función que tienen estos conjuntos en el algoritmo se describe a continuación:

conjunto C

- C *(clan)*: Se usa para almacenar los vértices de una *subgráfica completa de la gráfica* G. El algoritmo mueve un vértice v del conjunto P al conjunto C, solo si v es adyacente a todos los vértices de C. Con cada llamada recursiva del algoritmo, el conjunto C aumenta su cardinalidad en una unidad. Si en alguna llamada recursiva, los conjuntos S y P son vacíos, el algoritmo reporta que los vértices del conjunto Q forman un *clan* de la gráfica G.

conjunto P

- P *(posibles)*: Se usa para almacenar a todos los *posibles* vértices que son adyacentes a *todos* los vértices del conjunto C.

- N *(not)*: Se usa para guardar todos los vértices que han sido *utilizados* para extender a C. Los vértices en N *no deben* ser usados en posteriores iteraciones del bucle en la línea 6. *conjunto N*

Al iniciar el algoritmo 1, los conjuntos C y N son vacíos. El conjunto P se inicializa con todos los vértices de la gráfica G.

Observe que cada vértice v_i de P se selecciona de manera arbitraria en el bucle del algoritmo (línea 6). Tan pronto se selecciona el vértice v_i, este se elimina del conjunto P (línea 7) y se añade al conjunto C para formar un nuevo conjunto que denotaremos por C_{new} (línea 8).

Los conjuntos P_{new} y N_{new} se forman al realizar la intersección de la vecindad abierta de v_i con P y N respectivamente (líneas 9 y 10). Los conjuntos P_{new} y N_{new} contienen los vértices de P y N que son adyacentes al vértice v_i, por lo que en la siguiente llamada recursiva (línea 11) los conjuntos P y N contienen únicamente vértices que son adyacentes a v_i.

Teorema 4.1.1 (Koch [16, Teorema 3.3])**.** *Sea* G *una gráfica simple. Los conjuntos* C*,* P *y* N *que se usan en el procedimiento* BK1 *(algoritmo 1), tienen las siguientes propiedades:*

(i) *Cada vértice* $v \in P$ *es adyacente a todos los vértices de* C*.*

(ii) *Cada vértice* $v \in N$ *es adyacente a todos los vértices de* C*.*

(iii) *Todos los vértices en C son adyacentes por pares, es decir,* C *es una subgráfica completa de* G*.*

(iv) *Sea* v *un vértice de* G *que no está en el conjunto* C*. Si* v *es adyacente a todos los vértices en* C*, entonces* v *está en* P *o está en* N*.*

(v) *Si* $v \in N$*, entonces todos los clanes de* G *que contienen a* $C \cup \{v\}$*, ya han sido enumerados.*

(vi) *Al terminar la invocación del procedimiento* BK1*, todos los clanes de la gráfica* G *son enumerados exactamente una sola vez.*

Demostración.
Para los siguientes incisos, denotaremos por C_i, P_i y N_i a los valores que tienen los conjuntos C, P y N, en la línea 2, de la i-ésima llamada recursiva del algoritmo 1, respectivamente. Los conjuntos C_0, P_0 y N_0 denotarán los valores que tienen los conjuntos C, P y N en la invocación inicial del procedimiento BK1, respectivamente. Observe que al iniciar el procedimiento BK1, se tiene que $C_0 = \emptyset$, además, siempre se cumple que $|C_i| = i$.

También usaremos la convención de que si el conjunto C tiene vértices

$$C = \{v_1, v_2 \ldots, v_n\},$$

entonces v_1 es el primer vértice que se agregó a C, v_2 es el segundo vértice que se agregó a C y así sucesivamente.

(i), (ii) y (iii) La prueba se hará por inducción sobre la cardinalidad del conjunto C. Para $|C| = 1$, sea $C = \{v_1\}$. Por las líneas 8, 9 y 10 del algoritmo 1, se tiene que

$$\begin{aligned} C &= C_1 = C_0 \cup \{v_1\} = \emptyset \cup \{v_1\} = \{v_1\}, \\ P &= P_1 = P_0 \cap N(v_1), \\ N &= N_1 = N_0 \cap N(v_1). \end{aligned}$$

Por lo tanto, todos los vértices de los conjuntos P y N son adyacentes a v_1. El conjunto C_1 es una subgráfica completa de 1 vértice.

Suponga que la afirmación de los incisos *(i)*, *(ii)* y *(iii)* es cierta para n. Sea $C = \{v_1, v_2, \ldots, v_n, v_{n+1}\}$, por las líneas 8, 9 y 10 del algoritmo 1, se tiene que

$$\begin{aligned} C &= C_{n+1} = C_n \cup \{v_{n+1}\}, \\ P &= P_{n+1} = P_n \cap N(v_{n+1}), \\ N &= N_{n+1} = N_n \cap N(v_{n+1}). \end{aligned}$$

Como $|C_n| = n$, por la hipótesis de inducción C_n es una subgráfica completa, además, todos los vértices de P_n y N_n son adyacentes a C_n. Como los conjuntos P_{n+1} y N_{n+1} resultan de la intersección con la vecindad abierta de v_{n+1}, todos los vértices de los conjuntos P y N son adyacentes a todos los vértices de $C = C_n \cup \{v_{n+1}\}$.

Por la línea 6 del algoritmo 1, el vértice v_{n+1} se seleccionó del conjunto P_n. Por lo tanto, el vértice v_{n+1} es adyacente a todos los vértices del conjunto C_n y en consecuencia, C es una subgráfica completa.

(iv) Sea $C_n = \{v_1, \ldots, v_n\}$, los vértices del conjunto C en la recursión actual. Sean P_n y N_n los conjuntos P y N en la recursión actual, respectivamente. Sea v un vértice de G que no está en el conjunto C_n y que es adyacente a todo vértice de C_n. Por las líneas 7 y 12 del algoritmo 1, siempre se cumple que $V(G) = P_0 \cup N_0$. Por lo tanto, se tienen 2 casos:

- Caso 1: $v \in P_0$.

  Por la línea 9 del algoritmo 1, se tiene que $P_1 = P_0 \cap N(v_1)$. Como $N(v_1) \supseteq \{v_1\}$, el vértice v está en P_1. Un razonamiento similar nos lleva a concluir que el vértice v está en los conjuntos: $P_2, \ldots, P_n$.

- Caso 2: $v \in N_0$.

  Por la línea 10 del algoritmo 1, se tiene que $N_1 = N_0 \cap N(v_1)$. Como $N(v_1) \supseteq \{v_1\}$, el vértice v está en N_1. Un razonamiento similar nos lleva a concluir que el vértice v está en los conjuntos: $N_2, \ldots, N_n$.

De los casos anteriores, concluimos que el vértice v está en el conjunto P o en el conjunto N.

(v) Sea C_n, P_n y N_n los conjuntos C, P y N de la recursión actual, respectivamente. Sea v un vértice que está en el conjunto P_n.

Si $Q = C_n \cup \{v\}$ es un clan de la gráfica G, entonces por los incisos (i) y (ii), la llamada recursiva que se realizó después de seleccionar el vértice v para extender al conjunto C_n, tuvo conjuntos $P_{n+1} = \emptyset$ y $N_{n+1} = \emptyset$. Por la línea 3 del algoritmo 1, dicha llamada recursiva reportó al clan Q.

Si $Q \supsetneq C_n \cup \{v\}$ es un clan de la gráfica G, entonces se tiene que $R = Q \setminus (C_n \cup \{v\}) \neq \emptyset$. Sea C_{n+1} el conjunto C del siguiente nivel de recursión que resulta de elegir al vértice v, para extender a C_n. Por lo tanto, $C_{n+1} = C_n \cup \{v\}$ y se tienen 2 casos:

- Caso 1: $R \cap N_{n+1} = \emptyset$.

  Por el inciso *(iv)*, $P_{n+1} \supseteq R$. Por lo tanto, los vértices en el conjunto R son elegidos para extender a C_{n+1} en las sucesivas llamadas recursivas, hasta llegar a una llamada recursiva en la cual $P = \emptyset$ y $N = \emptyset$. Por la línea 3 del algoritmo 1, dicha llamada recursiva reportó al clan Q.
- Caso 2: $R \cap N_{n+1} \neq \emptyset$.

  Por el inciso *(ii)*, cada vértices en el conjunto $R' = R \cap N_{n+1}$ son adyacentes a todos los vértices en C_{n+1}. Por lo tanto, existe un nivel de recursión $i \leqslant n+1$, en el cual se agregó por primera vez un vértice r del conjunto R' en el conjunto N_i.

  Para el nivel de recursión i, se tiene que $C_{n+1} \supseteq C_i$ y $Q \supseteq C_i$. Como en el nivel *i*, ninguno de los vértices de $Q \setminus (C_i \cup \{r\})$ está en el conjunto N_i, por el inciso *(iv)* se tiene que $P_i \supseteq Q \setminus (C_i \cup \{r\})$. Por lo tanto, la elección del vértice r para extender a C_i reportó al clan

$$C_i \cup \{r\} \cup (Q \setminus (C_i \cup \{r\})) = Q.$$

(vi) Al iniciar el algoritmo 1, se tiene que $P_0 = V(G)$. Por la línea 6 del algoritmo 1, cada vértice en el conjunto P_0 se selecciona para extender a C_0 y por el inciso *(v)*, todos los clanes que contienen a cada vértice del conjunto P_0 son reportados. Por lo tanto todos los clanes de la gráfica G son reportados al terminar el algoritmo.

Para mostrar que los clanes de la gráfica son reportados 1 sola vez, sea v un vértice en N_0. Por el inciso (v), cualquier clan Q que contiene al vértice v ya ha sido reportado. Puesto que el vértice v ya no se puede volver a usar para extender a C_0, el algoritmo solo puede reportar al clan Q una vez.

Sea $S \subsetneq Q$ un subconjunto de un clan Q. Para mostrar que el algoritmo 1 no puede reportar a S como un clan de la gráfica G, suponga que existe un nivel de recursión i tal que $C_i = S$. Tenemos 2 casos:

- Caso 1: $(Q \setminus S) \cap N_i = \emptyset$.

  Por el inciso *(iv)*, se tiene que $P_i = Q \setminus S$. Por lo tanto, en la recursión i, no se cumple la condición de la línea 3 del algoritmo 1 y en consecuencia S no se puede reportar como clan.

- Caso 2: $(Q \setminus S) \cap N_i \neq \emptyset$.

  Entonces, en la recursión i, no se cumple la condición de la línea 3 del algoritmo 1 y en consecuencia S no se puede reportar como clan.

□

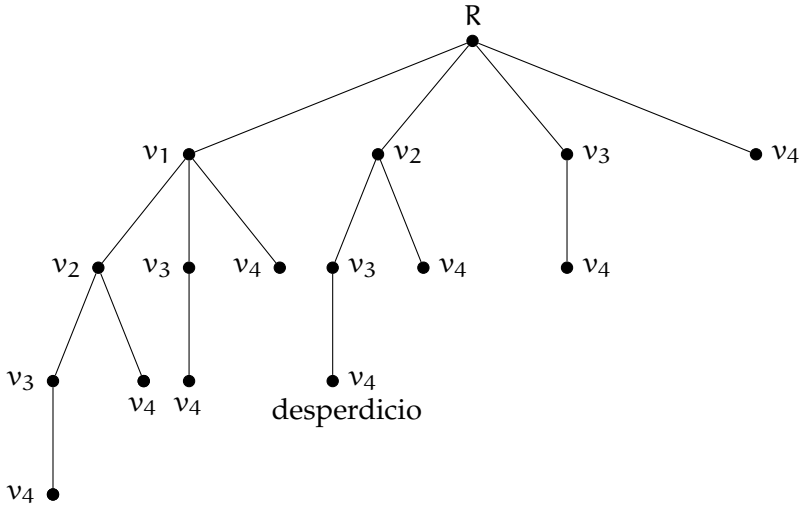
árbol de búsqueda

Es importante notar el algoritmo 1 requiere seleccionar al menos n vértices para reportar un clan Q, con $|Q| = n$. Sin embargo, el algoritmo también desperdicia tiempo seleccionando más de $\frac{n(n+1)}{2}$ vértices extras. Por ejemplo, la figura 8 muestra el *árbol de recursión* o *árbol de búsqueda* del algoritmo 1, cuando $G = K_4 = \{v_1, v_2, v_3, v_4\}$. Cada nodo del árbol en la figura 8, representa un vértice que fue seleccionado para extender el conjunto C. El nodo R representa la ejecución inicial del procedimiento BK1. Observe que el algoritmo solo requiere seleccionar 4 vértices (*trabajo*), para reportar que G es un K_4. Sin embargo, el algoritmo termina seleccionando 10 vértices más (*desperdicio*) que no reportan ningún clan.

vértices trabajo
vértices desperdicio

Los nodos del árbol de búsqueda que se requieren para reportar los clanes de la gráfica (incluyendo el nodo raíz), los denominaremos como *vértices trabajo*. El resto de los vértices los denominaremos *vértices desperdicio*.

Figura 8: Árbol de búsqueda del algoritmo 1 cuando $G = K_4 = \{v_1, v_2, v_3, v_4\}$.



En la siguiente sección, examinaremos una versión diferente del algoritmo de Bron-Kerbosch que tiene menor desperdicio que el algoritmo 1.

4.2 VERSIÓN 2 DEL ALGORITMO DE BRON-KERBOSCH

El árbol de búsqueda que se presenta en la figura 8, tiene varios vértices desperdicio debido a que en cada llamada del procedimiento BK1, el bucle en la línea 6 selecciona todos los vértices del conjunto P. Por ejemplo, los vértices desperdicio v_2, v_3 y v_4 que son hijos del vértice R, se producen al seleccionar a todos los vértices del conjunto $P = \{v_1, v_2, v_3, v_4\}$, de todos estos vértices, únicamente es necesario elegir al vértice v_1 a fin de que el algoritmo pueda reportar al clan K_4.

Surge entonces la pregunta de si es posible seleccionar menos vértices en el paso 6, de tal forma que las propiedades que se enuncian en el teorema 4.1.1 sigan siendo válidas. Esta cuestión nos lleva a formular la segunda versión del algoritmo de Bron-Kerbosch, el pseudocódigo de esta versión se muestra en el algoritmo 2.

Observe que el algoritmo 2, difiere del algoritmo 1, únicamente en las líneas 6 y 8. El vértice v_p que se selecciona en la línea 6, se le conoce como *pivote*. A diferencia del algoritmo 1, solo se seleccionan los vértices que no son adyacentes al pivote, esto no excluye al pivote de ser seleccionado ya que en una gráfica simple, el vértice v_p no es adyacente a sí mismo.

pivote

Algoritmo 2: Versión 2 del algoritmo de Bron-Kerbosch.

llamada inicial: BK2($\emptyset$, V(G), $\emptyset$)

```
1:  procedure BK2(C, P, N)
2:      Sea P = {v_1, ··· , v_k}
3:      if P = ∅ y N = ∅ then
4:          Reportar C como clan
5:      else
6:          Sea v_p ∈ P un vértice que maximiza a |P ∩ N(v_p)|
7:          for i ← 1 to k do
8:              if v_i ∉ N(v_p) then
9:                  P = P \ {v_i}
10:                 C_new = C ∪ {v_i}
11:                 P_new = P ∩ N(v_i)
12:                 N_new = N ∩ N(v_i)
13:                 BK2(C_new, P_new, N_new)
14:                 N = N ∪ {v_i}
15:             end if
16:         end for
17:     end if
18: end procedure
```

El siguiente teorema muestra que el algoritmo 2, cumple las mismas propiedades que el algoritmo 1. Observe que la restricción que se pide en la línea 6 del algoritmo 2 (seleccionar el pivote v_p que maximiza a $|P \cap N(v_p)|$), no es necesaria para la demostración.

Teorema 4.2.1. *Sea* G *una gráfica simple. Los conjuntos* C, P *y* N *que se usan en el procedimiento* BK2 *(algoritmo 2), tienen las siguientes propiedades:*

(i) *Cada vértice* $v \in P$ *es adyacente a todos los vértices de* C.

(ii) *Cada vértice* $v \in N$ *es adyacente a todos los vértices de* C.

(iii) *Todos los vértices en C son adyacentes por pares, es decir,* C *es una subgráfica completa de* G.

(iv) *Sea* v *un vértice de* G *que no está en el conjunto* C. *Si* v *es adyacente a todos los vértices en* C, *entonces* v *está en* P *o está en* N.

(v) *Si* $v \in N$, *entonces todos los clanes de* G *que contienen a* $C \cup \{v\}$, *ya han sido enumerados.*

(vi) *Al terminar la invocación del procedimiento* BK2, *todos los clanes de la gráfica* G *son enumerados exactamente una sola vez.*

Demostración. Los incisos *(i)*, *(ii)*, *(iii)*, *(iv)* y *(iv)* se demuestran igual que en el teorema 4.1.1.

Para la demostración del inciso *(vi)*, sea $v_p \in P$ el pivote que se elige en la línea 6 del algoritmo 2. Dado que v_p no es adyacente a sí mismo, el vértice v_p será seleccionado para extender al conjunto C y por el inciso *(v)*, al llegar a la línea 14 del algoritmo 2, todos los clanes que contienen a $C \cup \{v_p\}$ han sido reportados.

Suponga que posteriormente se selecciona un vértice $v_i \in N(v_p)$, entonces tenemos 2 casos:

1. Caso 1: $N(v_i) \setminus N(v_p) = \emptyset$.

   Como v_p está en el conjunto N y todos los vecinos de v_i son adyacentes a v_p, el conjunto N_{new} nunca es vacío. Por lo tanto, la condición en la línea 3 del algoritmo 2, no se cumple en las siguientes llamadas recursivas y en consecuencia, el conjunto C *no* se reporta como clan.
2. Caso 2: $N(v_i) \setminus N(v_p) \neq \emptyset$.

