# RESTRICCIONES PLANARES DE PROBLEMAS DUROS: UN ESTUDIO DE CASO

**JONNATHAN ALFREDO RAMOS CHAUX**

**UNIVERSIDAD INDUSTRIAL DE SANTANDER**
**FACULTAD DE INGENIERÍAS FISICO-MECÁNICAS**
**ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA**
**BUCARAMANGA**
**2011**

# RESTRICCIONES PLANARES DE PROBLEMAS DUROS: UN ESTUDIO DE CASO

**JONNATHAN ALFREDO RAMOS CHAUX**

**Trabajo de grado presentado para optar por el título de Ingeniero de Sistemas**

**Director**
**PhD. Juan Andrés Montoya Argüello**
**Profesor escuela de matemáticas**

**UNIVERSIDAD INDUSTRIAL DE SANTANDER**
**FACULTAD DE INGENIERÍAS FISICO-MECÁNICAS**
**ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA**
**BUCARAMANGA**
**2011**

A Dios

A mis padres Luis Alfredo y Blanca Lidia

A mi hermana Laura Katerinne

A mi novia Mary Hiovana

# AGRADECIMIENTOS

A Dios por darme la fortaleza para luchar y por ubicar en mi camino a las personas indicadas en el momento indicado.

A mis padres Luis Alfredo y Blanca Lidia porque gracias a su educación, su esfuerzo y su lucha constante pude alcanzar este logro que es más de ellos que mío.

A mi hermana Laura por su complicidad y los buenos momentos.

A mi novia Mary Hiovana por su amor y por apoyarme en mis momentos de flaqueza.

Al profesor Rafael Isaacs por introducirme en el fantástico mundo de la matemática discreta y al profesor Juan Andrés Montoya por su instrucción, paciencia, apoyo y guía.

A mis amigos Gustavo Fonseca, Sergio Gélvez, Cesar Ríos, Manuel Cuadrado, Wilfredo Gómez, Oscar Rodríguez, Eduardo Guerra, Darío Delgado, Julio Reyes, Andrés Plata, Federico Chaves y a todas aquellas personas que me han premiado con su amistad.

**Gracias** **Totales.**

# CONTENIDO

pág.

# LISTA DE ILUSTRACIONES

pág.

# RESUMEN


**TÍTULO**: RESTRICCIONES PLANARES DE PROBLEMAS DUROS: UN ESTUDIO DE CASO[*]

**AUTOR**: RAMOS CHAUX, Jonnathan Alfredo[**]

**PALABRAS CLAVE**: Isomorfismo de Grafos, Complejidad Computacional, Computación Teórica, Problemas Intermedios.

**CONTENIDO:**

Los grafos son estructuras matemáticas útiles en labores de modelado dentro de diversas áreas del conocimiento. En particular, el isomorfismo de grafos brinda la posibilidad de extender propiedades de interés, que se sabe presentes en un grafo, a otro, gracias a la existencia de una función denominada de isomorfismo que a cada vértice de un grafo le asigna como imagen un vértice en el otro grafo de modo tal que la relación de adyacencia es preservada.

Dados dos grafos, saber si son isomorfos (problema del isomorfismo de grafos) es un problema que a día de hoy no puede ser resuelto por una máquina determinista en tiempo polinomial. Sin embargo, es sabido que clases específicas como la que agrupa a los grafos planos puede ser solucionada de manera eficiente.

En este proyecto se estudia el problema del isomorfismo de grafos centrando la atención, particularmente, en su complejidad computacional y en la versión del problema restringida a grafos planos. También es estudiado un algoritmo que puede ser implementado capaz de, dados dos grafos planos como entrada, decidir si los grafos del input son isomorfos en un tiempo $O(n^2)$. La metodología utilizada por el algoritmo consiste en la generación de un código único que identifica cada grafo de modo tal que dos grafos serán isomorfos si y solo si dichos códigos son iguales.


[*] Proyecto de Investigación

[**] Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Juan Andrés Montoya Argüello.

# ABSTRACT

**TITLE**: PLANAR RESTRICTIONS OF HARD PROBLEMS: A CASE STUDY[*]

**AUTHOR**: RAMOS CHAUX, Jonnathan Alfredo[**]

**KEY WORDS**: Graph isomorphism, Computational complexity, Theoretical Computing, Restrictions, Hard Problems.

**DESCRIPTION:**

Graphs are mathematical structures useful in modeling labors within various fields of knowledge. In particular, the graph isomorphism provides the possibility of extending properties of interest from one graph to another, thanks to the existence of a function called “isomorphism” that assigns to each vertex of a graph, as image, a vertex in the other graph such that the adjacency relation is preserved from one graph to another.

Given two graphs, knowing if they are isomorphic (graph isomorphism problem) is a problem that cannot be solved by a deterministic Turing machine in polynomial time. However, it is known that to specific classes, such as planar graphs, the problem can be solved efficiently.

In this Project, the graph isomorphism problem is studied focusing the attention, particularly, on their computational complexity and the version of the problem restricted to planar graphs. It is also considered an algorithm, which can be implemented and capable to, given two planar graphs as input, decide in $O(n^2)$ units of time if they are isomorphic. The methodology used by the algorithm consists in construct a unique code that identifies each graph such that two graphs are isomorphic if and only if their codes are the same.

[*] Research Project

[**] Faculty of Physical-Mechanical Engineering. Systems Engineering and Informatics School. Advisor: Juan Andres Montoya Argüello.

# GLOSARIO

$\#\boldsymbol{P}$: Es la clase de problemas de función de la forma: "compute $f(x)$", donde $f$ es el número de caminos aceptantes de una máquina de Turing no determinista de tiempo polinomial.

$\boldsymbol{BQP}$: Es la clase de los problemas de decisión resolubles en tiempo polinomial por una máquina de Turing cuántica con una probabilidad máxima de error de $1/3$. (Del inglés Bounded-Error Quantum Polynomial time).

**CAMINO:** Secuencia alternada $v_0 e_1 v_1, \ldots, v_{r-1} e_r v_r$ de vértices y aristas donde aristas consecutivas son adyacentes tal que cada arista $e_i$ une a los vértices $v_{i-1}$ y $v_i$.

**CIRCUITO:** Camino cerrado de longitud positiva. Camino que inicia y termina en el mismo vértice $v_0 = v_r$.

**CIRCUITO EULERIANO:** circuito que contiene todas las aristas del grafo.

**CÓDIGO:** Secuencia de caracteres que identifican una estructura.

**COMPONENTE CORTANTE:** de un par cortante $\{u, v\}$ de un grafo $G$ es o una arista $e = (u, v)$ del grafo o un subgrafo maximal $C$ de $G$ tal que $\{u, v\}$ no es un par cortante de $C$.

**EMBEBIMIENTO DE UN GRAFO:** Es un ordenamiento de las aristas en torno a cada vértice de un grafo el cual permite dibujar el grafo en el plano sin que sin que exista cruce de aristas.

**GRAFO BICONEXO:** Es un grafo conexo sin puntos de articulación.

**GRAFO CONEXO:** es un grafo $G = (V, E)$ tal que para cualquier $v, w \in V(G)$ existe un camino que los conecta.

**GRAFO PLANO BICONEXO:** es un grafo $G = (V, E)$ que puede ser embebido en el plano y no tiene puntos de articulación.

**GRAFO PLANO TRICONEXO:** es un grafo $G = (V, E)$ que puede ser embebido en el plano y no tiene pares separadores.

**GRAFO TRICONEXO:** Es un grafo sin pares separadores.

$\boldsymbol{LOGSPACE}$: Es la clase de problemas de decisión resolubles por una máquina de Turing restringida a usar una cantidad de memoria logarítmica en el tamaño de la instancia.

$\boldsymbol{NP \cap coNP}$: Es la clase de problemas que se encuentran en $NP$ y en $coNP$.

**ORDENAMIENTO DE CÓDIGOS:** Organizar un conjunto de códigos de acuerdo al orden lexicográfico

**PAR CORTANTE (DE UN GRAFO):** es un par de vértices $\{u, v\}$ del grafo tal que o son un par separador o un par de vértices adyacentes.

**PAR MAXIMAL CORTANTE:** de un grafo $G$ con respecto a un al par cortante $\{s, t\}$ es tal que, para cualquier otro par cortante $\{u', v'\}$, los vértices $u, v, s, t$ están en la misma componente cortante.

**PAR SEPARADOR:** Es una pareja de vértices en un grafo que cuando son removidos del grafo lo desconecta.

**PUNTO DE ARTICULACIÓN:** Es un vértice en un grafo conexo que cuando es removido del grafo lo desconecta.

# INTRODUCCIÓN

Contrario a lo que pueda parecer, elaborar una radiografía de un problema como el del isomorfismo de grafos no resulta ser una tarea sencilla; es tanta la información al alcance, tan diversos los enfoques para abordarlo y tantos los aportes de la comunidad científica, que el proceso de selección de los pilares sobre los cuales se construirá una investigación a nivel de pregrado se convierte en una tarea bastante compleja.

Todas las fuentes que se consulten acerca del isomorfismo de grafos, siempre coincidirán en que este es un objeto de estudio de suma importancia no solo por su aplicabilidad en campos como la informática química o la minería de datos, sino también por su estructura y propiedades que lo hacen, aún hoy, objeto de nuevas investigaciones desde diversas áreas del conocimiento

En este proyecto se estudió el problema del isomorfismo de grafos desde la visión que de él ofrece la computación teórica, más concretamente desde la óptica de la teoría de la complejidad computacional. Para este fin, se desarrolló un proceso de aprendizaje a partir de la recopilación bibliográfica de fuentes especializadas en el área.

El proceso descrito dio como resultado la escritura de este documento, el cual puede considerarse dividido en dos secciones: la primera de ellas, descrita en los capítulos dos, tres y cuatro, recopila los fundamentos teóricos necesarios para aproximarse al problema desde el enfoque deseado. La primera parte, identificada con el numeral 2, presenta al lector los fundamentos de la teoría de la complejidad

computacional considerando como recurso computacional medible el tiempo de ejecución de las rutinas que solucionan un problema computacional dado; posteriormente se estudian algunos problemas que se conjeturan $NP$-intermedios. Como inciso final de la sección son expuestas un conjunto de metodologías que permiten abordar de un modo eficaz problemas que pueden ser complejos en términos computacionales.

La segunda sección presenta dos apartados. El primero de ellos, identificado con el numeral cinco, se enfoca en el estudio de la complejidad computacional del problema del isomorfismo de grafos mientras que el segundo, identificado como el numeral seis, centra su atención en el análisis de la restricción planar del problema y en un algoritmo capaz de resolver dicho problema de manera eficiente.

# 1. PRESENTACIÓN DEL PROYECTO

## 1.1. OBJETIVO GENERAL

Estudiar el problema de isomorfismo de grafos, cuando se restringe a grafos planos.

## 1.2. OBJETIVOS ESPECÍFICOS

Elaborar un documento que presente la información computacional general referente al problema de isomorfismo de grafos y las variaciones ocasionadas en el mismo al introducir restricciones planares.

Seleccionar un algoritmo capaz de detectar si dos grafos planares son isomorfos.

Estudiar en detalle el algortimo de detección de isomorfismo de dos grafos planares.

# 2. COMPLEJIDAD COMPUTACIONAL

*"Un hombre provisto de papel, lápiz y goma, y con sujeción a una disciplina estricta, es en efecto una máquina de Turing universal."*

*Alan Turing*

El término computación es definido como “el procedimiento de calcular o determinar algo por métodos matemáticos o lógicos”; esta definición no resulta ajena para alguien en la actualidad, incluso, es común que se establezca inmediatamente un vínculo entre el concepto (de computación) y la máquina que ha revolucionado la vida de la humanidad. Sin embargo, y por extraño que parezca, en los inicios de la computación no existía la computadora y en su lugar se encontraban seres humanos encargados de desarrollar tareas que implicaban grandes volúmenes de cálculo con las pocas herramientas de apoyo a su disposición, lo cual hacia de este un trabajo tedioso, poco eficiente y, más importante aún, bastante errático. En aquel entonces la mayor preocupación era conocer que problemas podían ser “computados” por un ser humano con los recursos a su disposición.

Con la aparición de las máquinas computadoras las tareas que implicaban un alto volumen de cálculo empezaron a ser resueltas con mayor eficiencia, razón por la cual el interés sobre los nuevos dispositivos, como aprovechar sus bondades y conocer sus limitaciones prácticas se acrecentó. Un importante punto de partida para formalizar el estudio de los límites de la naciente computación radicó en la categorización de las tareas computacionales en dos grandes grupos a saber:

- **Problemas de decisión:** en este tipo de problema se busca determinar si una instancia de un problema dado pertenece o no a un subconjunto que cumple con las condiciones enunciadas por el problema; la solución de

estos (problemas) es “si” o “no” entendiéndose la respuesta afirmativa como la pertenencia de la instancia al subconjunto mencionado.

- **Problemas de búsqueda:** los problemas de búsqueda se ajustan más a la convención de solucionar un problema. Resolver un problema de búsqueda consiste en especificar un conjunto de soluciones (que puede ser vacío $\{\emptyset\}$)

  para cada instancia del problema. Asociados a los problemas de búsqueda se encuentran los **problemas de conteo** (contar los elementos del conjunto solución asociado a la instancia del problema) y los **problemas de optimización** (hallar dentro del conjunto de soluciones asociado a la instancia del problema la “mejor solución” posible que satisfaga unos parámetros establecidos).

## 2.1. TEORIA DE LA COMPLEJIDAD COMPUTACIONAL

Una vez definidos los objetos de estudio, aparece la teoría de la complejidad computacional que actualmente se define como la rama de la ciencia de los computadores y de las matemáticas encargada de abordar y clasificar las tareas computacionales de acuerdo a su complejidad inherente, valiéndose de la cuantificación de los recursos computacionales disponibles (como tiempo de computo, espacio de almacenamiento, cantidad de procesadores), necesarios para resolver de manera eficiente un problema computacional.

La teoría de la complejidad está interesada primordialmente en la eficiencia de todos los algoritmos existentes o que lleguen a existir y como escalan las demandas computacionales de los mismos conforme varia el tamaño de la instancia del problema. Como es evidente, la medición de los recursos exigidos para solucionar el problema juega un papel dominante en el estudio y la clasificación del problema. Para tal tarea se introducen modelos matemáticos de computación que formalizan dicha actividad. El modelo estándar de medición es la

máquina de Turing que, básicamente, es un dispositivo teórico que lee símbolos que se encuentran escritos en una cinta y los computa para determinar una respuesta. A pesar de su sencillez, la máquina de Turing es un potente dispositivo capaz de simular modelos más complejos con apenas un aumento en el tiempo de ejecución. La Tesis de Church-Turing condensa la robustez de las máquinas de Turing al sostener que:

*"Todo lo que es computable es Turing-Computable"*

Esto significa que, para toda función efectivamente computable existe una máquina de Turing capaz de calcularla.

## 2.2. CLASES DE COMPLEJIDAD COMPUTACIONAL

Los problemas computacionales son agrupados en clases de complejidad de acuerdo a los recursos que demandan para ser resueltos.

Una clase de complejidad es un conjunto de problemas que pueden ser computados con una cantidad dada de recursos. Este conjunto puede ser definido fijando alguno(s) de los tres parámetros siguientes:

1. Tipo de problema computacional
2. Tipo de máquina necesario para resolver el problema
3. Recurso (o recursos) computacional que está siendo medido y una función limitante

### 2.2.1. Clase $P$

**Nombre de la clase de complejidad:** $P$

**Tipo de problema:** Problemas de Decisión

**Máquina necesaria:** Máquina de Turing determinista

**Recurso Computacional/ Función Limitante:** Tiempo de Computo/ Polinomio

Informalmente la clase $P$ es la clase de los problemas de decisión resolubles por algún algoritmo en un número de pasos acotado por un polinomio en función de la longitud de la entrada [5]. De un modo más preciso, y considerando a los elementos de $P$ como lenguajes se tiene que:

Sea:

$\Sigma$ Un alfabeto finito con al menos dos elementos

$\Sigma^*$ El conjunto de todas las palabras de longitud finita sobre $\Sigma$.

