Grafos: Fundamentos y Algoritmos

ISBN: 978-956-306-076-8
Registro de Propiedad Intelectual: 200.525
Colección: Herramientas para la formación de profesores de matemáticas.
Diseño: Jessica Jure de la Cerda.
Diseño de Ilustraciones: Cristina Felmer Plominsky, Catalina Frávega Thomas.
Diagramación: Pedro Montealegre Barba, Francisco Santibáñez Palma.
Financiamiento: Proyecto Fondef D05I-10211.

Datos de contacto para la adquisición de los libros:
Para Chile:
1. En librerías para clientes directos.
2. Instituciones privadas directamente con:
   Juan Carlos Sáez C.
   Director Gerente
   Comunicaciones Noreste Ltda.
   J.C. Sáez Editor
   jcsaezc@vtr.net
   www.jcsaezeditor.blogspot.com
   Oficina: (56 2) 3260104 - (56 2) 3253148
3. Instituciones públicas o fiscales: www.chilecompra.cl

Desde el extranjero:
1. Liberalia Ediciones: www.liberalia.cl
2. Librería Antártica: www.antartica.cl
3. Argentina: Ediciones Manantial: www.emanantial.com.ar
4. Colombia: Editorial Siglo del Hombre
   Fono: (571) 3377700
5. España: Tarahumara, tarahumara@tarahumaralibros.com
   Fono: (34 91) 3656221
6. México: Alejandría Distribución Bibliográfica, alejandria@alejandrialibros.com.mx
   Fono: (52 5) 556161319 - (52 5) 6167509
7. Perú: Librería La Familia, Avenida República de Chile # 661
8. Uruguay: Dolmen Ediciones del Uruguay
   Fono: 00-598-2-7124857

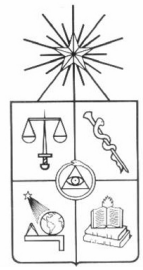
Grafos: Fundamentos y Algoritmos | Eduardo Moreno, Héctor Ramírez
Facultad de Igenieria y Ciencias, Universidad Adolfo Ibañez
Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile
eduardo.moreno@uai.cl, hramirez@dim.uchile.cl

ESTA PRIMERA EDICIÓN DE 2.000 EJEMPLARES
Se terminó de imprimir en febrero de 2011 en WORLDCOLOR CHILE S.A.

GRAFOS: FUNDAMENTOS Y ALGORITMOS

Eduardo Moreno
Universidad Adolfo Ibañez

Héctor Ramírez
Universidad de Chile



Dirección del Proyecto Fondef D05I-10211
Herramientas para la Formación de Profesores de Matemática

Patricio Felmer, Director del Proyecto
Universidad de Chile.

Leonor Varas, Directora Adjunta del Proyecto
Universidad de Chile.

Salomé Martínez, Subdirectora de Monografías
Universidad de Chile.

Cristián Reyes, Subdirector de Estudio de Casos
Universidad de Chile.

Presentación de la Colección

La colección de monografías que presentamos es el resultado del generoso esfuerzo de los autores, quienes han dedicado su tiempo y conocimiento a la tarea de escribir un texto de matemática. Pero este esfuerzo y generosidad no se encuentra plenamente representado en esta labor, sino que también en la enorme capacidad de aprendizaje que debieron mostrar, para entender y comprender las motivaciones y necesidades de los lectores: Futuros profesores de matemática.

Los autores, encantados una y otra vez por la matemática, sus abstracciones y aplicaciones, enfrentaron la tarea de buscar la mejor manera de traspasar ese encanto a un futuro profesor de matemática. Éste también se encanta y vibra con la matemática, pero además se apasiona con la posibilidad de explicarla, enseñarla y entregarla a los jóvenes estudiantes secundarios. Si la tarea parecía fácil en un comienzo, esta segunda dimensión puso al autor, matemático de profesión, un tremendo desafío. Tuvo que salir de su oficina a escuchar a los estudiantes de pedagogía, a los profesores, a los formadores de profesores y a sus pares. Tuvo que recibir críticas, someterse a la opinión de otros y reescribir una y otra vez su texto. Capítulos enteros resultaban inadecuados, el orden de los contenidos y de los ejemplos era inapropiado, se hacía necesario escribir una nueva versión y otra más. Conversaron con otros autores, escucharon sus opiniones, sostuvieron reuniones con los editores. Escuchar a los estudiantes de pedagogía significó, en muchos casos, realizar eventos de acercamiento, desarrollar cursos en base a la monografía, o formar parte de cursos ya establecidos. Es así que estas monografías recogen la experiencia de los autores y del equipo del proyecto, y también de formadores de profesores y estudiantes de pedagogía. Ellas son el fruto de un esfuerzo consciente y deliberado de acercamiento, de apertura de caminos, de despliegue de puentes entre mundos, muchas veces, separados por falta de comunicación y cuya unión es vital para el progreso de nuestra educación.

La colección de monografías que presentamos comprende una porción importante de los temas que usualmente encontramos en los currículos de formación de profesores de matemática de enseñanza media, pero en ningún caso pretende ser exhaustiva. Del mismo modo, se incorporan temas que sugieren nuevas formas de abordar los contenidos, con énfasis en una matemática más pertinente para el futuro profesor, la que difiere en su enfoque de la matemática para un ingeniero o para un licenciado en

matemática, por ejemplo. El formato de monografía, que aborda temas específicos con extensión moderada, les da flexibilidad para que sean usadas de muy diversas maneras, ya sea como texto de un curso, material complementario, documento básico de un seminario, tema de memoria y también como lectura personal. Su utilidad ciertamente va más allá de las aulas universitarias, pues esta colección puede convertirse en la base de una biblioteca personal del futuro profesor o profesora, puede ser usada como material de consulta por profesores en ejercicio y como texto en cursos de especialización y post-títulos. Esta colección de monografías puede ser usada en concepciones curriculares muy distintas. Es, en suma, una herramienta nueva y valiosa, que a partir de ahora estará a disposición de estudiantes de pedagogía en matemática, formadores de profesores y profesores en ejercicio.

El momento en que esta colección de monografías fue concebida, hace cuatro años, no es casual. Nuestro interés por la creación de herramientas que contribuyan a la formación de profesores de matemática coincide con un acercamiento entre matemáticos y formadores de profesores que ha estado ocurriendo en Chile y en otros lugares del mundo. Nuestra motivación nace a partir de una creciente preocupación en todos los niveles de la sociedad, que ha ido abriendo paso a una demanda social y a un interés nacional por la calidad de la educación, expresada de muy diversas formas. Esta preocupación y nuestro interés encontró eco inmediato en un grupo de matemáticos, inicialmente de la Universidad de Chile, pero que muy rápidamente fue involucrando a matemáticos de la Pontificia Universidad Católica de Chile, de la Universidad de Concepción, de la Universidad Andrés Bello, de la Universidad Federico Santa María, de la Universidad Adolfo Ibáñez, de la Universidad de La Serena y también de la Universidad de la República de Uruguay y de la Universidad de Colorado de Estados Unidos.

La matemática ha adquirido un rol central en la sociedad actual, siendo un pilar fundamental que sustenta el desarrollo en sus diversas expresiones. Constituye el cimiento creciente de todas las disciplinas científicas, de sus aplicaciones en la tecnología y es clave en las habilidades básicas para la vida. Es así que la matemática actualmente se encuentra en el corazón del currículo escolar en el mundo y en particular en Chile. No es posible que un país que pretenda lograr un desarrollo que involucre a toda la sociedad, descuide el cultivo de la matemática o la formación de quienes tienen la misión de traspasar de generación en generación los conocimientos que la sociedad ha acumulado a lo largo de su historia.

Nuestro país vive cambios importantes en educación. Se ha llegado a la convicción que la formación de profesores es la base que nos permitirá generar los cambios cualitativos en calidad que nuestra sociedad ha impuesto. Conscientes de que la tarea formativa de los profesores de matemática y de las futuras generaciones de jóvenes es extremadamente compleja, debido a que confluyen un sinnúmero de factores y disciplinas, a través de esta colección de monografías, sus editores, autores y todos los que han participado del proyecto en cada una de sus etapas, contribuyen a esta tarea, poniendo a disposición una herramienta adicional que ahora debe tomar vida propia en los formadores, estudiantes, futuros profesores y jóvenes de nuestro país.

Patricio Felmer y Salomé Martínez
Editores

Agradecimientos

Agradecemos a todos quienes han hecho posible la realización de este proyecto Fondef: “Herramientas para la formación de Profesores de Matemáticas". A Cristián Cox, quien apoyó con decisión la idea original y contribuyó de manera crucial para obtener la participación del Ministerio de Educación como institución asociada. Agradecemos a Carlos Eugenio Beca por su apoyo durante toda la realización del proyecto. A Rafael Correa, Edgar Kausel y Juan Carlos Sáez, miembros del Comité Directivo. Agradecemos a Rafael Benguria, Servet Martínez y Fidel Oteiza, miembros del Comité Editorial de la colección, quienes realizaron valiosos aportes a los textos. A Guillermo Marshall, Decano de la Facultad de Matemáticas de la Pontificia Universidad Católica de Chile y José Sánchez, entonces Decano de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Concepción, quienes contribuyeron de manera decisiva a lograr la integridad de la colección de 15 monografías. A Jaime San Martín, director del Centro de Modelamiento Matemático por su apoyo durante toda la realización del proyecto. Agradecemos a Víctor Campos, Ejecutivo de Proyectos de Fondef, por su colaboración y ayuda en las distintas etapas del proyecto.

Agradecemos también a Bárbara Ossandón de la Universidad de Santiago, a Jorge Ávila de la Universidad Católica Silva Henríquez, a Víctor Díaz de la Universidad de Magallanes, a Patricio Canelo de la Universidad de Playa Ancha en San Felipe y a Osvaldo Venegas y Silvia Vidal de la Universidad Católica de Temuco, quienes hicieron posible las visitas que realizamos a las carreras de pedagogía en matemática. Agradecemos a todos los evaluadores, alumnos, académicos y profesores -cuyos nombres no incluimos por ser más de una centena- quienes entregaron sugerencias, críticas y comentarios a los autores, que ayudaron a enriquecer cada uno de los textos.

Agradecemos a Marcela Lizana por su impecable aporte en todas las labores administrativas del proyecto, a Aldo Muzio por su colaboración en la etapa de evaluación, y también a Anyel Alfaro por sus contribuciones en la etapa final del proyecto y en la difusión de los logros alcanzados.

Dirección del Proyecto

Índice General

Prefacio 17

Capítulo 1: Introducción a los algoritmos 21

1.1 Introduciendo formalmente los algoritmos 21
1.2 Tipos de algoritmos 23
1.3 Eficiencia de un algoritmo 37
1.4 Ejercicios 45

Capítulo 2: Grafos 49

2.1 Introducción 49
2.2 Grafos no-dirigidos 51
2.3 Grafos dirigidos 64
2.9 Ejercicios 67

Capítulo 3. Árbol recubridor de costo mínimo 69

3.1 Árbol recubridor de costo mínimo 69
3.8 Ejercicios 74

Capítulo 4. Camino más corto 75

4.1 Camino más corto con costos positivos 75
4.2 Camino más corto con costos negativos 79
4.9 Ejercicios 83

Capítulo 5. Flujo en Redes: Problema de Flujo Máximo 85

5.1 Definiciones básicas 85
5.2 Problema de flujo máximo 88
5.3 Ejercicios 105

Capítulo 6. Ciclos 109

6.1 Ciclos Eulerianos 109
6.2 Ciclos Hamiltonianos 112

6.3 Las clases $\mathcal{P}$ y $\mathcal{NP}$ 114
6.4 Ejercicios 116

Bibliografía 119
Índice de Figuras 121
Índice de Términos 123

Prefacio

La teoría de grafos es un área relativamente nueva de las matemáticas, siendo considerado el trabajo de Leonhard Euler, sobre el problema de los puentes de Königsberg (1736), el primero en este tema. A partir de ese momento, hemos presenciado un vertiginoso crecimiento gracias a importantes aportes que han hecho matemáticos, ingenieros y otros científicos, quienes han encontrado en esta área las herramientas necesarias para modelar y resolver problemas de muy distinta índole. Es, sin duda, la simpleza de los conceptos estudiados lo que ha motivado esta importante evolución. Sin embargo, esta simpleza contrasta fuertemente con las dificultades a las cuales nos enfrentamos al estudiar las distintas conjeturas planteadas desde el primer trabajo de Euler, conjeturas que abarcan desde determinar si es posible colorear un mapa con sólo cuatro colores, probada sólo a mediados del siglo XX con la ayuda de los computadores, hasta la conjetura $\mathcal{P} \neq \mathcal{NP}$, que sigue siendo una pregunta abierta en nuestros días y se explica brevemente en el Capítulo 6 de esta monografía.

El estudio de la teoría de grafos ha ido de la mano con el desarrollo de algoritmos, para resolver los variados tipos de problemas que aparecen naturalmente en esta temática. Esto explica por qué la teoría de grafos ha sido también de vital importancia para las ciencias de la computación que, gracias al explosivo desarrollo que han tenido los computadores desde mediados del siglo XX, se ha consolidado como una ciencia fuerte e independiente, incentivando y potenciando, a su vez, la investigación en teoría de grafos. Esta interacción está en el centro de esta monografía. En efecto, esta monografía consta de seis capítulos. Los dos primeros son introductorios, dedicados a algoritmos y a los conceptos básicos en teoría de grafos, respectivamente. Los cuatro últimos capítulos tratan cada uno sobre un problema clásico en teoría de grafos, estudiando sus principales propiedades y su resolución mediante algoritmos apropiados para cada problema. Estos últimos capítulos han sido redactados de forma independiente y no requieren mayor conocimiento que el entregado en los dos primeros. A continuación, detallaremos los contenidos de cada uno de los seis capítulos de esta monografía.

En el Capítulo 1 se introduce el concepto de algoritmo, explicando qué es un algoritmo, identificando distintos tipos de algoritmos, definiendo la eficiencia de un algoritmo dado mediante el concepto de complejidad computacional e ilustrando cómo se calcula tal eficiencia. Estos conceptos son explicados con ejemplos de la vida cotidiana, que no están relacionados con grafos, por lo que este primer capítulo es

autocontenido e independiente del resto. Sin embargo, es fundamental para la lectura de los últimos cuatro. En el Capítulo 2 se explica qué es un grafo (dirigido y no-dirigido) y se introducen las principales definiciones y propiedades que éstos tienen. Éstas son ampliamente utilizadas en los capítulos posteriores. En el Capítulo 3 se estudia el problema de conectar un grafo, construyendo un árbol recubridor de peso mínimo. En el Capítulo 4 se analiza el problema de encontrar el camino más corto en un grafo, cuyas aristas pueden tener costos positivos o negativos. En el Capítulo 5 se introducen el concepto de red y nociones asociadas como corte y flujo factible. Luego, se estudia el problema de flujo en redes, que consiste en encontrar el máximo flujo que se puede enviar desde un origen a un destino dentro de la red. En cada uno de estos tres capítulos se estudia la manera de resolver eficientemente estos problemas, usando algoritmos apropiados para estos fines y analizando la complejidad obtenida para cada uno de ellos. Finalmente, en el Capítulo 6 se explica el problema de encontrar un ciclo que recorra, sin repetición, todos los vértices o aristas del grafo conocidos como ciclo Euleriano o Hamiltoniano, respectivamente. Además, usando estos problemas se entrega una breve intuición sobre los problemas de clase $\mathcal{P}$ y $\mathcal{NP}$.

Por motivos de espacio, hemos dejado de lado muchos interesantes tópicos en teoría de grafos como la coloración de grafos, la construcción de grafos a partir de una secuencia de grados, el problema del vendedor viajero, problemas en flujo en redes como el problema de transporte, entre otros. La elección de los temas tratados en esta monografía es responsabilidad de los autores. Para el lector interesado en profundizar en alguno de los temas que no han sido tratados aquí, hemos incluido en la bibliografía una selección de textos clásicos de teoría de grafos donde pueden encontrarse estos temas.

Esta monografía forma parte de la colección de monografías desarrolladas para el proyecto FONDEF D05I-10211 “Herramientas para la formación de profesores”, cuyo objetivo es fortalecer la formación inicial de profesores de matemáticas de enseñanza media. Por esta razón, nuestra orientación fue siempre la enseñanza de la teoría de grafos a alumnos de pedagogía en matemáticas. Teniendo esto en mente, tratamos de ilustrar los principales conceptos y problemas en teoría de grafos de una manera accesible, evitando el lenguaje árido que suele encontrarse en algunos libros de matemáticas universitarias, pero sin perder el formalismo propio de esta disciplina. Así, hemos puesto un especial énfasis en la modelación de problemas de la vida real (provenientes de la vida cotidiana o de la ingeniería) y su resolución vía los algoritmos introducidos en la monografía, incluyendo análisis de eficiencia e invitando al lector a comparar con otros algoritmos conocidos.

Queremos finalmente agradecer a las personas que de alguna manera han participado en la realización de esta monografía. A Patricio Felmer y Salomé Martínez, a cargo del proyecto FONDEF D05I-10211, por el constante apoyo y cuidadosas revisiones de versiones anteriores de este manuscrito. A los evaluadores externos; Eduardo Cabrera, Omar Gil, Alex Parada y Miryam Vicente, que aportaron con valiosas ideas y comentarios. A Iván Correa y a la Universidad Metropolitana de Ciencias de la Educación (Santiago, Chile) por permitirnos realizar un curso sobre teoría de grafos en esta institución, lo que nos permitió probar el uso de esta monografía, y a los alumnos de este curso por sus comentarios y opiniones que sin duda enriquecieron este texto.

Santiago, Julio del 2010 Eduardo Moreno, Héctor Ramírez

Capítulo 1: Introducción a los algoritmos

Los algoritmos son algo que utilizamos todos los días. Desde que despertamos realizamos un *algoritmo* que incluye generalmente ducharse, lavarse los dientes, vestirse, peinarse, tomar desayuno, entre otras actividades. De esta forma, todas las mañanas al realizar este *algoritmo* terminamos preparados para salir de la casa a trabajar y realizar nuestras labores diarias.

¿Qué es un algoritmo? No existe una definición precisa de este concepto, pero en palabras sencillas, podemos considerar que un *algoritmo* es *una secuencia finita y ordenada de pasos para realizar una tarea en forma precisa*. Bajo esta premisa, podemos ver que casi todo lo que hacemos en nuestra vida diaria son algoritmos. Sin embargo, el concepto de algoritmo es particularmente utilizado en la matemática y computación, y es este enfoque el que estudiaremos.

La palabra algoritmo viene del matemático persa al-Khwarizmi del siglo IX d.C., quien en su trabajo unió el sistema numérico indio con los conceptos algebraicos de occidente, permitiendo la introducción del *cero* a la matemática e incluso desarrollando *algoritmos* para resolver ecuaciones lineales y cuadráticas, dando origen así al álgebra.

1.1 Introduciendo formalmente los algoritmos

Para efectos de esta monografía consideraremos que un algoritmo tiene tres partes fundamentales: una *entrada*, que son los datos necesarios para que el algoritmo funcione, un *procedimiento* o *rutina* principal que consiste en una cantidad finita de pasos ordenados o *instrucciones* que son ejecutadas sobre los datos de entrada, y una *salida*, que es el resultado que se obtiene del algoritmo. Además, un algoritmo debe ser "preciso", es decir, que al ejecutar un algoritmo más de una vez sobre una misma entrada, deberíamos obtener siempre la misma salida. A modo de ejemplo describamos un algoritmo para preparar un "queque":

Algoritmo 1.1 Preparar un queque

Entrada: 2 tazas harina, 2 tazas azúcar, 100 gramos mantequilla, 1 taza leche, 3 huevos.