   Por el inciso (v), los clanes que contienen a $\{v_p, v_i\}$ se enumeran al seleccionar al vértice v_p. Para determinar qué sucede con los clanes que contienen a v_i pero no a v_p, considere un vértice v_j en $N(v_i) \setminus N(v_p)$. Como v_j no es adyacente al pivote v_p, dicho vértice debe ser seleccionado en la línea 8 del algoritmo 2. Cuando el procedimiento BK2 llega a la línea 14, por el inciso (v), los clanes que contienen a $\{v_i, v_j\}$ ya han sido reportados.

De los casos 1 y 2, concluimos que no es necesario seleccionar a los vértices que están en la vecindad del vértice pivote. Por lo tanto, para listar todos los clanes de la gráfica G es suficiente seleccionar a los vértices en $P \setminus N(p)$. La demostración de que los clanes se enumeran sin repetición es igual que en en el teorema 4.1.1 inciso (vi).

□

Puesto que las propiedades del teorema 4.2.1 se cumplen independientemente de como se seleccione el pivote, es posible usar alguna estrategia heurística a fin de reducir los vértices desperdicio. Una estrategia muy utilizada por su simplicidad y efectividad, es la de seleccionar el vértice con el mayor grado. Es por esta razón que la línea 6 del algoritmo 2, impone la restricción de que el pivote v_p debe maximizar a $|P \cap N(v_p)|$. Sin embargo, existen varios casos en los cuales esta estrategia no es suficiente para reducir el desperdicio, tal como lo demuestra el siguiente lema.

Lema 4.2.1. *[Koch [16, Teorema 3.5]] La elección del vértice $v_p \in P$ con el grado mayor, no necesariamente disminuye el desperdicio en el árbol de búsqueda del algoritmo 2.*

Demostración. Sea K_n una gráfica completa con n vértices, etiquetados de la siguiente manera

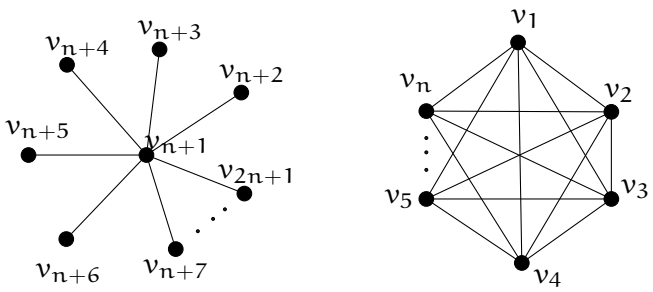
$$V(K_n) = \{v_1, v_2, \ldots, v_n\}.$$

Sea S_n la gráfica estrella con $n+1$ vértices etiquetados como

$$V(S_n) = \{v_{n+1}, v_{n+1}, \ldots, v_{2n+1}\},$$

donde v_{n+1} es el vértice de mayor grado de S_n. Sea G la gráfica con vértices $V(G) = V(K_n) \cup V(S_n)$ y con aristas $E(G) = E(K_n) \cup E(S_n)$. La gráfica G se muestra en la figura 9.

Figura 9: Gráfica G disconexa para el lema 4.2.1.



Al realizar la llamada inicial $BK2(\emptyset, V(G), \emptyset)$, el algoritmo 2 selecciona un pivote en la línea 6. Consideraremos 2 casos en la selección del pivote y denotaremos a el total de nodos de los árboles de recursión de los casos 1 y 2, como A_1 y A_2 respectivamente.

- *Caso 1:* $v_p = v_1$.

  El árbol de búsqueda que resulta de esta elección se muestra en la figura 10. Observe que en este caso el pivote no es el vértice de grado mayor. Los vértices que no son adyacentes al pivote v_1 son: v_{n+1}, v_{n+2}, $\ldots$, v_{2n+1}. Por lo tanto, el nodo raíz R del árbol de búsqueda tiene $n+2$ ramas.

  Por hipótesis, la primera rama el vértice v_1 se usa para extender a C y por el teorema 4.2.1 inciso (iv), dicha rama debe reportar al clan formado por los vértices $v_1, v_2, \ldots, v_p$. Por lo tanto, el total de nodos de esta rama (sin contar el nodo raíz) es n.

  La segunda rama se produce al seleccionar el vértice v_{n+1}. Como los vértices $v_{n+2}, \ldots, v_{2n+1}$ solo tienen como vecino a v_1, la siguiente

llamada recursiva tendrá que reportar a los n clanes formados por los vértices $v_{n+1}, \ldots, v_{2n+1}$. En total, la segunda rama tiene $n+1$ nodos (sin contar el nodo raíz).

Las n ramas restantes se producen al seleccionar los vértices $v_{n+2}, \ldots, v_{2n+1}$. Cada uno de estos vértices es vecino de v_{n+1}. Por lo tanto, el conjunto N nunca es vacío y en consecuencia estas ramas no reportan clanes. De esto se tiene que el total de nodos de las ramas restantes es n (sin contar el nodo raíz).

El total de nodos del árbol de busqueda es:

$$A_1 = n + (n+1) + n + 1 = 3n + 2$$

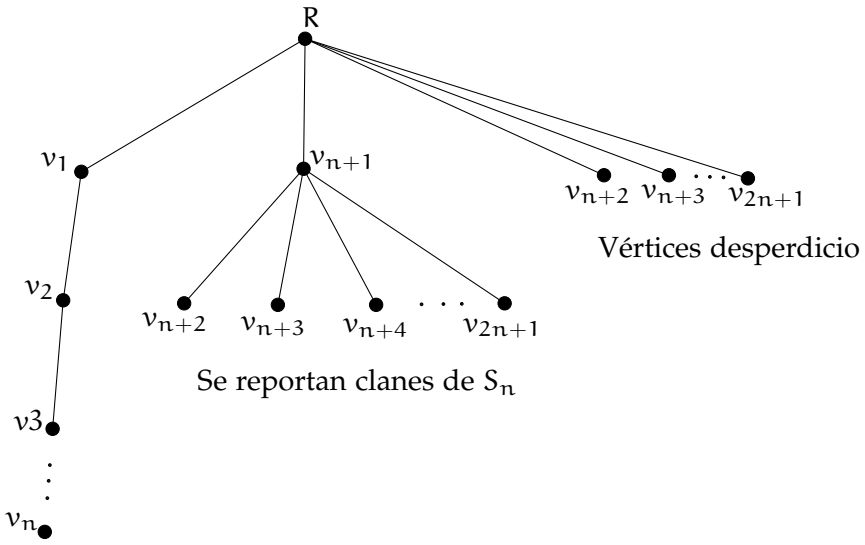


Figura 10: Variante 1, se elige a $v_p = v_1$.

- *Caso 2:* $v_p = v_{n+1}$.

El árbol de búsqueda que resulta de esta elección se muestra en la figura 11. Observe que en este caso, el pivote es el vértice de grado mayor. Los vértices que no son adyacentes al pivote son: v_1, ..., v_p. De esto se tiene que el nodo raíz R tiene $n+1$ ramas.

La primera rama se produce al seleccionar al pivote v_{n+1}, por el teorema 4.2.1 inciso *(v)*, dicha rama reporta los n clanes formados por los vértices v_{n+1}, v_{n+1}, ..., v_{2n+1}. En total, la primera rama tiene $n+1$ nodos (sin contar el nodo raíz).

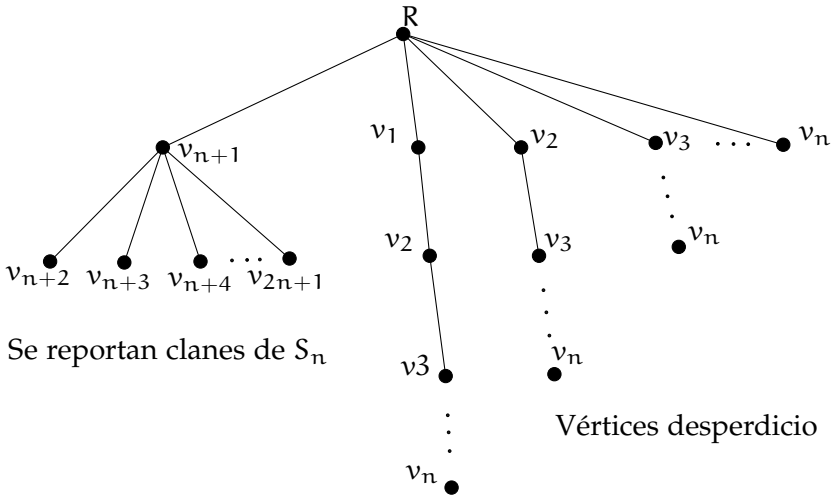
En la segunda rama se selecciona al vértice v_1, por el teorema 4.1.1 inciso (v), dicha rama debe de reportar al clan K_n. Por lo tanto, el total de nodos en esta rama es n. Al terminar la llamada recursiva que genera a esta rama, el vértice v_1 se agrega al conjunto N (paso 14).

La tercera rama se produce al seleccionar el vértice v_2. En las siguientes llamadas recursivas el conjunto P siempre es un subconjunto de $\{v_2, \ldots, v_n\}$ y el conjunto N nunca es vacío ya que $N[v_2] \subset N[v_1]$. Por lo tanto, esta rama enumera los vértices de la subgráfica completa con vértices $\{v_2, \ldots, v_p\}$, pero no la reporta como clan. En total esta rama tiene $n-1$ nodos. Por un razonamiento similar, las siguientes ramas encuentran subgráficas completas de cardinalidades $n-2$, $n-3$, ..., 2, 1. Sin embargo, ninguna de estas subgráficas se reporta como clan. El total de nodos desde la segunda rama hasta la última rama es $n + (n-1) + \ldots + 2 + 1 = \frac{n^2+n}{2}$.

El total de nodos del árbol de búsqueda (contando el nodo raíz) es:

$$A_2 = (n+1) + \frac{n^2+n}{2} + 1 = \frac{1}{2}n^2 + \frac{3}{2}n + 2$$

Figura 11: Variante 2, se elige a $v_p = v_{n+1}$.



En ambos casos, el total de vértices trabajo es el mismo. Cuando $n \geqslant 4$ entonces $A_2 > A_1$ y por lo tanto el caso 1 tiene menos desperdicio que el caso 2. □

La gráfica G que se definió en el lema 4.2.1 es disconexa. Dicha gráfica puede hacerse conexa agregando la arista $\{v_1, v_{2p+1}\}$. Por medio de un razonamiento similar al caso disconexo, se puede demostrar que también se cumple que $A_1 < A_2$.

El lema anterior nos lleva a enunciar la tercera versión del algoritmo de Bron-Kerbosch.

4.3 VERSIÓN 3 DEL ALGORITMO DE BRON-KERBOSCH

El argumento que se utilizó en el caso 2 del lema 4.2.1, muestra que el algoritmo 2 tiene el siguiente problema: Si un clan Q no se reporta en la primera rama, el árbol de búsqueda puede llegar a tener $\frac{(|Q|-1)^2+(|Q|-1)}{2}$ vértices desperdicio. La versión 3 del algoritmo de Bron-Kerbosch, soluciona este problema eligiendo el pivote de una manera diferente de como se hace en el algoritmo 2. El pseudocódigo de la versión 3 se muestra a continuación.

Algoritmo 3: Versión 3 del algoritmo de Bron-Kerbosch.

llamada inicial: BK3($\emptyset$, V(G), $\emptyset$)

```
1:  procedure BK3(C, P, N)
2:      Sea P = {v_1, ··· , v_k}
3:      if P = ∅ y N = ∅ then
4:          Reportar C como clan
5:      else
6:          Sea v_p un vértice en N ∪ P que maximiza a |P ∩ N(v_p)|
7:          for i ← 1 to k do
8:              if v_i ∉ N(v_p) then
9:                  P = P \ {v_i}
10:                 C_new = C ∪ {v_i}
11:                 P_new = P ∩ N(v_i)
12:                 N_new = N ∩ N(v_i)
13:                 BK3(C_new, P_new, N_new)
14:                 N = N ∪ {v_i}
15:             end if
16:         end for
17:     end if
18: end procedure
```

La única diferencia que tiene el algoritmo 3, con respecto al algoritmo 2, es la forma en como se elige el pivote en la línea 6. Mientras que en el

algoritmo 2, el pivote se elige del conjunto P, en el algoritmo 3, el pivote se elige de la unión de los conjuntos P y N. Es importante notar que las iteraciones de la instrucción for en la línea 7, se hacen sobre los índices de los vértices que están en P, por lo que si el pivote v_p se elige del conjunto N, entonces v_p no puede ser utilizado para extender a C.

El siguiente teorema muestra que las propiedades del algoritmo 2, también se cumplen para el algoritmo 3. Observe que al igual que en el teorema 4.2.1, la demostración del teorema 3, no requiere la condición de seleccionar el pivote v_p de tal forma que maximiza a $|P \cap N(v_p)|$. Esta última condición, se utiliza por su simplicidad y efectividad para reducir el número de vértices desperdicio del árbol de búsqueda del algoritmo 3.

Teorema 4.3.1 (Koch [16, Lema 3.6], Tomita et al. [29, Teorema A1])**.** *Sea* G *una gráfica simple. Los conjuntos* C, P *y* N *que se usan en el procedimiento* BK3 *(algoritmo 3), tienen las siguientes propiedades:*

(i) *Cada vértice* $v \in P$ *es adyacente a todos los vértices de* C.

(ii) *Cada vértice* $v \in N$ *es adyacente a todos los vértices de* C.

(iii) *Todos los vértices en C son adyacentes por pares, es decir,* C *es una subgráfica completa de* G.

(iv) *Sea* v *un vértice de* G *que no está en el conjunto* C. *Si* v *es adyacente a todos los vértices en* C, *entonces* v *está en* P *o está en* N.

(v) *Si* $v \in N$, *entonces todos los clanes de* G *que contienen a* $C \cup \{v\}$, *ya han sido enumerados.*

(vi) *Al terminar la invocación del procedimiento* BK3, *todos los clanes de la gráfica* G *son enumerados exactamente una sola vez.*

Demostración. Los incisos *(i)*, *(ii)*, *(iii)*, *(iv)* y *(v)* se demuestran igual que en el teorema 4.1.1.

Para la demostración del inciso *(vi)*, suponga que el pivote v_p está en el conjunto N. Por el inciso *(v)* los clanes que contienen a v_p ya han sido enumerados. Suponga que se selecciona un vértice $v_i \in N(v_p)$, entonces tenemos 2 casos:

1. Caso 1: $N(v_i) \setminus N(v_p) = \emptyset$.

   Debido a que todos los vecinos de v_i son adyacentes a v_p, el conjunto N_{new} nunca es vacío. Por lo tanto, la condición en la línea 3 del algoritmo 3, no se cumple en las siguientes llamadas recursivas y por tanto, el conjunto C *no* se reporta como clan.

2. Caso 2: $N(v_i) \setminus N(v_p) \neq \emptyset$.

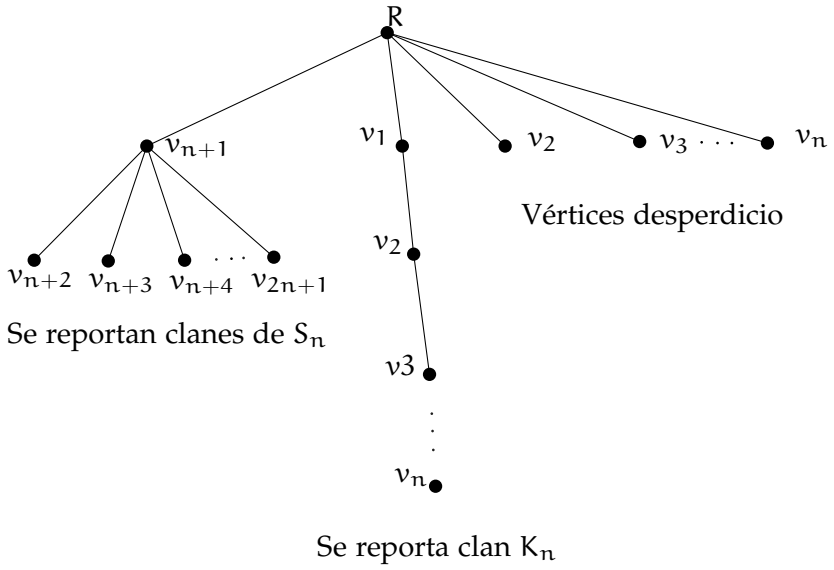
   Puesto que $v_p \in N$, los clanes que contienen a $\{v_p, v_i\}$ ya han sido reportados. Observe que los vértices que están en el conjunto $R = N(v_i) \setminus N(v_p)$ son vecinos de v_i pero no del vértice v_p, por lo tanto, dichos vértices deben ser seleccionados en la línea 8 del algoritmo 3. Por el inciso *(v)*, todos los clanes que contienen a v_i pero no a v_p, serán reportados al seleccionar los vértices en R.

De los casos anteriores, se concluye que no es necesario seleccionar a los vértices en la vecindad abierta de v_p. Por medio del mismo razonamiento que se utilizó en la prueba del teorema 4.2.1 inciso *(vi)*, podemos concluir que si el pivote v_p está en P, tampoco es necesario seleccionar sus vecinos. Por lo tanto, el pivote v_p se puede elegir de $N \cup P$. La demostración de que los clanes se reportan sin repetición, es igual que en la prueba del teorema 4.1.1 inciso *(vi)*.

□

El simple hecho de elegir el pivote del conjunto $P \cup N$, reduce drásticamente el desperdicio del árbol de búsqueda. Compare el árbol de búsqueda del algoritmo 3, cuando G es la gráfica del lema 4.2.1 (figura 12), con el árbol de búsqueda del algoritmo 2 (figura 11), para la misma gráfica G.

Figura 12: Árbol de búsqueda del algoritmo 3 cuando G es la gráfica de la figura 12.



En la siguiente sección presentamos el algoritmo TTT. Como se verá más adelante, los resultados del algoritmo TTT también aplican para el algoritmo de Bron-Kerbosch.