$\mathcal{L}$ Un lenguaje tal que $\mathcal{L} \subseteq \Sigma^*$

$\mathcal{M}$ Una máquina de Turing

Se dice que la máquina $\mathcal{M}$ acepta una cadena $w \in \Sigma^*$ si la computación realizada por $\mathcal{M}$ con input $w$ *termina* en un *estado de aceptación.* El lenguaje aceptado por $\mathcal{M}$, denotado como $\mathcal{L}(\mathcal{M})$, se define como:

$$\mathcal{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \mathcal{M}\ acepta\ a\ w\}$$

Sea:

$t_{\mathcal{M}}(w)$ = número de pasos realizados por $\mathcal{M}$ al computar $w$

Para todo $n \in \mathbb{N}$ se denota con $T_{\mathcal{M}}(n)$ al peor tiempo de ejecución de $\mathcal{M}$ en inputs de tamaño $n$, esto es:

$$T_{\mathcal{M}}(n) = max\{t_{\mathcal{M}}(w) \mid w \in \Sigma^n\}$$

Donde $\Sigma^n$ es el conjunto de todas las palabras sobre $\Sigma$ de longitud $n$. Se dice que $\mathcal{M}$ se ejecuta en tiempo polinomial si existe $k$ tal que $\forall n \in \mathbb{N}, T_{\mathcal{M}}(n) \leq n^k + k$.

**Definición 1: Clase de Complejidad $\mathbf{P}$**
Se define la clase $P$ como:

$$P = \left\{ \mathcal{L} \;\middle|\; \begin{array}{c} \mathcal{L} = \mathcal{L}(\mathcal{M}) \; para \; alguna \; máquina \; de \; Turing \; \mathcal{M} \\ que \; se \; ejecuta \; en \; tiempo \; polinomial \end{array} \right\}$$ [5]

### 2.2.2. Clase $NP$

**Nombre de la clase de complejidad: $NP$**

**Tipo de problema:** Problemas de Decisión

**Máquina necesaria:** Máquina de Turing no determinista

**Recurso Computacional/ Función Limitante:** Tiempo de Computo/ Polinomio

Oded Goldreich define en [13] la clase $NP$ como “el conjunto de problemas que pueden ser decididos en tiempo polinomial por un dispositivo ficticio llamado máquina de Turing no determinista”. Aunque la palabra “ficticio” pueda parecer peculiar, esta se encuentra perfectamente justificada debido a que todos los dispositivos reales actualmente conocidos son de naturaleza determinista. Una manera análoga de definir esta clase de complejidad es como la clase de problemas de decisión verificables por una máquina de Turing determinista en tiempo polinomial.

Formalmente se tiene:

**Definición 2: Clase de Complejidad $NP$**

Un lenguaje $\mathcal{L} \subseteq \Sigma^*$ está en $NP$ si existe un polinomio $p: \mathbb{N} \to \mathbb{N}$ y una máquina de Turing determinista $\mathcal{M}$, llamada verificador para $\mathcal{L}$, tal que $\forall w \in \Sigma^*$

$$w \in \mathcal{L} \Leftrightarrow \exists u \in \Sigma^{p(|w|)} \mid \mathcal{M}(w,u) = 1$$

Si $w \in \mathcal{L}$ y $u \in \Sigma^{p(|w|)}$ satisface que $\mathcal{M}(w,u) = 1$ entonces $u$ es llamado un certificado para $w$ con respecto al lenguaje $\mathcal{L}$ y la máquina de Turing $\mathcal{M}$ [2].

## 2.3. REDUCCIONES Y $NP$-COMPLETEZ

A pesar de tener estructuras similares, algunos problemas dentro de una misma clase de complejidad resultan más difíciles de resolver que otros, surgiendo entonces, un fenómeno interesante denominado “completez”; sin embargo, antes de desarrollarlo resulta conveniente y necesario introducir primero el concepto de reducción sobre el cual se sustenta.

### 2.3.1. Reducción

La idea fundamental de lo que es una reducción es bastante sencilla y, de hecho, difícilmente puede llegar a resultar ajena para alguien. Básicamente, reducir un problema $A$ a un problema $B$ consiste en plantear el problema $A$ de un modo tal que pueda ser resuelto con el conjunto de procedimientos con los que se resuelve el problema $B$.

Una reducción ha de ser una función fácil de calcular, de no ser así, solucionar un problema usando una reducción puede resultar menos eficiente que enfrentar el problema directamente.

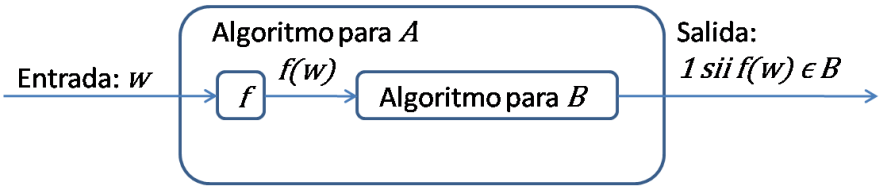


**Ilustración 1 Esquema gráfico de una reducción**

**Fuente: Computational Complexity: A Modern Approach**

De un modo más formal se tiene:

**Definición 3: Reducción**

Un lenguaje $\mathcal{L} \subseteq \Sigma^*$ es reducible a un lenguaje $\mathcal{L}' \subseteq \Sigma^*$ y notado como $\mathcal{L} \leq_* \mathcal{L}'$ si existe una función $f: \Sigma^* \to \Sigma^*$ computable en tiempo polinomial tal que:

$$\forall w \in \Sigma^*, w \in \mathcal{L} \Leftrightarrow f(w) \in \mathcal{L}' \text{ [2]}$$

El asterisco (*) en la notación de la reducción indica el tipo de, valga la redundancia, reducción usado (Cook, Turing, Karp entre otros) y es reemplazado por la letra que distingue cada prototipo. Un simple vistazo permite observar que si $\mathcal{L} \leq \mathcal{L}'$ y $\mathcal{L}$ está contenido en la clase $P$, entonces $\mathcal{L}'$ también estará contenido en $P$. Es importante mencionar que las reducciones son relaciones reflexivas y transitivas.

### 2.3.2. Problemas $NP$-Completos

La clase $NP$-Completo ($NPC$) es un subconjunto de la clase $NP$ en el cual se encuentran los problemas más difíciles en $NP$, siendo su característica más notable que no se conoce una solución eficiente para ellos ya que el tiempo

requerido para solucionarlos aumenta muy rápidamente con el tamaño de las instancias del problema haciéndose necesario, usualmente, tiempo superpolinomial al usar los algoritmos actualmente conocidos.

**Definición 4: Problema $NP$-Completo**
Sea $\mathcal{L} \subseteq \Sigma^*$ un lenguaje, $\mathcal{L}$ es llamado $NP$-Completo si cumple simultáneamente con las siguientes condiciones

i. $\mathcal{L} \in NP$.
ii. Cualquier $\mathcal{L}' \in NP$ es reducible en tiempo polinomial a $\mathcal{L}$, notado como $\mathcal{L}' \leq_p \mathcal{L}$.

Por consiguiente, la clase de complejidad $NPC$ está formada por todos los problemas $NP$-Completos. Vale la pena mencionar que el fenómeno de "completez" se usa para indicar cuáles son los problemas más difíciles de solucionar dentro de una clase de complejidad cualquiera pudiéndose intuir entonces que no es un fenómeno exclusivo de la clase $NP$.

## 2.4. $P$ VERSUS $NP$

¿Es $P = NP$? La anterior es una pregunta fundamental dentro de la teoría de la complejidad computacional y la teoría de la computabilidad. Como ya se mencionó anteriormente la clase $P$ es la clase de lenguajes $\mathcal{L} \subseteq \Sigma^*$ que pueden ser decididos en tiempo polinomial, mientras que la clase $NP$ es la clase de los lenguajes verificables en tiempo polinomial. Demostrar que $P \subseteq NP$ no resulta

complicado; para cada lenguaje $\mathcal{L}$ sobre $\Sigma$, si $\mathcal{L} \in P$ entonces se puede definir una relación de verificación en tiempo polinomial $R \subseteq \Sigma^* \cup \Sigma^*$ definida como:

$$R(w,y) \Leftrightarrow w \in \mathcal{L} \text{ y } |y| = |w|$$

Para todo $w \in \Sigma^*$.

Ahora bien, se desconoce si la relación de contenencia $P \subseteq NP$ es estricta o no. El problema $P$ versus $NP$ consiste en determinar si todo lenguaje reconocido por una máquina no determinista en tiempo polinomial puede ser reconocido también por un algoritmo determinista en tiempo polinomial.

La importancia de esta pregunta radica, entre otras cosas, en que un gran número de desarrollos teóricos dependen de la conjetura $P \neq NP$. Es claro que estos desarrollos colapsarían de no ser cierta dicha conjetura. Por ejemplo, los protocolos criptográficos usados en la actualidad se basan en que $P \neq NP$ y podrían ser fácilmente rotos de ser falsa dicha aseveración.

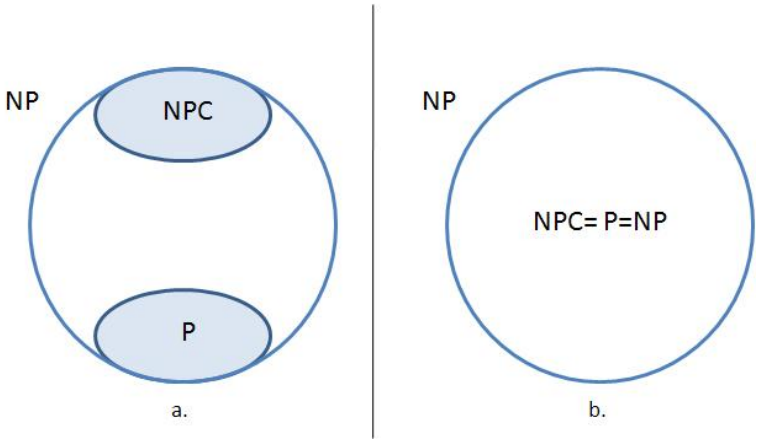


**Ilustración 2 Diagramas de Venn de la clase NP. a. $NP \neq P$ b. $NP = P$**

Para demostrar que $P = NP$ bastaría con exhibir un algoritmo que se ejecute en tiempo polinomial y que resuelva un problema $NP$-Completo cualquiera, ya que por definición de la clase $NPC$, si algún problema de la clase tiene solución algorítmica en tiempo polinomial, entonces todos los problemas de la clase serán reducibles a él y por ende también serán resolubles en tiempo polinomial. Por otro lado, demostrar que $P \neq NP$ resulta un poco más complejo y se hace necesario un argumento de validez universal o probar una cota inferior superpolinomial para cualquier familia de circuitos booleanos que resuelva algún problema $NP$-Completo [5].

Sin embargo, y a pesar de los grandes esfuerzos realizados por científicos provenientes de gran variedad de áreas, aún no se ha podido dar respuesta a tan fundamental cuestionamiento.

## 2.5. JERARQUÍA POLINOMIAL

Las máquinas de Turing que se han considerado (y considerarán) en este escrito tienen como función decidir de entre un conjunto de instancias cuales satisfacen los requerimientos del problema bajo estudio y cuáles no lo hacen. De este modo, el conjunto de instancias se divide en el conjunto de instancias aceptadas y en el conjunto de instancias rechazadas.

Dicho lo anterior se tiene que:

Sea $\mathcal{L} \subseteq \Sigma^*$ un lenguaje, se denotará como $\bar{\mathcal{L}}$ al complemento de $\mathcal{L}$; es decir que

$$\bar{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$$

De este modo se tiene que la clase $coNP$ se puede definir como:

$$coNP = \{\mathcal{L} \mid \bar{\mathcal{L}} \in NP\}$$

Ahora, definiendo la clase $coNP$ de un modo más formal:

**Definición 5: Clase $coNP$**
Para todo $\mathcal{L} \subseteq \Sigma^*$, se dice que $\mathcal{L} \in coNP$ si existe un polinomio $p: \mathbb{N} \to \mathbb{N}$ y una máquina de Turing polinomial tal que $\forall w \in \Sigma^*$

$$w \in \mathcal{L} \iff \forall u \in \Sigma^{p(|w|)}, \mathcal{M}(w, u) = 1$$

Ya introducida la clase de complejidad $coNP$ se puede decir que la Jerarquía Polinomial ($PH$, del inglés *Polynomial Hierarchy*) es, como su nombre lo indica, una jerarquía compuesta de un número infinito de subclases de complejidad que extienden a $P$, $NP$ y $coNP$.

Definiendo la jerarquía polinomial en función de cuantificadores universales (∀) y existenciales (∃) se tiene:

**Definición 6: Jerarquía Polinomial *($PH$)***
Para un número $k \in \mathbb{N}$, un problema de decisión $\mathcal{L} \subseteq \Sigma^*$ está en $\Sigma_k^p$ si existe un polinomio $p: \mathbb{N} \to \mathbb{N}$ y una máquina de Turing polinomial $\mathcal{M}$ tal que:

$$w \in \mathcal{L} \iff \exists y_1 \Sigma^{p(|w|)} \forall y_2 \Sigma^{p(|w|)} \dots Q_k y_k \Sigma^{p(|w|)} |$$

$$\mathcal{M}(w, y_1, y_2, \dots, y_k) = 1$$

Donde $Q_k$ es un cuantificador existencial si $k$ es impar y en el caso contrario es un cuantificador universal.

Es evidente que $\Sigma_1^p = NP$ y que $\Sigma_0^p = P$. Si se considera que $\Pi_k^p = co\Sigma_k^p$ donde $co\Sigma_k^p = \{\bar{\mathcal{L}} | \mathcal{L} \epsilon \Sigma_k^p\}$ los niveles de la jerarquía también pueden ser definidos de manera recursiva de la siguiente manera:

i. $\Sigma_0^p = \Pi_0^p = P$

ii. $\Sigma_{k+1}^p = \exists p \, \Pi_k^p$

iii. $\Pi_{k+1}^p = \forall p \, \Sigma_k^p$

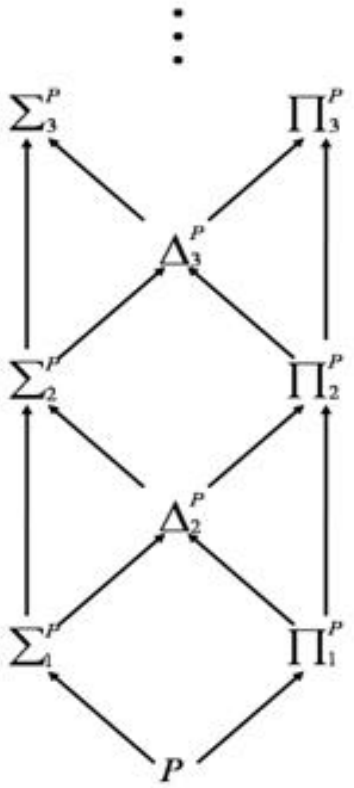


**Ilustración 3 Esquema gráfico de la jerarquía polinomial**

La definición implica que:

$$\Sigma_k^p \subseteq \Delta_{k+1}^p \subseteq \Sigma_{k+1}^p$$

$$\Pi_k^p \subseteq \Delta_{k+1}^p \subseteq \Pi_{k+1}^p$$

$$\Sigma_k^p = co\Pi_k^p$$

En la actualidad se desconoce si las relaciones de inclusión son propias aunque se cree que es así; sin embargo, si existiera un $k \in \mathbb{N}$ tal que $\Sigma_k^p = \Sigma_{k+1}^p$ o $\Pi_k^p = \Pi_{k+1}^p$ o que $\Sigma_k^p = \Pi_k^p$ la jerarquía dejaría de ser infinita y se diría que "colapsa" al nivel $k$-ésimo. Si $P = NP$ la jerarquía colapsaría completamente a $P$, es decir que $P = PH$ siendo $PH = \bigcup_{k\in\mathbb{N}} \Sigma_k^p$.

# 3. PROBLEMAS INTERMEDIOS

La mayoría de los problemas que pertenecen a la clase $NP$ han sido clasificados bien como problemas fáciles (en $P$) o bien como problemas duros ($NP$-Completos), este hecho puede llevar a creer que todo problema contenido en $NP$ o es fácil o es $NP$-Completo. Sin embargo, existen algunos problemas en $NP$ que se han resistido a ser categorizados de la forma antes mencionada, a pesar del intensivo estudio al que han sido sometidos y de los innumerables intentos hechos para clasificarlos. Surge entonces de manera lógica la idea que si $P \neq NP$, entonces el conjunto formado por $NP \setminus (P \cup NPC)$ es no vacío [4], naciendo de este modo una tercera clase dentro de $NP$ llamada $NP$-Intermedio y notada como $NPI$.