1: Mezclar el azúcar con la mantequilla hasta que quede cremosa.
2: Agregar los huevos y mezclar.
3: Agregar la harina.
4: Mezclar suavemente, agregando la leche de a poco.
5: Cocinar a horno bajo durante 45 minutos.

Salida: un queque.

El Algoritmo 1.1 tiene efectivamente una entrada (los ingredientes), un procedimiento, que consiste en el conjunto de pasos entre las líneas 1 y 5, necesarios para obtener una salida (un queque). Además, el algoritmo es preciso en el sentido antes definido, es decir, si lo realizamos diez veces deberíamos obtener diez queques iguales.

1.1.1 Pseudo-lenguaje de programación

En el ámbito de las matemáticas y la computación, necesitamos un *pseudo-lenguaje* para poder escribir los algoritmos, esto es, una Serie de comandos y sentencias que nos permiten escribir las rutinas o procedimiento necesarios para la creación de un algoritmo. Estos han sido elegidos adoptando una convención arbitraria, similar a la que se ve en lenguajes de programación como PASCAL, C, JAVA, MATLAB, entre otros. Estos comandos deben describir de manera general, pero precisa, las acciones que deseamos se ejecuten para obtener la salida deseada, pudiendo también ser fácilmente programados en alguno de los lenguajes antes mencionados.

Lo primero que debemos establecer es la forma de asignar valores a las variables. Esto lo escribiremos de la forma $a \leftarrow valor$. Por ejemplo, el siguiente algoritmo recibe como entrada un número, le suma 1 y retorna su nuevo valor.

Algoritmo 1.2 Incrementa en uno el valor de la entrada

Entrada: un número a

$a \leftarrow a + 1$ /* Se incrementa el valor de a */

Salida: el número a incrementado en 1

Nótese que para nosotros la variable a no necesitará ser definida previamente. Cuando queramos comentar algo dentro de una rutina, se usarán los símbolos "/*" y "*/". El comentario encerrado entre estos dos símbolos no ejecutará ninguna acción dentro del algoritmo, pues sólo servirá para darnos información sobre lo que se está programando.

Además de asignar valores, definiremos ciertas sentencias que nos permitirán realizar un conjunto de pasos si se cumple una condición, o durante un número predefinido de veces, o mientras cierta condición sea verdadera. Estas sentencias son:

si, entonces, sino: Realiza un conjunto de pasos si y sólo si se cumple una condición. Se utiliza de la siguiente forma:

si condición **entonces**
pasos a realizar
sino y **si** otra condición **entonces**
otros pasos a realizar
sino
otros pasos
fin

para: Repite una rutina un número predeterminado de veces, dado por un índice definido por el usuario. Este índice es muchas veces usado por dicha rutina:

para $j = 1$ hasta n
conjunto de pasos
fin

mientras: Se ejecuta una rutina mientras la condición dada en la primera sentencia sea cierta (es decir tenga valor "verdadero"). Su sintaxis es la siguiente:

mientras condición
pasos
fin

En las sentencias que hemos definido en esta sección, se ha usado el comando **fin** para indicar cuando termina la rutina que se ejecutará en una sentencia. Con el fin de ahorrar notación, omitiremos el uso del comando **fin** cuando exista sólo una instrucción a ejecutar en una sentencia, siempre que no haya posibilidad de confusión. En este caso, la instrucción se escribirá en la misma línea que la sentencia. En la próxima sección daremos ejemplos de algoritmos que nos permitirán entender mejor el uso de este pseudo-lenguaje.

1.2 Tipos de algoritmos

En esta sección describiremos tres nociones que aparecen reiteradamente en el contexto de algoritmos: *iteración, inducción* y *recursión*. La primera de ellas, *iteración*, es usada en los algoritmos para repetir una rutina una gran cantidad de veces sin que esta rutina sea especificada cada vez que se ejecuta. Por ejemplo, podemos realizar un procedimiento para subir una escalera de n escalones como sigue:

Repetir n veces: Subir un escalón

En lugar de repetir explícitamente la instrucción "subir un escalón" durante n veces:

Subir un escalón, Subir un escalón, ..., Subir un escalón

Esto nos permite establecer una *algoritmo iterativo* para subir una escalera usando el lenguaje de pseudo-programación definido en la Sección 1.1.1:

Entrada: n /* número de escalones */
  para $i = 1$ hasta n
    Subir un escalón
  fin

Dada la naturaleza de este algoritmo, no ha sido necesario especificar una salida. En el anterior algoritmo hemos supuesto que $n \geq 1$, es decir, que siempre hay, al menos, un escalón a subir.

Discutiremos más sobre la noción de iteración en la Sección 1.2.1.

La segunda de estas nociones, *inducción*, es utilizada en matemáticas para probar que cierta proposición es verdadera (*prueba por inducción*), y en matemáticas e informática para definir ciertos conceptos (*definiciones inductivas o recursivas*). Procederemos a explicar brevemente este primer uso en el próximo párrafo, mientras que el concepto de definición inductiva serátratado en la Sección 1.2.2.

En una prueba por inducción se considera una proposición lógica $S(n)$, la cual depende de una variable n (generalmente considerada como un número natural). Así, se busca probar que $S(n)$ es verdadera para todo valor de n. La manera de proceder es la siguiente: primero, se prueba una *base* o *caso base*, es decir $S(n)$ para un valor inicial $n = n_0$, usualmente elegido como $n_0 = 0$ o $n_0 = 1$. Luego, se prueba el *paso inductivo*, que corresponde a decir que el valor de verdad de $S(n+1)$, para un cierto n fijo, es consecuencia de los valores de verdad de las sentencias $S(i)$ anteriores ($i < n+1$). En el caso más simple, donde el valor de verdad de $S(n+1)$ sólo depende de la proposición inmediatamente anterior, el paso inductivo se puede identificar con la proposición lógica $S(n) \Rightarrow S(n+1)$. Estos dos pasos nos permiten concluir que $S(n)$ es cierta para todo n. Para dar una idea más concreta sobre cómo funciona una prueba inductiva, recurriremos a un tema que nos provee de numerosos ejemplos, donde las demostraciones por inducción aparecen naturalmente, las *sumatorias*. Podemos, a grandes rasgos, definir una sumatoria $\sum_{i=1}^{n} a_i$ como la suma de los elementos a_i's, indexados por i que varía de 1 a n, es decir:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + ... + a_n.$$

Ilustremos, entonces, el uso de la prueba inductiva mediante la demostración de la siguiente propiedad:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad \forall n \in \mathbb{N}, \tag{1.1}$$

es decir, demostremos por inducción que la suma de los primeros n naturales viene dada por $n(n+1)/2$. Si, para un número natural n fijo, llamamos $S(n)$ a la igualdad

en (1.1), notamos que $S(1)$ se verifica directamente pues $\sum_{i=1}^{1} i$ es claramente 1 y $1(1+1)/2 = 1$, esto prueba el *caso base*. Probemos ahora el paso inductivo $S(n) \implies S(n+1)$. Notemos que $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1)$, así, si la proposición $S(n)$ es cierta, la anterior igualdad nos lleva a

$$\sum_{i=1}^{n+1} i = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2},$$

que corresponde exactamente a decir que la proposición $S(n+1)$ es cierta (basta reemplazar n por $n+1$ en la definición de $S(n)$). Esto nos permite concluir que $S(n)$ es cierta para todo n, es decir, que (1.1) es cierta.

Finalmente, la última de estas nociones, *recursión*, se refiere a una técnica con la cual cierto concepto puede ser definido, directa o indirectamente, por él mismo. Por ejemplo, podemos observar que un paso clave para demostrar (1.1) fue el notar que $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1)$, es decir, que si denotamos por $P(n)$ a la sumatoria de los n primeros enteros, podríamos redefinir este operador o *procedimiento* (en el sentido que se puede implementar en un computador como una serie de pasos) de la siguiente forma:

$$P(1) = 1; \quad P(n+1) = P(n) + (n+1), \ \forall n \geq 1.$$

Esto es una definición recursiva o por recursión, hecha de manera directa, es decir, para definir $P(n)$ se usó o "se llamó", directamente al mismo procedimiento, pero para un n menor. Un procedimiento P cualquiera puede también "llamarse indirectamente", esto es, llamar a otro procedimiento P_1, el cual puede a su vez llamar a otro procedimiento P_2, y así, hasta llegar a un procedimiento P_r que finalmente llama a P. El uso de recursividad, cuando es aplicable, lleva a definir conceptos de manera más simple y directa, lo que resulta también más fácil de analizar y entender. Sin embargo, el buen uso de esta técnica necesita cierta práctica. Profundizaremos un poco más en este tema en la Sección 1.2.2.

Deseamos cerrar esta discusión con el problema de subir una escalera de n escalones. Usando recursión, podemos ahora dar un algoritmo alternativo al método iterativo entregado anteriormente. Este algoritmo será llamado *subir_ escalera* y es definido como sigue:

Algoritmo 1.3 subir_escalera

```
Entrada: n                    /* número de escalones */
  si n = 0 entonces
    Parar                     /* pues hemos llegado */
  sino
    Subir un escalón
    Ejecutar subir_escalera(n − 1)
  fin
```

Dada la naturaleza de esta rutina, no ha sido necesario especificar una salida. Como hemos notado en este último ejemplo, las nociones de iteración y recursión nos permiten también clasificar algunos algoritmos.

1.2.1 Definiciones y procedimientos iterativos

La iteración aparece naturalmente en los algoritmos con el uso de las rutinas **mientras** y **para** definidas en la Sección 1.1.1. En esta sección ilustraremos este hecho a través del algoritmo iterativo *selección*, que considera como entrada una *lista* y la "ordena" haciendo uso de estas rutinas iterativas. Para comenzar, demos una primera definición descriptiva de lo que entendemos por una lista:

Definición 1.1. Una *lista* es un conjunto finito de elementos que pueden estar repetidos, cuya posición en el conjunto está claramente definida.

Observación 1.1. Notemos que el mismo concepto de lista puede ser redefinido de manera iterativa como sigue: una lista es un conjunto vacío o es un elemento seguido por otro y luego otro, y así sucesivamente, hasta llegar a un elemento final (pues estamos considerando sólo listas finitas).

Precisaremos ahora a qué nos referimos con ordenar una lista de elementos. Supondremos que para los elementos de la lista en cuestión existe una relación de *orden total*, es decir, una relación de *orden*, que denotaremos por $\leq$, la cual satisface que, para todo par de elementos a, b de la lista, siempre se tiene que $a \leq b$ o que $b \leq a$. Bajo esta hipótesis, ordenar una lista de n elementos, digamos $(a_1, a_2, ..., a_n)$, corresponde a un procedimiento que permuta las posiciones de los elementos de la lista, obteniendo una nueva lista $(b_1, b_2, ..., b_n)$, que cumple con $b_1 \leq b_2 \leq ... \leq b_n$.

Ejemplo 1.1. Si consideramos la lista de números naturales (8,1,6,3,1,3,5,2) con el orden $\leq$ usual en los números reales $\mathbb{R}$, el procedimiento de ordenar esta lista corresponde a permutar sus elementos hasta obtener (1,1,2,3,3,5,6,8). Notemos que, debido a la definición de lista, la lista ordenada preserva la cantidad de veces que cada elemento aparece en la lista original. Así, vemos que los números 1 y 3 aparecen dos veces cada uno en la lista ordenada.

El ordenamiento de listas es un tema bien estudiado en la teoría de algoritmos y en las ciencias de computación, pues permite ejemplificar fácilmente los principales conceptos (tal como lo haremos en este primer capítulo) y tiene una gran cantidad de aplicaciones. Podemos nombrar, por ejemplo, el ordenamiento de la información de un grupo de personas en una base de datos, usando alguna característica como nombre, RUT, sueldo, puntaje en la PSU o cualquier otro campo de información que tenga una relación de orden total bien definida (en el caso del nombre es el *orden lexicográfico*). Esto es lo que planillas computacionales de cálculo hacen cuando les pedimos ordenar una base de datos según la información de algún campo específico y, para realizar rápidamente tal ordenamiento, recurren a los algoritmos que a continuación introduciremos.

Algoritmo iterativo *selección.* Consideraremos como entrada para este algoritmo la lista $(a_1, a_2, ..., a_n)$ dotada de un orden total $\leq$. Nuestra meta es entregar una lista ordenada $(b_1, b_2, ..., b_n)$ y, para realizar esto, iteraremos de la manera siguiente:

- Llamaremos también $(b_1, b_2, ..., b_n)$ a la lista que se utiliza en cada iteración[1].
- En el comienzo de cada iteración tendremos un índice $i \in \{1, 2, ..., n\}$ que nos mostrará los elementos que ya están ordenados. De esta manera, los primeros $i-1$ elementos de la lista de la actual iteración $(b_1, b_2, ..., b_n)$ ya están ordenados y, por lo tanto, nuestra iteración no hará intervención alguna en esta parte de la lista (por ejemplo, en un comienzo $i = 1$, es decir, *a priori* la lista de entrada puede estar completamente desordenada y entonces empezamos nuestro algoritmo desde el primer elemento).
- Sobre los $n - i + 1$ elementos restantes, elegimos o *seleccionamos* el más pequeño según el orden $\leq$ (o uno de los más pequeños, en caso que se repitan), digamos b_j con $j \geq i$.
- Intercambiamos b_j con b_i, aumentamos el índice i en uno ($i \leftarrow i + 1$) y repetimos este procedimiento iterativo.

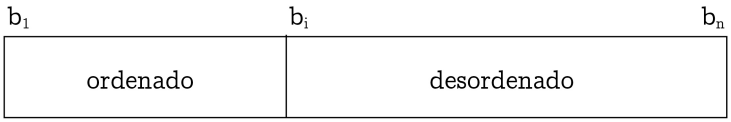


FIGURA 1.1. Lista $(b_1, b_2, ..., b_n)$ justo antes de la iteración i-ésima del algoritmo selección

Notemos que este procedimiento aumenta exactamente en un elemento la parte ordenada de la lista $(b_1, b_2, ..., b_n)$ en cada iteración, por lo que el algoritmo terminará con una lista ordenada en exactamente $n - 1$ pasos. Procederemos ahora a describir este algoritmo, usando las rutinas definidas en el Sección 1.1.1.

[1]Esto no debería dar lugar a confusión, pues al final de nuestro algoritmo esta lista será nuestra salida, coincidiendo con el significado que originalmente se le dio a esta notación.

Algoritmo 1.4 Selección

Entrada: $(a_1, a_2, ..., a_n)$

1: $(b_1, b_2, ..., b_n) \leftarrow (a_1, a_2, ..., a_n)$ /* asignación inicial */
2: **para** $i = 1$ hasta $n - 1$
3: $pequeno \leftarrow i$ /* La variable pequeño nos permite seleccionar el elemento más pequeño del final de la lista */
4: **para** $j = i + 1$ hasta n
5: **si** $b_j < b_{pequeno}$ **entonces**
6: $pequeno \leftarrow j$
7: **fin**
8: **fin** /* Al final de esta segunda rutina "**para**" se ha seleccionado el elemento más pequeño de la lista $(b_i, ..., b_n)$ */
9: $auxiliar \leftarrow b_{pequeno}$
10: $b_{pequeno} \leftarrow b_i$
11: $b_i \leftarrow auxiliar$ /* Se ha hecho uso de la variable auxiliar para intercambiar los valores de b_i y $b_{pequeno}$ */
12: **fin**

Salida: $(b_1, b_2, ..., b_n)$

Observación 1.2. Desde el punto de vista computacional la asignación inicial

$$(b_1, b_2, ..., b_n) \leftarrow (a_1, a_2, ..., a_n)$$

es innecesaria, pues se puede trabajar directamente sobre la lista $(a_1, a_2, ..., a_n)$. Esto sólo fue hecho para explicar más claramente el algoritmo. También se ha optado por no utilizar la letra ñ dentro del código del Algoritmo 1.4.

Ejemplo 1.2. Estudiemos el funcionamiento del algoritmo de selección en la siguiente lista $(4, 3, 8, 3, 6)$. La línea 1 del algoritmo asigna $(b_1, b_2, ..., b_n) = (4, 3, 8, 3, 6)$. Luego, se comienza el ciclo **para** con $i = 1$ en la línea 2, cuyo primer paso consiste en asignar, en la línea 3, el valor 1 a la variable *pequeno*. El ciclo **para** de las líneas 4 a 8 asigna el valor más pequeño de $(b_i, b_{i+1}, ..., b_n)$ a la variable *pequeno*, así, en nuestro ejemplo, podemos ver que la condición dada en la línea 5 se cumple para $j = 2$ (pues $b_2 = 3 < 4 = b_1$) y, por lo tanto, se asigna momentáneamente el valor 2 a la variable *pequeno*. Para el resto de los valores de j ($j = 3, 4, 5$) la condición de la sentencia **si** en la línea 5 no se cumple (pues $b_2 = 3$ no es mayor estricto que 8, 3 ó 6) y luego, una vez finalizado este ciclo **para**, la variable *pequeno* termina con el valor 2. Entre los pasos 9 y 12 se intercambian, entonces, los valores de b_1 y b_2, obteniendo la lista $b = (3, 4, 8, 3, 6)$ al final de la primera iteración del ciclo **para**, definido entre las líneas 2 y 12.

La segunda iteración de este ciclo **para**, comienza ahora con $i = 2$. Se asigna a la variable *pequeno* el valor 2 y se procede con el ciclo **para** de las líneas 4 a 8. La condición de la línea 5 no se cumple para $j = 3$, pues $b_2 = 4$ no es mayor estricto que

$b_3 = 8$, pero sí se cumple para $j = 4$, pues $b_4 = 3 < 4 = b_2$, luego, se asigna el valor 4 a la variable *pequeno* en la línea 6. Hecha esta asignación la condición de la línea 5 no se cumple para $j = 5$, pues $b_4 = 3$ no es mayor estricto que $b_5 = 6$. Se termina, entonces, este ciclo **para** con la variable *pequeno*, valiendo 4 y luego, se procede a intercambiar b_2 con b_4, obteniendo al final del ciclo **para** principal (líneas 2 a 12) la lista $b = (3, 3, 8, 4, 6)$.

La tercera y cuarta iteración de este ciclo **para** ($i = 3, 4$, respectivamente) no se explicarán en detalle, pero podemos ver que éstos terminan con los valores 4 y 5 asignados a la variable *pequeno*, respectivamente, y las listas de salida son $b = (3, 3, 4, 8, 6)$ y $b = (3, 3, 4, 6, 8)$, respectivamente.

Hemos resumido este ejemplo en la Tabla 1.1. La segunda columna nos indica el valor de la variable *pequeno* obtenido a partir del ciclo **para** de las líneas 4 hasta 8, mientras que la tercera columna nos muestra el valor de b obtenido al final de la iteración i del ciclo **para** de las líneas 2 a la 12.

iteración i	variable *pequeno*	b
INICIO	-	$(4, 3, 8, 3, 6)$
1	2	$(3, 4, 8, 3, 6)$
2	4	$(3, 3, 8, 4, 6)$
3	4	$(3, 3, 4, 8, 6)$
4	5	$(3, 3, 4, 6, 8)$

Tabla 1.1. Funcionamiento del algoritmo Selección

Ejercicio 1.1. Aplique el algoritmo Selección para ordenar las listas de números $(9, 3, 2, 1, 4, 1, 2, 5, 6)$ y $(5, 2, 3, 7, 5, 1, 2, 7, 5)$. Realice una tabla similar a la dada en 1.1.