4.4 EL ALGORITMO TTT

El algoritmo que se estudia en esta sección fue descrito por Tomita, Tanaka y Takahashi en [29]. De aquí en adelante denotaremos a este algoritmo como el *algoritmo TTT* .

algoritmo TTT

En este trabajo presentamos 2 versiones del algoritmo TTT, las cuales difieren principalmente en la forma en la que imprimen cada clan que encuentran. Aunque en [29] no se hace tal distinción, creemos que es más sencillo estudiar al algoritmo TTT usando este enfoque.

4.4.1 *Versión 1 del algoritmo TTT*

El pseudocódigo de la primera versión del algoritmo TTT se muestra a continuación. Usamos símbolos diferentes de los que se usan en [29], a fin de que resulte más fácil identificar las similitudes que tiene el algoritmo TTT con el algoritmo de Bron-Kerbosch.

Algoritmo 4: Versión 1 del algoritmo TTT.

llamada inicial: TTT1($\emptyset$, $V(G)$, $V(G)$)

```
1:  procedure TTT1(C, S, P)
2:      if S = ∅ then
3:          Reportar que C es un clan
4:      else
5:          Sea u un vértice en S que maximiza |P ∩ N(u)|
6:          while P \ N(u) ≠ ∅ do
7:              Seleccionar vértice q en P \ N(u)
8:              C_new = C ∪ {q}
9:              S_new = S ∩ N(q)
10:             P_new = P ∩ N(q)
11:             TTT1(C_new, S_new, P_new)
12:             P = P \ {q}
13:         end while
14:     end if
15: end procedure
```

El algoritmo 4 usa principalmente 3 conjuntos: C, S y P. La función que tienen estos conjuntos se describe a continuación:

- *C (clan)*: Contiene los vértices de una subgráfica completa de G. El algoritmo extiende este conjunto por medio de los vértices en el conjunto P. *conjunto C*
- *P (posibles)*: Se usa para almacenar a todos los *posibles* vértices que pueden usarse para extender al conjunto C. *conjunto P*
- *S (subgráfica)*: Contiene a todos los vértices de P. También contiene a todos los vértices que ya han sido utilizados para extender al conjunto C. *conjunto S*

Una simple comparación del algoritmo 4, con el algoritmo 3, muestra que hay múltiples similitudes entre estos algoritmos. El siguiente teorema es una recopilación de las observaciones hechas en [5], las cuales nos permiten concluir que ambos algoritmos son básicamente el mismo.

Teorema 4.4.1 (Cazals, Karande [5]). *La* versión 1 *del algoritmo TTT (algoritmo 4), tiene las siguientes propiedades con respecto a la versión 3 del algoritmo de Bron-Kerbosch (algoritmo 3).*

(i) Los conjuntos C *y* P*, tienen la misma función en el procedimiento* TTT1 *(algoritmo 4) y en el procedimiento* BK3 *(algoritmo 3).*

(ii) El conjunto N *del procedimiento* BK3*, es equivalente al conjunto* $(S \setminus P)$ *en el procedimiento* TTT1*.*

(iii) $S = \emptyset$ *en el procedimiento* TTT1 *si y solo si en el procedimiento* BK3 *se cumple que* $N = \emptyset$ *y* $P = \emptyset$*.*

Demostración.

(i) Al inicio, en ambos algoritmos el conjunto C es vacío y P contiene todos los vértices de G. En ambos algoritmos se elige un pivote u y los vértices que se seleccionan para extender a C, se toman del conjunto $P \setminus N(u)$. La creación de los conjuntos P_{new} y C_{new} es similar en ambos casos .

(ii) Al inicio del procedimiento TTT1, se tiene que $P = S = V(G)$ y por lo tanto, $S \setminus P = \emptyset$. Más aún, si q es el vértice que se selecciona en la línea 7 del algoritmo 4, entonces $S_{new} = N(q)$ y $P_{new} = N(q)$. En consecuencia, $(S_{new} \setminus P_{new}) = \emptyset$. Esto coincide con los valores del conjunto N al inicio del procedimiento BK3.

En las siguientes llamadas recursivas del procedimiento TTT1, se tiene que

$$(S_{new} \setminus P_{new}) = (S \setminus P) \cap N(q).$$

Como $N = S \setminus P$, entonces $N_{new} = (S_{new} \setminus P_{new})$. Puesto que P es un subconjunto de S, se tiene que

$$P \setminus (S \setminus \{q\}) = (P \setminus S) \cup \{q\} = N \cup \{q\}$$

Por lo tanto, la operación $P = P \setminus \{q\}$ que realiza en la línea 12 del procedimiento TTT1, de manera implícita realiza la operación $N = N \setminus \{q\}$.

(iii) Por el inciso *(ii)*, sabemos que $N = S \setminus P$. Puesto que P es un subconjunto de S, se tiene que $S = \emptyset$ si y solo si $P = \emptyset$ y $S = \emptyset$.

□

Corolario 4.4.1. *Suponga que en la línea 7 del procedimiento* TTT1*, todos los vértices se seleccionan siguiendo el mismo orden que el bucle for en la línea 7 del procedimiento* BK3*. Entonces, en todo momento los conjuntos* C*,* P *y* N *del procedimiento* BK3 *tienen los mismos vértices que los conjuntos* C*,* P *y* $(S \setminus P)$ *del procedimiento* TTT1*.*

Demostración. Inmediato de los incisos *(i)*, *(ii)* y *(iii)* del teorema 4.4.1. □

Corolario 4.4.2. *La versión 1 del algoritmo* TTT *cumple con las propiedades del teorema 4.3.1.*

Demostración. Usar el mismo argumento que se utilizó en la prueba del teorema 4.3.1, con los conjuntos S, P y $(S \setminus P)$. □

Más adelante demostraremos que la complejidad temporal del algoritmo 4 es $\mathcal{O}(n3^{\frac{n}{3}})$. Si se tiene cuidado con la implementación de la versión 3 del algoritmo de Bron-Kerbosch, se puede concluir por medio del teorema 4.4.1, que la complejidad de dicho algoritmo también es $\mathcal{O}(n3^{\frac{n}{3}})$.

4.4.2 *Versión 2 del algoritmo TTT*

La segunda versión del algoritmo TTT se basa en el hecho de que se puede omitir el conjunto C del algoritmo 4, sin que esto afecte a la forma del árbol de búsqueda. El pseudocódigo de la segunda versión se muestra en el algoritmo 5.

Algoritmo 5: Versión 2 del algoritmo TTT.

llamada inicial: TTT2(V(G), V(G))

```
1:  procedure TTT2(S, P)
2:      if S = ∅ then
3:          imprimir("clan,")
4:      else
5:          Sea u un vértice en S que maximiza |P ∩ N(u)|
6:          while P \ N(u) ≠ ∅ do
7:              Seleccionar vértice q en P \ N(u)
8:              imprimir(q, ",")
9:              S_new = S ∩ N(q)
10:             P_new = P ∩ N(q)
11:             TTT2(S_new, P_new)
12:             P = P \ {q}
13:             imprimir("regresar,")
14:         end while
15:     end if
16: end procedure
```

El algoritmo 5, difiere del algoritmo 4 (procedimiento TTT1), únicamente en las líneas 3, 8 y 13. Si u es el vértice pivote y q es el vértice que selecciona del conjunto $(P \setminus N(u))$, el algoritmo 5 imprime el vértice q seguido de una coma (en lugar de guardarlo en el conjunto C, como se hace en el procedimiento TTT1).

Observe que cuando el algoritmo 5 encuentra un clan, el procedimiento TTT2 imprime el texto "clan," y cuando termina la llamada recursiva del procedimiento TTT2, el algoritmo imprime "regresar,". Por ejemplo, si $G = K_4 = \{v_1, v_2, v_3, v_4\}$ entonces el algoritmo 5 imprime la siguiente cadena:

$$v_1,\ v_2,\ v_3,\ v_4,\ \texttt{clan},\ \texttt{regresar},\ \texttt{regresar},\ \texttt{regresar},\ \texttt{regresar},$$

Para ilustrar el significado de la cadena "regresar," suponga que el algoritmo 5 se usa para buscar los clanes de la gráfica que se muestra en la figura 13. Cuando el algoritmo 5 termina, el procedimiento TTT2 produce

la siguiente salida (hemos dividido el texto de salida en 3 renglones a fin de que sea más fácil de entender):

v_1, v_2, v_3, v_4, clan, regresar, regresar,
v_5, v_6, clan, regresar, regresar,
v_6, regresar, regresar,

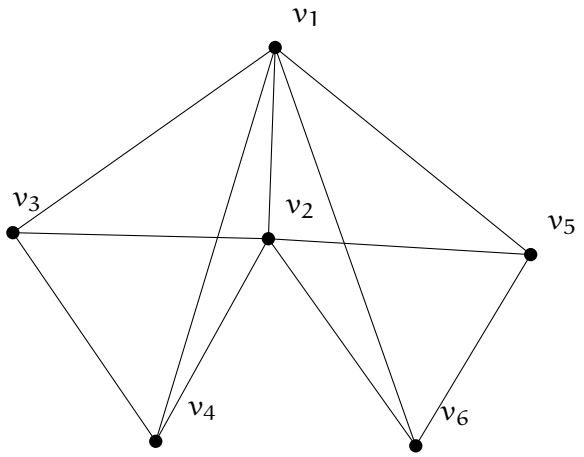
El primer renglón del texto anterior, indica que el algoritmo encontró un clan con vértices $Q = \{v_1, v_2, v_3, v_4\}$. Las 2 cadenas "regresar" a la derecha de la cadena "clan", indican que hay que remover los vértices v_4 y v_3 de Q, a fin de poder listar el siguiente clan que se encontró.

En el segundo renglón, la cadena "v_5, v_6" indica que se deben agregar los vértices v_5 y v_6 al conjunto Q. La cadena "clan", indica que el algoritmo encontró un clan con vértices $Q = \{v_1, v_2, v_5, v_6\}$. Las 2 cadenas "regresar" al final del renglón, indican que hay que remover los vértices v_5 y v_6 de Q para poder enumerar el siguiente clan que se encontró.

El tercer renglón indica que el algoritmo encontró una subgráfica completa con vértices $Q = \{v_1, v_2, v_6\}$, no es un clan ya que la cadena "regresar," se encuentra a la derecha de la cadena "v_6".

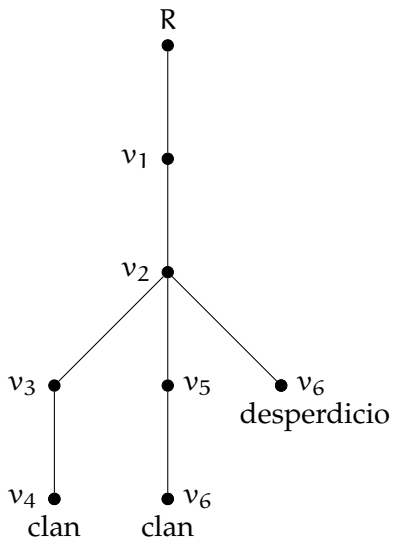
El árbol de búsqueda del algoritmo 5 para la gráfica 13, se muestra en la figura 14.

Figura 13: Gráfica de ejemplo para la segunda versión del algoritmo TTT.



La estrategia utilizada por el algoritmo 5 para imprimir clanes, es crucial para que su complejidad temporal sea $\mathcal{O}\left(3^{\frac{n}{3}}\right)$. Como era de esperarse, el algoritmo 5 cumple con las mismas propiedades que la versión 1 del algoritmo TTT.

Figura 14: Árbol de búsqueda del algoritmo 5 cuando G es la gráfica de la figura 13.



Teorema 4.4.2. *La versión 2 del algoritmo TTT cumple las propiedades del teorema 4.4.1.*

Demostración. Aunque el conjunto C no está presente de manera explícita en el algoritmo 5, podemos suponer que dicho conjunto es una variable auxiliar del algoritmo cuyo *contenido nunca se imprime*. Tomando en cuenta esta observación, el algoritmo 5 cumple las propiedades *(i)*, *(ii)* y *(iii)* del teorema 4.4.1. □

4.4.3 *Complejidad temporal del algoritmo TTT*

Antes de estudiar la complejidad temporal de los algoritmos 3, 4 y 5 conviene definir las siguientes funciones y constantes que usaremos para el resto de los lemas y teoremas de esta sección.

Definición 4.4.1. *Sea* $p_1 > 0$ *una constante en los reales, defina las constantes* q_1, q_2 *y* q_3 *como sigue:*

constantes q_1, q_2 *y* q_3

$$q_1 = \frac{p_1}{2} > 0,$$
$$q_2 = 9\frac{p_1}{2} > 0,$$
$$q_3 = 27\frac{p_1}{2} > 0.$$

función $Q(x)$ **Definición 4.4.2.** *Defina la función* $Q(x)$ *sobre los reales como sigue:*

$$Q(x) = q_1 x^2 + q_2 x + q_3.$$

constantes k_1, k_2, k_3 *y* k_m **Definición 4.4.3.** *Defina las constantes* k_1, k_2, k_3 *y* k_m *como sigue:*

$$k_1 = \frac{3q_2}{\ln(3)} = \frac{27p_1}{2 \cdot \ln(3)}$$
$$k_2 = \frac{39p_1}{2 \cdot 3^{1/3}}$$
$$k_3 = \max_{x \geqslant 0} \left\{ \frac{3Q(x-3)}{3^{(x/3)}\left(1 - 2 \cdot 3^{-(2/3)}\right)} \right\}$$
$$k_m = \max\{k_1, k_2, k_3\}.$$

Puesto que

$$\lim_{x \to \infty} \frac{Q(x-3)}{3^{(x/3)}} = 0,$$

la función $\frac{Q(x-3)}{3^{(x/3)}}$ *esta acotada superiormente, por lo tanto las constantes* k_3 *y* k_m *están bien definidas.*

función $R(x)$ **Definición 4.4.4.** *Defina la función* $R(x)$ *sobre los reales como sigue:*

$$R(x) = k_m(3^{x/3}) - Q(x).$$

Mas adelante se verá que la función $R(x)$ es una cota superior para la función de complejidad temporal del algoritmo TTT. En particular la función $R(x)$ tiene la siguiente propiedad:

Lema 4.4.1. *La función* $R(x)$ *es creciente para* $x \geqslant 0$.

Demostración. La primera derivada de la función $R(x)$ es

$$R'(x) = \ln(3) \cdot k_m \left(3^{x/3-1}\right) - 2q_1 x - q_2.$$

La segunda derivada de la función $R(x)$ es

$$R''(x) = \ln^2(3) \cdot k_m \left(3^{x/3-2}\right) - 2q_1.$$

Como $k_m \geqslant k_1$ se tiene que

$$R''(x) \geqslant \ln^2(3) \cdot k_1 \left(3^{x/3-2}\right) - 2q_1 = \ln(3) \cdot q_2 \left(3^{x/3-1}\right) - 2q_1.$$

Al reemplazar los valores de las constantes q_1 y q_2 por los de la definición 4.4.1 queda

$$R''(x) \geqslant \left(\frac{3 \cdot \ln(3)\left(3^{x/3}\right)}{2} - 1\right) p_1 > 0 \quad \forall x \geqslant 0.$$

Puesto que $R''(x)$ es la primera derivada de la función $R'(x)$, por la desigualdad anterior se tiene que $R'(x)$ es creciente para $x \geqslant 0$. Por lo tanto

$$R'(x) \geqslant R'(0) = \frac{\ln(3) \cdot k_m}{3} - q_2 \geqslant \frac{\ln(3) \cdot k_1}{3} - q_2 = 0 \qquad x \geqslant 0.$$

De la desigualdad anterior concluimos que $R(x)$ es creciente para $x \geqslant 0$.

□

Para poder determinar la complejidad temporal del algoritmo TTT, se debe tener muy en cuenta el siguiente resultado.

Lema 4.4.2. *Para todo entero positivo $n \neq 3$ se cumple que*

$$n\left(3^{-(n/3)}\right) \leqslant 2 \cdot 3^{-(2/3)}.$$

Demostración. El caso $n = 1$ es fácil de verificar. La igualdad se cumple cuando $n = 2$. Si $n > 3$, considere la función $g(x)$ sobre los reales que se define como

$$g(x) = x\left(3^{-(x/3)}\right) - 2 \cdot 3^{-(2/3)}.$$

La función $g(x)$ es continua y diferenciable en todo $\mathbb{R}$. Si $x > 3$, por el teorema del valor medio, existe una constante c en el intervalo $(3, x)$ tal que

$$g'(c) = \frac{g(x) - g(3)}{x - 3}.$$

Como $c > 3$ se tiene que

$$g'(c) = 3^{-(c/3)}\left(1 - \frac{\ln(3) \cdot c}{3}\right) < 0,$$

luego

$$\frac{g(x) - g(3)}{x - 3} < 0.$$

Como $x > 3$, el denominador de la desigualdad anterior se puede cancelar, entonces

$$g(x) < g(3) = \frac{1 - \ln(3)}{3} < 0.$$

De esto se concluye que

$$x\left(3^{-(x/3)}\right) < 2 \cdot 3^{-(2/3)} \quad \forall x > 3.$$

En particular, la desigualdad anterior es válida para todos los enteros $n > 3$. □

Las siguientes definiciones se hacen considerando la estructura del algoritmo 5, sin embargo, se pueden enunciar definiciones similares para el algoritmo 4.

Definición 4.4.5. *Sean S y P los conjuntos de la segunda versión del algoritmo TTT. Sea* TTT2 *el procedimiento del algoritmo 5. Sea $n \geqslant 1$ un entero positivo.*

tiempo $T(n)$
- *Denotaremos por $T(n)$, al tiempo de ejecución que tiene en el peor caso, la invocación del procedimiento* $TTT2(S, P)$ *cuando* $|S| = n$.

tiempo $T_k(n)$
- *Denotaremos por $T_k(n)$, al tiempo de ejecución que tiene en el peor caso, la invocación del procedimiento* $TTT2(S, P)$ *cuando* $|S| = n$ *y* $|P \setminus N(u)| = k$ *para algún vértice pivote* $u \in S$.

procedimiento $TTT2_\emptyset$
- *Denotaremos por* $TTT2_\emptyset(S, P)$, *al procedimiento que se obtiene de* $TTT2(S, P)$ *al reemplazar la línea 11:*

$$TTT2(S_{new}, P_{new})$$

por:

$$TTT2(\emptyset, \emptyset)$$

tiempo $T_\emptyset(n)$
- *Denotaremos por $T_\emptyset(n)$, al tiempo de ejecución que tiene en el peor caso, la invocación del procedimiento* $TTT2_\emptyset(S, P)$ *cuando* $|S| = n$.