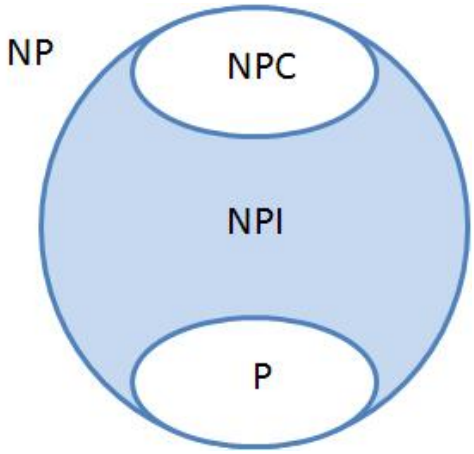


**Ilustración 4 Diagrama de Venn de la clase $NP$ si $P \neq NP$**

**Teorema de Ladner**
Si $P \neq NP$ existe entonces $\mathcal{L} \in (NP \setminus P)$ tal que $\mathcal{L}$ no es $NP$-Completo.

Las demostraciones conocidas del teorema anterior implican la definición de un lenguaje "artificial" que actúe como separador para la clase $NPI$ [4]**.** No se conoce un problema natural que pertenezca a la clase $NPI$, aunque existen algunos problemas interesantes de los cuales se conjetura, o se ha conjeturado, pertenecen a la clase $NPI$.

## 3.1. PROBLEMA DE LA PRIMALIDAD

**Nombre del Problema:** Prueba de Primalidad (Primality Testing; PRIMES).

**Entrada:** Un número $k \in \mathbb{N}$.

**Problema:** Determinar si $k$ es primo o no.

Uno de los primeros candidatos para ser un miembro "natural" de la clase $NPI$ fue el problema de la primalidad.

El interés en determinar si un número es primo o no surgió mucho antes de la existencia de los computadores debido a que los números primos son necesarios para probar algunas conjeturas matemáticas [2].

Gran cantidad de investigadores buscaron pruebas de primalidad eficientes partiendo del hecho que ya estaba demostrado que el problema PRIMES hacia parte de la clase $NP$. Dentro de las aproximaciones al problema están el algoritmo de Miller y el algoritmo probabilístico de Rabin-Scott [3]**;** sin embargo, fue solo hasta el año 2002 cuando Manindra Agrawal, Neeraj Kayal y Nitin Saxena exhibieron en [1] un algoritmo determinista de tiempo polinomial que determina si

un número es compuesto o primo, demostrando por ende que el problema PRIMES pertenece a $P$ y descartando entonces que sea un miembro de $NPI$.

A diferencia del problema recién mencionado, existen otros lenguajes que aún persisten como candidatos naturales para ser miembros de la clase $NPI$.

Destacan el problema de determinar si dos grafos finitos son isomorfismos (el cual será estudiado en profundidad posteriormente), el problema de determinar si dos grupos finitos son isomorfos, el problema de factorizar números enteros, la determinación de un ganador en juegos de paridad, el problema del logaritmo discreto y otros problemas relacionados con la criptografía.

Considere el problema a continuación:

## 3.2. FACTORIZACIÓN DE NÚMEROS ENTEROS

**Nombre del Problema:** Factorización de enteros (Integer Factorization. FACTORING).

**Entrada:** Un número $n \in \mathbb{N}$.

**Problema:** Determinar si $n$ tiene un divisor no trivial.

Dado un número natural $n$ hallar su factorización en números primos consiste en escribir a $n$ como un producto de potencias de sus divisores primos. Dicho de otro modo, si $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$, donde $p_i$ es primo y para todo $i \geq 1$ se tiene que $e_i \geq 1$

Calcular la factorización de $n$ consiste en calcular el vector $(p_1, e_1, p_2, e_2, \dots, p_k, e_k)$. La existencia de esta descomposición está garantizada por el teorema fundamental de la aritmética que sostiene que todo número natural $k > 1$ puede ser escrito como un único producto de números primos aunque,

como se puede notar, este no da pista alguna sobre el modo de determinar dicho producto.

Ahora, descomponer (o factorizar) no trivialmente un número natural $n$ consiste en escribirlo de la forma $n = ab$ de modo que $1 < a < n$ y $1 < b < n$ donde $a$ y $b$ son números no necesariamente primos.

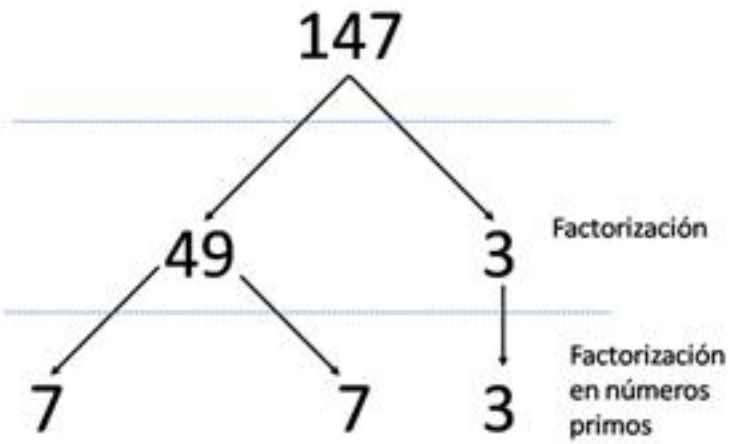


**Ilustración 5 Esquema de la factorización de un número natural**

El problema de factorizar enteros encuentra sus instancias más complejas en números semiprimos largos donde sus únicos factores $a$ y $b$ son números primos largos de un tamaño similar pero no igual. Este hecho fue capitalizado por los creadores del, ampliamente difundido, protocolo criptográfico RSA el cual explota la intratabilidad del problema de factorizar números enteros largos [23], aunque es importante aclarar que no se ha demostrado que dicho procedimiento no exista.

Los algoritmos más eficientes que existen en la actualidad para factorizar enteros en su orden son:

1. **General Number Field Sieve (GNFS):** es el algoritmo más rápido para factorizar números de más de cien dígitos [25].
2. **Quadratic Sieve:** es el segundo algoritmo más rápido, es más veloz que el algoritmo GNFS cuando se procesan instancias de menos de 100 dígitos [25].
3. **Elliptic Curved Method (ECM):** es considerado un algoritmo de propósito específico, es el procedimiento más rápido para factorizar números que no exceden los 25 dígitos [25].

Por otra parte, en el caso hipotético (actualmente) de poder desarrollar la computación cuántica el problema sería resuelto en tiempo polinomial gracias al algoritmo de Shor [27].

Respecto a la complejidad de la versión de decisión del problema de factorizar enteros se considera que está en la clase $NP \cap coNP$ [13]. Apoyado en el teorema de Shor, se sabe que el problema está en la clase $BQP$. Sin embargo, se sospecha que no hace parte de las clases $P$ o $NPC$ [2] debido a que esto implicaría que $NP = coNP$.

## 3.3. ISOMORFISMO DE GRUPOS

**Nombre del Problema:** Isomorfismo de Grupos (Group Isomorphism. GROUP-ISO).

**Entrada:** Dos grupos finitos $S_1 = (G, \star)$ y $S_2 = (H, \diamond)$.

**Problema:** Determinar si existe un isomorfismo entre $S_1$ y $S_2$.

Un grupo es una estructura algebraica $S = (G, \star)$ donde $G$ es un conjunto y $\star$ es una operación binaria que combina dos elementos de $G$ para asignarles un tercer elemento también de $G$.

Dicha estructura debe cumplir con cuatro condiciones denominadas axiomas de grupo que son:

i. Clausura: $\forall x, y \in G,\ x \star y \in G$.

ii. Asociatividad: $\forall x, y, z \in G,\ x \star (y \star z) = (x \star y) \star z$.

iii. Identidad o existencia de elemento neutro:

$$\exists! e : \forall x \in G\ , x \star e = e \star x = x.$$

iv. Invertibilidad o simetría: $\forall x \in G\ \exists \bar{x}\ \in G : x \star \bar{x} = \bar{x} \star x = e$.

Todo grupo $S = (G,\ \star)$ tiene la propiedad de que cada uno de sus $n$-elementos admite un sistema $X$ de como máximo $m = \lfloor \log(n) \rfloor$ generadores (donde $X \subseteq G$ es un sistema de generadores para $S$ sí y solo si todos los elementos de $G$ pueden ser obtenidos como el resultado de componer (o de operar) algunos de los elementos de $X$).

Se dice que dos grupos $S_1 = (G,\ \star)$ y $S_2 = (H,\ \diamond)$ son isomorfos si existe una función biyectiva $f: G \to H$ de modo tal que $\forall u, v \in G$ se tiene que:

$$f(u \star v) = f(u) \diamond f(v)$$

Y se nota como: $S_1 = (G,\ \star) \cong S_2 = (H,\ \diamond)$ o simplemente como $S_1 \cong S_2$.

Gracias a la propiedad de los grupos mencionada anteriormente, es posible obtener, en tiempo sub-exponencial, un sistema de generadores para un grupo $S = (G,\ \star)$ mediante la consideración de todas las posibles $m$-combinaciones de $G$ hasta que se halle una combinación tal que constituya un sistema de

generadores para $S$. Este hecho es aprovechado para detectar si dos grupos son isomorfos en el siguiente algoritmo:

1. Determinar un sistema de generadores $g_1, g_2, \dots, g_k$ para $S_1$.
2. Asociar a cada elemento de $x \in G$ su representación $x = g_{i1} \star g_{i2} \star \dots \star g_{ih}$.
3. Para cada $m$-permutación $z_1, z_2, \dots, z_m$ de $H$ evaluar si la función $f: G \to H$ es un isomorfismo entre $S_1$ y $S_2$, donde $f$ se define por

$$f(x) = \begin{cases} z_i & Si\ x = g_i \\ z_1 \diamond z_2 \diamond \dots \diamond z_m & Si\ x = g_{i1} \star g_{i2} \star \dots \star g_{ih} \end{cases}$$

4. Si al menos existe una permutación para la cual $f$ es un isomorfismo, entonces se acepta, de lo contrario se rechaza.

Revisar si una $m$-combinación es un sistema de generadores toma $O(n^2 m!)$ pasos, entonces el procedimiento completo tomará $O(n^2 m n^m)$. Además, verificar si la función $f$ es un isomorfismo requiere $O(n^2 m)$ pasos, mientras el número de $m$-permutaciones esta acotado por $O(n^{2m})$. De este modo el algoritmo requiere, al procesar un input de tamaño $n$

$$O(n^2 m n^m + n^2 m n^{2m}) \subseteq O(n^2 m n^{2m}) \subseteq O(n^{2\log(n)+3})$$

Unidades de tiempo. Del problema de isomorfismo de grupos se sabe que puede ser resuelto en espacio $O(\log^2 n)$, y que puede ser reducido en tiempo polinomial

al problema de isomorfismo de grafos [4]. Sin embargo, y aún a pesar del algoritmo exhibido (el cual requiere tiempo estrictamente menor que exponencial, pero no polinómial), se desconoce su ubicación dentro de $NP$, (esto es: se desconoce si esta en $P$, en $NPI$ o en $NPC$).

# 4. METODOLOGÍAS PARA SOLUCIONAR PROBLEMAS $NP \setminus P$

*"Cuanto mayor es la dificultad, mayor es la gloria."*

*Marco Tulio Cicerón*

En la computación al igual que en muchas áreas no basta con ser eficaz, es imprescindible ser eficiente. Basadas en esta premisa surgen las nociones de algoritmo bueno y la de algoritmo malo. Entendiéndose como una rutina buena aquella que además de resolver el problema, lo hace con una baja demanda de los recursos computacionales a disposición y como un procedimiento malo a aquel que falla al intentar resolver la tarea a la cual se enfrenta o que para lograrlo exige demasiados recursos; de manera intuitiva, se consideran como problemas fáciles a aquellos para los cuales existen buenos algoritmos para resolverlos, y como problemas difíciles aquellos donde el mejor procedimiento para solucionarlos es un mal algoritmo.

En la actualidad, la convención sugiere que un problema se puede considerar como fácil si puede ser resuelto por una rutina computacional que se ejecute en un tiempo acotado por una función polinómica, es decir, que pertenezca a la clase $P$.

Si se toma que $P = NP$, se tendrá entonces que todos los problemas de la clase $NP$ serán fáciles.

Ahora, asumiendo que $P \neq NP$, se considera que todos los problemas $\mathcal{L} \in NP \setminus P$ son difíciles y por ende no existen buenos algoritmos para resolverlos. Sin embargo, muchos de estos problemas son de vital importancia

práctica (sobre todo en labores de optimización), razón por la cual es absolutamente necesario hallar una solución incluso si ella no es tan buena como se quisiera.

Cuando se hace frente a un problema difícil y es necesaria una respuesta puede intentarse alguno de los tres enfoques que serán mencionados a continuación.

1. Usar el algoritmo determinista con el que se cuenta si la instancia del problema es pequeña o la rutina computacional tiene un comportamiento aceptable para dicha instancia.
2. Buscar soluciones cuasi óptimas en tiempo polinomial ya que en la práctica suelen ser lo suficientemente buenas.
3. Restringir las instancias del problema de modo que se puedan aislar casos especiales importantes del problema que se resuelvan en tiempo polinomial.

A continuación se hará énfasis en torno a la búsqueda de soluciones cuasi óptimas y al enfoque que sugiere el restringir las instancias del problema.

## 4.1. SOLUCIONES CUASI ÓPTIMAS

Aunque, los problemas de optimización son problemas de búsqueda, se puede aprovechar el hecho que también pueden ser planteados como problemas de decisión. Este enfoque es útil cuando hallar la solución óptima es difícil o resulta muy costosa en términos de recursos de cómputo. Para alcanzar respuestas cuasi óptimas se puede recurrir al uso de:

### 4.1.1. Algoritmos de Aproximación

Los algoritmos de aproximación constituyen uno de los caminos usados para obtener soluciones a problemas de búsqueda duros.

Para alcanzar una respuesta válida, el algoritmo renuncia a determinar la respuesta óptima (la cual es, en términos computacionales, altamente difícil de

alcanzar) a cambio de determinar en tiempo sub-exponencial una respuesta sub-óptima que se ajuste a los requerimientos de la aplicación práctica.

Los algoritmos de aproximación se caracterizan por presentar una relación de proximidad $\rho(n)$ definida como:

$Max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$, para toda entrada al algoritmo de tamaño $n$

Donde $C$ es el costo de producir una solución con el algoritmo de aproximación y $C^*$ es el costo de obtener la respuesta óptima al problema [6]. De un algoritmo con un índice de proximidad $\rho(n)$ se dice que es un $\rho(n)$-algoritmo de aproximación.

Como es evidente, la proporción $\rho(n)$ nunca será menor que uno; así mismo, para valores de $\rho(n) \neq 1$ se obtienen soluciones sub-optimas, mientras que un $1$-algotimo dará como salida la respuesta óptima del problema.

Para algunos problemas $\rho(n)$ es un valor constante, mientras que para otros, en el mejor algoritmo conocido que se ejecuta en tiempo polinomial, $\rho(n)$ crece en función del tamaño $n$ de la entrada.

Por otra parte, un sistema de aproximación para un problema de optimización es un algoritmo de aproximación que además de tomar como entrada la instancia $x$ del problema, considera también un parámetro $\varepsilon > 0$ de modo tal que con input $(x, \varepsilon)$ el sistema se comporta como un $(1 + \varepsilon)$-algoritmo de aproximación [6].

### 4.1.2. Algoritmos Aleatorios o Probabilísticos

Un algoritmo aleatorio es aquel que recibe además de los datos de entrada un flujo constante de bits aleatorios que son usados por el algoritmo para tomar decisiones aleatorias [19].

Características de los algoritmos aleatorios son la permisividad al fallo, siempre y cuando lo hagan con una baja probabilidad, el hecho que pueden dar respuestas diferentes con una misma entrada y que su rendimiento sobre una entrada fija es, en términos de tiempo de computo, una variable aleatoria [19] que depende de las decisiones tomadas durante la ejecución del algoritmo [26].

Para muchos problemas la mejor rutina conocida es un algoritmo aleatorio, incluso, si es posible comparar el desempeño de un algoritmo determinista y uno probabilista para solucionar una misma tarea, el algoritmo aleatorio es frecuentemente más simple de implementar y de mayor eficiencia que el determinista. Incluso, en algunos casos, el mejor algoritmo determinista para un problema dado es obtenido a partir de la eliminación de la aleatoriedad (Derandomization en inglés) de un algoritmo probabilista.