En la Sección 1.3 estudiaremos la *complejidad* del algoritmo de selección, es decir, estimaremos cuánto tiempo tarda este algoritmo en ordenar una lista.

1.2.2 Definiciones y procedimientos recursivos

Hemos mencionado que en una definición recursiva o inductiva, un concepto se define en términos de él mismo, ya sea directa o indirectamente. Sin embargo, esta definición no debe establecer una relación trivial que no aporte al entendimiento del concepto, como por ejemplo:

> *Una escalera es una escalera de cierto material o*
> *una lista es una lista de ciertos elementos*

ni tampoco puede ser una definición paradójica, como por ejemplo:

> *Un objeto es una escalera si y sólo si no es una escalera.*

De hecho, una definición recursiva siempre hace alusión a uno o varios casos simples, que llamaremos *base*, para los cuales el concepto está claramente definido y a un paso inductivo (como en las pruebas por inducción), donde el concepto que se define en términos de él mismo, pero para una caso "más pequeño" (en algún sentido que precisaremos a través de ejemplos).

Ejemplo 1.3. Retomemos la definición de una lista. Ésta ya fue hecha de dos maneras distintas, la definición original que podríamos llamar *descriptiva* y una definición iterativa dada en la Observación 1.1. También puede hacerse de manera recursiva, diciendo que "una lista es un conjunto vacío *(base)* o un elemento seguido de una lista *(paso inductivo)*". Notemos que la palabra "lista" que aparece al final del paso inductivo hace referencia a una lista que necesariamente tiene una cantidad menor de elementos que la lista original (de hecho tiene exactamente un elemento menos), pues las listas son finitas y estamos asegurando que hay un primer elemento que no está en esta segunda lista. Esto asegura que esta definición recursiva tiene sentido.

Otro ejemplo es el siguiente:

Ejemplo 1.4. El valor $n!$ (n factorial) corresponde a la multiplicación de los n primeros números naturales, es decir, a $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-1) \cdot n$. Este valor también puede ser definido de manera recursiva de la siguiente forma:

$$1! = 1 \text{ (base)}, \qquad n! = n \cdot (n-1)! \text{ (paso inductivo)}.$$

Ejercicio 1.2. Probar por inducción que ambas definiciones dadas, en el ejemplo anterior, coinciden.

De manera similar podemos definir un *procedimiento recursivo* como procedimiento que se "llama" a sí mismo, lo cual puede ser realizado de manera directa o indirecta, hasta llegar a un procedimiento base, cuya salida es calculada fácilmente. La utilización de estos procedimientos nos permiten programar de manera recursiva distintas tareas.

Programar en forma recursiva es muchas veces lo más natural, esto nos permite obtener procedimientos o algoritmos más concisos y fáciles de entender. Esta forma de programar está intrínsecamente ligada a la definición recursiva que describimos algunos párrafos atrás. En efecto, usando una definición recursiva, se procede a programar la *base* y luego, se llama recursivamente al mismo procedimiento para un caso "estrictamente más pequeño" (en el sentido que hemos mostrado en los ejemplos). Aclaremos esto con el siguiente ejemplo:

Ejemplo 1.5. Consideremos la definición recursiva de $n!$ dada en el Ejemplo 1.4. Una directa transcripción de esta definición nos lleva a describir el siguiente procedimiento recursivo para el cálculo de $n!$:

Algoritmo 1.5 Factorial(n)

Entrada: n /* número entero */
  si $n = 1$ **entonces**
    $factorial \leftarrow 1$ /* base */
  sino
    $factorial \leftarrow n * factorial(n-1)$ /* llamada recursiva */
  fin
Salida: $factorial = n!$

Para entender el funcionamiento de este algoritmo, y así dar una noción sobre cómo un procedimiento recursivo funciona, veamos cómo éste opera cuando se ejecuta *factorial(4)*. Según la definición, hará la llamada recursiva *factorial(3)*, la cual llamará a *factorial(2)*, que a su vez llamará a *factorial(1)*. En este caso hemos llegado a la base, por lo que se entrega un valor de salida a *factorial(1)* igual a 1, es decir, ésta es la primera vez que vemos una *salida* para nuestro procedimiento y corresponde a *Salida[factorial(1)]*$= 1$. Por lo tanto, el procedimiento *factorial* completa la primera llamada y, luego, en la sentencia **sino** asigna el valor *Salida[factorial(2)]*$= 2 \cdot 1 = 2$. Así, después de la segunda llamada se obtiene *Salida[factorial(3)]*$= 3 \cdot 2 = 6$, para finalmente completar la última llamada y dar como resultado final *Salida*=*Salida[factorial(4)]*$= 4 \cdot 6 = 24$. Este proceso ha sido resumido en la siguiente gráfica:

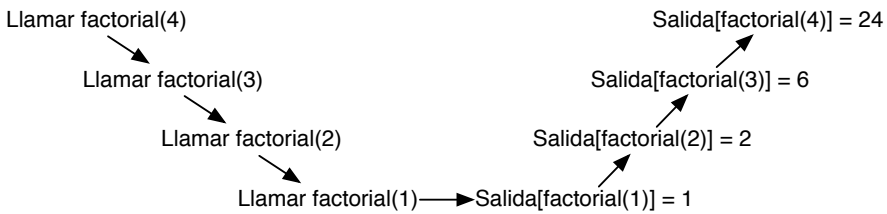


FIGURA 1.2. Llamadas y salidas realizadas por la rutina factorial(4)

Una manera muy conocida para resolver ciertos problemas es dividirlos en subproblemas, que son luego resueltos para combinar sus soluciones, obteniendo así la solución del problema original. Este apronte es generalmente llamado *dividir para conquistar*, y está fuertemente ligado a la noción de procedimiento recursivo. En efecto, si estos subproblemas tienen una estructura similar a la del problema original, entonces pueden ser resueltos de manera recursiva. Para que este enfoque funcione, los subproblemas no pueden ser más complicados de resolver que el problema original y debemos poder asegurar que después de una cantidad finita de pasos llegamos a

un subproblema que se resuelva directamente (es decir, a una *base*). A continuación describiremos un algoritmo o procedimiento que usa la técnica de dividir para conquistar, combinándola con recursión y que, al igual que el algoritmo de selección, nos permite ordenar una lista.

Algoritmo recursivo *fusión* (o *Mergesort* en inglés). Al igual que para el algoritmo selección, la entrada es una lista $(a_1, a_2, ..., a_n)$ dotada de un orden total $\leq$ y el objetivo es entregar una lista ordenada que llamaremos $(b_1, b_2, ..., b_n)$. Sin embargo, la manera de proceder de este algoritmo es muy distinta del de selección. Procede como sigue:

- Si la lista tiene un sólo elemento o es vacía se considera ordenada, por lo tanto, la salida es igual a la entrada (*base*).
- Una lista $(a_1, a_2, ..., a_n)$ de más de un elemento es divida en dos sublistas de un mismo tamaño: $a = (a_1, a_2, ..., a_k)$ y $\tilde{a} = (a_{k+1}, a_2, ..., a_n)$, donde $k = [n/2]$, esto es, $k = n/2$ si n es par, y $k = (n+1)/2$ si n es impar.[2]
- Se aplica a cada sublista el algoritmo *fusión*, de manera de obtener dos sublistas ordenadas.
- Las dos listas ordenadas obtenidas en el paso anterior se *fusionan* en una sola lista, la cual debe estar también ordenada, es decir, el procedimiento de fusión de estas dos listas debe ordenar también la lista resultante, que corresponderá a la salida del algoritmo. Ilustremos este último procedimiento: si tenemos dos listas (ya ordenadas) $(1, 1, 5, 6)$ y $(1, 3, 5, 7)$, la fusión de ambas listas nos debe entregar $(1, 1, 1, 3, 5, 5, 6, 7)$.

En la descripción anterior, no hemos explicado cómo se realiza el procedimiento de fusión de dos listas ordenadas que fue descrito en el último ítem. Este será descrito en los siguientes párrafos, luego, procederemos a dar una descripción en lenguaje de pseudo-programación del algoritmo de fusión.

Procedimiento de fusión de dos listas ordenadas. Para evitar cualquier confusión con la notación anterior, denotaremos las entradas de este procedimiento por $c = (c_1, c_2, ..., c_k)$ y $\tilde{c} = (\tilde{c}_1, \tilde{c}_2, ..., \tilde{c}_{\tilde{k}})$, que consisten en dos listas ordenadas. Por ejemplo, en el primer paso (donde se divide la lista $(a_1, a_2, ..., a_n)$) se tiene que la cantidad de elementos de ambas listas, k y $\tilde{k}$, suman n y satisfacen la relación $|k - \tilde{k}| \leq 1$, es decir, estas listas tienen la misma cantidad de elementos o bien una lista tiene sólo un elemento más que la otra. Por otro lado, la salida, que corresponde a una lista ordenada producto de la fusión de c y $\tilde{c}$, será denotada por $b = (b_1, b_2, ..., b_n)$ con $n = k + \tilde{k}$.[3] El procedimiento es descrito, entonces, como sigue:

[2]La elección de las sublistas puede ser hecha de manera arbitraria, siempre que estás tengan la misma cantidad de elementos o a lo más uno de diferencia. Por ejemplo, podríamos haber escogido la lista a como los elementos en posiciones pares y la lista $\tilde{a}$ como los elementos en posiciones impares.

[3]El uso de la la letra b, para denotar la salida de este procedimiento, no debería producir confusión, ya que la salida del procedimiento coincidirá con la salida final del algoritmo de fusión, que corresponde al significado original otorgado a la notación b.

- Si una de las dos listas, c o $\tilde{c}$, es vacía[4], entonces, la fusión de ambas corresponde simplemente a la otra lista. Es decir, si $c = \emptyset$ entonces $b = \tilde{c}$, y viceversa (*base*).
- Si ambas listas no son vacías, entonces, comparar los primeros elementos de cada lista, c_1 y $\tilde{c}_1$, seleccionar el menor de ellos y asignarlo a b_1, es decir $b_1 = \min\{c_1, \tilde{c}_1\}$.
- Si en el paso anterior el elemento seleccionado fue c_1, entonces, aplicar el procedimiento *fusión-2listas* a las listas $(c_2, ..., c_k)$ y $(\tilde{c}_1, \tilde{c}_2, ..., \tilde{c}_{\tilde{k}})$, la salida corresponderá a los valores $(b_2, b_3, ..., b_n)$. Proceder análogamente si el elemento seleccionado fue $\tilde{c}_1$.

Este procedimiento puede ser escrito en pseudo-lenguaje de programación (ver Sección 1.1.1) de la siguiente manera:

Algoritmo 1.6 *fusión-2listas*

Entrada: $c = (c_1, c_2, ..., c_k)$, $\tilde{c} = (\tilde{c}_1, \tilde{c}_2, ..., \tilde{c}_{\tilde{k}})$ /* las entradas son las listas ordenadas c y $\tilde{c}$ */
si c es vacía **entonces**
$b \leftarrow \tilde{c}$
sino y **si** $\tilde{c}$ es vacía **entonces**
$b \leftarrow c$ /* ambos casos corresponden a la base */
sino y **si** $c_1 \leq \tilde{c}_1$ **entonces**
$b \leftarrow (c_1, \textit{fusión-2listas}((c_2, ..., c_k), \tilde{c}))$
sino
$b \leftarrow (\tilde{c}_1, \textit{fusión-2listas}(c, (\tilde{c}_2, ..., \tilde{c}_k)))$ /* ambos casos corresponden a la llamada recursiva */
fin
Salida: b

Ejemplo 1.6. Ilustremos su funcionamiento con las listas $(1, 1, 5, 6)$ y $(1, 3, 5, 7)$. Hemos visto ya, que la fusión de ambas listas nos entrega $(1, 1, 1, 3, 5, 5, 6, 7)$. Esta lista coincide con la salida del algoritmo *fusión-2listas* cuyo funcionamiento se resume en la siguiente tabla:

[4]En este paso suponemos que el pseudo-lenguaje de programación que estamos usando permite identificar una lista vacía.

Lista c	Lista $\tilde{c}$	Lista final b
(1,1,5,6)	(1,3,5,7)	$\emptyset$
(1,5,6)	(1,3,5,7)	(1)
(5,6)	(1,3,5,7)	(1,1)
(5,6)	(3,5,7)	(1,1,1)
(5,6)	(5,7)	(1,1,1,3)
(6)	(5,7)	(1,1,1,3,5)
(6)	(7)	(1,1,1,3,5,5)
$\emptyset$	(7)	(1,1,1,3,5,5,6)
$\emptyset$	$\emptyset$	(1,1,1,3,5,5,6,7)

TABLA 1.2. Funcionamiento del algoritmo *fusión-2listas*

Ejercicio 1.3. El procedimiento de fusión de dos listas puede también ser efectuado de manera iterativa. Descríbalo, usando pseudo-lenguaje de programación.

Estamos ahora en condiciones de describir en lenguaje de pseudo-programación el algoritmo *fusión*, cuyo objetivo es ordenar una lista $(a_1, a_2, ..., a_n)$ dada. Según lo que hemos establecido anteriormente, para este algoritmo y la descripción del procedimiento para fusionar dos listas ya ordenadas, dadas por *fusión-2listas*, el algoritmo de fusión se escribe como sigue:

Algoritmo 1.7 *fusión*

Entrada: $a = (a_1, a_2, ..., a_n)$
1: **si** $n \leq 1$ **entonces**
2: $b \leftarrow a$ /* base */
3: **sino** /* La lista a tiene más de dos elementos */
4: $k \leftarrow [n/2]$
5: $c \leftarrow$ *fusión*$((a_1, a_2, ..., a_k))$
6: $\tilde{c} \leftarrow$ *fusión*$((a_{k+1}, a_{k+2}, ..., a_n))$ /* se divide la lista a en dos sublistas, las cuales se ordenan (llamada recursiva) y dan lugar a c y $\tilde{c}$ */
7: $b \leftarrow$ *fusion-2listas*$(c, \tilde{c})$ /* se fusionan las listas c y $\tilde{c}$ obteniendo b */
8: **fin**
Salida: b

Ejemplo 1.7. Mostremos cómo funciona el algoritmo de fusión, usando la lista $(4, 3, 8, 3, 6)$ del Ejemplo 1.2. Dado que la lista tiene $n = 5$ elementos la condición de la línea 1 no se cumple, por lo que se procede a ejecutar las sentencias a partir de la línea 3. En la línea 4 se calcula $k = [5] = 3$, lo que permite dividir la lista a en dos sublistas $(4, 3, 8)$ y $(3, 6)$, obteniendo las listas $c = (3, 4, 8)$ y $\tilde{c} = (3, 6)$ a partir de la aplicación recursiva del procedimiento fusión a estas dos listas en las líneas 5 y 6, respectivamente. Estas dos listas se encuentran ordenadas lo que nos permite aplicar

el Algoritmo 1.6 *fusión-2listas* en la línea 7, y así obtener como salida del algoritmo la lista ordenada $b = (3, 3, 4, 6, 8)$.

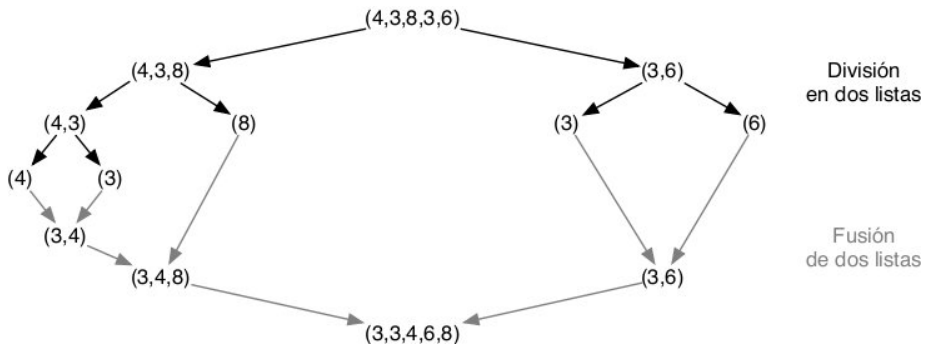


FIGURA 1.3. División y fusión recursiva

Ejercicio 1.4. Aplique el algoritmo Fusión para ordenar las listas de números dadas en el Ejercicio 1.1: $(9, 3, 2, 1, 4, 1, 2, 5, 6)$ y $(5, 2, 3, 7, 5, 1, 2, 7, 5)$. Realice una tabla similar a la Tabla 1.2.

1.2.3 Otros tipos de algoritmos

Áreas de las matemáticas como la optimización, el cálculo numérico, la teoría de números, el álgebra, entre otras, utilizan algoritmos para resolver problemas propios de sus dominios. Algoritmos ya clásicos en sus respectivas áreas como el método SIMPLEX para resolver un problema de optimización lineal, el método de Newton para encontrar los *ceros* de una función, el algoritmo de Euclides para calcular el máximo común divisor de dos números enteros y su posterior extensión para dividir polinomios, entre otros, son estudiados en otras monografías de la colección, por lo que no serán analizados en ésta. Sin embargo, cabe decir que las propiedades que usualmente se estudia en estos algoritmos tienen un enfoque distinto del que se aborda en esta monografía. En efecto, la mayoría de estos algoritmos podrían ser clasificados y analizados como algoritmos de iteración, por lo que la medición de su eficiencia (ver próxima Sección 1.3) es un asunto importante que puede ser formalmente estudiado con las herramientas de esta monografía. Sin embargo, el análisis del resto de las monografías se enfoca, principalmente, en la idea matemática usada para resolver el problema planteado, lo cual difiere de nuestro objetivo, que es mucho más general.

Es interesante mencionar algunos aprontes generales y unificadores para explicar algoritmos como, por ejemplo, en el contexto de la optimización, donde la mayoría de los métodos generan en cada iteración un buen candidato para resolver un problema de optimización dado, se han establecido esquemas basados en *multi-aplicaciones* o

también llamadas *funciones punto-conjunto* (funciones que a un elemento le asigna un conjunto, es decir, su imagen está contenida en el conjunto de las partes de un cierto espacio o conjunto). Estos trabajos, desarrollados por Willard I. Zangwill a fines de los años 60's, han permitido caracterizar de manera general la *convergencia* y, en algunos casos, la *velocidad de convergencia* (cuán rápido un criterio de optimalidad dado se mejora a medida que transcurren las iteraciones) de ciertos tipos de algoritmos. Recomendamos a un lector interesado en el tema consultar la Sección 1.4 del libro [**5**].

Otro tipo de algoritmos, que no serán abordados en profundidad en esta monografía, son los llamados *algoritmos heurísticos*. Estos algoritmos, independiente de su naturaleza (iterativo o recursivo), se caracterizan por el uso de una o varias reglas basadas en la experiencia, con las cuales se pretende resolver el problema planteado, pero sin necesariamente tener una demostración matemática que pruebe que la salida final del algoritmo resuelva satisfactoriamente el problema. Una estrategia de este tipo, que es muy utilizada en la práctica, es el *algoritmo glotón*. A grandes rasgos se puede definir como cualquier método que realice en cada iteración una regla "local" óptima (o eficiente, desde un cierto punto de vista) con la esperanza que ésta genere una salida que sea óptima en un sentido "global". Por ejemplo, veamos un algoritmo glotón que nos permita dar un vuelto de \$760 con la menor cantidad de monedas posibles. Las monedas que podemos utilizar tienen un valor de \$500, \$100, \$50, \$10, \$5 y \$1. Denotemos por a_i el vuelto que nos queda por dar en la iteración i del algoritmo (en un comienzo $a_1 = \$760$). Nuestra regla local consistirá en tratar siempre de dar la moneda de mayor valor, entre todas las que no superen el valor a_i. Formalmente, cada iteración i se describe como sigue:

- Si $a_i = 0$, entonces PARAR, pues ya hemos dado el vuelto requerido.
- Si $a_i \geq \$500$, entonces dar una moneda de \$500, y asignar $a_{i+1} = a_i - \$500$.
- Si no, pero si $a_i \geq \$100$, entonces dar una moneda de \$100, y asignar $a_{i+1} = a_i - \$100$.
- Si no, pero si $a_i \geq \$50$, entonces dar una moneda de \$50, y asignar $a_{i+1} = a_i - \$50$.
- etc.