Observación 4.4.1. *De las definiciones anteriores es inmediato el siguiente resultado:*

$$T(n) = \max_{1 \leqslant k \leqslant n} \{T_k(n)\}$$

El siguiente lema, muestra que la complejidad temporal del procedimiento $TTT2_\emptyset$ es polinomial.

Lema 4.4.3. *La complejidad temporal del procedimiento* $TTT2_\emptyset(S, P)$ *cuando* $|S| = n$ *es*

$$T_\emptyset(n) = \mathcal{O}(n^2).$$

Demostración. La selección del pivote en la línea 5 del algoritmo 5, puede hacerse de la siguiente manera:

1. Iterar sobre el conjunto S seleccionando un vértice $v \in S$, en cada ciclo.
2. Por cada vértice v, realizar la operación $r = |P \cap N(v)|$. Si r es mayor que la iteración previa, hacer $u = v$.

El paso 2 puede hacerse en tiempo $\mathcal{O}(n)$. Puesto que el paso 2 realiza n veces el paso 1, la selección del pivote u se puede hacer en tiempo $\mathcal{O}(n^2)$.

La selección del vértice q (línea 7 del algoritmo 5) y la creación de los conjuntos S_{new} y P_{new}, puede hacerse en tiempo $\mathcal{O}(n)$. Las operaciones en las líneas 3, 8, 11, 12 y 13 del procedimiento $TTT2_\emptyset$, pueden hacerse en tiempo constante. Por lo tanto, cada ciclo del bucle *while* (línea 6 del algoritmo 5) puede hacerse en tiempo $\mathcal{O}(n)$.

La instrucción *while* del procedimiento TTT2, realiza $|P \setminus N(u)| \leqslant n$ ciclos, en consecuencia, el tiempo total de ejecutar esta instrucción es $\mathcal{O}(n^2)$.

Concluimos que el tiempo de ejecución del procedimiento $TTT2_\emptyset(S, P)$ es

$$T_\emptyset(n) = \mathcal{O}(n^2).$$

□

Observación 4.4.2. *Como* $T_\emptyset(n) = \mathcal{O}(n^2)$ *existe una constante* $c > 0$ *y un entero* $n_0 \geqslant 1$ *tal que*

$$T_\emptyset(n) \leqslant c \cdot n^2 \qquad \forall n > n_0$$

De aquí en adelante se asumirá que la constante p_1 *que se usa en las definiciones 4.4.1 y 4.4.3 tiene el siguiente valor:*

$$p_1 = \max(\{T_\emptyset(1), T_\emptyset(2), \ldots, T_\emptyset(n_0)\} \cup \{c\})$$

Es importante notar, que la complejidad temporal del procedimiento $TTT2_\emptyset$ es $\mathcal{O}(n^2)$, gracias a que en la línea 3 del algoritmo 5, no se imprimen todos los vértices del clan. Si se imprimen todos los vértices que forman al clan C (tal como se hace en la primera versión del algoritmo TTT), podría ocurrir que $n^2 \leqslant |C|$ y por lo tanto el análisis del lema 4.4.3, no sería valido. Es por esta razón que la segunda versión del algoritmo TTT, usa la estrategia de impresión de clanes que se mencionó con anterioridad.

El siguiente lema es crucial para poder acotar a la complejidad temporal $T(n)$.

Lema 4.4.4 (Tomita, Tanaka y Takahashi [29, lema 2]). *Sea* $TTT2(S, P)$ *la invocación del algoritmo 5 cuando* $|S| = n$ *y* $|P \setminus N(u)| = k$, *para algún vértice pivote* $u \in S$.

Sea $P \setminus N(u) = \{v_1, v_2, \ldots, v_k\}$ *y suponga que los vértices en la línea 7 del algoritmo 5, se eligen siguiendo este orden. Defina* k *conjuntos de la siguiente manera*

$$S_i = S \cap N(v_i) \quad \textit{para todo } i = 1, 2, \ldots, k.$$

Entonces se cumplen las siguientes desigualdades:

(i) $T_k(|S|) \leqslant T_\emptyset(n) + \sum_{i=1}^{k} T(|S_i|)$.

(ii) $|S_i| \leqslant n - k \leqslant n - 1 \qquad \forall i = 1, 2, \ldots, k.$

Demostración.

(i) El tiempo de ejecución del procedimiento TTT2 *sin invocar* la llamada recursiva en la línea 11 del algoritmo 5, es el mismo que $TTT2_\emptyset$. Para cada vértice v_i, el tiempo de la llamada recursiva $TTT2(S_i, P \cap N(v_i))$ es $T(|S_i|)$. Por lo tanto

$$T_k(|S|) \leqslant \sum_{i=1}^{k} T(|S_i|) + T_\emptyset(n).$$

(ii) Como el vértice u se elige de tal forma que maximiza $|P \cap N(u)|$ y como $|P \setminus N(u)| = k$, para todo vértice v_i se cumple lo siguiente

$$|P \cap N(v_i)| \leqslant |P \cap N(u)| = |P| - k.$$

Como $S = (S \setminus P) \cup P$ y por la desigualdad anterior se tiene que

$$\begin{aligned} |S \cup N(v_i)| &= |(S \setminus P) \cap N(v_i)| + |P \cap N(v_i)| \\ &\leqslant |S \setminus P| + |P \cap N(v_i)| \\ &\leqslant (n - |P|) + (|P| - k) = n - k. \end{aligned}$$

Como $k \geqslant 1$ entonces $n - k \leqslant n - 1$.

□

El siguiente teorema nos permitirá concluir que la versión 2 del algoritmo TTT, tiene complejidad temporal $\mathcal{O}(3^{n/3})$.

Teorema 4.4.3 (Tomita, Tanaka y Takahashi [29, Teorema 3]). *Sea* p_1 *la constante cuyo valor esta determinado por la observación 4.4.2. Sea* k_m *la constante que se definió en 4.4.3. Sean* $Q(x)$ *y* $R(x)$ *las funciones que se definieron en 4.4.2 y 4.4.4 respectivamente. Sea* $T(n)$ *la función de complejidad temporal de* TTT2 *que se definió en 4.4.5.*

Para todo entero $n \geqslant 1$ *se cumple que*

$$T(n) \leqslant R(n) = k_m \cdot 3^{n/3} - Q(n).$$

Demostración. La prueba se hará por inducción sobre n.

Si $n = 1$, el procedimiento TTT2 es el mismo que el procedimiento $TTT2_\emptyset$. Como $k_2 = \frac{39p_1}{2\cdot 3^{(1/3)}}$, el valor de $R(1)$ cumple lo siguiente

$$R(1) = k_m \cdot 3^{(1/3)} - Q(1) \geqslant k_2 \cdot 3^{(1/3)} - \frac{37p_1}{2} = p_1.$$

Por la observación 4.4.2, se tiene que $T_\emptyset(n) \leqslant p_1 \cdot n^2$, luego

$$T(1) = T_\emptyset(1) \leqslant p_1 \leqslant R(1).$$

Suponga que el teorema es válido para $n \leqslant N$. Mostraremos que el teorema también se cumple para $n = N + 1$.

Considere el procedimiento $TTT2(S, P)$, con $|S| = n$ y con un pivote u tal que $P \setminus N(u) = \{v_1, v_2, \ldots, v_k\}$. Por el lema 4.4.4 inciso (i), se tiene que

$$T_k(n) = T_k(|S|) \leqslant T_\emptyset(n) + \sum_{i=1}^{k} T(|S_i|).$$

Por el lema 4.4.4 inciso (ii), se tiene que $|S_i| \leqslant n - k$ y por la hipótesis de inducción $T(|S_i|) \leqslant R(|S_i|)$. Por el lema 4.4.1, la función $R(n)$ es creciente. Al sustituir estas observaciones en la desigualdad anterior se tiene que

$$\begin{aligned} T_k(n) &\leqslant T_\emptyset(n) + \sum_{i=1}^{k} R(|S_i|) \leqslant T_\emptyset(n) + k \cdot R(n-k) \\ &= T_\emptyset(n) + k_m(k \cdot 3^{(n/3)}3^{-(k/3)}) - kQ(n-k). \end{aligned} \tag{3}$$

Si n es fijo, mostraremos que la parte derecha de la desigualdad (3), alcanza su valor máximo cuando $k = 3$. Este problema es equivalente a demostrar la siguiente desigualdad:

$$k_m(k \cdot 3^{(n/3)}3^{-(k/3)}) - kQ(n-k) \leqslant k_m(3^{(n/3)}) - 3Q(n-3). \tag{4}$$

Es evidente que la desigualdad (4) se cumple cuando $k = 3$. Suponga que $k \neq 3$, entonces (4) es equivalente a resolver la siguiente desigualdad

$$\frac{3Q(n-3) - kQ(n-k)}{3^{n/3} \cdot (1 - k \cdot 3^{-(k/3)})} \leqslant k_m. \tag{5}$$

Como $n - k \geqslant 0$, entonces $kQ(n-k) \geqslant 0$. Por el lema 4.4.2, se tiene que $k \cdot 3^{-(n/3)} \leqslant 2 \cdot 3^{-(2/3)}$ y por la definición 4.4.3 de k_3, en la parte izquierda de la desigualdad (5) obtenemos que

$$\frac{3Q(n-3) - kQ(n-k)}{3^{n/3} \cdot (1 - k \cdot 3^{-(k/3)})} \leqslant \frac{3Q(n-3)}{(1 - 2 \cdot 3^{-(2/3)}) \cdot 3^{n/3}} \leqslant k_3$$

Como $k_3 \leqslant k_m$, las desigualdades (5) y (4) son válidas. Por lo tanto, en la desigualdad (3) queda

$$\begin{aligned} T_k(n) &\leqslant T_\emptyset(n) + k_m 3^{(n/3)} - 3Q(n-3) \\ &\leqslant p_1 n + k_m 3^{(n/3)} - 3\left(\frac{p_1}{2}(n-3)^2 + \frac{9p_1}{2}(n-3) + \frac{27p_1}{2}\right) \\ &= k_m 3^{(n/3)} - \left(\frac{p_1}{2}n^2 + \frac{9p_1}{2}n + \frac{27p_1}{2}\right) \\ &= k_m 3^{(n/3)} - Q(n) = R(n). \end{aligned}$$

De la desigualdad anterior y de la observación 4.4.1, se concluye que:

$$T(n) = \max_{1 \leqslant k \leqslant n} \{T_k(n)\} \leqslant R(n).$$

□

Corolario 4.4.3. *La complejidad temporal de la versión 2 del algoritmo TTT es* $\mathcal{O}(3^{n/3})$.

Demostración. Inmediato del teorema 4.4.3 □

Corolario 4.4.4. *La complejidad temporal de la versión 1 del algoritmo TTT es* $\mathcal{O}(n3^{n/3})$.

Demostración. La versión 1 del algoritmo TTT difiere de la versión 2, únicamente en la forma en que reporta los clanes; en la versión 1 cada vez que se encuentra un clan se imprimen todos sus vértices.

Si una gráfica tiene n vértices, entonces cualquier clan Q de la gráfica cumple que $|Q| \leqslant n$. Por lo tanto, el tiempo que le lleva imprimir un clan Q, a la versión 1 del algoritmo TTT es $\mathcal{O}(n)$.

Si el algoritmo TTT no imprime todos los vértices de un clan, por el corolario 4.4.4, sabemos que el algoritmo requiere un tiempo $\mathcal{O}(3^n/3)$ para poder reportar todos los clanes de la gráfica. En consecuencia, la complejidad temporal de la versión 1 del algoritmo TTT es $\mathcal{O}(n3^{n/3})$. □

Corolario 4.4.5. *La complejidad temporal de la versión 3 del algoritmo de Bron-Kerbosch es* $\mathcal{O}(n3^{n/3})$

Demostración. Por el teorema 4.4.1, la versión 3 del algoritmo de Bron-Kerbosch y la versión 1 del algoritmo TTT son esencialmente el mismo algoritmo. El resultado es inmediato del corolario 4.4.4. □

4.5 COMPLEJIDAD TEMPORAL DE FAMILIAS DE GRÁFICAS BÁSICAS

En la sección anterior se demostró que el algoritmo de Bron-Kerbosch tiene complejidad temporal $\mathcal{O}(3^{n/3})$ (cuando usa la estrategia de reportar clanes propuesto por Tomita et al.). Aunque esta cota es válida para cualquier gráfica, nos gustaría saber cual es la complejidad temporal del algoritmo cuando la gráfica de entrada es una trayectoria, un ciclo, una estrella, una gráfica sin aristas o una gráfica completa. También nos gustaría saber cual es la complejidad temporal del algoritmo con la gráfica de Moon-Moser. Como se verá mas adelante, la cota $\mathcal{O}(3^{n/3})$ es mucho mayor que la cota de complejidad de la mayoría de las familias de gráficas que se estudian en esta sección.

observación importante

A menos que se indique lo contrario, *asumiremos en todo momento que estamos trabajando con la versión 2 del algoritmo TTT*. Por los resultados de la sección anterior, dicho algoritmo es equivalente a la versión 3 del algoritmo de Bron-Kerbosch (sin imprimir los clanes). Es por esto que *usaremos el pseudocódigo descrito en 5, como sinónimo del algoritmo de Bron-Kerbosch*.

Por un razonamiento similar al del corolario 4.4.5, basta con multiplicar por n a cada una de las cotas que se presentan en esta sección, para que también sean válidas para el caso en el que se imprimen todos los vértices del clan a reportar.

funciones $T(n)$, $T_k(n)$ *y* $T_\emptyset(n)$

En esta sección seguiremos utilizando las definiciones 4.4.5 para las funciones de complejidad temporal $T(n)$, $T_k(n)$ y $T_\emptyset(n)$. El valor de la constante p_1, se define igual que en la observación 4.4.2. Los conjuntos P y S así como el vértice pivote se definen igual que en 5.

4.5.1 *Complejidad temporal para gráficas completas*

Teorema 4.5.1. *Sea* K_n *una gráfica completa de* n *vértices. Sea* $T(n)$ *la complejidad temporal del algoritmo de Bron-Kerbosch cuando la gráfica de entrada es* K_n. *Para todo entero* $n \geqslant 1$ *se cumple que*

$$T(n) \leqslant p_1 \left(\frac{n(n+1)(2n+1)}{6} \right).$$

Demostración. Sea u el vértice pivote del algoritmo. Como u es adyacente a todos los vértices de la gráfica, tenemos que $P \setminus N(u) = \{u\}$. Por lo tanto, $T(n) = T_1(n)$. La desigualdad del teorema se demostrará por inducción sobre n.

Si $n = 1$, entonces

$$T(1) = T_1(1) = T_\emptyset(1) = p_1.$$

Suponga que la desigualdad del teorema es válida para $n \leqslant N$, mostraremos que la desigualdad también se cumple para $n = N + 1$.

Como $|S| = n$, entonces $|S \cup N(u)| = n - 1$. Por el lema 4.4.4, se tiene que

$$\begin{aligned} T(n) = T(|S|) &\leqslant T_\emptyset(n) + T_1(|S \cup N(u)|) \\ &= p_1 n^2 + T_1(n-1) \\ &= p_1 n^2 + T(n-1). \end{aligned}$$

Por la hipótesis de inducción, se tiene que $T(n-1) \leqslant p_1 \left(\frac{n(n-1)(2n-1)}{6} \right)$, al sustituir este valor en la desigualdad anterior se obtiene que

$$T(n) \leqslant p_1 \left(\frac{2n^2 + 3n + 1}{6} \right) = p_1 \left(\frac{n(n+1)(2n+1)}{6} \right).$$

□

Corolario 4.5.1. *La complejidad temporal del algoritmo de Bron-Kerbosch cuando la gráfica de entrada es una completa* K_n *con* n *vértices es* $\mathcal{O}(n^3)$.