La potencia introducida al algoritmo por la posibilidad de tomar decisiones aleatorias es tal que puede convertir un algoritmo determinista ingenuo con un mal desempeño en un procedimiento con un rendimiento con alta probabilidad de ser bueno en la mayoría de las instancias posibles.

### 4.1.3. Heurísticas y Meta-heurísticas

Las heurísticas son procedimientos que se implementan con la esperanza de alcanzar un buen desempeño en términos computacionales, a expensas de la exactitud y/o la precisión de la respuesta obtenida. Estos algoritmos, además, suelen estar basados en la experiencia y, a priori, se desconoce el comportamiento que presentaran ante diferentes tipos de instancias del problema al cual se enfrentan. Con el uso de heurísticas existe la posibilidad, al menos en teoría, de determinar la solución óptima para un problema de búsqueda, sin

embargo, es algo que en la práctica suele ocurrir poco debido a que esta clase de procedimientos tienden a “atascarse” en soluciones localmente óptimas [7].

Buscando solucionar el problema recién mencionado, las meta-heurísticas introducen reglas sistemáticas que les permiten, en la mayoría de los casos, continuar con la búsqueda del valor óptimo dejando de lado los óptimos locales [7], aunque esto no garantiza que una meta-heurística halle siempre el valor óptimo para un problema de búsqueda de optimización.

Glover y Laguna definen en [12] el término meta-heurística como “una estrategia maestra que guía y modifica otra heurística para producir soluciones más allá de aquellas que son normalmente obtenidas en una búsqueda de optimalidad local. La heurística guiada por meta-estrategias puede ser un procedimiento de alto nivel (de complejidad conceptual) o puede simplemente incluir una descripción de los pasos a seguir para transformar una solución en otra junto con una regla de evaluación asociada”. Glover y Laguna también proponen un método para clasificar meta-heurísticas con base en los parámetros $\alpha|\beta|\gamma$ a saber:

i. $\alpha$: Uso de memoria adaptativa.

ii. $\beta$: La forma en la cual se explora la vecindad.

iii. $\gamma$: La cantidad de soluciones factibles que son traspasadas de una iteración a la siguiente.

Cuyos valores pueden ser

- $\alpha = \begin{cases} A & \text{Si la meta-heurística tiene memoria adaptativa} \\ M & \text{Si el método no tiene memoria} \end{cases}$

- $\beta = \begin{cases} N & \text{Para un método usado en alguna clase de búsqueda sistemática} \\ S & \text{Para un método que utiliza muestreo al azar} \end{cases}$

- $\gamma = \begin{cases} 1 & \text{Si el método hace un movimiento en cada iteración} \\ P & \text{Para un enfoque poblacional con una población de tamaño } P \end{cases}$

Entre las meta-heurísticas más usadas destacan las redes neuronales artificiales (RNA), los algoritmos genéticos (AG) y la búsqueda tabú.

## 4.2. RESTRICCIONES SOBRE LAS INSTANCIAS DEL PROBLEMA

Cuando se hace frente a un problema computacional, es frecuente reducir su estudio a la búsqueda de un algoritmo capaz de solucionar cualquier instancia posible del problema planteado de manera eficiente, ignorando que muchas situaciones prácticas imponen restricciones adicionales sobre el dominio del mismo las cuales limitan el tipo de instancias que pueden aparecer.

Dado un problema computacional siempre es posible (y en ocasiones es útil) introducir *constraints* adicionales que restrinjan el conjunto de instancias, de esta manera es posible obtener (definir) nuevos problemas $\mathcal{L}'$ tales que $\mathcal{L}' \subsetneq \mathcal{L}$.

Si el problema $\mathcal{L}'$ es más fácil y contiene todas las instancias interesantes (todas las que puedan ocurrir en la práctica) se ha dado un paso adelante en la solución del problema computacional estudiado.

### 4.2.1. El Poder de Restringir

En un sinnúmero de oportunidades la búsqueda de un algoritmo capaz de resolver cualquier instancia de un problema dado resulta infructuosa. Ahora bien, existen algunos inputs que en la práctica no es posible que ocurran, por ello, limitarse a evaluar únicamente instancias probables se erige como una metodología eficaz, en la mayoría de casos, para abordar problemas altamente demandantes en términos computacionales especialmente cuando dichos problemas están íntimamente relacionados con ambientes prácticos.

Ahora, antes de continuar, se considera necesario introducir primero algunos conceptos propios de la teoría de grafos debido a que en lo sucesivo del presente escrito serán abordados problemas propios de esta temática.

De un modo formal, un grafo es un par de conjuntos disyuntos $G = (V, E)$ ($V \cap E = \emptyset$), donde los elementos del conjunto $V$ son llamados "vértices" y los elementos del conjunto $E$ se denominan "aristas" y se satisface que $E \subseteq [V]^2$.

Desde una óptica práctica, los vértices representan, en un modelo, a los elementos del sistema mientras que las aristas representan las relaciones binarias existentes entre cada par de dichos elementos. Cuando en un grafo $G = (V, E)$, una arista $e \in E$ establece un vínculo entre dos vértices $a$, $b$ se da origen a una relación de adyacencia de vértices. De $a$ y $b$ se dirá que son vértices adyacentes mientras que de la arista $e = (a, b)$ se dirá que incide (es incidente) sobre $a$ y $b$.

Existen diversas formas de representar un grafo; sin embargo, en este punto del documento se hará énfasis en la representación gráfica y posteriormente, a lo largo del escrito, se introducirán algunas de las otras representaciones.

Un grafo se representa gráficamente mediante la elaboración de un diagrama en el cual cada uno de los elementos del conjunto de vértices es representado por un punto específico que le diferencia de los demás, mientras que cada una de las aristas es caracterizada por un segmento de línea continuo que une a dos vértices no necesariamente distintos. Cuando se habla de grafos dirigidos el segmento de línea llevará en uno de sus extremos una cabeza de flecha que indicará la dirección de la relación que es representada por la arista. Además, resulta ideal que en el esquema los segmentos que representan a las aristas no tengan puntos

en común diferentes a sus puntos terminales que vienen siendo los vértices del grafo.

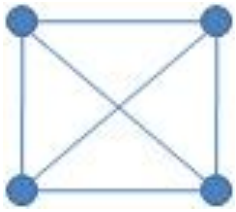

**Ilustración 6 Representación grafica de un grafo**

Dado un problema algorítmico $\mathcal{L}$, este define un conjunto $I(\mathcal{L})$ que es aquel formado por todas las instancias posibles de $\mathcal{L}$. Ahora, dado $A \subseteq I(\mathcal{L})$ se puede considerar el problema $\mathcal{L}[A]$ definido por:

**Nombre del problema:** $\mathcal{L}[A]$

**Entrada:** $x \in A$

**Problema:** Decida si $x \in \mathcal{L}$

Al definir $\mathcal{L}[A]$ se han eliminado algunas de las instancias de $\mathcal{L}$ (siempre y cuando se cumpla que $A \subsetneq I(\mathcal{L})$) y es posible que $\mathcal{L}[A]$ resulte ser fácil en comparación con $\mathcal{L}$ (si en la definición de $A$ han sido eliminadas las instancias más difíciles de $\mathcal{L}$). En lo que queda de este capítulo se mostrará cómo, en algunas ocasiones, esta intuición resulta correcta mientras en otras no lo es tanto.

#### *4.2.1.1. Cuando restringir funciona*

En esta sección se estudiara un caso en el cual restringir funciona. Son de especial interés para este trabajo problemas algorítmicos relacionados con grafos, y particularmente, las restricciones planares de los mismos.

Considérese el siguiente problema:

**Nombre del Problema:** Conteo de matchings perfectos. #MATCH.

**Entrada:** Un grafo $G$.

**Problema:** Calcular el número de matchings perfectos en un grafo $G$.

Dado un grafo $G = (V, E)$, un matching en $G$ es un conjunto de aristas que no comparten vértices en común. De un matching $M$ se dice que es perfecto si todos y cada uno de los vértices del grafo son incididos por una y solo una arista del matching.

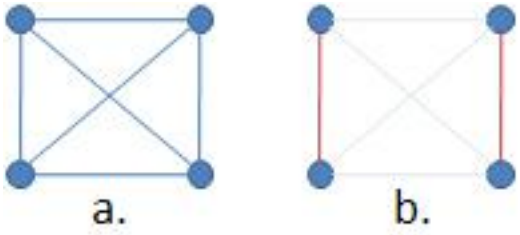


**Ilustración 7 Grafo y uno de sus matchings perfectos. a. Grafo b. Matching perfecto de a.**

Valiant en [32] probó que #MATCH es un problema $\#P$-completo, por otro lado Toda probó en [29] que todo problema $\#P$-completo es más difícil que cualquier problema en la jerarquía polinomial. Con esto se asegura que #MATCH es aún más difícil que todo problema dentro de la clase $NPC$.

Ahora, considérese el problema #MATCH$[\mathcal{P}]$ definido por:

**Nombre del Problema:** Conteo de matchings perfectos en grafos planos. #MATCH$[\mathcal{P}]$.

**Entrada:** Un grafo plano $G$.

**Problema:** Calcular el número de matchings perfectos en un grafo $G$.

Claramente #MATCH$[\mathcal{P}]$ es la restricción planar de #MATCH. Surge entonces la pregunta ¿qué tan difícil es #MATCH$[\mathcal{P}]$?

Kasteleyn probó en [20] que #MATCH$[\mathcal{P}]$ pertenece a la clase $P$. Esto significa que la restricción planar del (muy difícil) problema #MATCH es un problema fácil ya que puede ser resuelta en tiempo polinomial.

El ejemplo anterior muestra que restringir puede convertir problemas difíciles en problemas fáciles. El quid del asunto es que la restricción sea relevante, es decir, que el conjunto de instancias sea representativo. Particularmente, la representatividad de la clase de grafos planos está demostrada en el hecho que algunas aplicaciones solo precisan considerar este tipo de grafos. Así por ejemplo, a la física estadística que estudia el modelo de ISING solo le interesa el conteo de matching perfectos en grafos planos, y este problema, precisamente, es resuelto por el famoso algoritmo enunciado por Kasteleyn en [20] quien de por sí, era un físico estadístico.

#### *4.2.1.2. Cuando restringir NO funciona*

Considérese el problema:

> **Nombre del Problema:** Ciclo hamiltoniano. HAMCYCLE.
>
> **Entrada:** Un grafo $G$.
>
> **Problema:** Decidir si el grafo $G$ tiene un ciclo de Hamilton.

Dado un grafo $G = (V, E)$, un ciclo de Hamilton es una secuencia de aristas adyacentes que visita todos los vértices del grafo y el último vértice visitado es adyacente al primero.

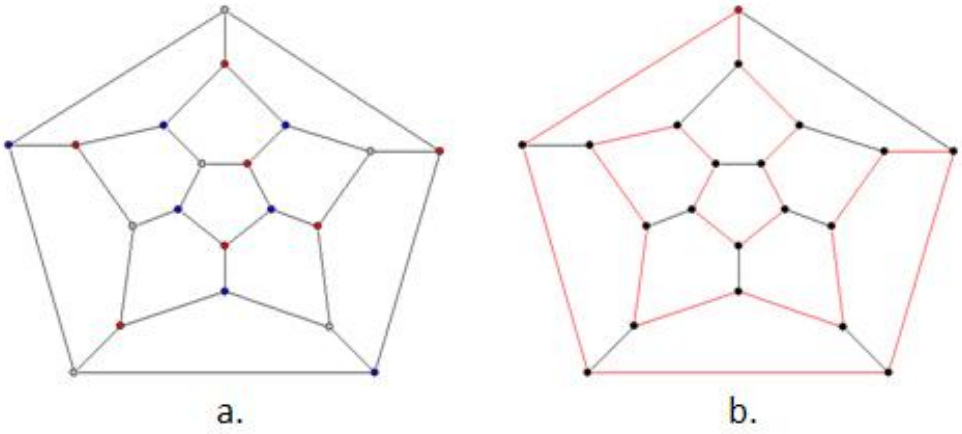


**Ilustración 8 Grafo y circuito de Hamilton. a. Grafo b. Circuito de Hamilton en el grafo a.**

El problema HAMCYCLE es un caso especial de la versión de decisión del problema del agente viajero (Travelling SalesmanProblem. TSP) [28] donde la ponderación de las aristas (que en TSP representa la distancia entre las ciudades) es reemplazada por el establecimiento de una constante que indica la adyacencia entre dos vértices e infinito en el caso que los vértices no sean adyacentes. HAMCYCLE es un problema $NP$-Completo [11].

Garey, Johnson y Tarjan probaron en [10] que la restricción planar del problema HAMCYCLE (nótese como HAMCYCLE$[\mathcal{P}]$) es $NP$-Completa. Lo cual implica que el problema HAMCYCLE y su restricción HAMCYCLE$[\mathcal{P}]$ son igualmente difíciles.

En los próximos capítulos la atención será centrada en el problema del isomorfismo de grafos, el cual, a diferencia de HAMCYCLE, se conjetura que hace parte de la clase $NPI$. En este estudio se considerarán algunas de sus restricciones y se hará un énfasis particular en la restricción planar de este problema.

# 5. EL PROBLEMA DEL ISOMORFISMO DE GRAFOS

Cuando Leonhard Euler publicó en 1736 su trabajo “Solutio problematis ad geometriam situs pertinentis” sobre el afamado problema de “los puentes de Königsberg” en el cual empleaba un esquema que simplificaba el sistema formado por las dos islas, las dos riberas y los siete puentes sobre el río Pregel seguramente jamás imaginó el potencial que tenía aquella sencilla abstracción.

La representación sugerida por Euler, quien nunca llegó a dibujarla, recibió el nombre de grafo y fue introducida en áreas como la química teórica gracias a trabajos como los de Arthur Cayley, en el análisis de circuitos a través de los estudios desarrollados por Gustav Kirchhoff y, de un modo similar, llegaron a penetrar en otras ciencias como las biológicas y las sociales a tal punto que llegaron a convertirse en elementos ubicuos y recurrentes en las tareas de modelado de sistemas sin hacer distinción alguna respecto a su origen.

En la teoría de grafos se condensa el estudio matemático de la estructura y de las relaciones existentes entre dos o más grafos. Entre los problemas destacados que aborda esta área de las matemáticas destaca el isomorfismo cuya importancia radica en que gracias a ellos es posible extender percepciones de un fenómeno a otro y, en particular, los grafos no son una excepción. Por otra parte, no siendo suficiente con esto, el problema de isomorfismo de grafos resulta ser un objeto de estudio interesante no solo por su aplicabilidad en áreas como la química informática, la minería de datos o el diseño electrónico automatizado sino que también por la “extraña” (por llamarla de algún modo) posición que ocupa dentro de la teoría de la complejidad computacional.

## 5.1. PLANTEAMIENTO DEL PROBLEMA

**Nombre del problema:** Problema de isomorfismo de grafos (Graph isomorphism Problem, GIP)

**Entrada:** Dos grafos finitos $G$ y $H$

**Problema:** Decidir si $G$ y $H$ son grafos isomorfos.

El problema de isomorfismo de grafos se puede enunciar informalmente de un modo muy sencillo: evaluar si dos grafos que lucen diferentes son, estructuralmente, el mismo grafo.

**Definición 7: Grafos Isomorfos**
Dados dos grafos $G$ y $H$, se dice que son grafos isomorfos ($G \equiv H$) si

$$\exists f \mid \forall a,b \in V_G, (a,b) \in E_G \Leftrightarrow (f(a),f(b)) \in E_H$$

Un isomorfismo es una función biyectiva $f: V_G \rightarrow V_H$ que preserva la relación de adyacencia existente entre los vértices de un grafo en el otro grafo. Y, además, $f^{-1}$ también es un isomorfismo.