Una vez ejecutada esta regla local, el algoritmo pasa a la próxima iteración $i + 1$.

Así, nuestro algoritmo entrega primero una moneda de \$500, luego dos de \$200 (en dos iteraciones distintas), continuamos con una de \$50 y finalizamos con una de \$10. Esto es ilustrado en la Figura 1.4.

Ejercicio 1.5. Escribir este algoritmo, de manera iterativa y recursiva, usando el pseudo-lenguaje de programación introducido en la Sección 1.1.1. ¿Puede entregar una prueba matemática que demuestre que este algoritmo da vuelto con la menor cantidad posible de monedas?

Los algoritmos glotones han servido para resolver eficientemente algunos problemas matemáticos, como, por ejemplo, los presentados en los Capítulos 3 (Algoritmos Prim y Kruskal) y 4 (Algoritmo Dijkstra), sin embargo, para muchos otros problemas

FIGURA 1.4. Algoritmo glotón usado para dar un vuelto de $760.

éstos no garantizan la obtención de una solución, debido principalmente a que no utilizan exhaustivamente toda la información del problema. Algunos problemas donde el algoritmo glotón falla son la construcción de ciclos Eulerianos y Hamiltonianos, que se introducen en el Capítulo 6.

1.3 Eficiencia de un algoritmo

La idea de utilizar algoritmos para resolver problemas concretos, ya sea en la vida diaria, en matemáticas, en computación o en otra área, es válida sólo si esta resolución tarda una cantidad "razonable" de tiempo. El objetivo de esta sección es explicar qué entendemos por "razonable" en este contexto, introduciendo la noción de *complejidad de un algoritmo*.

1.3.1 Complejidad de un algoritmo

Ilustremos la importancia de los principales conceptos a introducir en esta sección con un ejemplo. Supongamos que deseamos encontrar la clave numérica de un computador compuesta por n dígitos numéricos (1 al 10). Un posible algoritmo para encontrar esta clave es probar, una a una, cada combinación usando algún orden. Con esto, tendremos que probar, *en el peor caso*, las 10^n posibles combinaciones. Para ilustrar cuán poco eficiente es este algoritmo, supongamos que nos demoramos 0.1 segundos en probar

cada combinación, entonces nos demoraremos 1.000 segundos ($\approx$ 17 minutos) si la clave es de 4 dígitos ($n = 4$), 10.000 segundos ($\approx$ 2.8 horas) si es de 5 dígitos ($n = 5$), 100.000 segundos ($\approx$ 1.15 días) si es de 6 dígitos y, si pensamos en claves de 10 dígitos el tiempo que tardaríamos sería de 1.000.000.000 segundos que corresponde a más de 31 años!!. Notemos que, al comparar usando el tiempo del peor caso posible, un pequeño aumento en el "tamaño" del problema (en este caso el número de dígitos n) implica un enorme aumento en su tiempo de resolución.

En el ejemplo anterior hemos enfocado nuestra atención en dos aspectos: en el tiempo que demora el algoritmo en resolver algún problema, cuando el "tamaño" de este último crece y en el uso del tiempo de resolución en el *peor caso* como medida de eficiencia para un algoritmo. En efecto, en la teoría de algoritmos se estudia cuánto demora, en el *peor caso*, un algoritmo en resolver el problema que pretende solucionar, caracterizando el número de pasos u operaciones básicas que involucra la resolución como función del tamaño del problema. La forma como crece este tiempo será representada por una función $f : \mathbb{N} \to \mathbb{N}$ (generalmente se adoptan funciones bien conocidas como función constante, logaritmo, lineal, cuadrática, exponencial, etc.) y es lo que llamaremos la *complejidad del algoritmo*. Formalmente, establecemos lo siguiente:

Definición 1.2. Llamaremos *complejidad de un algoritmo* a la función $f : \mathbb{N} \to \mathbb{N}$ que a cada $n \in \mathbb{N}$ le asocia el valor máximo de pasos que puede demorar el algoritmo en resolver el problema estudiado, es decir, al tiempo de resolución del algoritmo en el *peor caso*.

Observación 1.3. Existen algoritmos, como el método SIMPLEX para resolver problemas de optimización lineal[5], que resultan satisfactoriamente rápidos en la práctica aun teniendo complejidad exponencial (ver definición en la Tabla 1.4), lo cual supondría una mala eficiencia en la resolución del problema. Situaciones como ésta han llevado a estudiar formas alternativas de medir la eficiencia de un algoritmo, que no sean basadas en el estudio del peor caso. Por ejemplo, otra medida de eficiencia comúnmente utilizada, es el análisis del *caso promedio*, que a grandes rasgos consiste en el promedio de cuánto tardaría nuestro algoritmo ejecutado sobre cada una de las entradas posibles. Si bien estas medidas alternativas pueden ser más adecuadas o más realistas para estudiar la eficiencia de ciertos algoritmos, son también mucho más difíciles de estimar, por lo que su uso es menor.

[5]Consultar detalles en la monografía "Optimización Lineal: una mirada introductoria"

1.3.2 Clases de complejidad

El cálculo de la complejidad de un algoritmo puede ser muy difícil y dado que sólo explica el tiempo de resolución en el peor caso, no siempre se considera útil un cálculo preciso de éste. Por ejemplo, en la Tabla 1.3 observamos que cuando multiplicamos por dos las funciones $f(n) = n^2$ y $f(n) = 2^n$, la forma en que estas funciones crecen no cambiar mayormente. Más aún, en la práctica estas constantes son difíciles de calcular con exactitud para un algoritmo dado, pudiendo sólo tener estimaciones poco precisas de ellas.

$n \setminus f$	$\log(n)$	n	$2n$	n^2	$2n^2$	2^n	$2 * 2^n$
1	0	0,1	0,2	0,1	0,2	0,2	0,4
2	0,1	0,2	0,4	0,4	0,8	0,4	0,8
3	0,158	0,3	0,6	0,9	1,8	0,8	1,6
4	0,2	0,4	0,8	1,6	3,2	1,6	3,2
5	0,232	0,5	1	2,5	5	3,2	6,4
6	0,258	0,6	1,2	3,6	7,2	6,4	12,8
7	0,28	0,7	1,4	4,9	9,8	12,8	25,6
8	0,3	0,8	1,6	6,4	12,8	25,6	51,2
9	0,317	0,9	1,8	8,1	16,2	51,2	102,4
10	0,332	1	2	10	20	102,4	204,8
100	0,664	10	20	1000	2000	$1{,}26765{\cdot}10^{29}$	$2{,}5353{\cdot}10^{29}$

TABLA 1.3. Tiempo que demora un algoritmo de $f(n)$ pasos, si realiza 1 paso en 0.1 segundos

Por estas razones, la eficiencia de un algoritmo generalmente se mide sólo usando un estimación superior de su complejidad, lo que permite agrupar las distintas funciones que las representan en *clases de complejidad*. Para formalizar estos conceptos, introducimos la siguiente noción de mayoración:

Definición 1.3. Sean dos funciones $f, g : \mathbb{N} \to \mathbb{N}$, decimos que $f(n)$ *está en* $O(g(n))$, si existen naturales c y n_0 tales que, para todo natural $n > n_0$

$$f(n) \leq c\ g(n).$$

El símbolo O suele llamarse *gran o*.

Una forma de interpretar esta definición es que, si $f(n)$ está en $O(g(n))$, entonces $f(n)$ está asintóticamente acotado superiormente por $g(n)$. Intuitivamente, esto significa que f es menor o igual que g, si hacemos caso omiso de diferencias menores que un factor constante. Por ejemplo, sea f la función $f(n) = 7n^4 + 2n^3 + 5n + 10$. Entonces, si tomamos el coeficiente de mayor orden ($7n^4$) y descartamos el coeficiente 7, obtenemos que $f(n)$ está en $O(n^4)$. En efecto, para $c = 8$ y $n_0 = 2$, se cumple con la Definición 1.3, ya que $7n^4 + 2n^3 + 5n + 10 < 8n^4$, para todo $n \geq 3$.

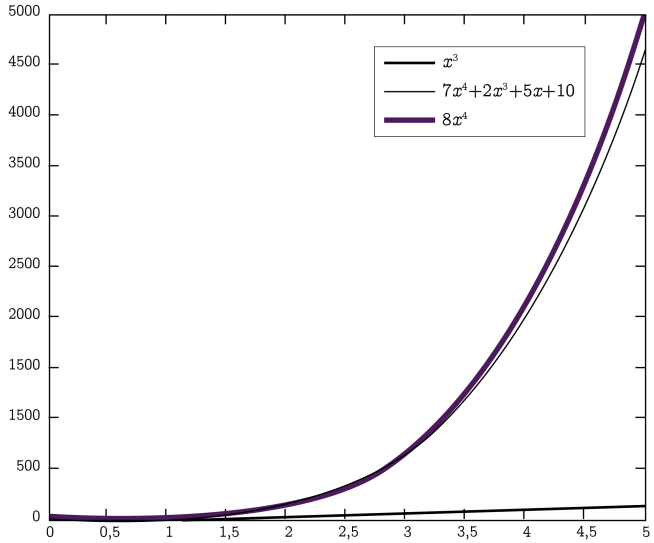


FIGURA 1.5. Crecimiento de funciones del ejemplo

Podemos también notar que $f(n)$ está en $O(n^5)$, y en $O(n^6)$, pero $f(n)$ **no** está en $O(n^3)$. Estas relaciones de mayoración son en parte explicadas en el siguiente resultado de *transitividad* cuya demostración se deja como ejercicio al lector.

Teorema 1.4. *Si $f(n)$ está en $O(g(n))$ y $g(n)$ está en $O(h(n))$, entonces $f(n)$ está en $O(h(n))$.*

Luego, aun cuando es válido hablar de que la complejidad $f(n)$ está en $O(g(n))$ para una función g cualquiera, se acostumbra considerar funciones "sencillas", como n, n^2, 2^n, etc., que sean minimales para la relación dada por la Definición 1.3, es decir, que cumpliendo que $f(n)$ está en $O(g(n))$, no se pueda encontrar otra función sencilla h tal que $f(n)$ está en $O(h(n))$ y $g(n)$ está en $O(h(n))$. Dependiendo de cuál sea esta función g, diremos que un algoritmo es "lineal", "cuadrático", "exponencial", respectivamente. Llamaremos *clases de complejidad* a estas categorías usadas para clasificar la complejidad de un algoritmo.

En la Tabla 1.4 se muestran las clases de complejidades más comunes y el nombre con que suele llamárseles. La tabla está ordenada en forma creciente según la relación dada por la Definición 1.3.

Observación 1.4. Como las constantes no juegan ningún rol en la definición de las clases de complejidad, las propiedades de cambio de base de los logaritmos nos permiten asumir, sin pérdida de generalidad, que las complejidades que involucran la función logaritmo son calculadas en base 2.

Complejidad	Nombre
$O(1)$	Constante
$O(\log n)$	Logarítmico
$O((\log n)^k)$ $(k \in \mathbb{N})$	Poli-logarítmico
$O(n)$	Lineal
$O(n \log n)$	Log-lineal
$O(n^2)$	Cuadrático
$O(n^3)$	Cúbico
$O(n^k)$ $(k \in \mathbb{N})$	Polinomial
$O(c^n)$ $(c > 1)$	Exponencial
$O(n!)$	Factorial

TABLA 1.4. Complejidades y sus nombres.

Finalizamos esta sección con un resultado sobre el álgebra básica asociada a la notación O, éste nos permitirá medir más fácilmente la complejidad de un algoritmo. Su demostración se puede obtener a partir de la definición de complejidad y se deja como ejercicio para el lector.

Teorema 1.5. *Si f, g son dos funciones tales que $f(n)$ está en $O(u(n))$ y $g(n)$ está en $O(v(n))$, entonces*

1. *$f(n) + g(n)$ está en $O(u(n) + v(n))$.*
2. *$f(n) \cdot g(n)$ está en $O(u(n) \cdot v(n))$.*

En términos de la notación gran o, podemos reescribir lo anterior como:

1. $O(u(n)) + O(v(n)) = O(u(n) + v(n))$.
2. $O(u(n)) \cdot O(v(n)) = O(u(n) \cdot v(n))$.

Ejercicio 1.6. Muestre que:

1. $f(n) = 5n^3 + 2n^2 + 100$ está en $O(n^3)$.
2. $f(n) + g(n)$ está en $O(\max\{f(n), g(n)\})$.

1.3.3 Midiendo la complejidad de un algoritmo

Las propiedades de los Teoremas 1.4 y 1.5 nos permiten, a partir de un algoritmo escrito en un pseudo-lenguaje de programación, obtener la complejidad de un algoritmo. En efecto, si logramos escribir el algoritmo de forma que cada paso sea *atómico*, es decir, que tome una cantidad constante de tiempo u operaciones (o equivalentemente que cada paso este en $O(1)$), podemos entonces contar las veces que se repiten estas instrucciones y calcular directamente su complejidad. Por ejemplo, pensemos en el problema de identificar los números que sean pares en un conjunto de n números dados. Un primer algoritmo que daremos para resolver este problema consiste en verificar, número por número, si este es par o no (Algoritmo 1.8).

Algoritmo 1.8 Buscar pares

Entrada: Conjunto de números $A = \{a_1, a_2, ...a_n\}$.

1: $P \leftarrow \emptyset$ /* Conjunto de números pares */
2: **para todo** $a_i \in A$
3: **si** a_i es par **entonces**
4: $P \leftarrow P \cup \{a_i\}$
5: **fin**
6: **fin**

Salida: Subconjunto P de números pares en A

En el Algoritmo 1.8, la asignación inicial de P (línea 1) es de complejidad $O(1)$. Verificar que un número es par (línea 3) y eventualmente agregarlo a P (línea 4) también tienen complejidad $O(1)$, por lo que la sentencia **si** de las líneas 3 a 5 son de complejidad $O(1) + O(1) = O(1)$ (gracias la propiedad 1 del Teorema 1.5). Sin embargo, esta secuencia de pasos debemos repetirlas para cada a_i en A, o sea, n veces. Por lo que el ciclo **para** de las líneas 2 a la 6 tienen complejidad $nO(1)$ que está en $O(n)$ (gracias a la propiedad 2 del Teorema 1.5). Finalmente como el paso de la línea 1 está en $O(1)$, podemos concluir que la complejidad del algoritmo es $O(1)+O(n) = O(n)$, es decir, lineal. Notemos que cada paso del algoritmo es *atómico*, esto hizo muy sencillo el cálculo de su complejidad.

Pensemos en otro algoritmo para resolver el mismo problema:

Algoritmo 1.9 Buscar pares

Entrada: Conjunto de números $A = \{a_1, a_2, ...a_n\}$.

1: **para todo** P subconjunto de A (en orden decreciente de tamaño)
2: **si** algún elemento de P es impar **entonces**
3: Continuar con el siguiente subconjunto P
4: **sino**
5: TERMINAR
6: **fin**
7: **fin**

Salida: Subconjunto P de números pares en A

Suponiendo que la verificación la condición de la línea 2 se realiza con el Algoritmo 1.9, es decir, verificando la paridad número por número, obtenemos que esta condición tarda una cantidad de pasos igual a $|P|$ (cardinalidad de P), que está en $O(n)$. Sin embargo, incluso aunque esta verificación la pudiésemos hacer en tiempo $O(1)$, la secuencia de pasos 2-6 hay que repetirla para cada subconjunto de A, cuya cardinalidad es igual 2^n (ver Ejercicio 3.c) de la Sección 1.4) y, por lo tanto en el peor de los casos, deberíamos repetir estos pasos 2^n veces. Concluimos así que la complejidad de este algoritmo es $nO(2^n)$ que está en $O(4^n)$, es decir, exponencial.

Estudiemos ahora las complejidades de los algoritmos de ordenamiento de listas Selección (Algoritmo 1.4) y Fusión (Algoritmo 1.7). En el algoritmo Selección, todas las asignaciones y verificaciones de condiciones son atómicas, es decir, se pueden hacer en tiempo $O(1)$, por lo que la complejidad de este algoritmo se estima directamente de los dos ciclos **para** anidados de la línea 2 y 4. Es claro que el primer ciclo **para** (líneas 2-12) se repite $n-1$ veces, y el segundo ciclo **para** (líneas 4-8) se repite (para un i fijo), en el peor caso, $n-1$ veces, por lo que podemos decir que la complejidad está en $O(n-1)\cdot O(n-1) = O((n-1)^2) = O(n^2)$ (la primera igualdad proviene de la propiedad 2 del Teorema 1.5, mientras que la segunda se obtiene directamente de la Definición 1.3). Podemos ser más finos en contar cuántas veces se repiten las instrucciones de las líneas 4-8. Una forma de contar es pensar en un cuadro de $(n-1)\cdot(n-1)$ y marcar los posibles valores de i y j del algoritmo (ver Figura 1.6). En la figura se puede apreciar que aproximadamente la mitad de los cuadros están marcados, o sea, los pasos de las líneas 4-7 se repiten aproximadamente $\frac{(n-1)^2}{2}$ en total, que está también en $O(n^2)$. En resumen, la complejidad del algoritmo de Selección para ordenar una lista de n elementos está en $O(n^2)$, es decir, es cuadrática.

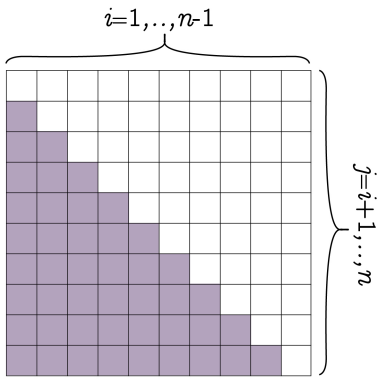


FIGURA 1.6. Índices i y j utilizados en el Algoritmo 1.4

Estudiemos ahora el algoritmo Fusión. El estudio de la complejidad esta vez es más complicado, dada la definición recursiva de éste. Denotemos por $f(n)$, para alguna función f, a la complejidad del algoritmo Fusión. Se puede mostrar que el Algoritmo 1.6, *fusión-2listas*, tiene complejidad lineal, es decir, está en $O(n)$ (el lector puede probar esto como ejercicio). Por otro lado, dada la definición recursiva del algoritmo, sabemos que el número de pasos para ordenar una lista de tamaño n es igual al número de pasos que toma ordenar una lista de tamaño $\lfloor n/2 \rfloor$, más el número de pasos que toma ordenar una lista de tamaño $\lceil n/2 \rceil$, más el número de pasos que toma fusionar ambas listas con el algoritmo *fusión-2listas*. Esto se puede aproximar como sigue:

$$f(n) \approx 2f(\lceil n/2 \rceil) + O(n).$$

Intuitivamente[6], si pensamos que la relación anterior se da con una igualdad, reemplazando $O(n)$ por una función lineal, obtenemos:

$$f(n) = 2f(n/2) + \alpha n \qquad \text{(para cierto } \alpha > 0\text{)}, \tag{1.2}$$

Se puede mostrar usando inducción (ver págs. 128-131 del libro [**1**]) que la función f, que cumple con la igualdad, viene dada por:

$$f(n) = \alpha n \log n + \beta n, \tag{1.3}$$

donde β corresponde a cuánto se demora el algoritmo Fusión si la lista tiene tamaño $n = 1$.[7] Por lo tanto, se concluye que $f(n)$ está en $O(n \log n) + O(n) = O(n \log n)$, es decir, el algoritmo Fusión tiene complejidad log-lineal. En el Ejercicio 12 de la próxima sección veremos un argumento ligeramente distinto para demostrar que el algoritmo Fusión tiene complejidad $O(n \log n)$, el que también está basado en la Ecuación (1.2) y en la función (1.3).