Demostración. Inmediato del teorema 4.5.1. □

4.5.2 *Complejidad temporal para gráficas sin aristas*

Teorema 4.5.2. *Sea $\overline{K_n}$ una gráfica sin aristas con conjunto de vértices V. Sea $T(n)$ la complejidad temporal del algoritmo de Bron-Kerbosch, cuando G es la gráfica de entrada. Para todo entero $n \geqslant 1$ se cumple que*

$$T(n) \leqslant p_1 n^2.$$

Demostración. Como la gráfica G no tiene aristas, cualquier vértice $u \in V$ puede ser seleccionado como pivote. Si v es un vértice del conjunto $P \setminus N(u)$, entonces

$$S_{new} = S \cap N(v) = \emptyset\,,$$
$$P_{new} = P \cap N(v) = \emptyset\,.$$

Por lo tanto, el procedimiento TTT2 se detiene inmediatamente en la siguiente llamada recursiva, lo cual es equivalente al procedimiento $TTT_\emptyset$ y en consecuencia:

$$T(n) = T_\emptyset(n) \leqslant p_1 n^2.$$

□

Corolario 4.5.2. *Sea $\overline{K_n}$ una gráfica sin aristas con n vértices. La complejidad temporal del algoritmo de Bron-Kerbosch con la gráfica $\overline{K_n}$ es $\mathcal{O}(n^2)$.*

Demostración. Inmediato del teorema 4.5.2. □

4.5.3 *Complejidad temporal para gráficas estrella*

Teorema 4.5.3. *Sea S_n una gráfica estrella con n vértices. Sea $T(n)$ la complejidad temporal del algoritmo de Bron-Kerbosch cuando S_n es la gráfica de entrada. Para todo entero $n \geqslant 1$ se cumple que*

$$T(n) \leqslant p_1 \left(2n^2 - 2n + 1\right).$$

Demostración. Por definición, la gráfica S_n tiene un vértice u con grado $|N(u)| = n-1$, el resto de los vértices de la gráfica S_n tienen al vértice u como único vecino. Al iniciar el algoritmo, el vértice u se elige como pivote y en consecuencia $P \setminus N(u) = \{u\}$. Además, al llegar a las líneas 8 y 9 del algoritmo 5, se tiene que

$$P_{new} = P \cap N(u) = V(S_n) \setminus \{u\},$$
$$S_{new} = S \cap N(u) = V(S_n) \setminus \{u\}.$$

Los vértices del conjunto S_{new}, inducen una subgráfica sin vértices y por el teorema 4.5.2, el tiempo de ejecución de TTT2(S_{new}, P_{new}) está acotado por $p_1|S_new| = p_1(n-1)$. Por lo tanto

$$\begin{aligned} T(n) &\leqslant T_\emptyset(n) + p_1(n-1) = p_1\left(n^2 + (n-1)^2\right) \\ &= p_1\left(2n^2 - 2n + 1\right). \end{aligned}$$

□

Corolario 4.5.3. *La complejidad temporal del algoritmo de Bron-Kerbosch cuando la gráfica de entrada es una estrella S_n con n vértices es $\mathcal{O}(n^2)$.*

Demostración. Inmediato del teorema 4.5.3. □

4.5.4 *Complejidad temporal para trayectorias y ciclos*

Teorema 4.5.4. *Sea P_{n-1} una trayectoria con n vértices. Sea $T(n)$ la complejidad temporal del algoritmo de Bron-Kerbosch cuando P_{n-1} es la gráfica de entrada. Para todo entero $n \geqslant 1$ se cumple que*

$$T(n) \leqslant p_1\left(n^2 + 4n - 4\right).$$

Demostración. Es fácil verificar que la desigualdad del teorema es válida para $n = 1$.

Para $n > 1$ sabemos que al inicio, el algoritmo elige un vértice pivote u del conjunto $V(P_{n-1})$. Si v es un vértice en el conjunto $P \setminus N(u)$, por la definición de trayectoria se cumple que

$$|S_{new}| = |S \cap N(v)| \leqslant 2\,,$$
$$|P_{new}| = |P \cap N(v)| \leqslant 2\,.$$

Los vértices en el conjunto S_{new} inducen una subgráfica sin aristas de 1 o 2 vértices. Por el teorema 4.5.2, el tiempo de ejecución de $TTT2(S_{new}, P_{new})$ está acotado por $p_1|S_{new}|^2 \leqslant 4p_1$ y como $|P \setminus N(u)| \leqslant n-1$ el tiempo de ejecución del algoritmo es:

$$\begin{aligned} T(n) &\leqslant T_{\emptyset}(n) + |P \setminus N(u)|(4p_1) \leqslant p_1 n^2 + (n-1)(4p_1) \\ &= p_1(n^2 + 4n - 4). \end{aligned}$$

□

Corolario 4.5.4. *La complejidad temporal del algoritmo de Bron-Kerbosch cuando la gráfica de entrada es una trayectoria P_{n-1} con n vértices es $\mathcal{O}(n^2)$.*

Demostración. Inmediato del teorema 4.5.4. □

Corolario 4.5.5. *Sea C_{n-1} un ciclo con n vértices. Sea $T(n)$ la complejidad temporal del algoritmo de Bron-Kerbosch cuando C_{n-1} es la gráfica de entrada. Para todo entero $n \geqslant 3$ se cumple que*

$$T(n) \leqslant p_1\left(n^2 + 4n - 4\right).$$

Demostración. La prueba es idéntica a la del teorema 4.5.4 □

Corolario 4.5.6. *La complejidad temporal del algoritmo de Bron-Kerbosch cuando la gráfica de entrada es un ciclo C_{n-1} con n vértices es $\mathcal{O}(n^2)$.*

Demostración. Inmediato del corolario 4.5.5. □

4.5.5 *Complejidad temporal para la gráfica de Moon-Moser*

En esta sección mostramos que la gráfica de Moon-Moser alcanza la cota de complejidad $\mathcal{O}(3^{n/3})$ del teorema 4.4.3. Con este fin enunciamos el siguiente lema:

Lema 4.5.1. *La serie infinita* $\sum_{n=1}^{\infty} \frac{n^2}{3^{n/3}}$ *es convergente.*

Demostración. Primero demostraremos la siguiente desigualdad:

$$\frac{n^2}{3^{n/3}} \leqslant \left(\frac{1}{3}\right)^n \qquad \forall n \geqslant 3^3. \tag{6}$$

La desigualdad (6) es equivalente a las siguientes desigualdades

$$\begin{aligned} & n^2 \leqslant 3^{n/3}\left(\frac{1}{3}\right)^n \\ \Leftrightarrow \quad & 2\cdot \ln(n) \leqslant 3\cdot \ln(n/3) + (1/3)\ln(n) \\ \Leftrightarrow \quad & 5\cdot \ln(n) \leqslant 9\cdot \ln(n/3) \\ \Leftrightarrow \quad & n^5 \leqslant \left(\frac{n}{3}\right)^9 \\ \Leftrightarrow \quad & n^5\left(1-\frac{n^4}{3^9}\right) \leqslant 0. \end{aligned}$$

Como la última desigualdad se cumple para $n \geqslant 3^3$, concluimos que la desigualdad (6) es verdadera. Por lo tanto para $N > 3^5$ se tiene

$$\sum_{n=1}^{N} \frac{n^2}{3^{n/3}} \leqslant \sum_{n=3^3+1}^{N} \left(\frac{1}{3}\right)^n + \max_{1\leqslant n\leqslant 3^3}\left\{\frac{n^2}{3^{n/3}}\right\}.$$

Puesto que la serie geométrica $\sum_{n=1}^{\infty}\left(\frac{1}{3}\right)^n$ es convergente, el término derecho de la desigualdad anterior está acotado cuando $N \to \infty$. Por lo tanto, la serie $\sum_{n=1}^{\infty} \frac{n^2}{3^{n/3}}$ es convergente. □

Teorema 4.5.5. *Sea* G *la gráfica de Moon-Moser con* n *vértices. Sea* $M(n)$ *la función sobre los enteros que se define de la siguiente manera*

$$M(n) = \sum_{i=0}^{\frac{n}{3}-1} 3^i(n-3i)^2.$$

Sea $T(n)$ *la complejidad temporal del algoritmo de Bron-Kerbosch, cuando* G *es la gráfica de entrada. Si* $n \equiv 3 \pmod 3$ *y* $n \geqslant 3$ *entonces se cumple la siguiente desigualdad*

$$T(n) \leqslant p_1 M(n).$$

Demostración. La prueba se hará por inducción sobre el número de vértices.

Si $n = 1$, entonces $M(1) = p_1$, por lo tanto

$$T(1) = T\emptyset(1) \leqslant p_1.$$

Suponga que el teorema es verdadero para $n \leqslant N$, se demostrará que también es válido para $n = N + 1$.

Sea $u \in V(G)$ el vértice pivote que se elige al inicio del algoritmo y sea $P \setminus N(u) = \{v_1, v_2, \ldots, v_k\}$. Defina los conjuntos $S_i = S \cup N(v_i)$ para $i = 1, 2, \ldots, k$. Por el lema 4.4.4 inciso *(i)*, se tiene que

$$T_k(n) \leqslant T_\emptyset(n) + \sum_{i=1}^{k} T(|S_i|) \leqslant p_1 n^2 + \sum_{i=1}^{k} T(|S_i|).$$

Por la definición 3.3.1, la gráfica de Moon-Moser tiene $k = 3$, si $n \equiv 0 \pmod 3$ y por el lema 4.4.4 inciso (i), se cumple que $|S_i| \leqslant n - k = n - 3$. Por la hipótesis de inducción y por el hecho de que $M(n)$ es una función creciente, en la desigualdad anterior se tiene que

$$\begin{aligned} T_3(n) &\leqslant p_1 n^2 + \sum_{i=1}^{3} T(|S_i|) \leqslant p_1 n^2 + 3p_1 M(n-3) \\ &= p_1 n^2 + p_1 \sum_{i=0}^{\frac{n-3}{3}-1} 3^{i+1}(n - 3(i+1))^2. \end{aligned}$$

Al hacer el cambio de variable $j = i + 1$ en la desigualdad anterior, obtenemos que

$$\begin{aligned} T_3(n) &\leqslant p_1 n^2 + p_1 \sum_{j=1}^{\frac{n}{3}-1} 3^j (n - 3j)^2 \\ &= p_1 \sum_{j=1}^{\frac{n}{3}-1} 3^j (n - 3j)^2 = p_1 M(n). \end{aligned}$$

Finalmente por la observación 4.4.1, concluimos que

$$T(n) = T_3(n) \leqslant p_1 M(n).$$

□

Teorema 4.5.6. *Sea* G *la gráfica de Moon-Moser con* n *vértices. Si* $n \equiv 0 \pmod 3$, *la complejidad temporal del algoritmo de Bron-Kerbosch, cuando* G *es la gráfica de entrada es* $\mathcal{O}(3^{n/3})$.

Demostración. Sea $T(n)$ la complejidad temporal del algoritmo de Bron-Kerbosch con la gráfica G. Por el teorema 4.5.5, para $n \geqslant 3$ se cumple la siguiente desigualdad:

$$T(n) \leqslant p_1 \sum_{i=1}^{\frac{n}{3}-1} 3^i (n-3i)^2.$$

Haciendo el cambio de variable $k = n - 3i$ en la desigualdad anterior, se tiene que

$$\begin{aligned} T(n) &\leqslant p_1 \sum_{k=3}^{n-3} 3^{\frac{n-k}{3}} k^2 = p_1 3^{n/3} \sum_{k=3}^{n-3} \frac{k^2}{3^{k/3}} \\ &\leqslant p_1 3^{n/3} \sum_{k=1}^{\infty} \frac{k^2}{3^{k/3}} \end{aligned} \tag{7}$$

Por el lema 4.5.1, la serie infinita $\sum_{k=1}^{\infty} \frac{k^2}{3^{k/3}}$ es convergente. Por lo tanto, existe una constante $c \in \mathbb{R}$ tal que

$$\sum_{k=1}^{\infty} \frac{k^2}{3^{k/3}} \leqslant c. \tag{8}$$

Al sustituir la desigualdad (8) en (7), tenemos que

$$T(n) \leqslant p_1 c 3^{n/3}$$

Por lo tanto, $T(n) = \mathcal{O}(3^{n/3})$. □

La cota del teorema 4.5.6 se comprueba en el capítulo 5, por medio de experimentos computacionales.

4.6 CONJETURAS SOBRE LA COMPLEJIDAD TEMPORAL

Resulta claro de los resultados de la sección anterior, que la complejidad temporal del algoritmo de Bron-Kerbosch puede variar drásticamente en función de la gráfica de entrada. Una de las cuestiones que se investigaron en el desarrollo de este trabajo fue el determinar una cota de complejidad temporal en función de las aristas y vértices de una gráfica. Como ya se mencionó, dicha cota es crucial para poder hacer comparaciones mas adecuadas con otros algoritmos.

Durante el desarrollo de este trabajo, se hicieron múltiples conjeturas para tratar de determinar una cota de complejidad en función de vértices y aristas. La conjetura más general que tenemos para la complejidad temporal parametrizada del algoritmo de Bron-Kerbosch, se enuncia a continuación. Aunque no se pudo demostrar dicha cota de complejidad, si se restringe nuestra cota en función de vértices y aristas, los resultados de los experimentos computacionales del capítulo 5, parecen respaldar estos resultados.

Conjetura 4.6.1. *Sea* G *una gráfica con* n *vértices,* n *aristas y* μ *clanes. La complejidad temporal del algoritmo TTT (versión 2) es*

$$\mathcal{O}(n^2 + nm + \mu).$$

Observación 4.6.1. *Para obtener una cota para el algoritmo de Bron-Kerbosch (versión 3) basta con multiplicar por* n*, a la cota de complejidad de la conjetura 4.6.1.*

Una evidencia que tenemos para suponer que la conjetura 4.6.1 es correcta, es que dicha conjetura permite obtener las cotas de complejidad que se demostraron en las secciones anteriores (gráficas completas, sin aristas, ciclos, trayectorias y de Moon-Moser). En particular, observe que también es posible obtener la cota de complejidad del corolario 4.4.5, para el algoritmo de Bron-Kerbosch (versión 3).

De ser cierta la conjetura 4.6.1, sería posible derivar una cota de complejidad que esté en función de el número de vértices n y el número de aristas m. Para esto es necesario tener una formula para el número máximo de clanes que puede tener una gráfica en función de n y m, esto nos lleva a enunciar la siguiente conjetura.

Conjetura 4.6.2. *Sea* $\mu(n, m)$ *el número máximo de clanes que puede tener una gráfica con* n *vértices y* m *aristas, entonces*

$$\mu(n, m) \leqslant \begin{cases} n + c_1 3^{r_1/3} & \text{Si } 0 \leqslant m < \frac{n(n-3)}{2} \\ n + c_2 3^{(n-r_2)/3} & \text{Si } m \geqslant \frac{n(n-3)}{2} \end{cases}$$

donde c_1 *y* c_2 *son constantes positivas y* r_1 *y* r_2 *son enteros que se definen de la siguiente manera:*

$$r_1 = \text{máx}\left\{ k \in \mathbb{N} \;\middle|\; \frac{k(k-3)}{2} \leqslant m \right\},$$
$$r_2 = m - \frac{n(n-3)}{2}.$$

La idea intuitiva que nos llevó a proponer la conjetura 4.6.2, para el número máximo de clanes, se basa en las siguientes observaciones.

Observación 4.6.2.

(i) *La gráfica de Moon-Moser de n vértices tiene un número de clanes que está en el orden $\mathcal{O}(3^{n/3})$. Por los teoremas 3.3.1 y 3.3.2, este es el mayor número de clanes que puede tener una gráfica de n vértices.*

(ii) *Es posible que una situación similar a la del inciso (i), ocurra para una gráfica G con n vértices y m aristas. En este caso, tenemos la conjetura de que el mayor número de clanes en la gráfica G, ocurre con una gráfica que resulta de usar las m aristas para construir una subgráfica M que es "similar" a la gráfica de Moon-Moser. Tenemos los siguientes casos:*

- *Si $m < \frac{n(n-3)}{2}$, entonces el número de aristas de G, es* menor *que el número máximo de aristas que puede tener una gráfica de Moon-Moser con n vértices. Por lo tanto, aproximamos el número de vértices de la subgráfica M con el entero r_1. El total de clanes de la subgráfica M, está acotado por $c_1 3^{r_1/3}$ y puede haber a lo más $n - r_1 \leqslant n$ vértices aislados.*
- *Si $m \geqslant \frac{n(n-3)}{2}$, entonces el número de aristas de G, es* mayor *que el número máximo de aristas que puede tener una gráfica de Moon-Moser con n vértices. Por lo tanto, aproximamos el número de vértices de la subgráfica M con $n - r_2$. El total de clanes de la subgráfica M, esta acotado por $c_1 3^{(n-r_2)/3}$ y puede haber a lo más $r_2 \leqslant n$ subgráficas completas.*

Al usar la conjetura 4.6.2, en la conjetura 4.6.1, obtenemos las siguientes conjeturas para la complejidad temporal en función de vértices y aristas.

Conjetura 4.6.3. *Sea G una gráfica con n vértices y m aristas. Sea $T(n, m)$ la complejidad temporal del algoritmo de Bron-Kerbosch, cuando la gráfica de entrada es G. Sea p_1 la constante que se definió en la observación 4.4.2. Sea $C \geqslant p_1$, una constante en los reales y sea $Q_1(n)$ una función de orden polinomial.*

Para $0 \leqslant m < \frac{n(n-3)}{2}$ y para todo $n \geqslant 3$ se cumple que

$$T(n, m) \leqslant C\left(n^2 + nm + 3^{(r_1/3)}\right) + Q_1(n),$$

donde r_1 es un entero que se define de la siguiente manera:

$$r_1 = \max\left\{k \in \mathbb{N} \;\middle|\; \frac{k(k-3)}{2} \leqslant m\right\}.$$

El determinar el valor específico de la función polinomial $Q_1(n)$, es uno de los problemas que impiden probar formalmente la conjetura 4.6.3. Es-

ta función tiene el objetivo de facilitar el paso inductivo, de una manera similar como se hace en el teorema 4.4.3.

Si se observa con cuidado la demostración del teorema 4.4.3, se notará que es crucial la condición $|S_i| \leqslant n - k$. Para la demostración de la conjetura 4.6.3, es necesario encontrar una condición similar que dependa de n, m y k.

Conjetura 4.6.4. *Sea* G *una gráfica con* n *vértices y* m *aristas. Sea* $T(n, m)$ *la complejidad temporal del algoritmo de Bron-Kerbosch, cuando la gráfica de entrada es* G. *Sea* p_1 *la constante que se definió en la observación 4.4.2. Sea* $C \geqslant p_1$, *una constante en los reales y sea* $Q_2(n)$ *una función de orden polinomial.*

Para $m \geqslant \frac{n(n-3)}{2}$ *y para todo* $n \geqslant 3$ *se cumple que*

$$T(n, m) \leqslant C\left(nm + 3^{(n-r_2)/3}\right) + Q_2(n),$$

donde r_2 *es un entero que se define de la siguiente manera:*

$$r_2 = m - \frac{n(n-3)}{2}.$$

La función $Q_2(n)$ tiene el objetivo de facilitar el paso inductivo de una manera similar como se hizo en el teorema 4.4.3. Las razones por las que no hemos podido demostrar la conjetura 4.6.4, son similares a las razones por las que no hemos podido demostrar la conjetura 4.6.3.