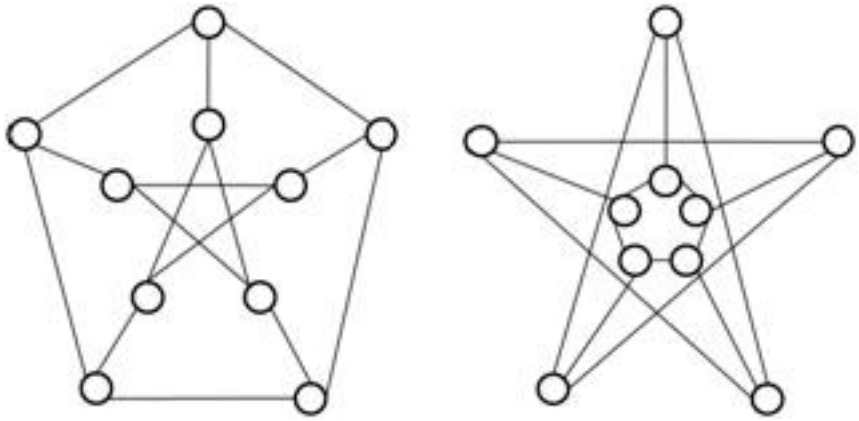

**Ilustración 9 Grafos isomorfos**

## 5.2. COMPLEJIDAD DEL PROBLEMA DE ISOMORFISMO DE GRAFOS

Como se mencionó previamente, el problema de isomorfismo de grafos ocupa un lugar especial dentro del universo de la teoría de la complejidad computacional;

resulta sencillo demostrar que es un lenguaje contenido en la clase de complejidad $NP$ pero, contrario a lo que ocurre con la mayoría de los elementos de la clase, no ha podido ser catalogado como un problema contenido en $P$ o como $NP$-Completo siendo esta la razón por la cual emerge como uno de los candidatos naturales más sólidos para corroborar el teorema de Ladner.

GIP no se considera contenido en $P$ porque, a día de hoy, no existe una máquina de Turing determinística capaz de decidir si dos grafos dados cualquiera son isomorfos [21], en contraposición no se le considera $NP$-Completo porque de serlo la jerarquía polinomial colapsaría en el segundo nivel y además está demostrado que la versión de conteo del problema es polinomialmente equivalente a GIP y existe la fuerte creencia que este fenómeno no se presenta en los problemas $NP$-Completos [31].

Adicional a lo recién mencionado, es bien sabido que cuando se restringe el problema de isomorfismo de grafos a clases (de grafos) con ciertas propiedades fijas (nótese el problema restringido como GIP[$\mathcal{R}$]) se presentan dos fenómenos a saber. Por una parte se encuentran clases denominadas isomórficamente completas como la de los grafos eulerianos bipartitos o los grafos cordales para las cuales resolver GIP[$\mathcal{R}$] es tan difícil como resolver GIP. Mientras que, en el otro extremo se encuentran clases como las de grado acotado, árboles o grafos planos para las cuales GIP[$\mathcal{R}$] es un elemento de la clase $P$.

## 5.3. RESOLVIENDO EL PROBLEMA DE ISOMORFISMO DE GRAFOS

Si bien es cierto que se cree que GIP no es un problema $NP$-Completo, también resulta cierto que el hecho que no ha sido posible demostrar que sea un lenguaje

de la clase $P$ es una prueba que indica que no puede ser resuelto, a día de hoy, en tiempo polinomial, aún cuando es uno de los problemas que más ha sido abordado.

Algoritmos que intentan resolver GIP abundan en la literatura especializada, siendo el resultado obtenido por Luks y Zemlyachenko de $e^{\sqrt{cn \log n}}$ la mejor cota superior conocida en la actualidad aunque no existen evidencias que indiquen que esta cota sea óptima [31]. Ahora bien, cuando se revisan los algoritmos desarrollados para resolver GIP se pueden distinguir dos metodologías para hacer frente al problema, una de ellas consiste en computar los dos grafos de tal manera que se busca entre el conjunto de permutaciones un isomorfismo entre ellos mientras que la otra metodología sugiere procesar los grafos de modo tal que de cada uno de ellos se obtenga un código único para cada grafo el cual será igual si se trata de dos grafos isomorfos.

Por otra parte, y como ya se mencionó previamente, existen clases de grafos para las cuales el problema de isomorfismo de grafos puede ser resuelto en tiempo polinomial; y justamente el siguiente apartado de este capítulo se centrará en presentar un algoritmo que resuelve el problema de isomorfismo de árboles de una manera eficiente. La razón por la cual se opta por presentar este algoritmo es que constituye la piedra angular de otros procedimientos desarrollados para abordar el isomorfismo de grafos en clases más complejas como la de grafos planos la cual es la clase central de estudio dentro de este proyecto.

## 5.4. ISOMORFISMO DE ÁRBOLES: UN ALGORITMO POLINOMIAL

Existen problemas que pertenecen a la teoría de grafos que son intratables algorítmicamente; sin embargo, cuando las instancias de estos son restringidas a la clase especial de grafos denominada “árboles” la mayor parte de los problemas pueden ser resueltos en tiempo polinomial. Esto es importante para este trabajo porque el objeto de estudio de este, el isomorfismo de grafos, no es una excepción

a la situación recién expuesta. A continuación serán introducidos algunos conceptos necesarios para poder presentar un algoritmo determinista que resuelve GIP restringido a árboles.

**Definición 8: Árbol**
Un grafo $T = (V, E)$ será llamado árbol si es un grafo simple, no dirigido y no tiene ciclos.

Si dos vértices $v, w \in V(T)$, la arista $e = (v, w) \in E(T)$ será denominada rama. Por otra parte, todo vértice $v$ con grado igual a uno que pertenezca a $T$ será llamado hoja.

**Definición 9: Árbol con raíz**
Un árbol con raíz es un par $(T, r)$ donde $T$ es un árbol y $r \in V(T)$ es un vértice distinguido llamado raíz.

Si $e = (v, w) \in E(T)$ es una rama y el vértice $v$ está en el único camino desde $w$ hasta la raíz, se dice que $v$ es el *padre* de $w$ y a su vez, $w$ es el *hijo* de $v$.

**Definición 10: Árbol Plantado**
Un árbol plantado $(T, r)$ es un árbol con raíz y un dibujo $T$ en el plano. En ese dibujo $T$ la raíz del árbol $r$ se indica con una flecha que apunta había abajo, y los hijos de cada uno de sus vértices estás situados encima del vértice correspondiente.

Diferenciar un vértice $r$ de un árbol $T$ y fijarlo como la raíz del árbol impone un ordenamiento parcial sobre $V(T)$. Con esto en mente se puede definir un árbol plantado como una tripla $(T, r, v)$ donde $v$ es una colección de ordenaciones lineales, una ordenación lineal para el conjunto de los hijos de cada vértice.

De acuerdo al tipo de árbol la definición de la función isomorfismo varía del siguiente modo:

$f: V(T) \longrightarrow V(T')$ es una función isomorfismo si es una función biyectiva y además:

- Isomorfismo de Árboles: Dados $T$ y $T'$ dos árboles

  $\forall x, y \in V(T), (x, y) \in E(T) \Leftrightarrow (f(x), f(y)) \in E(T')$

- Isomorfismo de Árboles con raíz: Dados $(T, r)$ y $(T', r')$ dos árboles con raíz

  $\forall x, y \in V(T), (x, y) \in E(T) \Leftrightarrow \big(f(x), f(y)\big) \in E(T') \wedge f(r) = r'$

- Isomorfismo de árboles plantados: Dados $(T, r, v)$ y $(T', r', v')$ dos árboles con raíz

  $\forall x, y \in V(T), (x, y) \in E(T) \Leftrightarrow \big(f(x), f(y)\big) \in E(T') \wedge f(r) = r'$

  $\wedge \forall v \in V(T)$ *se preserva la ordenación izquierda a derecha de los hijos*

### 5.4.1. Construyendo el Algoritmo

El procedimiento que se sigue en la de la carrera para construir el algoritmo para decidir si dos árboles son isomorfos tiene como punto de partida los árboles plantados y la definición de la función isomorfismo para los mismo debido a que al ser más restrictivos, codificarlos resulta menos complicado que con casos más generales. Dados $T$ y $T'$, el algoritmo consiste, básicamente, en asignar una secuencia de $0s$ y $1s$ de longitud $2n$ (donde $n$ es la cantidad de vértices del grafo) llamada código del árbol $T$. Árboles isomorfos producen secuencias idénticas, mientras que árboles no isomorfos producen secuencias distintas.

**Algoritmo 1: Isomorfismo de árboles plantados**

1. Generar el código del árbol para cada $T$

   1.1. A cada hoja del árbol (si la raíz tiene grado 1 obviarla) se le asigna el código $01$.

   1.2. Generar para cada vértice del árbol su código correspondiente.

      1.2.1. Sea $v$ un vértice con hijos $v_1, v_2, \dots, v_t$ (escritos de izquierda a derecha). Si $A_i$ es el código del hijo $v_i$, entonces el vértice $v$ recibe el código $0A_1A_2 \dots A_t1$.

   1.3. El código del árbol será el código de la raíz.

2. Comparar el código de $T$ con el código de $T'$. Si los códigos son iguales los árboles son isomorfos, en caso contrario, no son isomorfos.

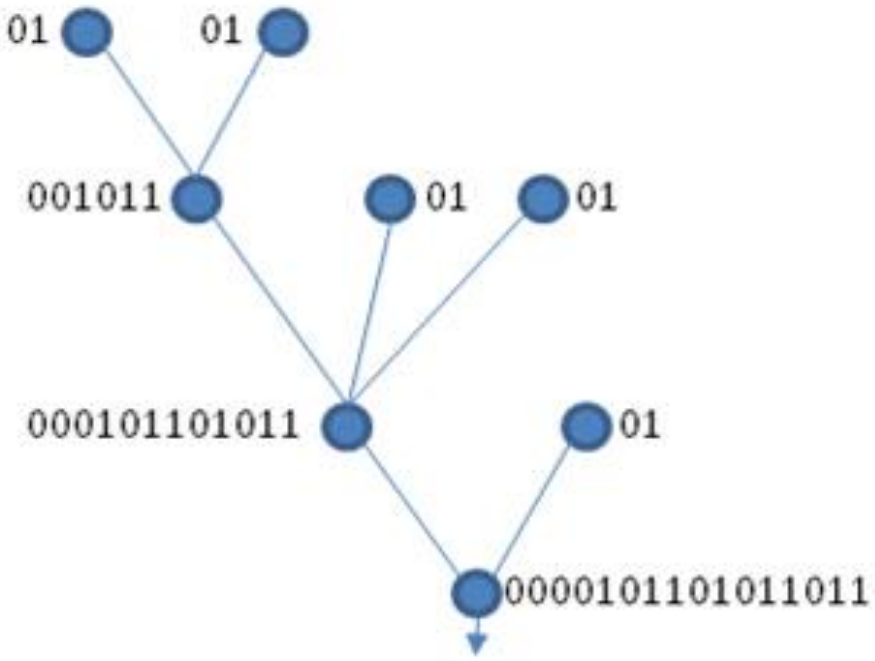


**Ilustración 10 Árbol y etiquetas que conforman su código único**

**Algoritmo 2: Isomorfismo de árboles con raíz**

1. Generar el código del árbol para cada $T$

   1.1. A cada hoja del árbol (si la raíz tiene grado 1 obviarla) se le asigna el código $01$.

   1.2. Generar para cada vértice del árbol su código correspondiente.

      1.2.1. Supóngase que cada hijo $w$ de un vértice $v$ ya tiene asignado un código $A(w)$. Se denota por $w_1, w_2, \ldots, w_t$ a los hijos de $v$, de manera que $A(w_1) \leq A(w_2) \leq \cdots \leq A(w_t)$.[3] El vértice $v$ recibe el código $0A_1A_2 \ldots A_t1$.

   1.3. El código del árbol será el código de la raíz.

2. Comparar el código de $T$ con el código de $T'$. Si los códigos son iguales los árboles son isomorfos, en caso contrario, no son isomorfos.

### 5.4.2. El Algoritmo

Como se ha podido notar en los dos algoritmos ya exhibidos, conocer la raíz del árbol constituye una necesidad ya que en dicho nodo se forma el código descriptor del árbol a partir de la recopilación de los códigos de cada uno de los vértices del grafo. Ahora, dado un árbol $T$, es posible designar cualquier $v \in V(T)$ como la raíz del árbol, lo cual lleva a pensar que bastaría con definir una forma estándar

---

[3] Aquí $A \leq B$ significa que la secuencia $A$ es menor que la secuencia $B$ usando el orden lexicográfico.
Dos secuencias distintas $A = (a_1, a_2, \ldots, a_n)$ y $B = (b_1, b_2, \ldots, b_m)$ se comparan como sigue:

- Si $A$ es un segmento inicial de $B$ entonces $A < B$. Si $B$ es un segmento inicial de $A$ entonces B$< A$.
- Si no es así, sea $j$ el menor índice con $a_j \neq b_j$. Entonces, si $a_j < b_j$ se dice que $A < B$, y si $a_j > b_j$ se dice que $A > B$

para seleccionar el vértice de modo tal que una vez desarrollado ese proceso, el árbol podría ser visto como un árbol con raíz.

Para desarrollar esta idea es necesario introducir primero algunos conceptos que permitirán presentar el proceso antes mencionado de una forma más clara y precisa.

**Definición 11: Excentricidad y centro de un grafo**
Dado un grafo $G = (V, E)$, se le llama excentricidad del vértice $v$ en el grafo $G$ y notado como $ex_G(v)$ al número máximo de las distancias de $v$ a los otros vértices de $G$. Ahora se le llamará centro del grafo $G$ al conjunto de todos los vértices de $G$ con excentricidad mínima y lo notaremos como $C(G)$.

Ahora, gracias a que para cualquier árbol $T = (V, E)$, se tiene que $C(T)$ tiene como máximo dos vértices  y, si este es el caso, se puede asegurar que existe una rama entre dichos vértices [22] resulta posible desarrollar una rutina que permite seleccionar un vértice del árbol como la raíz del mismo.

**Algoritmo 3: Isomorfismo de árboles**

1. Seleccionar la raíz del árbol
   1.1. Si el centro de $T$ tiene un solo vértice $v$, entonces se define el código de $T$ como el código del árbol con raíz $v$, $(T, v)$.
   1.2. Si el centro de $T$ consiste de una rama $e = (v, w)$, se considera el grafo $T - e$. Este grafo tiene exactamente dos componentes conexas [22] $T_v$ y $T_w$ con $v \in V(T_v)$ y $w \in V(T_w)$. Ahora, se dice que $A$

representa el código del árbol con raíz $(T_v, v)$ y $B$ representa el código del árbol con raíz $(T_w, w)$. Si $A \leq B$ en la odenación lexicográfica, el árbol $T$ se codifica mediante el código del árbol con raíz $(T, v)$, de lo contrario, se codifica con el código del árbol con raíz $(T, w)$.

2. Generar el código del árbol para cada $T$

   2.1. A cada hoja del árbol (si la raíz tiene grado 1 obviarla) se le asigna el código $01$.

   2.2. Generar para cada vértice del árbol su código correspondiente.

      2.2.1. Supóngase que cada hijo $w$ de un vértice $v$ ya tiene asignado un código $A(w)$. Denotamos por $w_1, w_2, \ldots, w_t$ a los hijos de $v$, de manera que $A(w_1) \leq A(w_2) \leq \cdots \leq A(w_t)$ gracias a un ordenamiento lexicográfico. El vértice $v$ recibe el código $0A_1A_2 \ldots A_t1$.

   2.3. El código del árbol será el código de la raíz.

3. Comparar el código de $T$ con el código de $T'$. Si los códigos son iguales los árboles son isomorfos, en caso contrario, no son isomorfos.

# 6. EL PROBLEMA DEL ISOMORFISMO DE GRAFOS PLANOS

*"Divide et vinces" (Divide y vencerás)*

*Gaius Julius Caesar (Julio Cesar)*

La clase formada por los grafos planos es importante porque ocurre tanto en labores de modelado como en el diseño de redes o en la planeación de rutas de transporte eficientes. Un grafo $G = (V, E)$ se dice que es planar si puede ser embebido en el plano, dicho de otro modo, $G$ es planar (o plano) si existe al menos una manera de representarlo gráficamente en un plano de modo tal que no se presenten intersecciones de aristas. De una manera más formal se tiene que:

**Definición 12: Grafo Plano (planar)**
Sea $CS(\mathbb{R}^2)$ la colección de todas las curvas planas simples y sea $G = (V, E)$ un grafo. Se dirá que $G$ es plano si y solo si existen $f: V \to \mathbb{R}^2$ y $g: E \to CS(\mathbb{R}^2)$ de manera tal que:

i. $f$ y $g$ son inyectivas.

ii. Si $e = \{v, w\}$ se cumple entonces que:

$$(g(e)(0) = v \text{ y } g(e)(1) = w) \text{ o } (g(e)(0) = w \text{ y } g(e)(1) = v).$$

iii. Dadas $e, e' \in E(G)$ si $e \neq e'$ entonces $|g(e) \cap g(e')| = |e \cap e'|$.