Es por esto que el algoritmo Fusión es uno de los más rápidos en términos de complejidad, de hecho, no existen algoritmos generales que permitan ordenar una lista con complejidad menor que $O(n \log n)$.

[6]La demostración formal de este teorema necesita ciertos conocimientos fuera del alcance de esta monografía.

[7]Determinar exactamente esta cantidad no es importante para el análisis de complejidad, pero podemos ver directamente, contando la verificación de la línea 1 y la asignación de la línea 2 del Algoritmo 1.7, que $\beta = 2$.

1.4 Ejercicios

1. Inspirándose en los Algoritmos 1.1, 1.2 y 1.2, imagine situaciones cotidianas que pueden ser resueltas usando algoritmos y escríbalos utilizando pseudo-lenguaje de programación.
2. Demuestre, usando inducción, las siguientes identidades:
$$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6 \quad \text{y} \quad \sum_{i=1}^{n} i^3 = n^2(n+1)^2/4.$$
3. Pruebe, usando inducción, que las siguientes aseveraciones son ciertas:
   a) Para todo $n \in \mathbb{N}$, $3^{2n+1} + 2^{n+2}$ es divisible por 7.
   b) $n \geq 1$ puntos distintos sobre una recta determinan $n+1$ segmentos.
   c) Sea A un conjunto de cardinalidad $n \in \mathbb{N}$. El conjunto de las partes de A, es decir, el conjunto de todos los subconjuntos posibles de A, tiene cardinalidad 2^n.
4. Podemos definir recursivamente n^2 como sigue:
   $1^2 = 1$ (base),
   Si $n^2 = m$, entonces $(n+1)^2 = m + 2n + 1$ (paso inductivo).

   Escriba un algoritmo recursivo que implemente lo anterior, usando pseudo-lenguaje de programación. Pruebe, utilizando inducción sobre n, que esta definición recursiva calcula correctamente n^2.
5. En el país de *Akinostán* sólo existen monedas de 3 y 5 pesos. Muestre, usando inducción, que cualquier cantidad de pesos mayor o igual que 8 puede ser pagada con las monedas disponibles.

   Indicación: Considere como casos bases el pago de 8, 9 y 10 pesos. Luego, para demostrar que esta aseveración es cierta para $n+1$ pesos, utilice como hipótesis inductiva que es cierta para n, $n-1$ y $n-2$ pesos.
6. El *máximo común divisor* (MCD) de dos números naturales a y b es el mayor número natural que divide a ambos números. Por ejemplo
$$MCD(9,12) = 3,\ MCD(24,30) = 6 \text{ y } MCD(12,25) = 1.$$

   Escriba un algoritmo recursivo que calcule el máximo común divisor de dos números naturales a y b, asumiendo que $a \geq b$.

   Indicación: Puede utilizar la siguiente definición recursiva del máximo común divisor:

   Si b divide a a, entonces, $MCD(a,b) = b$ (base), si b no divide a a, entonces, sea r el resto de la división a/b (es decir, $a = kb + r$ para cierto $k \in \mathbb{N}$)[8], se tiene que $MCD(a,b) = MCD(b,r)$ (paso inductivo).

   Pruebe, usando inducción, que esta definición recursiva calcula correctamente máximo común divisor de a y b (con $a \geq b$).

[8]La operación que a dos enteros a y b le asocia su resto se llama *módulo* y se denota $r = a \bmod b$.

7. El algoritmo de Euclides, que calcula el máximo común divisor de a y b (con $a \geq b$), se puede describir iterativamente como sigue: "Si $b = 0$, entonces $MCD(a, b) = a$ (pues 0 es divisible por cualquier entero). Si no, mientras $b > 0$, asigne b a a y el resto r de la división a/b a b. La salida será a".

   Escriba este algoritmo usando pseudo-lenguaje de programación.

8. Demuestre que:
   a) a^n está en $O(b^n)$ si $1 < a \leq b$, pero a^n no está en $O(b^n)$ si $1 < b < a$.
   b) n^a está en $O(b^n)$, para todo a y para todo $b > 1$.
   c) $\log n^a$ está en $O(n^b)$, para todo a y para todo $b > 0$.
   d) $n!$ está en $O(n^n)$.
   e) $n^{\log n}$ está en $O(2^n)$.

9. Demuestre que el algoritmo de Euclides dado en el Ejercicio 7 tiene complejidad $O(\log a)$ (con $a \geq b$).

   Indicación: Pruebe que $a \mod b < a/2$. Con esto se deduce que la variable a se hace, al menos, más pequeña que $a/2$ después de dos iteraciones del ciclo **mientras**. Se concluye que este ciclo **mientras** será utilizado, a lo más, una cantidad $2 \log a$ veces.

10. La versión original del algoritmo de Euclides usaba la sustracción repetida en lugar del residuo de la división, éste puede describirse iterativamente como sigue: "Si $b = 0$, entonces $MCD(a, b) = a$ (pues 0 es divisible por cualquier entero). Si no, mientras $b > 0$, asigne b a a y $a - b$ a b. La salida será a". Esta versión resulta ser significativamente menos eficiente.

    Escriba este algoritmo, usando pseudo-lenguaje de programación y calcule su complejidad. Compare con el Ejercicio 9.

11. Los números de Fibonacci son una secuencia de números enteros $\{F(n)\}$, con $n \in \mathbb{N} \setminus \{0\}$, cuyo n-ésimo término $F(n)$ se calcula sumando los dos números precedentes y donde los dos primeros son iguales a 1. Por ejemplo, los primeros siete números de Fibonacci vienen dados por: 1,1,2,3,5,8, y 13.
    a) Escriba $F(n)$ en función de $F(n-1)$ y $F(n-2)$, para $n \geq 3$.
    b) A partir de a), escriba un algoritmo recursivo usando pseudo-lenguaje de programación, cuya entrada sea n y cuya salida sea el n-ésimo número de Fibonacci $F(n)$.
    c) La constante $r = (1 + \sqrt{5})/2 \approx 1,62$ es llamada *la razón áurea* (se deja al lector investigar también sobre *la sección áurea*). Demuestre que $F(n)$ está en $O(r^n)$.
    Indicación: Pruebe por inducción que $F(n) \leq ar^n$, para cierta constante a. La base debe incorporar los casos $n = 1$ y $n = 2$. Para el paso inductivo, le puede ser útil notar que $r^2 = r + 1$.
    d) Demuestre que $F(n)$ puede ser calculado usando la fórmula de Cauchy-Binet:

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

e) A partir de la parte d), escriba un algoritmo iterativo, usando pseudo-lenguaje de programación, cuya entrada sea n y cuya salida sea el n-ésimo número de Fibonacci $F(n)$. Calcule su complejidad y compare con la parte c).

12. El objetivo de este ejercicio es justificar que la complejidad $f(n)$ del algoritmo Fusión está en $O(n \log n)$. Para esto procederemos como sigue:
    a) Supongamos primero que n es una potencia de 2, es decir, que $n = 2^i$ con $i \in \mathbb{N}$. Demuestre, usando inducción sobre i, que la función $i \to f(2^i) = 2^i(\alpha i + \beta)$, donde f está dada en (1.3), satisface la ecuación (1.2), para todo $n = 2^i$ con $i \in \mathbb{N} \setminus \{0\}$. Se deduce, entonces, que $f(n)$ está en $O(n \log n)$, para todo n que sea una potencia de 2.
    b) Usando la parte anterior y el hecho que la complejidad $f(n)$ es creciente (justifique), pruebe que $f(n)$ está en $O(n \log n)$, para cualquier $n \in \mathbb{N}$.

Capítulo 2: Grafos

En este capítulo introduciremos las nociones básicas de teoría de grafos, así como el lenguaje y las propiedades que formarán la base de los temas abordados en los próximos capítulos.

Presentaremos también varios ejemplos, ejercicios e ilustraciones que esperamos permitan al lector adueñarse de este tema.

En cuanto a los algoritmos, en el resto de esta monografía estudiaremos formas de resolver problemas que se plantean sobre las estructuras de grafos que definiremos en este capítulo. Estos problemas tienen gran interés en matemáticas y ciencias de la computación por lo simple que resulta su planteamiento y lo difícil que generalmente es su resolución.

Aparecerán así, de manera natural, las nociones de recursión, iteración, complejidad, etc., que estudiamos en el Capítulo 1.

2.1 Introducción

Un *grafo* es una dupla $G = (V, E)$ compuesta por un conjunto finito $V = V(G)$ de *vértices*, también llamados *nodos* y típicamente dibujados por círculos y un conjunto $E = E(G) \subseteq V_*^2$, que llamaremos conjunto de *aristas*[1], donde hemos definido $V_*^2 = (V \times V) \setminus (\cup_{v \in V}(v, v))$, pues en esta monografía no estamos interesados en estudiar aristas de la forma (v, v), también conocidas como *bucles*.

Un *grafo no-dirigido* (que simplemente llamaremos *grafo*) es una tupla $G = (V, E)$ con las características mencionadas anteriormente, donde una arista $e \in E$ representa una conexión (sin una dirección establecida) entre dos vértices de V y se dibuja por una línea que une estos elementos. Así, la arista $e = (v_1, v_2)$ representa una conexión entre los vértices v_1 y v_2, la cual puede también escribirse como $e = (v_2, v_1)$, en este caso, diremos que los vértices v_1 y v_2 son *adyacentes*. Para simplificar la notación, en esta monografía utilizaremos la convención que la arista e de un grafo no-dirigido aparecerá escrita sólo una vez en la descripción de E y, en caso de existir alguna relación de orden $\leq$ definida sobre el conjunto de vértices V, está será descrita de la forma $e = (v_1, v_2)$ cuando $v_1 \leq v_2$. Sin embargo, para definir correctamente ciertas operaciones entre grafos que veremos en la Sección 2.2.1, las cuales se establecerán mediante operaciones

[1]Hemos adoptado la letra E motivados por el término inglés *edge*, con el cual se llama a este objeto matemático.

usuales entre conjuntos (unión, intersección, complemento, diferencia, etc.), debemos tener en cuenta que, si el par (v_1, v_2) está en E, entonces también lo está el par (v_2, v_1).

Ejemplo 2.1. Sea el grafo $G = (V, E)$, con los vértices $V = \{1, 2, 3, 4, 5, 6\}$ y las aristas $E = \{(1, 2), (1, 3), (2, 4), (2, 6)\}$. Este grafo se representa gráficamente en la Figura 2.1.

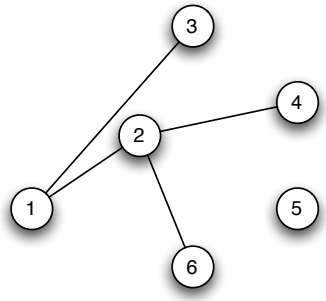


FIGURA 2.1. Ejemplo de un grafo

Por otro lado, un *grafo dirigido* $G = (V, A)$ es un grafo cuyas aristas $a = (v_1, v_2) \in A$ representan una conexión entre los dos vértices v_1 y v_2 en la dirección desde v_1 a v_2, usualmente denotada por una flecha que va desde v_1 hacia v_2. Luego, en un grafo dirigido, una arista nos entrega más información que una arista en un grafo no-dirigido. Dada esta importante diferencia, llamaremos *arco* a un elemento $a = (v_1, v_2) \in A$ y diremos que A es el conjunto de arcos del grafo dirigido G, explicando el cambio de notación de E a A y de e a a. En la Figura 2.2 dibujamos el grafo dirigido $G = (V, A)$, donde sus vértices son $V = \{1, 2, 3, 4, 5, 6\}$ y sus arcos vienen dados por $A = \{(1, 2), (2, 4), (3, 1), (4, 2), (6, 2)\}$.

Resumiendo, la principal diferencia entre grafos dirigidos y no-dirigidos es que, para dos vértices v_1 y v_2 de un grafo dirigido, (v_1, v_2) no representa el mismo arco que (v_2, v_1), pues el primero sólo sirve para "llegar" desde v_1 a v_2, mientras que el segundo tiene el sentido contrario.

Una característica intrínseca a un grafo (no-dirigido y dirigido) es la cardinalidad de los conjuntos de vértices y aristas (o arcos según sea el caso). Estas serán denotadas por $|V|$ y $|E|$ (o $|A|$), respectivamente. En particular, el valor $|E|$ (o $|A|$) es llamado el *tamaño del grafo* G.

Ejercicio 2.1. Mostrar que el tamaño de un grafo no-dirigido G es a lo menos 0 y a lo más $\binom{n}{2}$, siendo n el número de vértices de G. ¿Cuales son las cotas para un grafo dirigido?

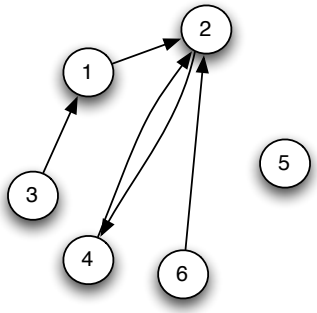


FIGURA 2.2. Ejemplo de un grafo dirigido

2.2 Grafos no-dirigidos

En esta sección a un grafo no-dirigido lo llamaremos simplemente grafo. Un grafo queda completamente definido por sus vértices V y aristas E, aunque su representación gráfica puede variar. Por ejemplo, el grafo de la Figura 2.1, también puede ser representado como en los grafos de la Figura 2.3.

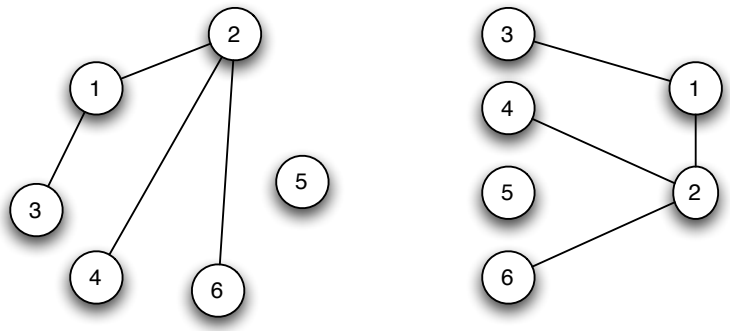


FIGURA 2.3. Otras representaciones gráficas del grafo G

En el contexto de teoría de grafos, en especial en lo referente a grafos no-dirigidos, muchas veces nos enfocamos en la estructura que un grafo pueda tener y a las propiedades que resulten de ésta, quedando relegado a segundo plano el significado que tienen sus vértices y aristas. Es por esto que resulta útil entender cuándo dos grafos tienen propiedades equivalentes aun cuando no sean iguales como objetos matemáticos, ya que los nombres de sus vértices (y, en consecuencia, sus aristas) puedan diferir.

Así, diremos que dos grafos $G = (V, E)$ y $G' = (V', E')$ son *isomorfos* si existen biyecciones, tanto entre sus vértices, como entre sus aristas, es decir, existen dos funciones biyectivas $\varphi_V : V \to V'$ y $\varphi_E : E \to E'$, que satisfacen que $e = (u, v) \in E$ si y sólo si $\varphi_E(e) = (\varphi_V(u), \varphi_V(v)) \in E'$. Por ejemplo, los grafos de la Figura 2.4 son isomorfos, pero distintos.

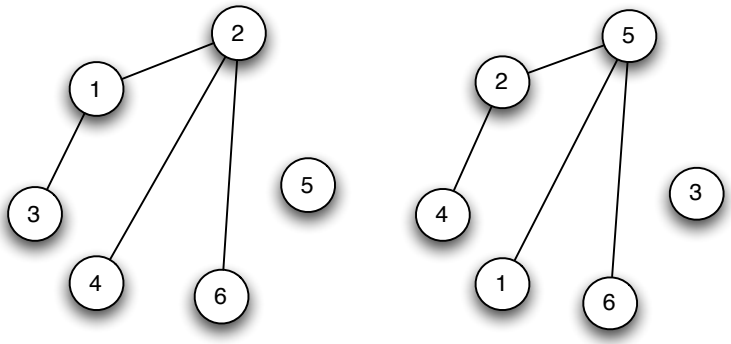


FIGURA 2.4. Grafos isomorfos

En la Sección 2.2.2 estudiaremos algunas estructuras básicas, que se comparten para grafos isomorfos, con algunas de sus propiedades principales.

Una característica muy importante en un grafo no-dirigido es la noción de *arista incidente* y grado de un vértice que definimos a continuación.

Definición 2.1. Diremos que una arista e es *incidente* a un vértice v, si v es uno de los elementos de la arista e. El número de aristas incidentes a un vértice v le llamaremos el *grado* de v y lo denotaremos por $\delta(v)$. En el caso particular que el grado de un vértice v sea 0, diremos que v es un *vértice aislado*.

Para ilustrar esta definición, veamos que en el grafo del Ejemplo 2.1 el grado del vértice 2 es 3, es decir, $\delta(2) = 3$, pues las aristas $(1, 2)$, $(2, 4)$ y $(2, 6)$ inciden en el vértice 2.

Estamos ahora en condiciones de probar nuestro primer resultado en teoría de grafos.

Teorema 2.2. *En todo grafo, el número de vértices de grado impar es par.*

Demostración. Sea $G = (V, E)$ el grafo en consideración. Notemos que cada arista contribuye exactamente 2 veces a la suma total de los grados de los vértices (de hecho podemos calcular el grado de un vértice contando las veces que aparece en el conjunto

E), entonces, sumando los grados de todos los vértices $v \in V$, obtenemos[2]:

$$\sum_{v \in V} \delta(v) = 2|E|. \tag{2.1}$$

Si denotamos por $V' \subseteq V$ el subconjunto de todos los vértices de grado impar, entonces $V'' = V \setminus V'$ es el subconjunto de todos los vértices de grado par y $\sum_{v \in V} \delta(v) = \sum_{v \in V'} \delta(v) + \sum_{v \in V''} \delta(v)$. Obteniendo de (2.1) que

$$\sum_{v \in V'} \delta(v) = 2|E| - \sum_{v \in V''} \delta(v).$$

Pero como el término de la derecha es par, deducimos que la $\sum_{v \in V'} \delta(v)$ es par y, por lo tanto, dado que $\delta(v)$ es impar, para todo $v \in V'$, la única posibilidad para que esto suceda es que el número de vértices de grado impar sea par. □

Observación 2.1. La igualdad (2.1) puede ser reescrita como:

$$\sum_{v \in V} \delta(v) = 0 \mod 2, \tag{2.2}$$

donde la operación $a \mod b$ nos entrega, para dos números enteros a y b, el resto de la división a/b (ver también la Nota 8 al pie de la página 45 del Capítulo 1). Esta última igualdad es llamada el *lema del apretón de manos*[3], dado que expresa el hecho que **¡en cualquier fiesta el número de manos apretadas es par!**.