Observe que con $m = \frac{n(n-3)}{2}$ aristas, es posible formar una gráfica de Moon-Moser. Es interesante notar que para este número de aristas, la cota de la conjetura 4.6.4 está en $\mathcal{O}(3^{n/3})$, este resultado coincide con el corolario 4.4.4. Si $m = \frac{n(n-1)}{2}$, entonces se tiene una gráfica completa y para este número de aristas, la conjetura 4.6.4 está en $\mathcal{O}(n^3)$, esto coincide con el corolario 4.5.1.

5

EXPERIMENTOS COMPUTACIONALES

En esta sección presentamos varios resultados de experimentos computacionales, que confirman los teoremas y corolarios sobre complejidad temporal para las familias de gráficas que se discutieron en el capítulo anterior. Al final de esta sección, mostramos resultados experimentales que sirven de evidencia para las conjeturas 4.6.3 y 4.6.4.

Puesto que las cotas de complejidad son modelos teóricos del mundo real, es importante tomar en cuenta los criterios de falsabilidad propuestos por Karl Popper[1]; a fin de considerar que cierta teoría tiene validez, es necesario tratar de mostrar que es falsa mediante evidencia experimental, si estos experimentos no pueden demostrar que la teoría es falsa, se puede tener cierto grado de confianza en la teoría. Los experimentos de este capitulo por una parte sirven como evidencia de que las cotas de complejidad desarrolladas son correctas y permiten visualizar cuan parecido es el modelo teórico, con el comportamiento del algoritmo en un entorno real.

Para realizar las mediciones de tiempo de cada uno de los experimentos de esta sección, se han tomado en cuenta las recomendaciones que se describen en [22]. La implementación de todos los experimentos que se muestran hasta antes de la sección 5.7, se realizaron siguiendo la siguiente estrategia (para cada familia de gráficas):

1. Para $n = 10, 20, 30, \ldots, 70$ generar una gráfica G de n vértices.
2. Realizar 100 invocaciones del procedimiento $\mathrm{TTT2}(V(G), V(G))$. Por cada invocación tomar el tiempo que le lleva al procedimiento TTT2 terminar con la gráfica G. El tiempo final reportado para n, es el tiempo promedio de todas las invocaciones.

En cada una de las gráficas que presentamos en las siguientes secciones, también se incluye una estimación de la función de complejidad temporal $T(n)$, por medio de una regresión con mínimos cuadrados. Para ello considere los conjuntos $N = \{n_1, n_2, \ldots, n_k\}$ de enteros positivos y sea $T = \{T(n_1), T(n_2), \ldots, T(n_k)\}$ los tiempos $T(n)$ que se obtuvieron en el

[1] Filosofo y teórico de la Ciencia Austro-Británico (1902-1994). Propuso la teoría de Falsacionismo como criterio de demarcación para la Ciencia.

experimento, para $n = n_1, n_2, \ldots, n_k$. Los modelos que utilizamos para las estimaciones son los siguientes [1] :

1. Regresión exponencial por mínimos cuadrados: *Regresión exponencial*

$$T(n) = Ae^{Bn}.$$

Las constantes A y B, se calculan de acuerdo a las siguientes formulas:

$$a = \frac{\sum_{i=1}^{k} \ln(T(n_i)) \sum_{i=1}^{k} (n_i)^2 - \sum_{i=1}^{k} n_i \sum_{i=1}^{k} n_i \ln(T(n_i))}{n \sum_{i=1}^{k} (n_i)^2 - \left(\sum_{i=1}^{k} n_i\right)^2}$$

$$B = \frac{n \sum_{i=1}^{k} n_i \ln(T(n_i)) - \sum_{i=1}^{k} n_i \sum_{i=1}^{k} \ln(T(n_i))}{n \sum_{i=1}^{k} (n_i)^2 - \left(\sum_{i=1}^{k} n_i\right)^2}$$

$$A = e^a$$

2. Regresión logarítmica por mínimos cuadrados: *Regresión logarítmica*

$$\ln(T(n)) = A + B\ln(n).$$

La expresión anterior es equivalente a

$$T(n) = A_0 n^B.$$

Las constantes A, A_0 y B, se calculan de acuerdo a las siguientes formulas:

$$B = \frac{n \sum_{i=1}^{k} (T(n_i)\ln(n_i)) - \sum_{i=1}^{k} T(n_i) \sum_{i=1}^{k} \ln(n_i)}{n \sum_{i=1}^{k} (\ln(n_i))^2 - \left(\sum_{i=1}^{k} \ln(n_i)\right)^2}$$

$$A = \frac{\sum_{i=1}^{k} T(n_i) - b \sum_{i=1}^{k} \ln(T(n_i))}{n}$$

$$A_0 = e^A.$$

La razón por la cual utilizamos estos 2 tipos de regresión lineal, se debe a que en algunos casos los tiempos reportados por los experimentos tienen complejidad exponencial y en otros casos tienen complejidad polinomial.

[1] Para más información sobre métodos de regresión y sobre el método de mínimos cuadrados puede consultar [32].

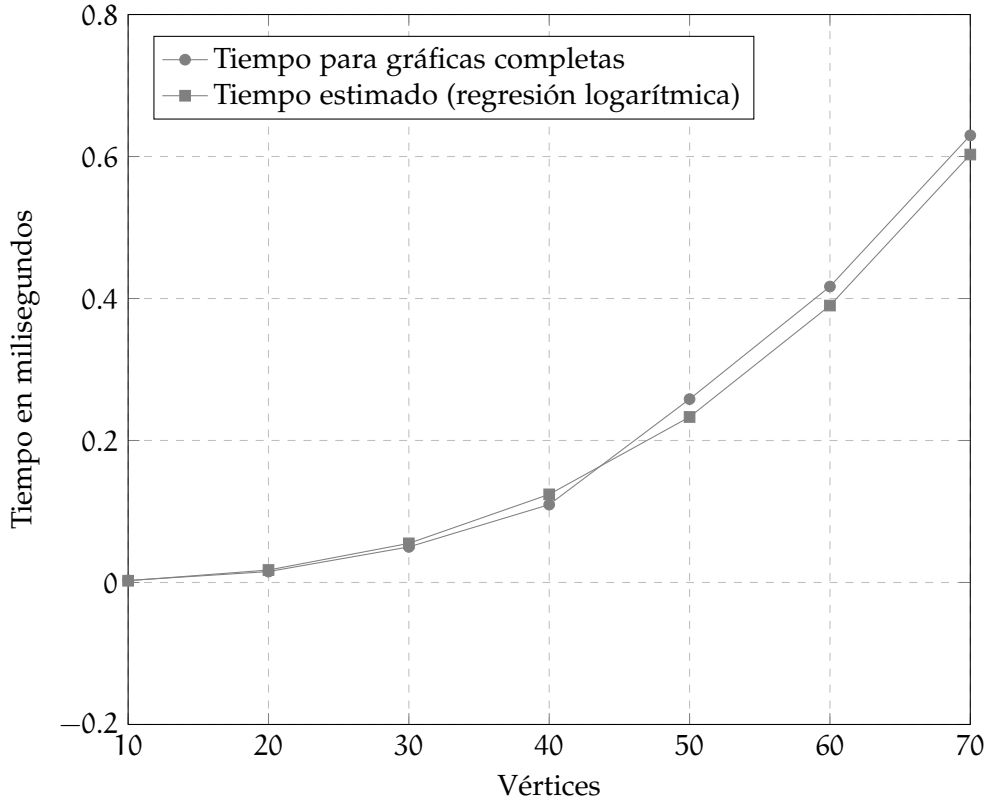
Coeficientes de $A_0 n^B$ para gráficas completas	
A_0	0.00000371927727239 5287
B	2.8235443825782625

Tabla 1: Coeficientes de la regresión logarítmica para la figura 15 (gráficas completas).

5.1 EXPERIMENTOS PARA GRÁFICAS COMPLETAS

La figura 15 confirma lo que se demostró en el corolario 4.5.1; la complejidad temporal de una gráfica completa es $\mathcal{O}(n^3)$. Como era de esperarse, el valor que se reporta en la tabla 1, para el coeficiente B de la regresión $T(n) = A_0 n^B$, es muy cercano a 3.

Figura 15: Resultados del experimento y de la regresión logarítmica para gráficas completas.



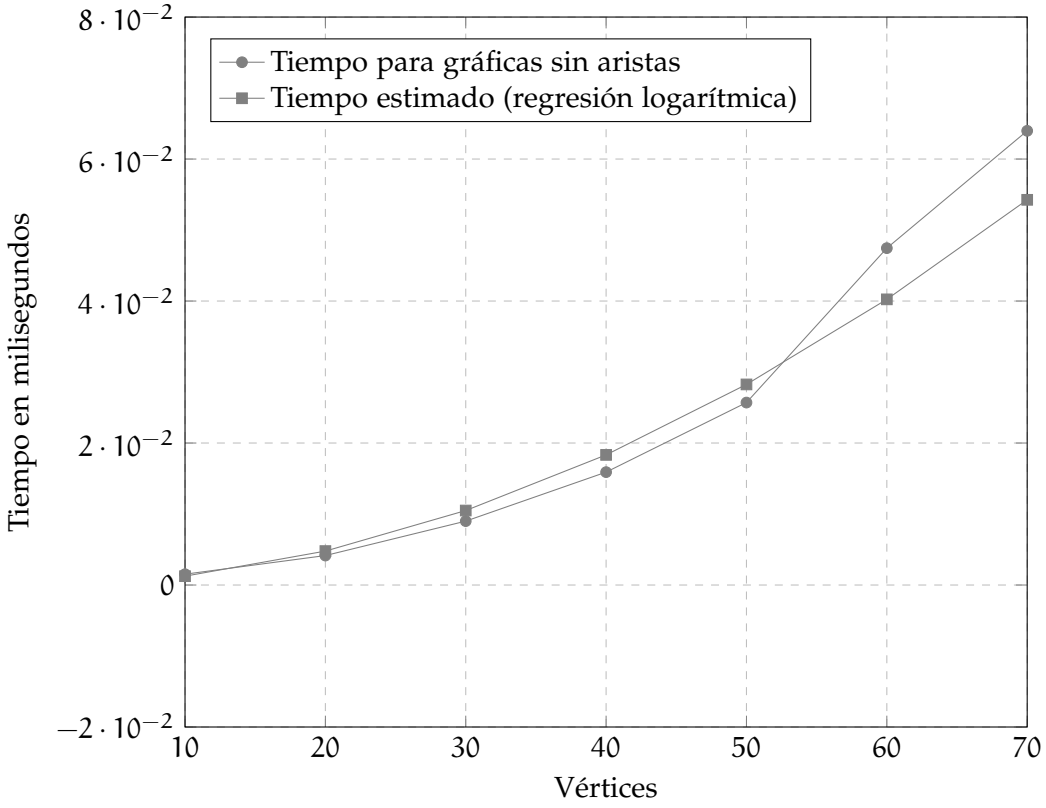
Coeficientes de $A_0 n^B$ para gráficas sin aristas	
A_0	0.000014332191338383747
B	1.939274350420881

Tabla 2: Coeficientes de la regresión logarítmica para la figura 16 (gráficas sin aristas).

5.2 EXPERIMENTOS PARA GRÁFICAS SIN ARISTAS

La figura 16 confirma lo que se demostró en el corolario 4.5.2; la complejidad temporal de una gráfica sin aristas es $\mathcal{O}(n^2)$. Como era de esperarse, el valor que se reporta en la tabla 2, para el coeficiente B de la regresión $T(n) = A_0 n^B$, es muy cercano a 2.

Figura 16: Resultados del experimento y de la regresión logarítmica para gráficas sin aristas.



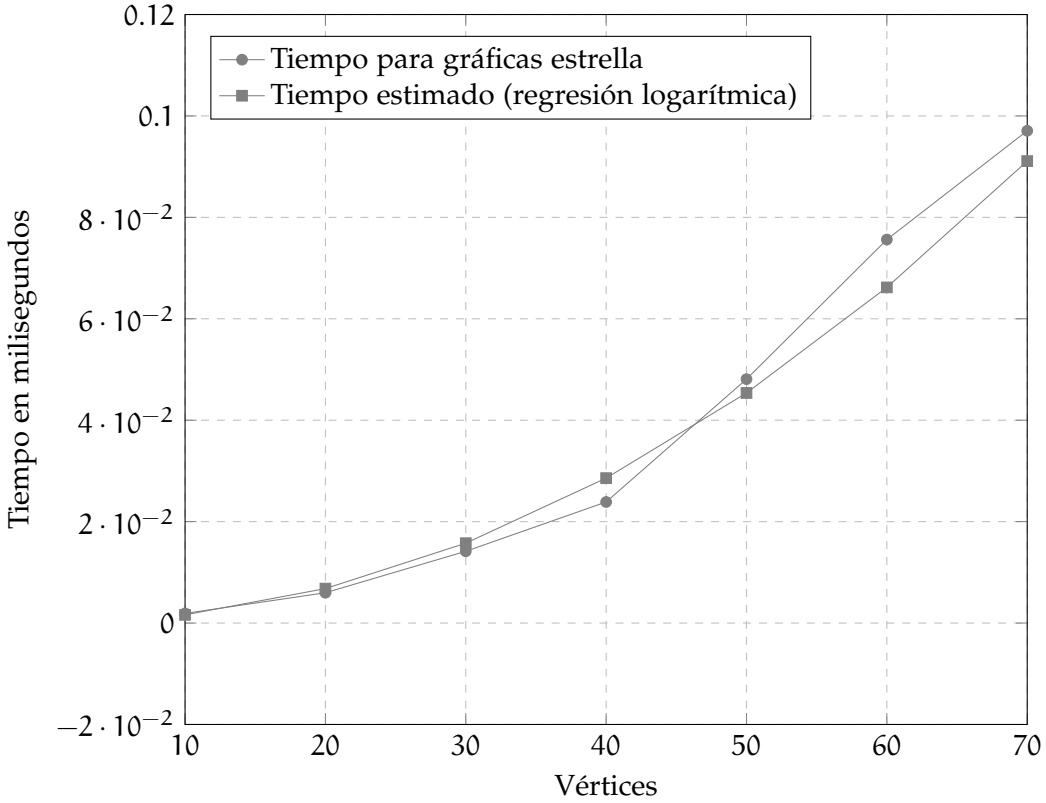
Coeficientes de $A_0 n^B$ para gráficas estrella	
A_0	0.00001372210754810494 8
B	2.0714699699939985

Tabla 3: Coeficientes de la regresión logarítmica para la figura 17 (gráficas estrella).

5.3 EXPERIMENTOS PARA GRÁFICAS ESTRELLA

La figura 17 confirma lo que se demostró en el corolario 4.5.3; la complejidad temporal de una gráfica estrella es $\mathcal{O}(n^2)$. Como era de esperarse, el valor que se reporta en la tabla 3, para el coeficiente B de la regresión $T(n) = A_0 n^B$, es muy cercano a 2.

Figura 17: Resultados del experimento y de la regresión logarítmica para gráficas estrella.



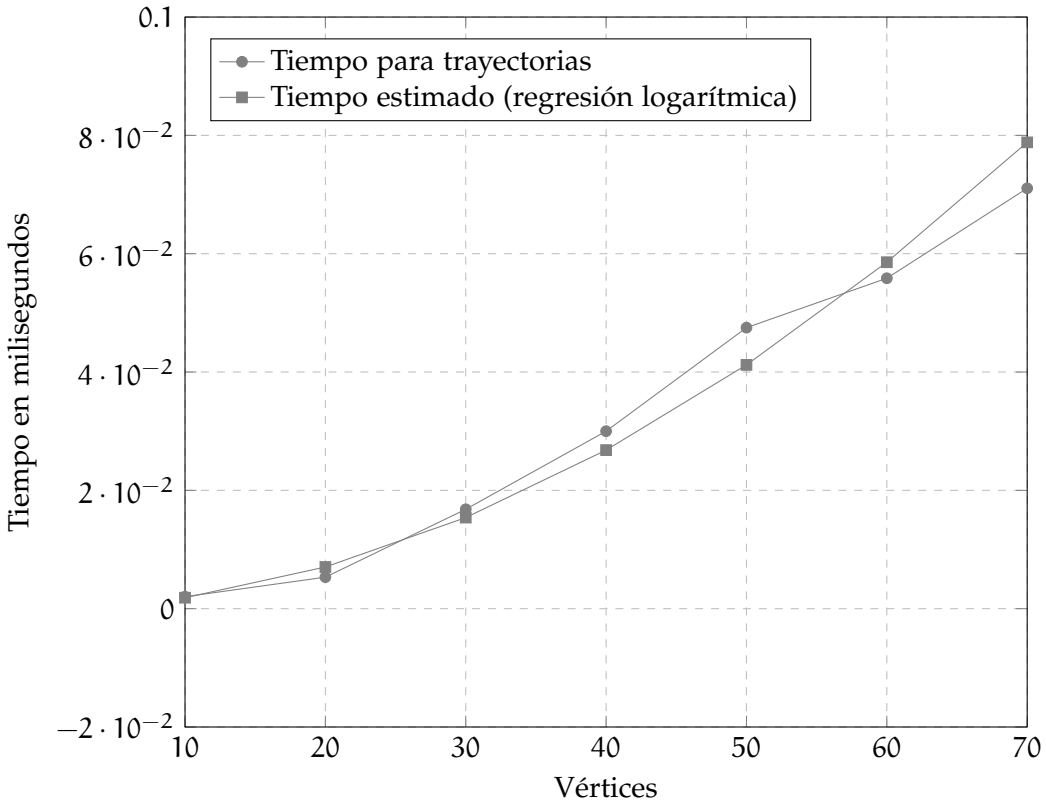
Coeficientes de $A_0 n^B$ para trayectorias	
A_0	0.000021892796774310257
B	1.9274451028491058

Tabla 4: Coeficientes de la regresión logarítmica para la figura 18 (trayectorias).

5.4 EXPERIMENTOS PARA TRAYECTORIAS

La figura 18 confirma lo que se demostró en el corolario 4.5.4; la complejidad temporal de una trayectoria es $\mathcal{O}(n^2)$. Como era de esperarse, el valor que se reporta en la tabla 4, para el coeficiente B de la regresión $T(n) = A_0 n^B$, es muy cercano a 2.