En este capítulo, el último de este trabajo, será estudiado el problema del isomorfismo de grafos restringido a grafos planos y se revisará con un cierto nivel

de detalle un algoritmo capaz de decidir en tiempo polinomial si dos grafos planos son isomorfos.

## 6.1. PLANTEAMIENTO DEL PROBLEMA DEL ISOMORFISMO DE GRAFOS RESTRINGIDO A PLANOS

**Nombre del problema:**Problema del isomorfismo de grafos planos (Planar graph isomorphism problem, PGIP, GIP$[\mathcal{P}]$).

**Entrada:**Dos grafos planos finitos $G$ y $H$.

**Problema:** Decidir si $G$ y $H$ son grafos isomorfos.

Como fue mencionado previamente en este documento, la falta de capacidad para determinar la posición de GIP dentro del universo de clases de complejidad ha sido una gran motivación para estudiar el problema, de hecho, ha promovido aproximaciones al problema alternas a las habituales como el estudio del problema restringido a clases de grafos específicas.

Varias subclases contenidas en la clase de grafos planos han sido abordadas y las cotas superiores e inferiores para muchas de estas clases casan, es decir, tienen el mismo valor. Del problema del isomorfismo de grafos plano es sabido, desde hace ya varios años, que este puede ser resuelto en tiempo polinomial. Sin embargo la complejidad exacta del problema solo pudo ser determinada recientemente [30]. La siguiente tabla resume los resultados conocidos para la clase de grafos planos y algunas subclases de ella.

| Clase de Grafos | Cota Inferior | Cota Superior |
|---|---|---|
| Árboles | $L$ | $L$ |
| 2-árboles paciales | $L$ | $L$ |
| Grafos planos 3-conexos | $L$ | $L$ |
| Grafos planos | $L$ | $AC^1$ |

Donde $L$ denota la clase $LOGSPACE$ y $AC^1$ denota la colección de todos los problemas que pueden ser resueltos en tiempo $O(\log n)$ usando un numero polinomial de maquinas RAM.

En lo que queda del capítulo se estudiará un algoritmo que resuelve el problema del isomorfismo de grafos planos en tiempo cuadrático.

## 6.2. SELECCIÓN DE UN ALGORITMO PARA RESOLVER GIP[$\mathcal{P}$]

A pesar de ser un problema estudiado desde los albores de las ciencias de la computación son pocos los algoritmos capaces de decidir GIP$[\mathcal{P}]$ en tiempo polinomial y resulta aún más llamativo que la primera rutina capaz de lograrlo date apenas del año 1972.

En [17] Hopcroft y Tarjan extendieron las ideas usadas por Weinberg, que permiten decidir si dos grafos planos triconexos son isomorfos, para diseñar un algoritmo de tiempo polinomial que resuelve el problema del isomorfismo de grafos cuando este se restringe a la clase de grafos planos, el algoritmo de Hopcroft y Tarjan resuelve el problema GIP$[\mathcal{P}]$ en tiempo $O(n^2)$. Posteriormente los mismos autores mejoraron el desempeño del algoritmo logrando un tiempo de $O(n \log n)$ [30]**.** Luego de esto, el trabajo conjunto de Hopcroft y Wong dio origen en [18] al procedimiento más rápido, de entre los conocidos en la actualidad, el cual es capaz de resolver GIP$[\mathcal{P}]$ en $O(n)$ unidades de tiempo donde $n$ es el número de vértices del grafo. Sin embargo, los autores hacen hincapié en el carácter teórico del algoritmo al asegurar que en el estado actual de la computación una implementación de este procedimiento resultaría ineficiente debido al gran valor de las constantes involucradas en la rutina [18].

Otra aproximación al problema del isomorfismo de grafos planos es el algoritmo paralelo propuesto por Miller y Reif en [24] el cual empleando un número polinomial de procesadores resuelve el problema en $O(\log n)$ unidades de tiempo apelando al modelo CRCW PRAM.[4]

El algoritmo seleccionado para desarrollar este trabajo fue el propuesto por Kukluk, Holder y Cook en [21], el cual es capaz de solucionar GIP$[\mathcal{P}]$ en un tiempo $O(n^2)$. Esta escogencia está fundamentada en que esta rutina computacional puede llegar a ser implementada en un computador convencional a diferencia de las propuestas de Hopcroft-Tarjan y Miller-Reif.

## 6.3. TRASFONDO DEL ALGORITMO SELECCIONADO

Para iniciar el estudio de la rutina seleccionada se probará, inicialmente, la existencia de un algoritmo de tiempo $O(n^2)$ que permite decidir si dos grafos planos triconexos son isomorfos y posteriormente se reducirá el problema de decidir si dos grafos planos son isomorfos al caso de los grafos triconexos. La idea central de estos algoritmos es similar a la usada en el diseño de la rutina de tiempo lineal que resuelve el problema del isomorfismo de árboles; esto es: Los algoritmos que serán estudiados en este capítulo asignan a cada grafo una secuencia de caracteres a manera de código de modo tal que dos grafos serán isomorfos si y solo si sus códigos son iguales.

### 6.3.1. El problema del isomorfismo de grafos planos triconexos

Los grafos planos pueden tener muchos embebimientos en el plano, sin embargo, fue demostrado por Whitney en [34] que los grafos planos triconexos tienen exactamente dos embebimientos. En esta sección será estudiado el algoritmo

[4]Máquina de acceso aleatorio paralela con escritura y lectura concurrentes (Concurrent Read Concurrent Write Parallel Random Access Machine)

propuesto por Weinberg en [33] que permite resolver el problema del isomorfismo de grafos planos triconexos.

Sea $G = (V, E)$ un grafo plano triconexo y sea $\mu_1$ uno de sus dos embebimientos, si se reemplaza cada arista de $G$ por dos aristas dirigidas con sentidos opuestos entre sí, se obtendrá un grafo dirigido $\vec{G}$ con la siguiente propiedad:

> Dado $v \in V(\vec{G})$ el grado de salida y el grado de entrada del vértice $v$ son iguales.

Gracias al teorema de Euler es posible garantizar que dado $v \in V(\vec{G})$ existe un circuito euleriano que empieza en $v$.

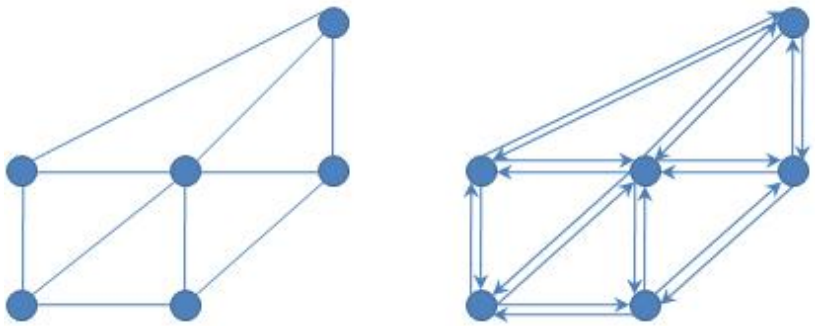


**Ilustración 11 Grafo plano triconexo $G$ y grafo plano triconexo dirigido $\vec{G}$. a. Grafo $G$ o b. Grafo $\vec{G}$**

#### *6.3.1.1. Procedimiento de Weinberg*

El procedimiento propuesto empieza fijando $\mu_1$, un embebimiento en el plano del grafo $G$ y a partir de él se construye $\vec{G}$; luego de esto se escoge un vértice $v \in V(\vec{G})$ y se fija un circuito euleriano $\gamma$ en $\vec{G}$ con inicio en $v$. Hecho esto se

tiene una tripla $(\mu, v, \gamma)$ a partir de la cual le será asignado a $G$ un código $c^{\mu,v,\gamma}(G)$. Debido a que el número de circuitos eulerianos posibles puede ser excesivamente grande debe ser escogido un circuito canónico que inicie en $v$. Para lograr este cometido se fija una arista $e \in V(\vec{G})$ incidente en $v$ y se aplica la regla propuesta por Weinberg:

1. Salga por primera vez de $v$ usando la arista $e$.
2. Dado $w \in V(G)$, si $w$ es visitado por primera vez, tome la primera arista de salida a la derecha (sentido anti horario) de la arista empleada para entrar a $w$.
3. Si $w$ ya fue visitado previamente, use el lado dual para salir de $w$ siempre y cuando no haya sido usado con anterioridad.
4. Si $w$ ya fue visitado y el lado dual fue usado con anterioridad, salga del vértice usando la primera arista de salida a la derecha de la arista empleada para entrar en $w$, que no haya sido usada con anterioridad.

La regla de Weinberg determina una manera de recorrer el grafo $G$ y de este modo determina un circuito euleriano $\gamma_{\mu,v,e}$. Hecho lo anterior podemos usar el circuito $\gamma_{\mu,v,e}$ para asignar un código a $G$; el código $c^{\mu,v,\gamma_{\mu,v,e}}(G)$ se construye de la siguiente manera:

1. Asígnese el número 1 a $v$.

2. Sea $\widehat{W}$ el código parcial construido, cuando se visita un vértice $w$ puede ocurrir uno de dos casos:
   2.1. $w$ no ha sido visitado previamente: asígnese a esté vértice un número $i$ igual al número de vértices que ya han sido visitados más uno. En este caso el código parcial se transforma en $\widehat{W}(i+1)$.
   2.2. $w$ ya fue visitado y por ende ya le fue asignado un número $j$: en este caso el código parcial se transformará en $\widehat{W}j$.

La construcción del código termina cuando se regresa al vértice inicial del ciclo y todas las aristas incidentes en $v$ ya han sido usadas.

El procedimiento antes descrito es repetido para cada una de las aristas del grafo $\vec{G}$ obteniendo así un conjunto de códigos que evidentemente dependen del vértice y la arista fijados.

Sea $c^{\mu}(G)$ el código

$$\min\{c^{\mu,v,\gamma_{\mu,v,e}}(G): v \in V(G)\ y\ e\ es\ incidente\ con\ v\}$$

Donde el mínimo se calcula respecto al orden lexicográfico. Es claro que este mínimo es único y depende del embebimiento $\mu$. Como $G$ es triconexo se puede asociar una pareja de códigos $(c^{\mu_1}(G), c^{\mu_2}(G))$ donde $\mu_1$ y $\mu_2$ son los dos embebimientos del grafo $G$ en el plano. El hecho clave es el siguiente:

***Lema 1:*** Sean $G_1$ y $G_2$ dos grafos planos triconexos, $\mu_1$ y $\mu_2$ los dos embebimientos de $G_1$ y $k_1$ y $k_2$ los dos embebimientos de $G_2$, $G_1$ es isomorfo a $G_2$ si y solo si $\exists i, j \in \{1,2\}$ tal que $c^{\mu_i}(G_1) = c^{k_j}(G_2)$.

Por otra parte, dado $\mu$ uno de los dos embebimientos en el plano de un grafo plano triconexo $G$, es posible construir los códigos $c^{\mu_1}(G)$ y $c^{\mu_2}(G)$ a partir de $\mu$ simplemente con cambiar la regla de ir a la derecha (sentido anti horario) por la regla de ir a la izquierda (sentido horario) en el procedimiento que fija el circuito canónico a través del grafo. Gracias a ello se habla de un "código a la derecha" y un "código a la izquierda".

El procedimiento de Weinberg es el algoritmo $\mathcal{M}$ que con entradas $(G_1, G_2)$ (dos grafos planos triconexos) hace lo siguiente:

**Algoritmo 4: Algoritmo de Weinberg**

1. Construye $\mu_1$ y $\mu_2$ dos embebimientos diferentes de $G_1$.
2. Construye $k_1$ y $k_2$ dos embebimientos diferentes de $G_2$.
3. Construye $c^{\mu_1}(G_1)$ y $c^{\mu_2}(G_1)$.
4. Construye $c^{k_1}(G_2)$ y $c^{k_2}(G_2)$.
5. Calcula $X = \min\{c^{\mu_1}(G_1), c^{\mu_2}(G_1)\}$ y $Y = \min\{c^{k_1}(G_2), c^{k_2}(G_2)\}$.

   Verifica si $X = Y$ si es el caso acepta, de lo contrario rechaza.

De lo dicho anteriormente es claro que $\mathcal{M}$ es correcto. Por otra parte, es posible chequear que el tiempo de cómputo del algoritmo $\mathcal{M}$ es $O(n^2)$. Para una prueba el lector interesado puede consultarla referencia [33]**.**

Llegados a este punto es posible afirmar que se ha logrado resolver el problema del isomorfismo de grafos para el caso de grafos planos triconexos,( lo cual no es suficiente). Sin embargo se puede tomar este progreso como punto de partida para diseñar algoritmos capaces de resolver el caso más general, el que atañe a este proyecto, el de los grafos planos. Las ideas básicas tras los algoritmos que se estudiarán son las siguientes:

i. Todo grafo plano biconexo puede ser descompuesto en un árbol de componentes triconexas. Los códigos de estas componentes (que se calculan usando el algoritmo de Weinberg) y del árbol mismo pueden ser usados para construir un código que caracteriza el grafo en cuestión.
ii. Todo grafo plano se puede descomponer como un árbol de componentes biconexas. Los códigos de estas componentes (que de acuerdo con i. ya pueden ser calculadas) y el árbol mismo pueden ser usados para construir un código que caracteriza el grafo en cuestión.

### 6.3.2. Grafos planos biconexos

Siguiendo con la hoja de ruta demarcada para resolver el problema de interés, el paso a dar consiste en considerar los grafos planos biconexos. Para abordar esta restricción de GIP se empleará una estructura de datos denominada árbol SPQR que será introducida a continuación

#### *6.3.2.1. Árboles SPQR y componentes triconexas*

Sea $G = (V, E)$ un multi-grafo biconexo, $\{a, b\}$un par separador de $G$, y $E_1, \dots, E_k$ las clases de separación de $G$ con respecto a $\{a, b\}$. Sean, entonces,

$E' = \bigcup_{i=1}^{l} E_i$ y $E'' = \bigcup_{i=l+1}^{k} E_i$, donde $|E'| \geq 2$ y $|E''| \geq 2$. Los grafos $G' = (V(E'), E' \cup \{e\})$y $G'' = (V(E''), E'' \cup \{e\})$ son llamados grafos de división y $e = (a, b)$ es una arista nueva que identifica la operación de separación[5] y es llamada arista virtual.

Descomponer a $G$ radica en aplicar la operación de separación inicialmente sobre el grafo $G$ y luego sobre sus grafos de división tantas veces como sea posible hasta que cada uno de los grafos obtenidos, llamados componentes de separación, sean de uno de los siguientes tres tipos:

i. Un conjunto de tres aristas múltiples (triple ligadura).
ii. Un ciclo de longitud tres (triángulos).
iii. Grafos triconexos simples.

Es importante señalar que las componentes de separación no son necesariamente únicas [14].

> **Lema 2:** [14] Sea $G = (V, E)$ un multi-grafo. Cada arista de $E$ está contenida en exactamente una componente de separación, y cada arista virtual está contenida en exactamente dos componentes de separación.

Sean $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ dos componentes de separación que contienen la misma arista virtual $e$. El grafo $G = (V_1 \cup V_2, (E_1 \cup E_2) \setminus \{e\})$ es llamado el grafo fusión de $G_1$ y $G_2$. Las componentes triconexas de $G$ son obtenidas como resultado de fusionar[6] las componentes de separación de los tipos

[5]La operación separación consiste en reemplazar un multi-grafo $G$ por sus dos grafos de división.
[6]Fusionar consiste en reemplazar dos componentes de separación $G_1$ y $G_2$ por el grafo fusión respectivo

i. y ii. en conjuntos maximales de múltiples aristas y en ciclos (polígonos) respectivamente. Las componentes triconexas de un grafo $G = (V, E)$ son únicas [14].

Las componentes triconexas están íntimamente relacionadas con los árboles SPQR. De hecho, los árboles SPQR son estructuras usadas para agrupar las componentes triconexas de un grafo biconexo descompuesto en sus componentes triconexas.