Ejercicio 2.2. Un grafo cúbico es un grafo de n vértices, donde cada vértice tiene grado 3. Pruebe que si G es un grafo cúbico, entonces $n = |V(G)|$ es par. ¿Cuál es el grafo cúbico de menos vértices?

A lo largo de esta monografía nos resultará útil identificar "partes" de un grafo dado, es decir, un subconjunto de vértices y las aristas que incidan en éstos. Formalmente, diremos que un grafo $G' = (V', E')$ es un *subgrafo* del grafo $G = (V, E)$, si $V' \subseteq V$ y $E' \subseteq E \cap (V' \times V')$. Esta relación la denotaremos por $G' \subseteq G$ y diremos que G *contiene* a G' o que G' *está contenido* en G. Por ejemplo, dos posibles subgrafos del grafo G, dado en el Ejemplo 2.1, son: $G' = (V', E')$, con los vértices $V' = \{1, 2, 4, 6\}$ y las aristas $E' = \{(1, 2), (2, 4), (2, 6)\}$ y $G'' = (V'', E'')$ con $V'' = V'$ y $E' = \{(1, 2), (2, 4)\}$.

Una primera aplicación del estudio de subgrafos es la siguiente definición:

Definición 2.3. Definimos un *camino* como un grafo $P = (V, E)$ donde $V = \{v_0, v_1, \ldots, v_k\}$ y $E = \{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}$. En este caso, diremos que P es un camino que *conecta* o *une* los vértices v_0 y v_k. Definimos también el *largo del camino* de P como el tamaño de este grafo, es decir, el número de aristas k de P. En el caso particular que los extremos del camino P sean el mismo vértice (es decir, $v_0 = v_n$),

[2]En el Ejercicio 2 de la Sección 1.4 de ejercicios se sugiere una forma alternativa para probar la igualdad (2.1).

[3]Se conoce en inglés como *handshaking lemma*.

diremos que el camino es un *ciclo* y lo denotaremos por C. En particular, diremos que P (o C) es un camino (o un ciclo, respectivamente) del grafo G, si además se tiene que $P \subseteq G$ (o $C \subseteq G$, respectivamente).

Observación 2.2. Muchas veces un camino (o un ciclo) de largo k se denota por P_k (o C_k, respectivamente), esta notación será usada en esta monografía sólo cuando no haya posibilidad alguna de confusión.

En el grafo del Ejemplo 2.1 podemos identificar varios caminos posibles, por ejemplo, entre los vértices 3 y 6 existe un único camino que los conecta dado por $P = (\{1, 2, 3, 6\}, \{(1, 2), (1, 3), (2, 6)\})$. Para simplificar la notación, un camino será denotado por la secuencia de vértices o de aristas que lo representan. Así, en el caso del ejemplo anterior, el camino P puede ser descrito simplemente como $(3, 1, 2, 6)$ o $((1, 2), (1, 3), (2, 6))$, respectivamente.

Ejercicio 2.3. Verifique que estas notaciones son coherentes, es decir, que efectivamente describen el mismo camino. Describa, usando los tres tipos de notaciones, el camino que conecta los vértices 3 y 4 en el grafo del Ejemplo 2.1 .

Para establecer propiedades sobre grafos o realizar algoritmos sobre éstos, es muchas veces necesario que los grafos estén "conectados" en algún sentido. En teoría de grafos, esta noción corresponde a la de *grafos conexos*. Usando la Definición 2.3, podemos decir que un grafo es *conexo* si existe un camino entre cada par de vértices distintos del grafo. Por ejemplo, en la Figura 2.5, el grafo de la izquierda es conexo mientras que el grafo de la derecha no lo es, pues no existe un camino que conecte a los vértices 1, 2 ó 3 con los vértices 4 ó 5.

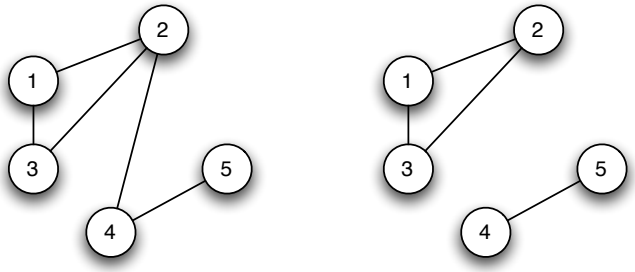


FIGURA 2.5. Ejemplo de un grafo conexo y uno que no lo es

Siempre podemos identificar las *componentes conexas* de un grafo $G = (V, E)$ (que llamaremos simplemente *componentes*) como los subgrafos conexos G_i de G que no están conectados entre ellos y, por lo tanto, sus vértices $V(G_i)$ particionan el conjunto de vértices de G (es decir, $V(G_i) \cap V(G_j) = \emptyset$, para todo $i \neq j$ y $\cup_i V(G_i) = V$) . Por

ejemplo, el grafo de la derecha de la Figura 2.5 tiene dos componentes formadas por los vértices $\{1, 2, 3\}$ y $\{4, 5\}$ y sus respectivas aristas.

Observación 2.3. Notemos que un grafo G compuesto por un solo vértice (i.e. $V(G) = \{v\}$ y $E(G) = \emptyset$) es un grafo conexo, ya que la definición se satisface por vacuidad, pues no existen dos vértices distintos en V.

2.2.1 Operaciones sobre grafos

Vamos a necesitar construir nuevos grafos a partir de algunos ya existentes, para esto, definiremos las siguientes operaciones básicas sobre grafos que nos permitirán sumar o restar vértices y aristas a un grafo $G = (V, E)$ dado.

Si $V' \subset V$ es un subconjunto (estricto) de vértices de G, entonces la *diferencia* entre G y V', denotada por $G \setminus V'$, corresponde al grafo cuyos vértices son $V \setminus V'$ y sus aristas son las aristas E, menos las que inciden en algún vértice de V', es decir, $E(G \setminus V') = E \cap (V \setminus V') \times (V \setminus V')$. Similarmente, si $E' \subset E$ es un subconjunto (estricto) de aristas de G, entonces la *diferencia* entre G y E', denotada por $G \setminus E'$, corresponde al grafo cuyos vértices son los vértices V y cuyas aristas son $E \setminus E'$, es decir, $G \setminus E' = (V, E \setminus E')$. En el caso que estos conjuntos sean un sólo elemento, por ejemplo, $V' = \{v\}$ y $E' = \{e\}$, simplificaremos la notación escribiendo $G \setminus v$ y $G \setminus e$, respectivamente.

En las Figuras 2.6 y 2.7 se ilustra las diferencias entre el grafo del Ejemplo 2.1 y un vértice y de este mismo grafo y un conjunto de aristas, respectivamente.

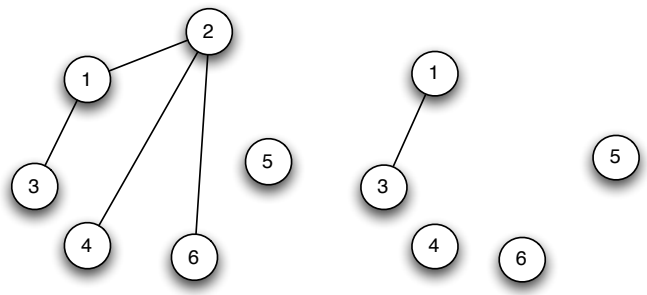


FIGURA 2.6. Diferencia entre un grafo y un conjunto de vértices. Izquierda: G, Derecha: $G \setminus \{2\}$

Para una arista e que une dos vértices *no adyacentes* de un grafo $G = (V, E)$, es decir, $e \in V_*^2 \setminus E$, definimos la *suma* de G y e, denotada por $G + e$, como el grafo obtenido al incluir la arista e en G. En la Figura 2.8 se ilustra la suma del grafo del Ejemplo 2.1 y la arista $(2, 5)$.

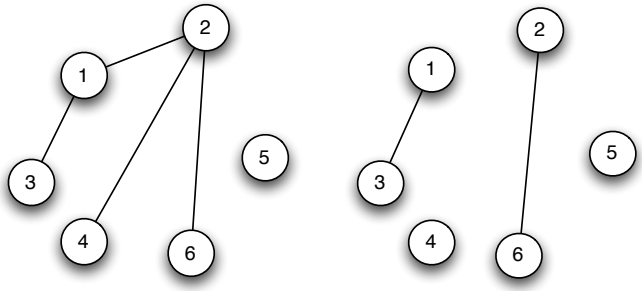


FIGURA 2.7. Diferencia entre un grafo y un conjunto de aristas. Izquierda: G, Derecha: $G \setminus \{(1,2),(2,4)\}$

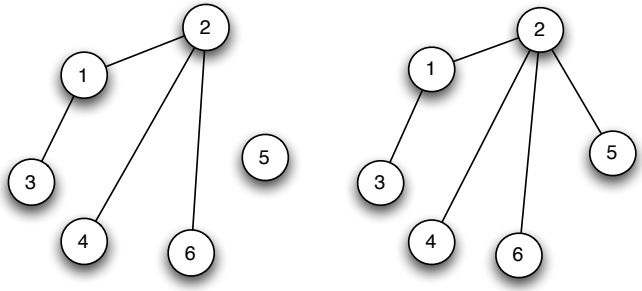


FIGURA 2.8. Suma entre un grafo y una arista. Izquierda: G, Derecha: $G + (2,5)$

Ejercicio 2.4. Sean dos grafos $G = (V, E)$ y $G' = (V', E')$. Deseamos estudiar una posible definición para la operación suma $G + G'$.

(i) Supongamos primero que $V = V'$. Defina $G + G'$ de manera que, si G' tiene sólo una arista, entonces $G + G'$ coincida con la suma entre un grafo y una arista definida anteriormente.

(ii) Usando lo aprendido en el Capítulo 1 construya un algoritmo iterativo que realice tal suma y calcule su complejidad.

(iii) ¿Cómo intentaría definir la suma cuando $V \cap V' = \emptyset$? ¿o cuando $V \cap V' \neq \emptyset$, pero $V \neq V'$?

Finalmente, definimos el *complemento* de un grafo $G = (V, E)$ como el grafo $\bar{G}$ cuyos vértices son los mismos de G, es decir, $V(\bar{G}) = V$ y dos vértices son adyacentes

en G', si y sólo si no lo son en G, es decir, $E(G) = V_*^2 \setminus E$. Se deja como ejercicio al lector calcular y dibujar el complemento del grafo del Ejemplo 2.1. Notemos que el complemento del complemento de un grafo es el grafo original, es decir, $\overline{(\bar{G})} = G$.

Ejercicio 2.5. Probar que, para todo grafo que no sea conexo, se tiene que su complemento necesariamente lo es. Investigar la recíproca, es decir, ver si el complemento de un grafo conexo es conexo o no.

Ejercicio 2.6. Demostrar que para dos grafos G y G', cuyos conjuntos de vértices coinciden, se tiene que: $G \subseteq G'$ si y sólo si $\bar{G}' \subseteq \bar{G}$. ¿Sigue siendo cierta esta propiedad si los vértices de G' están contenidos en los de G?

2.2.2 Algunos grafos particulares

En esta sección presentaremos algunas estructuras particulares de grafos y sub-grafos que nos permitirán plantear los problemas y algoritmos de los próximos capítulos. Estas estructuras se preservan entre grafos isomorfos y tienen también importancia en sí mismas, ya que nos permiten plantear propiedades inherentes a ellas que siguen siendo objeto de investigación en esta área de las matemáticas.

Definición 2.4. Definimos un *árbol* como un grafo conexo que no contiene ciclos.

Un ejemplo de un árbol es el grafo de la Figura 2.9. Planteemos un primer ejercicio asociado a esta estructura:

Ejercicio 2.7. Muestre que en todo árbol siempre hay vértices de grado 1, más aún, la conexidad nos dice que éstos corresponden a los vértices de grado mínimo del árbol. A estos vértices los llamaremos las *hojas* del árbol.

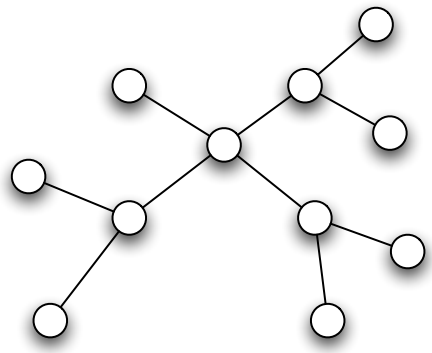


FIGURA 2.9. Ejemplo de un árbol

Una simple caracterización de esta estructura tan particular se plantea a continuación.

Proposición 2.5. *Sea G un grafo conexo con n vértices, entonces G es un árbol si y sólo si G tiene $n-1$ aristas.*

Demostración. Probemos esto por inducción en n. Para $n = 1$ el resultado es directo, pues G es un sólo vértice, que es conexo (ver Observación 2.3) y no contiene ciclos, luego es un árbol y el número de aristas es $n - 1 = 0$. Supongamos ahora que la equivalencia es cierta para grafos de $n - 1$ vértices y sea G un grafo con n vértices, para el cual probaremos la equivalencia establecida en la proposición. En el resto de esta demostración v denotará el vértice de menor grado en G.

Supongamos primero que G es un árbol. Del Ejercicio 2.7 se tiene que v es una hoja, es decir, $\delta(v) = 1$. Si removemos v y su arista incidente, el grafo resultante $G \setminus v$ seguirá siendo un árbol, pues sigue siendo conexo y no contiene ciclos. Aplicando la hipótesis de inducción, obtenemos que $G \setminus v$ tiene $(n-1)-1$ aristas, concluyendo así, que G tenía $n - 1$ aristas.

Recíprocamente, si ahora G tiene $n - 1$ aristas, entonces $\delta(v) = 1$, pues en caso contrario, del lema del apretón de manos (ecuación (2.1)) se deduce que el número de aristas de G sería mayor o igual que n. Luego $G \setminus v$ tiene $(n - 1) - 1$ aristas y, por la hipótesis de inducción, obtenemos que $G \setminus v$ es un árbol. Finalmente, notamos que si volvemos a agregar v y su arista incidente al grafo $G \setminus v$, no se formarán nuevos ciclos (ya que $\delta(v) = 1$), concluyendo que G también es un árbol, lo que prueba la proposición. □

Ejercicio 2.8. Sea $G = (V, E)$ un grafo con n vértices, pruebe que G es un árbol si y sólo si se cumple una de las siguientes caracterizaciones:

1. Cualquier par de vértices de G están conectados exactamente por un camino.
2. Al incluir cualquier arista e que no esté originalmente en G, pero que conecte dos vértices de G (es decir $e \in V_*^2 \setminus E$), el grafo resultante $G + e$ contiene exactamente un ciclo.

Otro tipo de grafo de gran interés es el siguiente:

Definición 2.6. El grafo completo K_n es el grafo de n vértices que contiene todas las aristas posibles, es decir, para cada par de vértices existe una arista que los une.

Ejercicio 2.9. Pruebe que K_n tiene exactamente $\frac{n(n-1)}{2}$ aristas.

El complemento $\bar{K}_n$ de un grafo completo consiste simplemente en n vértices sin conexiones entre sí. Esta estructura tan particular nos permite establecer la siguiente propiedad: K_n está contenido en un grafo G si y sólo si $\bar{K}_n$ está contenido en su complemento $\bar{G}$.

Ejercicio 2.10. Muestre que la aseveración anterior es cierta. ¿Qué sucede si se remplaza K_n por un grafo arbitrario G'? Compare con el Ejercicio 2.6.

Algunos grafos completos particulares son de gran importancia en teoría de grafos, uno de ellos es el *triángulo*, definido como el grafo completo de tres vértices y denotado por K_3. Claramente este grafo también se puede identificar con el ciclo de largo 3, que denotaremos aquí por C_3.

Un hermoso resultado, basado en el lema del apretón de manos (Ecuación (2.1)) y en la *desigualdad de Cauchy-Schwartz*, es el que veremos a continuación.

Teorema 2.7 (Mantel 1907)**.** *Todo grafo con n vértices y tamaño mayor estricto que $\lfloor n^2/4 \rfloor$ contiene un triángulo.*

Demostración. Mostremos la contra-recíproca. Sea $G = (V, E)$ un grafo con n vértices y que no contiene un triángulo, probemos que el tamaño $|E|$ es menor o igual que $\lfloor n^2/4 \rfloor$.

Como G no contiene triángulos, necesariamente se tiene que, para toda arista $e = (v_1, v_2) \in E$, no existen vértices que sean al mismo tiempo adyacentes a v_1 y a v_2. Luego, si denotamos por E_v al conjunto de todas las aristas que inciden en un vértice v, obtenemos que E_{v_1} y E_{v_2} son disjuntos para todo par de vértices distintos v_1 y v_2, lo que en particular implica que los conjuntos E_{v_i}, con $i = 1, ..., n$ (donde hemos denotado por $\{v_1, v_2, ..., v_n\}$ al conjunto de vértices V), son una partición de E, es decir:

$$E = \cup_{i=1}^{n} E_{v_i} \quad \text{y} \quad E_{v_1} \cap E_{v_2} = \emptyset, \forall v_1, v_2 \in V : v_1 \neq v_2. \tag{2.3}$$

Por lo tanto,

$$\delta(v_1) + \delta(v_2) \leq n, \qquad \forall e = (v_1, v_2) \in E. \tag{2.4}$$

Notemos además que, si fijamos un vértice $v_1 \in V$, se tiene que

$$\delta(v_1)^2 = \sum_{e=(v_1,v_2)\in E_{v_1}} \delta(v_1) \leq \sum_{e=(v_1,v_2)\in E_{v_1}} (\delta(v_1) + \delta(v_2)), \tag{2.5}$$

pues en la primera igualdad hemos usado que $\delta(v_1) = \sum_{e=(v_1,v_2)\in E_{v_1}} 1$ y que v_1 está fijo, por lo que podemos pasarlo adentro de la sumatoria y en la desigualdad, que el grado de un vértice es siempre mayor o igual que 0. Entonces, sumando las desigualdades (2.5), para todos los vértices $v_1 \in V$, usando (2.3) y sumando las desigualdades (2.4), para todas las aristas $e = (v_1, v_2) \in E$, respectivamente, obtenemos que

$$\begin{aligned} \sum_{v_1 \in V} \delta(v_1)^2 &\leq \sum_{v_1 \in V} \sum_{e=(v_1,v_2)\in E_{v_1}} (\delta(v_1) + \delta(v_2)) \\ &= \sum_{e=(v_1,v_2)\in E} (\delta(v_1) + \delta(v_2)) \leq n|E|. \end{aligned} \tag{2.6}$$

Por otro lado, del lema del apretón de manos (2.1) y la desigualdad de Cauchy-Schwartz, aplicada a los vectores $(\delta(v_1), \delta(v_2), ..., \delta(v_n))^\top$ y $(1, 1, ..., 1)^\top$ (vector con

n unos), se llega a

$$(2|E|)^2 = \left(\sum_{v_1 \in V} \delta(v_1)\right)^2 \leq n \sum_{v_1 \in V} \delta(v_1)^2.$$

Así, de la ecuación (2.6) se concluye que

$$(2|E|)^2 \leq n^2|E|,$$

es decir, que $|E| \leq n^2/4$. □

Números de Ramsey. Un entretenido juego de ingenio consiste en **probar que en cualquier fiesta de seis personas, siempre hay tres personas que se conocen entre sí o tres personas que no**. Este problema es usualmente llamado el *problema de la fiesta de seis personas.*

Modelaremos este problema usando un grafo G, donde cada vértice representa a una persona y cada arista entre dos vértices significa que las personas representadas, por dichos vértices, se conocen. Con esto, el hecho que tres personas se conozcan entre sí, queda representado por un triángulo K_3 contenido en G.