Figura 18: Resultados del experimento y de la regresión logarítmica para trayectorias.



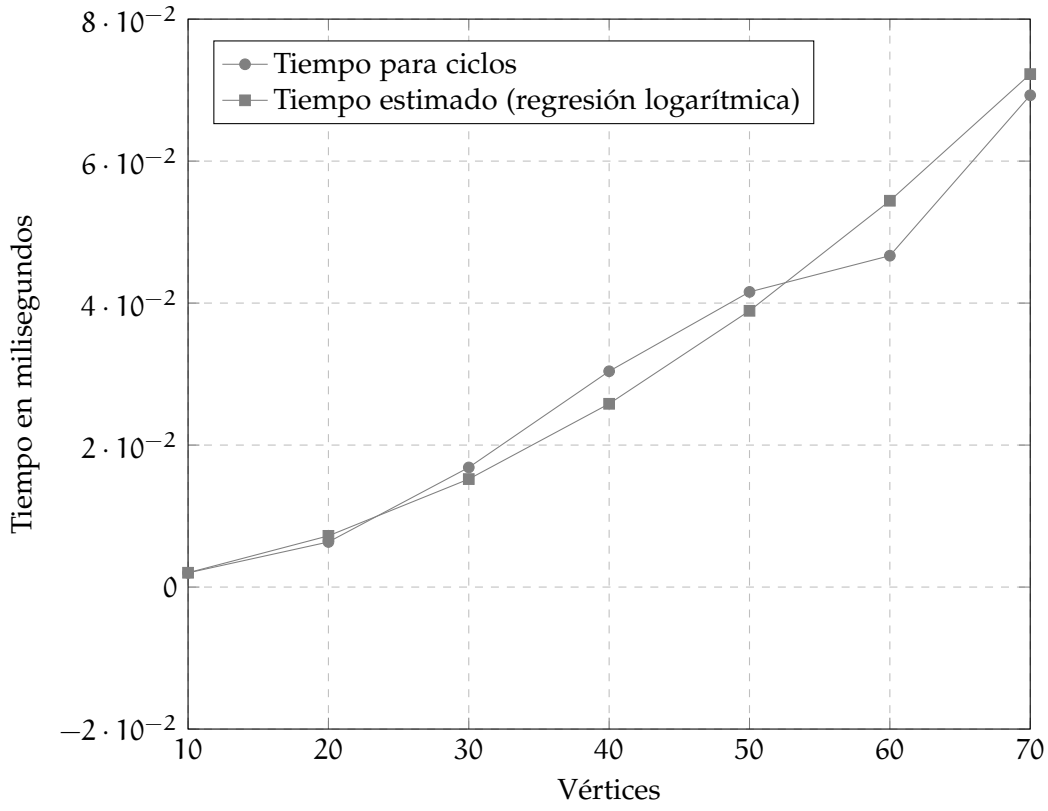
Coeficientes de $A_0 n^B$ para ciclos	
A_0	0.00002922500456268621
B	1.8389736937097538

Tabla 5: Coeficientes de la regresión logarítmica para la figura 19 (ciclos).

5.5 EXPERIMENTOS PARA CICLOS

La figura 19 confirma lo que se demostró en el corolario 4.5.6; la complejidad temporal de un ciclo es $\mathcal{O}(n^2)$. Como era de esperarse, el valor que se reporta en la tabla 5, para el coeficiente B de la regresión $T(n) = A_0 n^B$, es muy cercano a 2.

Figura 19: Resultados del experimento y de la regresión logarítmica para ciclos.



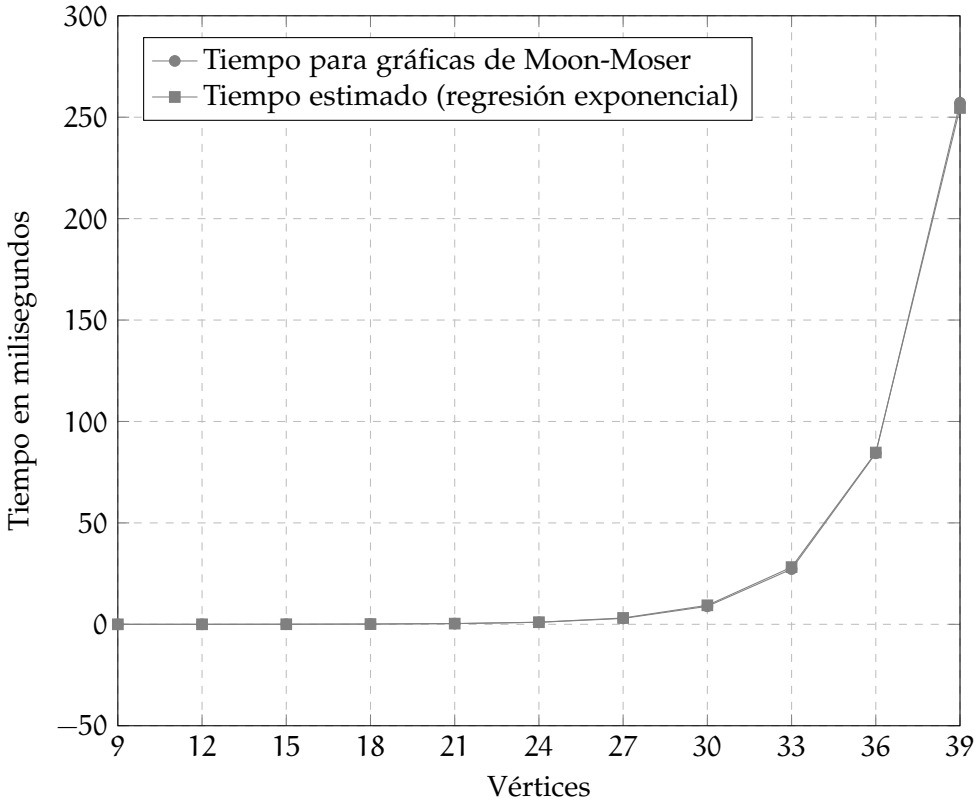
Coeficientes de Ae^{Bn} para gráficas de Moon-Moser	
A	0.00015571262267099 81
B	0.36684745659223744
e^B	1.44317775507

Tabla 6: Coeficientes de la regresión exponencial para la figura 20 (gráfica de Moon-Moser).

5.6 EXPERIMENTOS PARA GRÁFICAS DE MOON-MOSER

La figura 20 confirma lo que se demostró en el teorema 4.5.6; ja complejidad temporal de la gráfica de Moon-Moser con $n \equiv 0 \pmod 3$, es $O(3^{n/3})$.

Figura 20: Resultados del experimento y de la regresión exponencial para gráficas de Moon-Moser.



Observe que el coeficiente B en la tabla 6, tiene la peculiaridad de que $e^B = 1.446$, observe también que $3^{(1/3)} \approx 1.446$. Por lo tanto, la regresión exponencial nos dice que un modelo adecuado para el tiempo del experimento 20, es $T(n) = A \cdot 3^{n/3}$. Este es un resultado que era de esperarse, ya que por el corolario 4.4.5 y por el teorema 4.5.6, el peor tiempo que pue-

de tener el algoritmo de Bron-Kerbosch es $\mathcal{O}(3^{n/3})$ y dicho tiempo ocurre cuando la gráfica de entrada es una gráfica de Moon-Moser.

5.7 EXPERIMENTOS PARA GRÁFICAS CON n VÉRTICES Y m ARISTAS

5.7.1 *Número de clanes en función de vértices y aristas*

En esta sección presentamos datos experimentales que sirven de evidencia para la conjetura 4.6.2. Para este experimento, usamos la lista de gráficas que se ofrecen en la página de Brendan McKay [23]. Dicha lista contiene *todas* las gráficas (salvo isomorfismos) con número de vértices $n \leqslant 10$. Para realizar el experimento, nuestro algoritmo previamente guarda todas las gráficas con $n \leqslant 10$ vértices en una lista L y usa un arreglo R bidimensional, en el que la entrada $R[n][m]$, contiene el número de clanes máximo que tiene una gráfica de n vértices y m aristas. Nuestro algoritmo que realiza el experimento usa las siguientes reglas:

- Para cada gráfica $G \in L$, calcular el número de vértices n, el número de aristas m y el número de clanes μ.
- Si $\mu > R[n][m]$, hacer $R[n][m] = \mu$

observación importante

En la tabla 7 se muestra los valores del arreglo R para $n = 8, 9, 10$. La columna μ', es el *piso* de la cota superior de la conjetura 4.6.2. Para $n = 10$, se han omitido algunos renglones en la tabla 7 a fin de mostrar el mayor rango posible de resultados, para el número de aristas.

5.7.2 *Complejidad temporal en función de vértices y aristas*

conjetura sobre Bron-Kerbosch

En la sección 4.6, enunciamos una conjetura sobre la forma que tiene la cota de complejidad temporal de la versión 2 del algoritmo TTT o equivalentemente; la versión 3 del algoritmo de Bron-Kerbosch sin imprimir los clanes. Recuerde que cuando la cota está en función de n vértices y m aristas, la conjetura 4.6.3 indica que la cota de complejidad temporal para $m \leqslant \frac{n(n-3)}{2}$ es de la forma:

$$T(n, m) \leqslant C\left(n^2 + nm + 3^{(r_1/3)}\right) + Q_1(n),$$
$$\text{donde} \quad r_1 = \max\left\{k \in \mathbb{N} \;\middle|\; \frac{k(k-3)}{2} \leqslant m\right\}.$$

n	m	μ	μ′
8	0	8	14
8	1	7	14
8	2	7	16
8	3	7	16
8	4	8	16
8	5	8	20
8	6	9	20
8	7	9	20
8	8	10	20
8	9	11	26
8	10	11	26
8	11	12	26
8	12	13	26
8	13	13	26
8	14	14	33
8	15	15	33
8	16	16	33
8	17	13	33
8	18	14	33
8	19	15	33
8	20	16	45
8	21	18	33
8	22	12	26
8	23	12	20
8	24	16	16
8	25	8	14
8	26	4	12
8	27	2	10
8	28	1	10
9	0	9	15
9	1	8	15
9	2	8	17
9	3	8	17
9	4	9	17
9	5	9	21
9	6	10	21
9	7	10	21
9	8	11	21
9	9	12	27
9	10	12	27
9	11	13	27
9	12	14	27
9	13	14	27
9	14	15	34
9	15	16	34
9	16	17	34
9	17	17	34
9	18	18	34
9	19	19	34
9	20	20	46
9	21	19	46
9	22	19	46
9	23	20	46
9	24	21	46
9	25	22	46
9	26	24	46
9	27	27	63
9	28	20	46
9	29	20	34
9	30	24	27
9	31	16	21
9	32	16	17
9	33	8	15
9	34	4	13
9	35	2	11
9	36	1	11
10	0	10	16
10	1	9	16
10	1	9	16
10	2	9	18
10	3	9	18
10	4	10	18
10	5	10	22
10	6	11	22
10	7	11	22
10	8	12	22
10	9	13	28
10	10	13	28
10	11	14	28
10	12	15	28
10	13	15	28
10	14	16	35
10	15	17	35
10	16	18	35
10	17	18	35
10	18	19	35
10	19	20	35
10	20	21	47
10	21	21	47
10	25	25	47
10	26	25	47
10	28	28	64
10	29	29	64
10	30	30	64
10	31	31	64
10	32	33	64
10	33	36	64
10	35	30	87
10	36	32	64
10	39	24	28
10	44	2	12.
10	45	1	12.

Tabla 7: Valores de μ y μ′ para n vértices y m aristas.

La constante C y la función $Q_1(n)$ se definieron en 4.6.3. Lo importante a notar de la conjetura 4.6.3, es que si m es una constante, entonces r_1 también es una constante. Por lo tanto, la complejidad temporal en este caso, debería de ser algo muy cercano a $\mathcal{O}(n^2)$. Los experimentos de esta sección, confirman que efectivamente dicha condición se cumple.

procedimiento para experimentos

El procedimiento 6, describe el código que se utilizó para realizar los experimentos de esta sección. Los procedimientos

$$\texttt{ConstruirGraficaAleatoria y}$$
$$\texttt{ConstruirGraficaMoonMoser,}$$

que se listan en las líneas 10 y 11 del algoritmo 6, construyen gráficas aleatorias y de Moon-Moser con n vértices y m aristas. Omitimos la implementación de dichos procedimientos por se demasiado extensa.

El procedimiento TIME() en las línea 16 y 18 del algoritmo 6, regresa el tiempo actual. El procedimiento ARRAY(k) en la línea 3 del algoritmo 6, construye un arreglo de tamaño k. Este arreglo se utiliza para guardar el tiempo máximo que se obtuvo en el experimento, para n vértices y m aristas.

procedimiento $\text{TIEMPO}_{n,m}$

El procedimiento $\text{TIEMPO}_{n,m}(N, M)$ en el algoritmo 6, recibe arreglos de enteros N y M tales que $k = |N| = |M|$. Para cada entero $i = 1, 2, \ldots, k$, el procedimiento asigna $n = N[i]$ y $m = M[i]$ y a continuación construye 100 gráficas de n vértices y m aristas. Por cada una de estas 100 gráficas, el procedimiento revisa con cual de ellas se alcanza el tiempo máximo; para ello, por cada gráfica, el procedimiento realiza 100 llamadas al algoritmo de Bron-Kerbosch y obtiene el promedio del tiempo de ejecución.

Observe que en el procedimiento $\text{TIEMPO}_{n,m}$, para cada n y m, la gráfica número 100 es la gráfica de Moon-Moser con n vértices y m aristas. Esto se debe a que tenemos la conjetura de que dicha gráfica, es la que tiene el mayor tiempo de ejecución (conjeturas 4.6.2 y 4.6.1).

La figura 21 muestra el resultado de ejecutar el procedimiento 6 con gráficas en las que el total de aristas siempre es $m = 27$ (constante) y el total de vértices n es un múltiplo de 3. Observe que el total de aristas *es mucho menor* que el número de aristas que podría tener la gráfica G, si fuera una gráfica completa.

Es muy importante notar que el valor de la constante B en la tabla 8, es casi $B = 2$. Por lo tanto, los experimentos con un número de aristas constante que se muestran en la figura 21, están en $\mathcal{O}(n^2)$. Este comportamiento es el que se espera, en caso de que la conjetura 4.6.3 sea verdadera.

Algoritmo 6: Procedimiento que realiza experimentos para medir la complejidad temporal de gráficas con n vértices y m aristas.

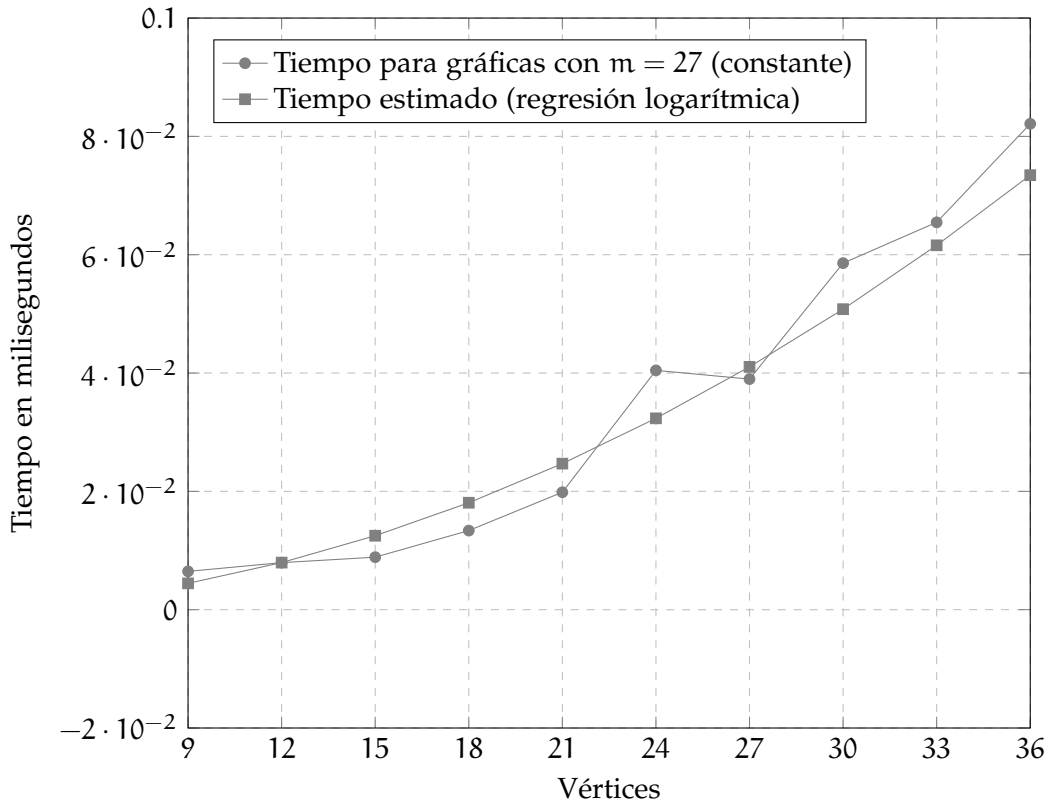
Sea $N = \{n_1, n_2, \ldots, n_k\}$ un conjunto de enteros.
Sea $M = \{m_1, m_2, \ldots, n_k\}$ un conjunto de enteros tales que

$$0 \leqslant m_i \leqslant \frac{n_i(n_i - 1)}{2} \qquad \forall i = 1, \ldots, k.$$

llamada inicial: TIEMPO$_{n,m}$(N,M)

```
1: procedure TIEMPO_{n,m}(N,M)
2:     k = |N|
3:     T = ARRAY(k)
4:     for i ← 1 to k do
5:         n = N[i]
6:         m = M[i]
7:         t_máximo = 0
8:         for j ← 1 to 100 do
9:             if j ⩽ 99 then
10:                G = ConstruirGraficaAleatoria(n, m)
11:            else
12:                G = ConstruirGraficaMoonMoser(n, m)
13:            end if
14:            t_promedio = 0
15:            for l ← 1 to 100 do
16:                t_inicial = TIME()
17:                TTT2(V(G), V(G))
18:                t_final = TIME()
19:                t_promedio = t_promedio + (t_final − t_inicial)
20:            end for
21:            t_promedio = t_promedio/100
22:            if t_máximo < t_promedio then
23:                t_máximo = t_promedio
24:            end if
25:        end for
26:        T[i] = t_máximo
27:    end for
28:    return T
29: end procedure
```

Figura 21: Resultados del experimento y de la regresión logarítmica para gráficas con $n \geqslant 9$ y $m = 27$ (constante).