Sea $G = (V, E)$ un grafo plano biconexo, dada $e \in E$ una arista de $G$ se mostrará cómo construir un árbol $\mathcal{T}_e(G)$ que será llamado *el árbol SPQR de* $G$ *asociado a la arista* $e$ la cual será denominada arista de referencia. El árbol $\mathcal{T}_e(G)$ será un grafo etiquetado tal que dado $s$ un nodo de $\mathcal{T}_e(G)$ la etiqueta asociada a $s$ es un multígrafo, para más señas una componente triconexa, que será llamado esqueleto de $s$.

Sea $e = \{s, t\}$ un par cortante, el árbol $\mathcal{T}_e(G)$ puede tener vértices de cuatro tipos a saber:

- Q-nodos
  El árbol $\mathcal{T}_e(G)$ contendrá este tipo de nodos si y solo si $G$ consiste de dos aristas paralelas conectando a $s$ y $t$; en este caso $\mathcal{T}_e(G)$ es un árbol de un único nodo y la etiqueta de este es $G$.
- P-nodos

Si $\{s,t\}$ tiene al menos tres componentes cortantes $G_1,\dots,G_k$ $(k \geq 3)$, entonces la raíz de $\mathcal{T}_e(G)$ es un P-nodo cuyo esqueleto es un grafo con dos vértices $s,t$ y $k$-aristas paralelas entre los nodos antes mencionados.

- S-nodos
  Si $\{s,t\}$ tiene exactamente dos componentes cortantes, una de ellas se reduce a la arista de referencia $e=(s,t)$ mientras que la otra es un subgrafo maximal de $G$ en el que $\{s,t\}$ no es un par cortante, esta segunda componente se notará como $G'$. Para este caso la raíz del árbol será un nodo del tipo S y su esqueleto se construye del siguiente modo: sean $c_1,\dots,c_{k-1}$ $(k \geq 2)$ los puntos de articulación de $G'$ que lo separan en $k$-bloques, el esqueleto será un ciclo $e_0,e_1,\dots,e_k$, donde $e_0 = e = (s,t)$, $c_0 = s, c_k = t$ y $e_i = (c_{i-1},c_i)$.

- R-nodos
  Si ninguno de los casos recién mencionados aplica; sean $\{s_1,t_1\},\dots,\{s_k,t_k\}$ las componentes maximales cortantes de $G$ con respecto a $\{s,t\}(k \geq 1)$. Dado $i \in \{1,\dots,k\}$ se usará $G_i$ para denotar la unión de las componentes cortantes de $\{s_i,t_i\}$ que no contienen al par $\{s,t\}$. La raíz de $\mathcal{T}_e(G)$ será un R-nodo cuyo esqueleto es el grafo obtenido de $G$ reemplazando cada subgrafo $G_i$ con el lado $e=\{s_i,t_i\}$.

Dado $G$ un grafo no trivial y $e$ una arista de $G$, se construye $\mathcal{T}_e(G)$ de la siguiente manera:

1. Construir el esqueleto de la raíz.
2. Dada $s$ una arista en el esqueleto de la raíz, si $s$ es una arista de $G$ se le asigna la etiqueta 0, en caso contrario le es asignada la etiqueta 1.
3. Dado $p$ un nodo ya construido y dado $S(p)$ el esqueleto de $p$, se etiquetan las aristas de $S(p)$ de acuerdo a la regla establecida en el paso 2.
4. Si $\tilde{e}$ es una arista de $S(p)$ con etiqueta 1, se construye $\mathcal{T}_{\tilde{e}}(S(p))$ y se pone una arista entre $p$ y la raíz de $\mathcal{T}_{\tilde{e}}(S(p))$.
5. Repetir el proceso hasta que ningún nodo pueda ser descompuesto.

Los lemas a continuación son dos lemas fundamentales para nuestros propósitos:

**Lema 3:** Dado $G$ un grafo biconexo y dadas $e$ y $e'$ dos aristas de $G$ se tiene que $\mathcal{T}_e(G) \cong \mathcal{T}_{e'}(G)$ y el isomorfismo respeta las etiquetas (dado $r$ un nodo de $\mathcal{T}_e(G)$, el esqueleto de $r$ y el de $f(r)$ coinciden).

**Lema 4**: Dados $G = (V, E)$ un grafo plano biconexo, $e \in E(G), v$ un nodo de $\mathcal{T}_e(G)$ y $S(v)$ su esqueleto. Se tiene que $S(v)$ es triconexo.

El lector interesado en la verificación de los lemas puede consultar la referencia [8].

Los lemas anteriores sugieren una manera de asignar a un grafo plano biconexo $G = (V, E)$ un código aplicando el siguiente procedimiento:

1. Escoja $e \in E(G)$.
2. Construya $\mathcal{T}_e(G)$.

3. Use los códigos de $\mathcal{T}_e(G)$ y de las etiquetas (esqueletos) de sus nodos (que son triconexos) para asignarle un código a $G$.

En estos momentos (exceptuando que aún se desconoce la rutina por medio de la cual se genera el código que describe el grafo) es posible asegurar que ya se ha demarcado una metodología que hace posible evaluar si dos grafos planos biconexos son isomorfos.

Por el momento se va a aplazar la introducción del algoritmo generador del código caracterizador del grafo biconexo y se hará énfasis en la estructura del grafo plano y en cómo se le puede descomponer en un árbol de componentes biconexas que permita aprovechar el camino recorrido hasta ahora.

### 6.3.3. Descomposición de grafos planos en componentes biconexas.

Sea $G = (V, E)$ un grafo plano no dirigido, y $u_1, \dots, u_n$ los puntos de articulación de $G$. Los puntos de articulación separan a $G$ en subgrafos biconexos $G_1, \dots, G_k$. Cada subgrafo $G_i$ comparte un punto de articulación $u_i$ con al menos otro subgrafo $G_j$. Sea el árbol $T$ de componentes biconexas formado por vértices de dos tipos:

i. Nodos biconexos $B_1, \dots, B_k$ que corresponden a los subgrafos biconexos de $G$.
ii. Nodos de articulación $v_1, \dots, v_n$ que corresponden a los puntos de articulación de $G$.

Un vértice de articulación $v$ es adyacente a los nodos biconexos $B_l, \ldots, B_m$ en $T$ si los correspondientes subgrafos $G_l, \ldots, G_m$tiene como punto de articulación común el vértice $v$.

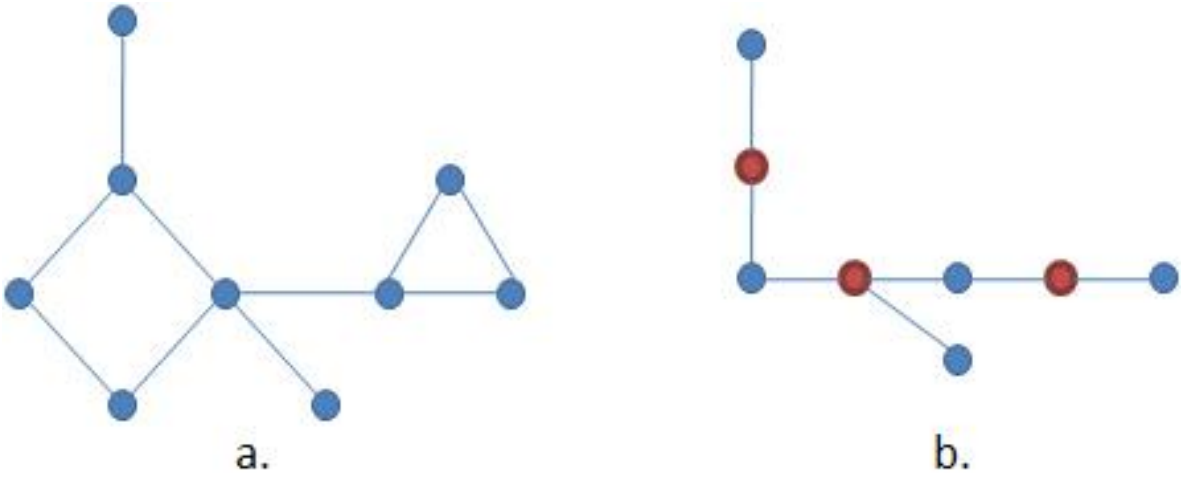


**Ilustración 12 Grafo conexo y su árbol de componentes biconexas. a. Grafo $G$ b. Árbol de componentes de biconexas de $G$; nodos de articulación en color rojo, nodos biconexas en color azul**

Hopcroft y Tarjan proponen en [16] un algoritmo eficiente para descomponer un grafo conexo en sus componentes biconexas realizando una primera búsqueda en anchura (DFS) a través de las aristas del grafo. A grandes rasgos, el algoritmo recorre los arcos del grafo tratando siempre de alcanzar nuevos vértices, cuando no es posible alcanzar un nuevo nodo y se detecta un punto de articulación, se considera que la componente biconexa estará formada por las aristas recorridas justo después de la primera aparición del punto de articulación. Una vez extraída la componente de $G$, el proceso se repite desde el último punto de articulación hallado hasta cubrir todos los elementos del grafo.

Dado un grafo $G$, una descripción más detallada del algoritmo es la siguiente:

1. Escoja un vértice $v \in V(G)$, selecciónelo como el punto inicial y registre la pareja $(x_1, x_2) = (v, \#)$ en la pila de vértices donde $\#$ es un símbolo fijado que denota que es el primer elemento de la pila.
2. Salga del vértice en el tope de la pila usando una arista cualquiera $e = (x_1, u)$ buscando, siempre que sea posible, que $u$ sea un vértice que aún no haya sido visitado. Registre la arista $e$ en una pila destinada para tal fin (se hará referencia a ella como pila de aristas).
   2.1. Si a través de $e$, desde $x_1$ en el tope de la pila, se alcanza un nuevo vértice, registre en la pila de vértices la pareja $(x_1, x_2) = (u, v)$. De lo contrario haga en el tope de la pila $x_2 = u$ si $u < x_2$.
   2.2. Si no existe arista alguna para salir de $x_1$, el vértice en el tope de la pila, y en ella (la pila) hay más de un registro, evalúe si la componente $x_2$ en el registro que se encuentra en el tope de la pila es igual a la componente $x_1$ del siguiente registro; de ser así, la componente biconexa se formará con los arcos que se eliminarán sucesivamente de la pila de aristas hasta encontrar una que conecte a la componente $x_1$ del registro en el tope de la pila con el $x_1$ del siguiente registro. Elimine estos dos registros y repita el paso 2 hasta que todos los vértices y aristas sean cubiertos. Si $x_2$ en el tope es diferente del valor de $x_1$ en el siguiente registro en la pila haga al $x_2$ de este registro (el penúltimo de la pila) igual al $x_2$ del tope de la pila, elimine el registro en el tope de la pila de vértices y repita 2.

Las componentes biconexas de un grafo conexo (que también reciben el nombre de bloques) pueden ser de tres tipos:

i. Aristas puente
ii. Subgrafos biconexos
iii. Vértices aislados [9]

El grafo de componentes biconexas de un grafo conexo es siempre un árbol [9]. Para crear el árbol se hace lo siguiente:

1. Para cada componente biconexa $B_i$ identifique los nodos comunes con las demás componentes $B_j$ (puntos de articulación).
2. Cree un nodo de articulación $v_i$ que será adyacente a las componentes $B_l, \ldots, B_m$ que lo contengan como vértice.

### 6.3.4. Isomorfismo de árboles etiquetados

Como fue mencionado recientemente en esta sección, la metodología de solución de GIP[$\mathcal{P}$] en la cual esta cimentado el algoritmo que se está estudiando consiste en asignar un código descriptor del grafo de modo tal que dos grafos serán isomorfos si y solo si sus códigos coinciden.

Ahora bien, el código del que se habla es, en esencia, una secuencia de caracteres, una palabra sobre un alfabeto definido $\Sigma$; dicha palabra está constituida, a su vez, por la concatenación de códigos parciales asignados a los elementos del grafo a modo de identificador. Dichas palabras reciben el nombre de etiquetas y son palabras del conjunto $\Sigma^*$.

El hecho de que todo grafo con puntos de articulación, incluidos los grafos planos, pueda ser descompuesto en un árbol de componentes biconexas, lleva a pensar

en que una buena manera de abordar GIP[$\mathcal{P}$] es intentar reducir este problema al problema de isomorfismo de árboles etiquetados donde a cada uno de los nodos biconexos del árbol le correspondería una etiqueta que describiría la componente representada por el nodo en la estructura.

La anterior idea estaría sustentada por el siguiente lema:

> **Lema 5:** Sean $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ dos grafos y $T_1$, $T_2$ sus árboles de componentes biconexas respectivamente. $G_1$ y $G_2$ serán isomorfos si y solo si $T_1$ y $T_2$ también son isomorfos entre sí.

La presencia de las etiquetas en los árboles hace que resulte necesario redefinir la función de isomorfismo de la siguiente manera:

> **Definición13: Isomorfismo de árboles etiquetados**
> Dados $T_1$ y $T_2$ dos grafos etiquetados. $T_1$ y $T_2$ serán isomorfos si y solo si existe $f: V(T_1) \longrightarrow V(T_2)$ tal que:
>
> i. $\forall x, y \in V(T_1), (x, y) \in E(T_1) \Leftrightarrow (f(x), f(y)) \in E(T_2)$.
>
> ii. Sea $e(v)$ la etiqueta del vértice $v$. $\forall v \in V(T_1), e(v) = e(f(v))$.

Una rutina para determinar si dos árboles etiquetados $T_1$ y $T_2$ son isomorfos es la siguiente:

**Algoritmo 5: Isomorfismo de árboles etiquetados**

1. Chequee los árboles $T_1$ y $T_2$ para isomorfismo obviando sus etiquetas.
   1.1. Si los árboles son isomorfos, calcule el centro de cada árbol. De lo contrario, rechace.
2. Sean $v_1, v_2$ y $w_1, w_2$ los vérties del centro. Para cada pareja $(v_i, w_j)$ haga lo siguiente:

2.1. Plante los árboles $T_1$ y $T_2$ en torno a los nodos $v_i$ y $w_j$ y ordene los hijos de cada uno de los nodos (de cada uno de los árboles) de acuerdo al orden lexicográfico de sus etiquetas.

3. Construya $\mathcal{C}_{v_i}^{T_1}$ los códigos del árbol $T_1$ plantado en torno a la raíz $v_i$ y $\mathcal{C}_{w_j}^{T_2}$ los códigos del árbol $T_2$ plantado en torno a la raíz $w_j$ con $i,j \in \{1,2\}$. El código se logra mediante una primera búsqueda en anchura a través del árbol de modo tal que en cada paso concatene al código parcial ya construido la etiqueta de cada nuevo vértice visitado.
4. Para $i,j \in \{1,2\}$ compare los códigos $\mathcal{C}_{v_i}^{T_1}$ y $\mathcal{C}_{w_j}^{T_2}$ si existe algún valor de $i$ y algún valor de $j$ tal que $\mathcal{C}_{v_i}^{T_1} = \mathcal{C}_{w_j}^{T_2}$ acepte los árboles como isomorfos, de lo contrario rechácelos.

## 6.4. ALGORITMO DE KUKLUK, HOLDER Y COOK

La rutina propuesto para solucionar GIP[$\mathcal{P}$] actúa sobre grafos planos, conexos, no dirigidos y sin etiquetar [21]**;** dados dos grafos $G_1$ y $G_2$, el algoritmo presentado por los autores es el siguiente:

**Algoritmo 6: Algoritmo de Kukluk, Holder y Cook**

1. Testear si $G_1$ y $G_2$son grafos planos. De no serlo, rechazar mencionando que los grafos no son planos.
2. Descomponer $G_1$ y $G_2$ en componentes biconexas y construir el árbol de componentes biconexas.

3. Descomponer cada componente biconexas en sus componentes triconexas y construir su árbol SPQR correspondiente
4. Construir el código único para cada árbol SPQR y desde las hojas hacia la raíz construir el código único para el árbol de componentes biconexas.
5. Si $Código(G_1) = Código(G_2)$ aceptar a $G_1$ y $G_2$ como grafos isomorfos, de lo contrario rechazar.

Exceptuando la forma en la cual se construyen los códigos para los árboles SPQR y árboles de componentes biconexas todas las rutinas invocadas por el algoritmo propuesto ya han sido presentadas al lector; se procede, entonces, a introducir las rutinas constructoras de los códigos y posteriormente se exhibirá un análisis del tiempo de ejecución del algoritmo estudiado en el presente documento.