Dada esta modelación, en el complemento del grafo G, denotado por $\bar{G}$, dos vértices son adyacentes sólo si las personas representadas por dichos vértices no se conocen. Luego, el hecho que tres personas no se conozcan entre sí, queda representado por un triángulo K_3 contenido en $\bar{G}$.

En estos términos, el problema de la fiesta de seis personas puede ser enunciado a través del siguiente teorema. La demostración es dejada como ejercicio.

Teorema 2.8. *Para todo grafo G de seis vértices, se tiene que, G contiene un triángulo, o bien su complemento $\bar{G}$ lo contiene.*

Lo anterior nos permite plantear la siguiente pregunta (conocida como el *problema de la fiesta*): **¿Cuál es la cantidad mínima necesaria de invitados en una fiesta, para que, al menos p se conozcan entre sí, o bien, al menos q no se conozcan entre sí?**

Similarmente al problema de la fiesta de seis personas, el hecho que p personas se conozcan entre sí, queda representado por un grafo completo K_p, contenido en G, y el hecho que q personas no se conozcan entre sí, queda representado por un grafo completo K_q contenido en el complemento $\bar{G}$. Podemos, entonces, establecer la pregunta anterior en términos de grafos como sigue: **¿Cuál es el número natural $r(p,q)$ más pequeño, tal que todo grafo con $r(p,q)$ vértices, contenga a K_p o a $\bar{K}_q$?**[4]

Los valores de $r(p,q)$ son conocidos como los *números de Ramsey*, en honor al matemático inglés Frank P. Ramsey (Cambridge 1903- Londres 1930), quien en sus

[4]Notemos que, gracias al Ejercicio 2.10, que un grafo contenga a $\bar{K}_q$ equivale a decir que su complemento contenga a K_q, lo que para el problema de la fiesta corresponde a que, al menos q no se conozcan entre sí.

sólo 27 años de vida produjo una cantidad extraordinaria de trabajos originales en economía, lógica, filosofía y, por supuesto, matemáticas.

Ejercicio 2.11. Demuestre que $r(p,q) = r(q,p)$, para todo $p, q \in \mathbb{N} \setminus \{0\}$. Muestre además que $r(1,q) = 1$ y $r(2,q) = q$, para todo $q \in \mathbb{N} \setminus \{0\}$. Interprete en términos del problema de la fiesta.

La determinación de los números de Ramsey es un problema abierto en matemáticas y sólo se conocen con exactitud para algunos valores específicos de p y q (ver Tabla 2.1). Sin embargo, una cota superior fácil de calcular, escrita en términos de números combinatoriales, es conocida desde el año 1935 gracias al trabajo de Erdös y Szekeres:

$$r(p,q) \leq \binom{p+q-2}{p-1}. \tag{2.7}$$

p	q	$r(p,q)$
3	3	6
3	4	9
3	5	14
3	6	18
3	7	23
3	8	28
3	9	36
4	4	18
4	5	25

Tabla 2.1. Números de Ramsey conocidos

Otra interesante estructura relacionado con el concepto de subgrafos es la que se presenta a continuación:

Definición 2.9. Un grafo *bipartito* es un grafo $G = (V, E)$ en que el conjunto de vértices se puede particionar en dos subconjuntos V^1 y V^2 (es decir, $V = V^1 \cup V^2$ y $V^1 \cap V^2 = \emptyset$), de manera que las aristas de G tienen un extremo en V^1 y otro en V^2 (es decir, $E \subseteq V^1 \times V^2$).

En la Figura 2.10, el grafo de la izquierda es bipartito, sin embargo, el grafo de la derecha, que corresponde al triángulo K_3, no es bipartito, pues no es posible particionar el conjunto de vértices sin dejar una arista con ambos extremos en el mismo conjunto.

Ejercicio 2.12. Probar que todo subgrafo de un grafo bipartito, es también un grafo bipartito.

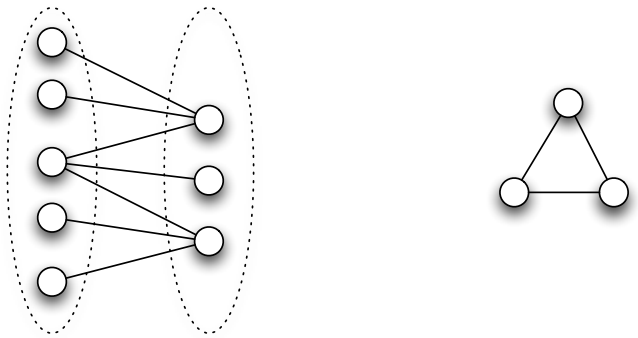


FIGURA 2.10. Ejemplo de un grafo bipartito y un no-bipartito

El ejemplo del lado derecho de la Figura 2.10 podemos generalizarlo a un ciclo de tamaño n. No es difícil ver que la única forma para crear una bipartición en el conjunto de vértices de un ciclo, es dejar a un vértice en V^1 y sus vértices adyacentes en V^2, y así sucesivamente. Esto efectivamente bi-particionará el ciclo si y sólo si n es par.

Ejercicio 2.13. Formalizar la demostración de que un ciclo de tamaño n es bipartito si y sólo si n es par.

Por lo mismo, la existencia de ciclos impares caracteriza a los grafos bipartitos como sigue:

Proposición 2.10. *Un grafo G es bipartito si y sólo si no contiene un ciclo de largo impar.*

Demostración. Debido a los Ejercicios 2.12 y 2.13, el hecho de no contener ciclos impares es claramente una condición necesaria.

Probemos ahora que es también una condición suficiente. Se verifica rápidamente que un grafo es bipartito si y sólo si cada una de sus componentes lo son, luego podemos asumir, sin perdida de generalidad, que el grafo es conexo.

Sea v un vértice de G y definamos V^1 como todos los vértices $v^1 \in V$, tales que, el camino más corto que conecta v y v^1 es de largo par[5]. Definamos también $V^2 = V \backslash V^1$. Notemos que no hay ningúna arista que conecte dos elementos en un mismo conjunto V^i, con $i = 1, 2$, pues esto implicaría la existencia de un ciclo de largo impar. En efecto, denotemos esta arista por $e = (v', v'')$ y llamemos P' y P'' los caminos que

[5]Podemos definir $V^1 = \{v^1 : d(v, v^1) \text{ es par }\}$, donde $d(v, v^1)$ es la distancia entre v y v^1 en un grafo conexo G, la cual, a su vez, se define como el largo del camino más corto que conecta v y v^1 en G, es decir, $d(v, v^1) = \min |P|$; P es un camino que conecta v y v^1. Ver también Ejercicio 4 en la Sección 1.4 de ejercicios.

conectan v con v' y v'', respectivamente. Si $v', v'' \in V^1$, los caminos P' y P'' tienen largo par y entonces el ciclo $C = (P', e, P'')$ tiene largo impar. Ahora, si $v', v'' \in V^2$, entonces los caminos P' y P'' tienen largo impar, pero el ciclo $C = (P', e, P'')$ sigue teniendo largo impar.

Por lo tanto, podemos particionar el conjunto de vértices, usando V^1 y V^2, y lo anterior nos dice que las aristas de G están en $V^1 \times V^2$, concluyendo que G es un grafo bipartito. □

A continuación describimos una estructura de grafos que combina la de grafo completo y bipartito, dadas en las Definiciones 2.6 y 2.9, respectivamente.

Definición 2.11. Diremos que un grafo $G = (V, E)$ es *bipartito-completo*, si éste es bipartito, cuya partición del conjunto de vértices V está dada por V^1 y V^2, y si las aristas de G corresponden a todas las aristas que conectan un vértice de V^1 con uno de V^2 (es decir, $E = V^1 \times V^2$). Si las cardinalidades de V^1 y V^2 son n_1 y n_2, respectivamente, denotaremos a este grafo por K_{n_1,n_2}.

Observación 2.4. El tamaño de un grafo bipartito-completo K_{n_1,n_2} es $n_1 n_2$.

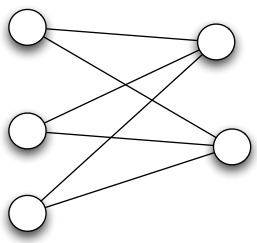


FIGURA 2.11. Ejemplo del grafo bipartito-completo $K_{3,2}$

Ejercicio 2.14. Dibuje los siguientes grafos completos-bipartitos $K_{1,3}$ y $K_{1,n}$. Muestre que los grafos K_{n_1,n_2} y K_{n_2,n_1} son isomorfos.

La introducción de grafos completos-bipartitos nos permite ver que la cota mostrada en el Teorema 2.7 es la mejor posible. En efecto, para todo $n \in \mathbb{N} \setminus \{0\}$ el grafo completo-bipartito $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$ tiene n vértices, tamaño igual a $\lfloor n^2/4 \rfloor$ y no contiene triángulos gracias a la Propiedad 2.10. Mostrando, así, que el Teorema 2.7 de Mantel no se cumple para grafos de tamaño $\lfloor n^2/4 \rfloor$.

Ejercicio 2.15. ¿Cuál de los siguientes grafos es isomorfo a $K_{2,2}$: P_4, C_4 o K_4?

Terminamos esta sección dejando propuesto al lector un entretenido ejercicio que muestra lo simple que es modelar situaciones cotidianas, usando teoría de grafos.

Ejercicio 2.16. El *dilema de los maridos celosos* se trata de tres maridos y sus respectivas esposas que desean cruzar un río. Para aquello cuentan con un sólo bote con capacidad máxima para dos personas. Dado que estos tres maridos son muy celosos, ninguno deja a su mujer en compañía de otro hombre, a menos que él también esté presente. Dibuje un grafo con las distribuciones posibles de las personas y aconseje a estos viajeros cómo cruzar el río.

2.3 Grafos dirigidos

En esta sección adaptaremos algunas definiciones y propiedades establecidas en las secciones anteriores de este capítulo, a un *grafo dirigido*.

Recordemos que un *grafo dirigido* $G = (V, A)$ es una dupla compuesta por un conjunto de vértices $V = V(G)$ y un conjunto de *arcos* $A = A(G) \subseteq V_*^2$, cuyos elementos $a = (v_1, v_2) \in A$ representan una conexión entre los dos vértices v_1 y v_2 en la dirección desde v_1 a v_2. Luego, la principal diferencia entre grafos dirigidos y no-dirigidos es que, para dos vértices v_1 y v_2 del grafo, $a = (v_1, v_2)$ no representa el mismo arco que $a' = (v_2, v_1)$, pues a sólo sirve para "llegar" desde v_1 a v_2, mientras que a' tiene el sentido contrario. Un ejemplo de grafo dirigido fue dado en la Figura 2.2.

Notemos que, si por un instante no consideramos los sentidos en los arcos del grafo de la Figura 2.2, es decir, que sólo nos interesamos en las conexiones establecidas entre los vértices del grafo, obtenemos el grafo no-dirigido introducido en el Ejemplo 2.1 (ver Figura 2.12). Formalizaremos esto en la próxima definición:

Definición 2.12. Sea $G = (V, A)$ un grafo dirigido. La *versión no-dirigida* de (o el *grafo no-dirigido asociado a*) G es el grafo no-dirigido $G' = (V, E)$ con los mismos vértices que G y cuyas aristas conectan dos vértices sólo si existe algún arco en A entre éstos dos vértices, independiente de su dirección. En otras palabras, para dos vértices $v, v' \in V$, $e = (v, v') \in E$ si y sólo si $(v, v') \in A$ o $(v', v) \in A$.

Antes de continuar, explicaremos un ligero cambio de notación con respecto a lo visto en las secciones anteriores. Cuando trabajamos con grafos dirigidos, en general, se necesita establecer claramente la dirección de sus arcos, por lo que en la descripción de un arco, usualmente se explicitan los dos vértices que lo definen y en el orden que lo hacen. Debido a esto, a menudo simplificaremos la notación de los vértices, denotándolos con índices i, j, k, etc., los cuales, a su vez, corresponderán a los subíndices que nos permitan identificar fácilmente a un arco de la forma $a_{ij} = (i, j)$. Esta notación facilitará los cálculos sobre estos arcos y será de gran utilidad en el Capítulo 5 de esta monografía.

A continuación, adaptamos la Definición 2.3 de camino, realizada en el contexto de grafos no-dirigidos, al caso de grafo dirigidos.

Definición 2.13. Sea $G = (V, A)$ un grafo dirigido. Un *camino* entre dos vértices i y j en V es un subgrafo $P = (V', A')$ de G (i.e. $V' \subseteq V$ y $A' \subseteq A$), donde $V' =$

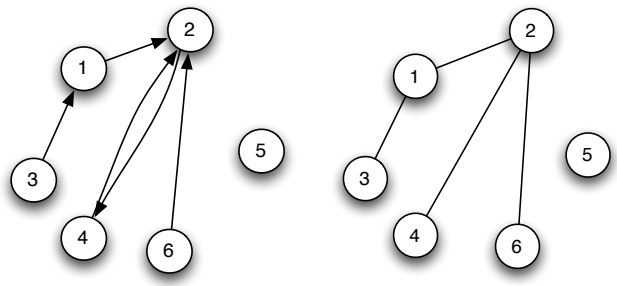


FIGURA 2.12. Ejemplo de un grafo dirigido y su versión no-dirigida

$\{i_0, i_1, \ldots i_l\}$ y $A' = \{(i_0, i_1), (i_1, i_2), \ldots, (i_{l-1}, i_l)\}$, donde $i_0 = i$ e $i_l = j$. Usualmente, el camino P es también identificado como una secuencia de arcos $a_k = (i_k, i_{k+1}) \in A$, con $k = 0, .., l$, o como una secuencia de los vértices $(i_0, i_1, ..., i_{l+1})$, debido a que no hay posibilidad de confusión. Si $i = j$, es decir, si los vértices que se encuentran en los extremos son los mismos, diremos que el camino es un *ciclo*.

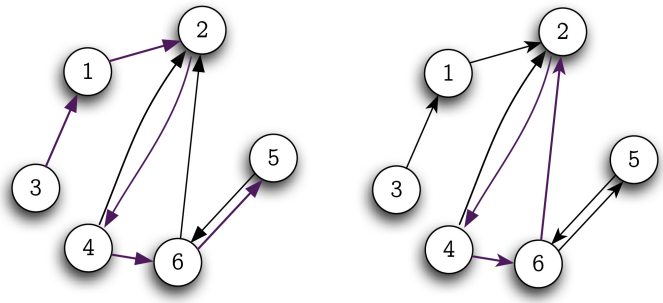


FIGURA 2.13. En púrpura se describe un camino y un ciclo, respectivamente

Una extensión de esta definición, propia de grafos dirigidos, es la siguiente:

Definición 2.14. Sea $G = (V, A)$ un grafo dirigido. Una *cadena* entre dos vértices i y j en V es un camino entre estos vértices en la versión no dirigida de G.

Notemos que si P es una cadena de G y $a = (k, l)$ es uno de los arcos que forman esta cadena, entonces se tiene que $(k, l) \in A$ o bien $(l, k) \in A$. En el primer caso diremos que a es un *arco hacia adelante* y en el segundo caso diremos que a es un *arco hacia atrás* o *en reversa*. Así, una cadena P no es un camino en G si y sólo si existe un arco hacia atrás en P.

Por ejemplo, en el grafo de la Figura 2.13, la secuencia de arcos

$$P = ((3,1),(1,2),(4,2),(4,6),(6,5))$$

es una cadena, pero no es un camino, pues el arco $(4,2)$ es un arco hacia atrás.

Como veremos en la próxima definición, las nociones de grafo conexo, árbol, grafo completo y bipartito se extiende directamente a grafos dirigidos, vía la versión no-dirigida del grafo.

Definición 2.15. Diremos que un grafo dirigido es *conexo (un árbol, completo, bipartito o bipartito-completo)* si su versión no-dirigida es conexa (un árbol, completa, bipartita o bipartita-completa, respectivamente).

Las distintas propiedades asociadas a estas definiciones tienen un interés menor en el caso de grafos dirigidos, por lo que no indagaremos en ellas. Sin embargo, algunas de estas estructuras serán fundamentales para desarrollar algunos razonamientos en los Capítulos 4 y 5.

Las operaciones diferencias, complemento y suma de un arco, se definen de la misma forma que para grafos dirigidos. Se deja propuesto al lector el establecer formalmente estas operaciones.

Cerramos este capítulo, extendiendo la definición de grado de un vértice al contexto de grafos dirigidos.

Definición 2.16. Sea $G = (V, A)$ un grafo dirigido. Para un vértice $v \in V$, el *grado* de v corresponde a la cantidad total de arcos en G que parten o terminan en v (i.e. $(v, v') \in A$ o $(v', v) \in A$), el cual será denotado por $\delta(v)$. Por simplicidad de notación, cuando el vértice sea denotado por un índice i, su grado será denotado simplemente por $\delta_i = \delta(i)$.

En el caso particular que el grado de un vértice v sea 0, diremos que v es un *vértice aislado*.

La principal diferencia entre grafos dirigidos y no-dirigidos con respecto a esta definición, es que, en el caso de grafos dirigidos, podemos contar los arcos que parten y los que salen de un vértice v separadamente, dando origen a los conceptos de *grado saliente* $\delta_+(v) = |\{a = (v, v') \in A\}|$ y *grado entrante* $\delta_-(v) = |\{a = (v', v) \in A\}|$, respectivamente. Obteniendo la relación

$$\delta(v) = \delta_+(v) + \delta_-(v), \quad \forall v \in V.$$

Por ejemplo, en el grafo de la Figura 2.2 notamos que el grado del vértice 2 es igual a $\delta(2) = 4$ y sus grados salientes y entrantes son $\delta_+(2) = 1$ y $\delta_-(2) = 3$.

Ejercicio 2.17. Modele una vecindad del lugar donde vive (por ejemplo, las diez calles más cercanas), usando un grafo dirigido, donde cada vértice represente una intersección entre dos calles y los arcos representen las calles con sus respectivos sentidos. Calcule los grados entrantes y salientes de cada uno de los vértices e identifique si su vecindad tiene alguna estructura particular (conexo, árbol, bipartito, etc.).