Sean M_1 y N_1 arreglos que se le pasan al procedimiento $\text{TIEMPO}_{n,m}$, cuyos valores son los siguientes:

$$M_1 = [27,\ 44,\ 70,\ 125,\ 169,\ 232,\ 304,\ 385,\ 445],$$
$$N_1 = [\ 9,\ 12,\ 15,\ \ 18,\ \ 21,\ \ 24,\ \ \ 27,\ \ \ 30,\ \ \ 33].$$

Si para cada entero $n \in N_1$, calculamos el número $\frac{n(n-3)}{2}$ y dicho valor se guarda en un arreglo M_1', entonces los valores de M_1' son los siguientes:

$$M_1' = [27,\ 54,\ 90,\ 135,\ 189,\ 252,\ 324,\ 405,\ 495].$$

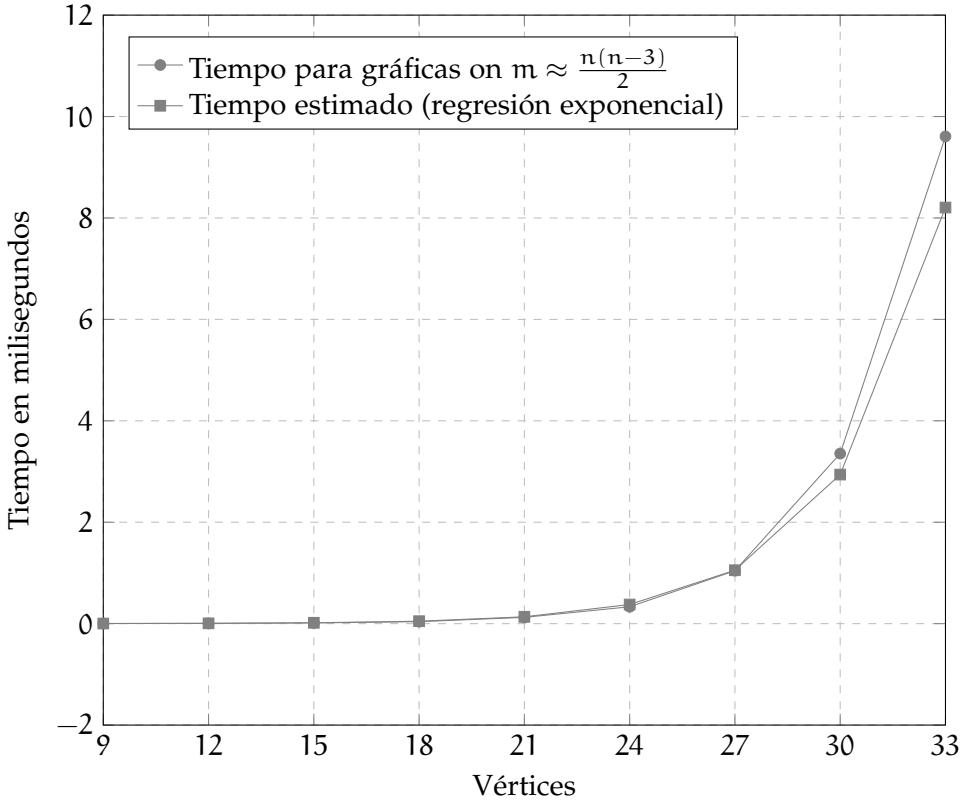
Observe que cada número en la i-ésima entrada del arreglo M_1, es un valor cercano al número que contiene la i-ésima entrada del arreglo M_1'.

Coeficientes de gráficas con $m = 27$ (constante)	
A_0	0.00005226426908717166
B	2.022562573150225

Tabla 8: Coeficientes para la regresión logarítmica de la figura 21 (gráficas con $n \geqslant 9$ y $m = 27$).

Los conjuntos M_1 y N_1 se utilizaron para evaluar el comportamiento de la conjetura 4.6.3 cuando la gráfica G tiene aristas $m \approx \frac{n(n-3)}{2}$, en este caso se tiene que $r_1 \approx n$ (r_1 es el entero que se definió en 4.6.3). De ser cierta la conjetura 4.6.3, la complejidad temporal debe estar en $\mathcal{O}(3^{n/3})$ y en consecuencia, una regresión exponencial de la forma Ae^{Bn}, debe tener B cercano a $3^{1/3} \approx 1.44$. La tabla 9 y la figura 22, muestran que esta condición se cumple para los experimentos realizados por el procedimiento $\text{TIEMPO}_{n,m}$, cuando recibe como entrada los arreglos M_1 y N_1.

Figura 22: Resultados del experimento y de la regresión exponencial para gráficas con $m \approx \frac{n(n-3)}{2}$, $M_1 = [27, 44, 70, 125, 169, 232, 304, 385, 445]$ y $N_1 = [9, 12, 15, 18, 21, 24, 27, 30, 33]$.



Si $n \geqslant \frac{n(n-3)}{2}$, la conjetura 4.6.4 afirma que el tiempo $T(n, m)$ esta acotado por la desigualdad

$$T(n, m) \leqslant C\left(nm + 3^{(n-r_2)/3}\right) + Q_2(n),$$
$$\text{donde} \quad r_2 = m - \frac{n(n-3)}{2}.$$

La constante C y la función $Q_2(n)$ se definieron en 4.6.4. Lo importante a notar de la conjetura 4.6.4, es que si el número de aristas se acerca a $\frac{n(n-1)}{2}$, entonces $r_2 \approx n$ y en consecuencia, la complejidad temporal debería estar

Coeficientes de Ae^{Bn} para gráficas con $n \approx \frac{n(n-3)}{2}$	
A	0.00010247734746275625
B	0.3421424036041071
e^B	1.40796078193

Tabla 9: Coeficientes de la regresión exponencial para la figura 22 (gráficas con $m \approx \frac{n(n-3)}{2}$).

Coeficientes de $A_0 n^B$ para gráficas con $m > \frac{n(n-3)}{2}$	
A_0	0.000006765287174738636
B	2.7727315462394557

Tabla 10: Coeficientes de la regresión logarítmica para la figura 23 (gráficas con $m > \frac{n(n-3)}{2}$).

en $\mathcal{O}(n^3)$. Para probar experimentalmente este caso, sean M_2 y N_2 arreglos cuyos valores son los siguientes:

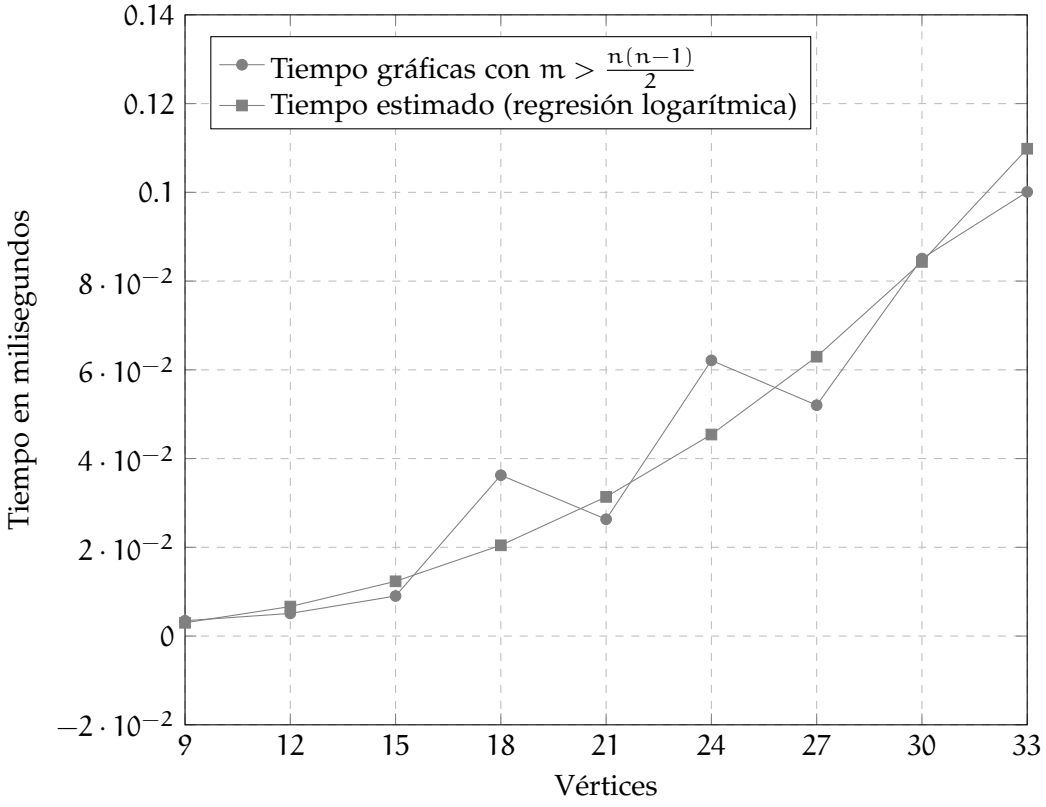
$$M_2 = [30,\ 59,\ 96,\ 140,\ 198,\ 263,\ 341,\ 420,\ 515],$$
$$N_2 = [\ 9,\ 12,\ 15,\ \ 18,\ \ 21,\ \ 24,\ \ 27,\ \ 30,\ \ 33].$$

Si $n \in N_2$, el número máximo de aristas que puede tener una gráfica con n vértices es $\frac{n(n-1)}{2}$, estos valores se muestran en el siguiente arreglo:

$$M_2'' = [36,\ 66,\ 105,\ 153,\ 210,\ 276,\ 351,\ 435,\ 528].$$

Observe que el número en la i-ésima entrada del arreglo M_2, es un valor cercano al número en la i-ésima entrada, del arreglo M_2''. De ser cierta la conjetura 4.6.4, los experimentos realizados por el procedimiento $\text{TIEMPO}_{n,m}$ con los arreglos de entrada N_2 y M_2, deberían tener complejidad en $\mathcal{O}(n^3)$. Por lo tanto, la regresión exponencial $A_0 e^B$, debe tener a la constante B cercana a 3. La tabla 10 y la figura 23 confirman que esta condición se cumple.

Figura 23: Resultados del experimento y de la regresión logarítmica para gráficas con $m > \frac{n(n-3)}{2}$, $M_2 = [30, 59, 96, 140, 195, 263, 342, 420, 515]$ y $N_2 = [9, 12, 15, 18, 21, 24, 27, 30, 33]$.



CONCLUSIONES

En este trabajo estudiamos teoremas fundamentales de la teoría de la complejidad computacional. Hicimos un extenso análisis de esos conceptos a fin de buscar herramientas que puedan ser de utilidad para entender mejor la complejidad temporal del algoritmo de Bron-Kerbosch, que es uno de los algoritmos más utilizados para buscar clanes en una gráfica. Al estudiar las clase de complejidad de los problemas LClique, MClique y Clique, surgieron múltiples ideas y problemas relacionados con la complejidad temporal de los algoritmos que buscan clanes. En particular, en una buena parte de este trabajo, nos centramos en estudiar el problema abierto de determinar una cota de complejidad temporal para el algoritmo de Bron-Kerbosch, en función de aristas y vértices.

Aunque de entrada el problema parece sencillo, en la práctica resultó mucho más difícil de lo que creíamos, no por nada, el simple problema de determinar la complejidad temporal del algoritmo de Bron-Kerbosch en función de n vértices, permaneció abierto por más de 30 años.

Durante el desarrollo de este trabajo, surgieron múltiples conjeturas que resultaron ser falsas pero que fueron de gran ayuda para determinar las conjeturas que presentamos en 4.6.4 y en 4.6.3. Estudiamos a fondo los lemas y teoremas necesarios para demostrar las propiedades y la complejidad temporal del algoritmo de Bron-Kerbosch. También estudiamos otros algoritmos que buscan clanes en una gráfica, que ya no incluimos en este trabajo, pero que resultaron útiles para posibles ideas que permitan una demostración de las conjeturas que se presentan en esta tesis.

Existen múltiples problemas técnicos a resolver para demostrar formalmente la conjeturas que tenemos, sin embargo, parece haber mucha evidencia experimental que apoya la idea de que nuestras conjeturas son correctas. Quizás un análisis posterior del problema, nos permita demostrar alguna de las conjeturas presentadas. Creemos que hay incógnitas y problemas abiertos muy interesantes, relacionados con la complejidad del algoritmo de Bron-Kerbosch.

BIBLIOGRAFÍA

[1] B. Bollobás. *Modern graph theory*. First Edition. Elsevier Science Publishing, 2000.

[2] J. Bondy. *Graph theory with applications*. Fifth Edition. Elsevier Science Publishing, 1982.

[3] A. T. Brint y P. Willett. «Algorithms for the identification of three-dimensional maximal common substructures.» En: *Journal of Chemical Information and Computer Sciences* 27.4 (1987), págs. 152-158.

[4] C. Bron y J. Kerbosch. «Finding all cliques of an undirected graph–Algorithm 457». En: *Communications of the ACM* 16 (1973), págs. 575-577.

[5] F. Cazals y C. Karande. «A note on the problem of reporting maximal cliques». En: *Theoret. Comput. Sci.* 407.1-3 (2008), págs. 564-568.

[6] A. Church. «An Unsolvable Problem of Elementary Number Theory». En: *American Journal of Mathematics* 58.2 (1936), págs. 345-363.

[7] S. Cook. «The complexity of theorem-proving procedures». En: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: ACM, 1971, págs. 151-158.

[8] J. Copic, M. O. Jackson y A. Kirman. «Identifying community structures from network data via maximum likelihood methods». En: *B. E. J. Theor. Econ.* 9.1 (2009), Art. 30, 40.

[9] D. Eppstein y D. Strash. «Listing All Maximal Cliques in Large Sparse Real-World Graphs.» En: *SEA*. Ed. por P. M. Pardalos y S. Rebennack. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, págs. 364-375.

[10] S. Fortunato. «Community detection in graphs». En: *Phys. Rep.* 486.3-5 (2010), págs. 75-174.

[11] M. Garey y D. Johnson. *Computers and intractability; A guide to the theory of NP-completeness*. First Edition. W.H. Freeman & Co., 1976.

[12] D. Hilbert y W. Ackerman. *Principles of Mathematical Logic*. First Edition. Julius Springer, 1928.

[13] J. E. Hopcroft, R. Motwani y J. Jeffrey. *Introduction to automata theory, languages and computation*. Second Edition. Addison-Wesley, 2001.

[14] R. M. Karp. «Reducibility among combinatorial problems». En: *Complexity of Computer Computations*. Ed. por R. Miller y J. Thatcher. Plenum Press, 1972, págs. 85-103.

[15] D. E. Knuth. «Big Omicron and big Omega and big Theta». En: *SIGACT News* 8.2 (1976), págs. 18-24.

[16] J. I. Koch. «Enumerating all connected maximal common subgraphs in two graphs». En: *Theoret. Comput. Sci.* 250.1-2 (2001), págs. 1-30.

[17] D. C. Kozen. *Automata and Computability*. First Edition. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[18] E. H. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Volume 2. Leipzig: B. G. Teubner, 1909.

[19] E. Leggett y D. J. Moore. «Optimization problems and the polynomial hierarchy». En: *Theoretical Computer Science* 15.3 (1981), págs. 279 -289.

[20] L. A. Levin. «Universal sequential search problems». En: *Problemy Peredachi Informatsii* 9.3 (1972), págs. 115-116.

[21] M. Li y P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Second Edition. Springer, 1997.

[22] C. C. McGeoch. *A Guide to Experimental Algorithmics*. Firs Edition. Cambridge University Press, 2012.

[23] B. McKay. *Various simple graphs*. 2016. URL: http://users.cecs.anu.edu.au/~bdm/data/graphs.html (visitado 24-01-2016).

[24] R. Miller y D. Muller. «A problem of maximum consistent subsets». En: *IBM Research Report* RC-240 (1960).

[25] J. Moon y L. Moser. «On cliques in graphs». En: *Israel Journal of Mathematics* 3 (1 1965), págs. 23-28.

[26] T. J. Ottosen y J. Vomlel. «All roads lead to Rome—new search methods for the optimal triangulation problem». En: *Internat. J. Approx. Reason.* 53.9 (2012), págs. 1350-1366.

[27] R. A. Shore y T. A. Slaman. «Defining the Turing Jump». En: *Mathematical Research Letters* 6 (1999), págs. 711-722.

[28] M. Sipser. *Introduction to the theory of computation*. Second Edition. Thomson Course Technology, 2006.

[29] E. Tomita, A. Tanaka y H. Takahashi. «The worst-case time complexity for generating all maximal cliques and computational experiments». En: *Theoret. Comput. Sci.* 363.1 (2006), págs. 28-42.

[30] S. Tsukiyama, M. Ide, H. Ariyoshi e I. Shirakawa. «A new algorithm for generating all the maximal independent sets». En: *SIAM J. Comput.* 6.3 (1977), págs. 505-517.

[31] A. M. Turing. «On Computable Numbers, with an Application to the Entscheidungsproblem». En: *Proceedings of the London Mathematical Society* 2.42 (1936), págs. 230-265.

[32] S. Weisberg. *Applied Linear Regression*. Third Edition. John Wiley & Sons, 2005.

[33] D. Wood. «On the number of maximal independent sets in a graph». En: *Discrete Math. Theor. Comput. Sci.* 13.3 (2011), págs. 17-19.

ÍNDICE ALFABÉTICO