### 6.4.1. Rutina para la generación de un código único para el árbol SPQR de un grafo biconexo.

El procedimiento que se describe a continuación asigna un código único a un grafo biconexo. Debido a que la rutina de construcción del código varía de acuerdo al nodo para el cual es elaborado se usa el conjunto de símbolos '$_P$(', ')$_P$', '$_S$(', ')$_S$', '$_R$(', ')$_R$', denominados de control, los cuales actúan como identificadores y son parte integral del grupo. Estos símbolos están ordenados lexicográficamente del siguiente modo: '$_P$('<')$_P$'<'$_S$('<')$_S$'<'$_R$('<')$_R$'.

Dado $G = (V, E)$ un grafo plano biconexo como input, la rutina hace lo siguiente:

**Algoritmo 7: Asignación de un código único a un grafo plano biconexo**

1. Descomponga el grafo biconexo en sus componentes triconexas y construya el correspondiente árbol SPQR.
2. Reemplace cada arista en los esqueletos de los nodos del árbol SPQR con dos aristas dirigidas en direcciones opuestas entre sí.

3. Identifique el centro del árbol y selecciónelo como raíz. Las aristas virtuales en los esqueletos de los nodos internos del árbol ($d(v) > 1$) están asociadas con las ramas del árbol por ello su código dependerá también de los nodos a los cuales están conectadas. La dirección de las ramas del árbol es determinada por la raíz del árbol.

   3.1. Si el centro del árbol es una arista $e = (a, b)$ y los tipos de los nodos $a$ y $b$ son diferentes fije una regla que especifique que tipo de nodo debe ser seleccionado como el centro del árbol. Es fácil verificar que dos nodos del mismo tipo solo pueden ser adyacentes si son tipo R; para este caso deben computarse dos casos por separado uno considerando $a$ y luego otro considerando a $b$ como raíz.

4. Construye el código a partir de la raíz del árbol de acuerdo a las siguientes reglas:

   **Nodo tipo S**

   El código inicia con '$_S$(' y termina con ')$_S$'.

   **Como Raíz**

   Agregue el número de aristas del esqueleto de S al código. Determine los códigos asociados a las aristas virtuales. Escoja la arista con el menor código de acuerdo al orden lexicográfico y considérela la arista de referencia. Recorra el circuito euleriano determinado por la arista referencia iniciando con el lado inmediatamente siguiente a ella. Cuente las aristas durante el recorrido. Si una arista virtual es hallada registre en el código el número que le correspondió durante la exploración del esqueleto. Al llegar a la arista de referencia el recorrido finaliza. Concatene al código hallado los códigos correspondientes a las aristas virtuales en el orden en el que fueron halladas durante el recorrido.

   Si el nodo raíz no tiene aristas virtuales ni puntos de articulación, el código del nodo será el número de aristas de su esqueleto.

Para los casos en los cuales la arista de referencia no puede ser seleccionada porque hay varias aristas con el menor código, el procedimiento descrito deberá repetirse para cada una de ellas y el menor de los códigos hallados será el código único de la raíz tipo S. Si en algún punto durante el recorrido es hallado un punto de articulación, recuerde en cual arista del tour ocurrió y agregue el número de esta arista al código marcándolo como un punto de articulación.

**Nodo que no es raíz**

Construir el código para este tipo de nodos difiere solo en dos aspectos respecto a la construcción del código para los nodos tipo S que son raíces:

1. La manera en la que se escoge la arista de referencia: Cuando el nodo S no es raíz la arista de referencia es aquella asociada a la arista de entrada; por ello solo se considera dicha arista.
2. Solamente las aristas virtuales diferentes a la arista de referencia son consideradas en el momento de concatenar el código.

**Nodo tipo P**

El código inicia con '$_P$(' y termina con ')$_P$'.

**Como raíz**

Encuentre el número de aristas totales y de aristas virtuales en el esqueleto del nodo tipo P. Agregue al código primero el número de aristas totales seguido del número de aristas virtuales. Si $a$ y $b$ son los vértices del esqueleto construya el código para todas las aristas virtuales en una dirección. Agregue los códigos de todas las aristas virtuales dirigidas desde $a$ hacia $b$ al código del nodo P. Los códigos agregados deben estar en orden creciente. Si $a$ o $b$ es un punto de articulación agregue una marca al código indicando si el vértice de articulación está en la cabeza o en la cola

de la arista dirigida desde $a$ hacia $b$. Construya el segundo código del mismo modo pero en dirección desde $b$ hacia $a$. Compare los dos códigos. El menor código es el código del nodo P

**Nodo que no es raíz**

Construya el código del mismo modo que se crea para el nodo tipo P como raíz pero solamente en una dirección la cual es determinada por la arista de entrada.

**Nodo Tipo R**

El código inicia con '$_R$(' y termina con ')$_R$'.

**Como raíz**

Para todas las aristas virtuales del esqueleto de un nodo tipo R encuentre los códigos asociados a ellas. Determine el menor de los códigos y las aristas con ese código serán las aristas iniciales. Para cada una de las aristas de ese conjunto construya el código de acuerdo al procedimiento de Weinberg. Cuando una arista virtual es hallada concatene su código. Para cada arista deben considerarse los casos de "ir a la derecha" e "ir a la izquierda". Finalmente, escoja el menor código para representar el nodo raíz tipo R. Si durante el tour se halla algún punto de articulación, marque este punto en el código.

**Nodo que no es raíz**

Solamente dos casos han de ser considerados ("ir a la derecha" e "ir a la izquierda") porque la arista inicial es determinada por la arista de entrada al nodo. Solamente son consideradas en el momento de concatenar los códigos de las aristas virtuales aquellas diferentes a la arista de entrada.

### 6.4.2. Rutina para la generación de un código único para el árbol de componentes biconexas de un grafo plano.

A diferencia del procedimiento recientemente descrito que asigna un código único para los grafos biconexos partiendo de la raíz del árbol, esta rutina construye el código partiendo de las hojas del árbol e iteración tras iteración avanza hacia el centro de la estructura de datos. Una peculiaridad de los árboles de componentes biconexas es que, a diferencia de otras estructuras de datos tipo árbol, su centro siempre será un nodo ya sea de articulación o biconexo [21].

Para esta rutina se hace necesario ampliar el conjunto de los símbolos de control con la inclusión de los elementos $'_A('$, $')_A'$, $'_B('$ y $')_B'$ quedando definido el orden lexicográfico para ellos del siguiente modo:

$$'_A(' < ')_A' < '_B(' < ')_B' < '_P(' < ')_P' < '_S(' < ')_S' < '_R(' < ')_R'$$

Ahora, dado $T = (V, E)$ un árbol de componentes biconexas como input, el algoritmo asigna un código único al árbol del siguiente modo:

1. Construya el código para las hojas del árbol.
   1.1. Si la hoja es un nodo biconexo $B_i$.

   Construya el código $C$ del grafo biconexo que le corresponde al nodo $B_i$. El código para la componente biconexa $B_i$ será $C(B_i) = {}_B(C)_B$ .

   1.2. Si la hoja del árbol es un nodo de articulación $v_i$, el código del nodo de articulación $v_i$ será $C(v_i) = {}_A(C)_A$ donde $C$ es la sarta formada por la concatenación de los códigos de las componentes $B_1 \dots B_m$ adyacentes a $v_i$ ordenados de manera creciente de acuerdo al orden lexicográfico.

2. Si solo queda un nodo en el árbol, el último código calculado es el código del árbol y por ende el código identificador del grafo plano. De lo contrario elimine las hojas del árbol, generando así nuevas hojas y repita el paso 1.

## 6.5. ANÁLISIS DEL TIEMPO DE EJECUCIÓN DEL ALGORITMO DE KUKLUK, HOLDER Y COOK PARA TESTEAR ISOMORFISMO EN GRAFOS PLANOS

Como pudo notarse, el algoritmo propuesto por Kukluk, Holder y Cook y estudiado en este escrito está compuesto por varias subrutinas con tiempos de ejecución propios, por ello, para poder determinar el tiempo que requiere el procedimiento de manera global resulta necesario conocer el de cada uno de los subprocesos.

Hopcroft y Tarjan presentan en [15] un algoritmo implementable que requiere tiempo $O(n)$ para testear la planaridad de un grafo con $n$ vértices y en [16] exhiben, entre otros, un algoritmo capaz de descomponer un grafo en sus componentes biconexas en tiempo lineal, siempre y cuando el grafo esté expresado como una lista de aristas. El trabajo Desarrollado por Gutwenger y Mutzel en [14] muestra que es posible implementar una rutina de tiempo lineal que dado un grafo biconexo como input calcula su árbol SPQR. En [21] es demostrado que la rutina aquí presentada para construir un código único para un grafo biconexo requiere de tiempo $O(n^2)$. Los resultados presentados en [33] implican que es posible construir un código para evaluar el isomorfismo de grafos planos triconexos en tiempo $O(n^2)$.

Por las razones descritas recientemente resulta posible asegurar que el algoritmo de Kukluk, Holder y Cook resuelve el problema del isomorfismo de grafos planos en tiempo $O(n^2)$.

# 7. CONCLUSIONES

En situaciones reales, solucionar un problema de optimización con un algoritmo determinista puede no ser la mejor opción dado que existen rutinas probabilísticas y/o meta-heurísticas capaces de hallar la solución óptima, o al menos una cuasi óptima, con demandas de cómputo menores.

Restringir las instancias de un problema computacional es una acción válida ya que, en ambientes reales, el objeto abstraído es quien determina la estructura que tendrá el modelo que lo representa.

Existen algoritmos que a pesar de ser considerados teóricamente eficientes no son implementables en la práctica ya que su tiempo de ejecución, aunque polinomial, está determinado por una constante de gran magnitud.

Restringir las instancias posibles de un problema computacional a un conjunto determinado puede facilitar el proceso de resolverlo eficientemente ya que es posible aprovechar la estructura que define a dicho conjunto para diseñar buenos algoritmos.

Transformar las instancias de un problema complejo en instancias de un problema para el cual es conocida una rutina eficiente que lo soluciona (reducirlo) es una

buena estrategia para el diseño de algoritmos ya que permite aprovechar rutinas cuya eficacia es ya conocida.

Aunque a día de hoy no ha sido posible determinar la posición exacta de GIP dentro de la clase $NP$, al ser la computación teórica un área dinámica en constante desarrollo, resulta posible asegurar que es muy probable que tan fundamental cuestionamiento sea resuelto en los próximos años.

# 8. RECOMENDACIONES

Estudiar otras clases de restricciones de GIP permitiría contrastar los resultados exhibidos en este escrito y ampliaría el panorama que se posee del problema del isomorfismo de grafos.

Construir una biblioteca de programas enfocada en el desarrollo de rutinas útiles para abordar problemas con grafos, buscando con esto facilitar el desarrollo de futuros proyectos de desarrollo en temáticas relacionadas con la teoría de grafos.

Estudiar en detalle los algoritmos para solucionar el problema del isomorfismo de grafos propuestos por Hopcroft y Wong en [18] y Miller y Reif propuesto en [24] buscando con esto construir un hit algorítmico para resolver el problema del isomorfismo de grafos planos.

# BIBLIOGRAFIA


1. **AGRAWAL, Manindra; KAYAL, Neeraj y SAXENA, Nitin.** PRIMES is in P. Kanpur: Annals of Mathematics, Vol. 160. 2004.

2. **ARORA, Sanjeev y BARAK, Boaz.** Computational Complexity: A Modern Approach. Cambridge, UK: Cambridge University Press, 2009. ISBN 978-0-521-42426-4.

3. **ARRINGTON, Chelsea.** Primality Testing. 2010.

4. **BOVET, Daniel P. y CRESCENZY, Piereluigi.** Introduction to the theory of complexity. 2006.

5. **COOK, Stephen.** The P versus NP Problem. Toronto: Clay Mathematics Institute, 2000.

6. **CORMEN, Thomas; LEISERSON, Charles y RIVEST, Ronald.** Introduction to algorithms. Cambridge, Massachusetts, USA: The MIT Press, 2001. ISBN 0262032937.

7. **DEPUY, Gail; MORAGA, Reinaldo y WHITEHOUSE, Gary.** Metaheuristics: A solution methodology for optimization problems.En: BADIRU Adedeji. Handbook of industrial and systems engineering. [s.l.]: CRC/Taylor & Francis, 2006.

8. **DI BATTISTA, Giuseppe y TAMASSIA, Robert.** On-Line Maintenance of Triconnected Components with SPQR-Trees. Algorithmica, Vol. 15, págs. 302-318. 1996.

9. **DIESTEL, Reinhard.** Graph theory. Edición Electrónica. New York: Springer-Verlag, 2000.

10. **GAREY, M. R.; JOHNSON, David y TARJAN, Robert E.** The Planar Hamiltonian Circuit Problem is NP-Complete. SIAM Journal of computing, Vol. 5(4), págs. 704-714. 1976.

11. **GAREY, Michael y JOHNSON, David.** Computers and Intractability. San Francisco : W. H. Freeman and company, 1979. ISBN: 0716710447.

12. **GLOVER, Fred y LAGUNA, Manuel.** Tabu search. Dordrecht : Kluwer Academic Publishers, 1997.

13. **GOLDREICH, Oded.** Computational Complexity: A Conceptual Perspective. Cambridge, UK : Cambridge University Press, 2008. ISBN 9780521884730.

14. **GUTWENGER, Carsten y MUTZEL, Petra.** A linear time implementation of SPQR-trees. Springer- Verlag, 2001.

15. **HOPCROFT, John y TARJAN, Robert.** Efficient planarity testing. Journal of the association for computing machinery, Vol. 21(4), págs. 549-568. 1974.

16. **HOPCROFT, John y TARJAN, Robert.** Efficient algorithms for graph manipulation. Communications of the ACM, Vol. 16(6). 1973.

17. **HOPCROFT, John y TARJAN, Robert.** Isomorphism of planar graphs (working paper). 1972.

18. **HOPCROFT, John y WONG, J. K.** Linear time algorithm for isomorphism of planar graphs (Preliminary report). Proceedings of the sixth annual ACM symposium on Theory of computing, págs. 172-184. 1974.

19. **KARP, Richard.** An introduction to randomized algorithms. Discrete applied Mahematics, Vol. 34, 1991.

20. **KASTELEYN, P. W.** Graph theory and crystal physics. In Frank Harary, editor, Graph Theory and theoretical physics, págs. 43-110. Academic Press, 1967.

21. **KUKLUK, Jacek; HOLDER, Lawrence y COOK, Diane.** Algorithm and experiments in testing planar graphs for isomorphism. Journal of Graph algorithms and applications, Vol. 8(3), págs. 313-356. 2004.

22. **MATOUŠEK, Jiří y NEŠETŘIL, Jaroslav**. Invitación a la matemática discreta. Barcelona : Editorial Reverté, 2008.

23. **MENEZES, Alfred; VAN OORSCHOT, Paul y VANSTONE, Scott.** Handbook of applied cryptography. Boca Raton, USA : 1996. ISBN: 0849385237.

24. **MILLER, Gary y REIF, Jhon.** Parallel tree contraction part 2: further applications. SIAM Journal of computing, 20(6). 1991.

25. **MORAIN, François.** Thirty years of integer factorization. Lix, École Polytechnique, 2001.

26. **RANGAN, Pandu.** Randomized Dictionary Structures. En: MEHTA, Dinesh y SAHNI, Sartaj (eds). Handbook of data structures and applications. [s.l.] : Chapman & Hall/ CRC Press, 2004.

27. **SHOR, Peter W.** Algorithms for quantum computation: Discrete logarithms and factoring. Proceedings of the 35th Annual Symposium on Foundations of Computer Science. 1994.

28. **SKIENA, Steve S.** The Algorithm Design Manual. New York : Springer-Verlag, 2008. ISBN: 978-1-84800-069-8.

29. **TODA, Seinosuke.** PP is as hard as the polynomial-time hierarchy. SIAM Journal on computing, 20 (5): págs. 865-877. 1991.

30. **TORÁN, Jacobo, WAGNER, Fabián.** The complexity of planar graph isomorphism. 2009.

31. **TORÁN, Jacobo.** On the hardness of graph isomorphism. Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on: págs. 180-186. 2000.

32. **VALIANT, L. G.** The complexity of computing the permanent., Theoretical Computer Science, 8: págs. 189-201. 1979.

33. **WEINBERG, Louis.** A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. Circuit theory, 13(2). 1966.

34. **WHITNEY, Hassler.** A set of topological invariants for graphs. American journal of mathematics Vol. 55: págs. 231-235, 1933.