2.4 Ejercicios

1. Pruebe que todo grafo (con más de dos vértices) contiene dos vértices de igual grado.
2. Para un grafo $G = (V, E)$, definimos la *matriz de incidencia* A como la matriz de $|V|$ por $|E|$ elementos, cuyas filas están indexadas por los vértices y cuyas columnas están indexadas por las aristas, de tal forma que $A = (A_{ve})$, con $A_{ve} = 1$ si e es incidente en v y $A_{ve} = 0$ si no.
   a) Escriba la matriz de incidencia de todos los grafos que aparecen en este capítulo.
   b) ¿Qué se obtiene al sumar una fila de A? ¿Qué se obtiene al sumar una columna de A?
   c) Muestre el lema del apretón de manos (2.1), usando la matriz de incidencia de un grafo.
   Indicación: Note que puede contar la cantidad de unos que aparecen en A de dos formas distintas: sumando las filas o sumando las columnas.
3. Demuestre que el conjunto de aristas de un grafo $G = (V, E)$ puede ser particionado en ciclos (es decir, existen ciclos C_i, $i = 1, ..., k$ en G tales que $V(C_i) \cap V(C_j) = \emptyset$, para todo $i \neq j$, y $\cup_{i=1}^{k} V(C_i) = V$) si y sólo si cada vértice de G tiene grado par.
4. Definiremos la distancia $d(v, v')$ entre dos vértices v y v' de un grafo conexo G como el largo del camino más corto que conecta v y v' en G, es decir, $d(v, v') = \min |P|$; P es un camino que conecta v y v'. Muestre que d define una *métrica* sobre G.
5. Sea $G = (V, E)$ un grafo de tamaño superior o igual a 2 ($|G| \geq 2$). Mostrar que las siguientes aseveraciones son equivalentes:
   a) Cualquier par de vértices de G forman parte de algún ciclo.
   b) Cualquier par de aristas de G forman parte de algún ciclo.
   c) Para cualquier trío de vértices v_1, v_2 y v_3 de G, existe un camino entre v_1 y v_2 que pasa por v_2.
6. Muestre que para todo grafo conexo G, con al menos dos vértices, existen vértices v_1 y v_2, tales que $G \setminus v_1$ y $G \setminus v_2$ son conexos.
7. Pruebe que un grafo con n vértices y grado mínimo $\delta \geq \frac{n}{2}$ es conexo.
8. Dibuje todos los árboles posibles (salvo isomorfismo) de 4,5 y 9 vértices.
9. Demuestre que un árbol con al menos 2 vértices, contiene al menos 2 vértices de grado 1.
   Indicación: Utilice la igualdad (2.1).
10. Un *árbol generador* T de un grafo G es un árbol que contiene todos los vértices de G (es decir, $V(T) = V(G)$). Muestre que todo grafo conexo contiene un árbol generador.
11. Un grafo que no contiene ciclos lo llamaremos un *bosque*.
    a) Defina un árbol en términos de un bosque.

b) Muestre que un grafo G es un bosque si y sólo si para todo par de vértices distintos v_1 y v_2 de G, existe a lo más un camino entre v_1 y v_2 en G.

12. Demuestre que un bosque de n vértices y k componentes (es decir, k subgrafos conexos que no están conectados entre ellos) tiene $n-k$ aristas. Plantee y estudie la recíproca.
13. La unión entre dos grafos $G = (V, E)$ y $G' = (V', E')$ se define $G \cup G' = (V \cup V', E \cup E')$. Suponiendo que los vértices V y V' son disjuntos y, usando las notaciones dadas en este capítulo, dibuje las uniones de los grafos $G = P_4$ con $G' = C_5$, $G = P_3$ con $G' = K_4$, $G = K_3$ con $G' = K_3$, $G = C_3$ con $G' = K_{1,4}$, y $G = K_3$ con $G' = K_{2,3}$. ¿cómo cambian estos dibujos si ahora asume que los grafos G y G' tienen un vértice en común?.
14. Pruebe formalmente que la unión de dos caminos distintos que conectan los mismos vértices contiene un ciclo.
15. Demuestre que los números de Ramsey satisfacen la siguiente relación de recurrencia (debida a Erdös y Szekeres [**9**]):

$$r(p, q) \leq r(p-1, q) + r(p, q-1).$$

16. Muestre que todo árbol es un grafo bipartito. ¿qué árboles son grafos bipartitos-completos?
17. Pruebe que si G es un grafo con n vértices y $\lfloor n^2/4 \rfloor$ aristas, que no contiene triángulos, entonces G es isomorfo al grafo bipartito-completo $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$.
18. Sea v un vértice arbitrario de un grafo conexo G. Usando la definición de distancia entre dos vértices $d(v, v')$ dada en el Ejercicio 4, muestre que G es bipartito si y sólo si $d(v, v') \neq d(v, v'')$, para toda arista (v', v'') de G.
19. Considere un grafo dirigido $G = (V, A)$ con la propiedad adicional que para cada par de vértices $i, j \in V$ uno y sólo uno de los arcos (i, j) o (j, i) existe. Pruebe que G contiene un camino P que recorre todos los vértices de G sólo una vez.
20. Sea $G = (V, E)$ un grafo no-dirigido conexo con una cantidad par de vértices. Demuestre que siempre se puede asignar una dirección a los vértices de G (obteniendo así un grafo dirigido $G' = (V, A)$, donde cada par de vértices son adyacentes por, a lo más, un arco) de manera que el grado saliente $\delta_+(v)$ en G' sea par, para todo vértice $v \in V$.

Capítulo 3: Árbol recubridor de costo mínimo

3.1 Árbol recubridor de costo mínimo

Suponga que usted está encargado de construir los caminos que unirán a un conjunto de pueblos que hasta ahora no tienen contacto entre sí. Se estudiaron las posibles conexiones entre los pueblos y el costo asociado a construir estos caminos, obteniendo la situación de la Figura 3.1.

Usted debe decidir cuál de estos caminos construir, de forma que todas las ciudades queden conectadas y, dado que sus recursos son escasos, que el costo total de construir estos caminos sea el menor posible. ¿Cómo encontrar la solución de este problema?

Pensemos ahora la situación como si fuera un grafo $G = (V, E)$ donde, los pueblos son los vértices V y los posibles caminos son las aristas E. Además, se tiene un costo c_e asociado a cada arista e, que asumiremos que es positivo ($c_e \geq 0 \ \forall e \in E$).

Lo que tenemos que decidir, es qué conjunto de caminos (subconjunto E' de aristas del grafo) construir, de forma que todas las ciudades estén conectadas (que el grafo (V, E') sea conexo) y que el "costo" total ($\sum_{e \in E'} c_e$) sea el menor posible.

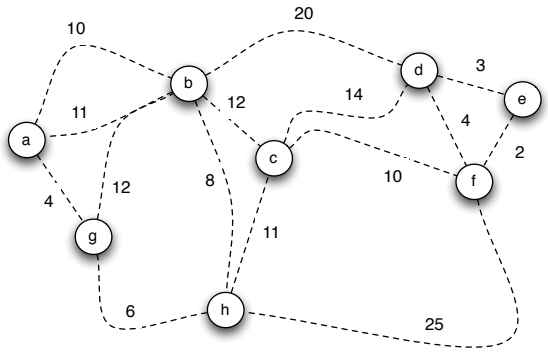


FIGURA 3.1. Grafo de los posibles caminos entre los pueblos. El número al lado de cada arista corresponde al costo de construcción de ese camino

Algunas cosas están claras: en la solución no deberían aparecer ciclos, pues si los hubieran, entonces al sacar una arista del ciclo, se obtiene una solución con menos aristas (o sea, menos costosa) y el grafo seguiría siendo conexo. O sea, lo que necesitamos es un subgrafo que:

1. incluya a todos los vértices del grafo original
2. sea conexo
3. no tenga ciclos

Recordemos del Capítulo 2 que un grafo conexo sin ciclos es un árbol. En nuestro caso, necesitamos un árbol que contenga todos los vértices de V, este tipo de árbol se conoce como *árbol recubridor* de G.

En general, hay muchos árboles recubridores para un grafo dado, pero en nuestro caso, buscamos el árbol recubridor cuyo "costo" (o sea, la suma de los costos de sus aristas) sea el menor posible.

Problema 3.1 (Árbol recubridor de costo mínimo)**.** *Dado un grafo $G = (V, E)$ y un costo c_e, para cada arista $e \in E$, encontrar un árbol recubridor de G de costo mínimo.*

En la Figura 3.2, se pueden ver dos posibles soluciones a este problema: ¿Habrá una mejor? ¿cómo resolvemos este problema? ¿qué tipo de algoritmo utilizamos para resolver este problema?

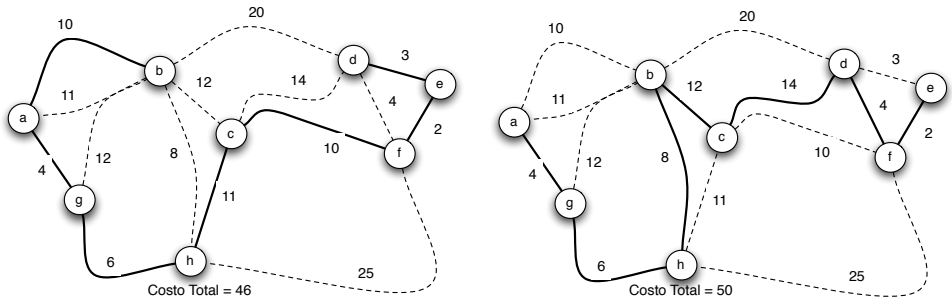


FIGURA 3.2. Dos soluciones al problema anterior

Nuestra intuición puede ayudarnos: si dos pueblos se unen por un camino con muy bajo costo, posiblemente voy a construir ese camino. Este sencillo razonamiento lo podemos llevar a un algoritmo del tipo glotón: ir agregando una a una las aristas de menor costo, mientras el subgrafo sea un árbol. Este algoritmo es conocido como el algoritmo de Prim, en honor a Robert C. Prim quien lo descubrió en 1957.

Solución 3.1 (Algoritmo de Prim)**.** *Sea $T = \emptyset$ un conjunto de aristas, agregar a T la arista en $E \setminus T$ de menor costo, tal que $H = (V, T)$ no tenga ciclos. Detenerse cuando H sea un árbol recubridor.*

En la Figura 3.3 se muestra la solución de nuestro problema inicial, cuyo costo total es 44.

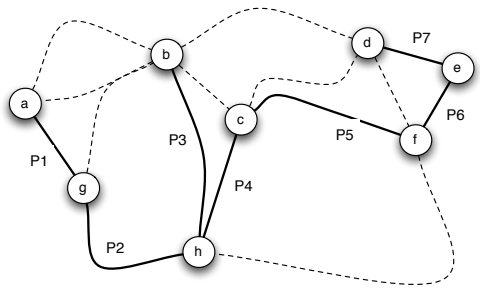


FIGURA 3.3. Pasos del Algoritmo de Prim

Escribamos nuevamente el algoritmo de forma más algorítmica para poder analizar su correctitud y calcular su complejidad.

Algoritmo 3.1 Algoritmo de Prim

Entrada: $G = (V, E)$: grafo
1: $T \leftarrow \emptyset$ /* conjunto de aristas */
2: $u \leftarrow$ vértice inicial cualquiera
3: $U \leftarrow \{u\}$ /* conjunto de vértices */
4: **mientras** $U \neq V$
5: $(u, v) \leftarrow$ arista de menor costo, tal que $u \in U$ y $v \in V \setminus U$.
6: $T = T \bigcup \{(u, v)\}$
7: $U = U \bigcup \{v\}$
8: **fin**
Salida: T árbol recubridor de costo mínimo.

Probemos que el algoritmo de Prim encuentra un árbol recubridor de costo mínimo:

Demostración. Probaremos por inducción en el tamaño de U que en cada paso el subgrafo (U, T) es un sub-árbol de un árbol recubridor de costo mínimo.

El caso base es trivialmente cierto: si el grafo sólo tiene un vértice $\{u\}$, entonces $(\{u\}, \emptyset)$ es un sub-árbol del árbol recubridor de costo mínimo. Supongamos ahora que es cierto para (U, T) y llamemos $U' = U \bigcup \{v\}$ y $T' = T \bigcup \{(u, v)\}$ para el arco (u, v) del paso 5. Podemos observar que (U', T') es conexo y sigue siendo un sub-árbol, ya que se agregó una arista y un vértice, es decir, $|T'| = |U'| - 1$. Sólo falta demostrar que es un sub-árbol de un árbol recubridor de costo mínimo.

Por hipótesis de inducción, T es un sub-árbol de un árbol recubridor de costo mínimo T_M. Si suponemos que la arista $e = (u, v)$ no está en T_M, entonces $T_M \cup e$

posee un ciclo. Luego, hay otra arista e' de ese ciclo que conecta U con $V \setminus U$. Por la definición de la arista e (línea 5), el costo de e' tiene que ser mayor o igual al costo de e, por lo tanto, el árbol recubridor $T_M \cup \{e\} \setminus \{e'\}$ tiene costo menor o igual que T_M y este árbol contiene a T' como sub-árbol, probando el resultado. □

Estudiemos ahora la complejidad del algoritmo de Prim en función de la cantidad de vértices $|V|$. El ciclo **mientras** (líneas 4-8) se repite $|V| - 1$ veces y cada iteración de este ciclo toma $\mathcal{O}(|V|)$, pues debe encontrar la arista de la línea 5, por lo que la complejidad de este algoritmo es $\mathcal{O}(|V|^2)$.

Teorema 3.1. *El algoritmo de Prim corre en tiempo* $\mathcal{O}(|V|^2)$.

Existe otro algoritmo glotón para encontrar un árbol recubridor de costo mínimo, que en ciertos casos puede ser más eficiente que el algoritmo de Prim.

La idea es muy parecida: en cada iteración, agregar la arista de menor costo (¡aunque no quede conectada!) mientras no se forme un ciclo. Este algoritmo se conoce como el Algoritmo de Kruskal, en honor a Joseph Kruskal, quien lo publicó en el año 1956.

Algoritmo 3.2 Algoritmo de Kruskal

Entrada: $G = (V, E)$: grafo
1: $T \leftarrow \emptyset$ /* conjunto de aristas */
2: $u \leftarrow$ vértice inicial cualquiera
3: $U \leftarrow \{u\}$ /* conjunto de vértices */
4: **mientras** $U \neq V$
5: $(u, v) \leftarrow$ arista de menor costo, tal que $T \bigcup \{(u, v\}$ no tiene ciclos.
6: $T = T \bigcup \{(u, v)\}$
7: $U = U \bigcup \{v\}$
8: **fin**
Salida: T árbol recubridor de costo mínimo.

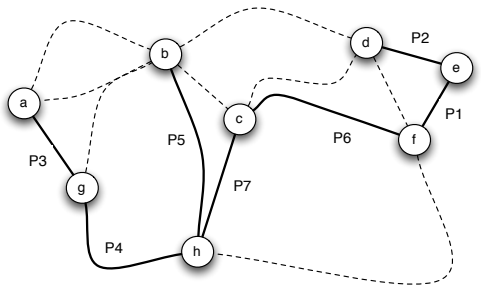


FIGURA 3.4. Pasos del Algoritmo de Kruskal (comparar con Figura 3.3)

Podemos ver que la única diferencia entre el algoritmo de Prim y el algoritmo de Kruskal está en la línea 5 del algoritmo. Dejamos como ejercicio para el lector demostrar que el algoritmo de Kruskal termina con un árbol recubridor de costo mínimo.

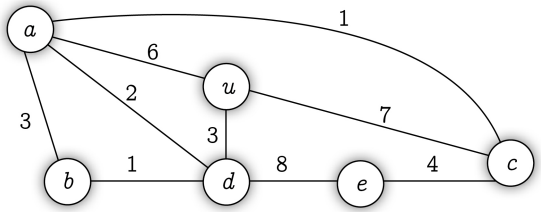
Podemos implementar el algoritmo de Kruskal de una forma más eficiente: antes de ejecutar el algoritmo, ordenamos las aristas del grafo por su costo en forma creciente, lo que requiere un tiempo $\mathcal{O}(|E|\log|E|)$ (ver Capítulo 1). Teniendo las aristas ordenadas, el paso de la línea 5 se puede realizar en forma más eficiente, obteniendo que la complejidad del algoritmo de Kruskal es $\mathcal{O}(|E|\log|E|)$.

Teorema 3.2. *El algoritmo de Kruskal corre en tiempo* $\mathcal{O}(|E|\log|E|)$.

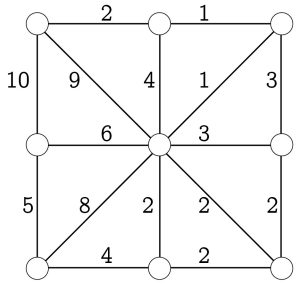
¿Cuál algoritmo conviene usar? Eso depende del número de aristas. Si casi todas las aristas posibles están en el grafo (es decir, $|E| = \mathcal{O}(|V|^2)$), la complejidad del algoritmo de Kruskal sería $\mathcal{O}(|V|^2\log|V|)$ y, por lo tanto, el algoritmo de Prim sería más eficiente. Por otro lado, si el número de aristas es $|E| = \mathcal{O}(|V|)$, el algoritmo de Kruskal correría en tiempo $\mathcal{O}(|V|\log|V|)$ que es menor que el tiempo necesario por el algoritmo de Prim. En nuestro problema inicial, podemos ver que el número de aristas es bastante bajo, pues a una ciudad le interesa conectarse con sus vecinos más cercanos, pero no con aquellos a los que puedo acceder pasando por otra ciudad antes. Por esto, es posible que el algoritmo de Kruskal sea el más adecuado para resolver este tipo de problemas.

3.2 Ejercicios

1. ¿Cuántos árboles recubridores tienen los grafos completos K_3, K_4 y K_5?
2. En el siguiente grafo, encuentre el árbol recubridor de costo mínimo, usando los algoritmos de Prim y de Kruskal.



3. En el siguiente grafo, encuentre el árbol recubridor de costo mínimo, usando los algoritmos de Prim y de Kruskal.



4. En el grafo de la figura anterior, encuentre dos árboles recubridores de peso mínimo distintos.
5. Supongamos que ahora deseamos encontrar el árbol recubridor de peso máximo. ¿Cómo resolver este problema?
6. ¿Cuántos árboles recubridores tiene el grafo completo K_n?
7. Se desea construir un acueducto que una las ciudades de Mafil, Valdivia, Corral, Paillaco y Los Lagos. El costo en miles de millones de pesos de cada tramo viene dado en la siguiente tabla:

	Mafil	Valdivia	Corral	Los Lagos
Valdivia	7			
Corral	19	6		
Los Lagos	5	17	9	
Paillaco	13	5	7	14

¿Qué tramos deberían construirse para gastar la menor cantidad posible de dinero?

Capítulo 4: Camino más corto

4.1 Camino más corto con costos positivos

Supongamos que estamos a cargo de una pizzería a domicilio y, como es usual, nos comprometemos a entregar el pedido en menos de 30 minutos. Para esto, debemos evaluar si es efectivamente posible llegar desde el local a todos los lugares de despacho en menos de ese tiempo. El problema es complicado, hay muchos caminos posible entre la pizzería y cada lugar de despacho, hay calles que son más lentas que otras y, además tiene que preocuparse de respetar el sentido de las calles. ¿Podemos llegar a cada lugar de despacho en menos de 30 minutos? ¿Cuál es el camino más corto entre la pizzería y un domicilio dado?

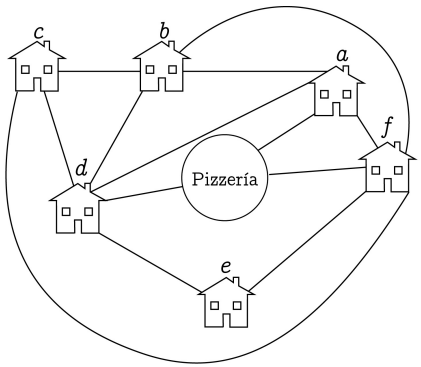


FIGURA 4.1. Ejemplo de una ciudad

Para modelar nuestro problema, representaremos nuestra ciudad por un grafo $G = (V, E)$ donde cada arista es una calle de la ciudad, donde cada arista $e \in E$ tiene un costo asociado c_e (por ejemplo, el tiempo que demora atravesar la calle e), cuyo valor es positivo.

Podemos incluso modelar el sentido de las calles usando grafos dirigidos, la dirección del arco corresponde al sentido de la calle (si la calle es doble sentido, podemos agregar dos arcos, uno en cada sentido de la calle). En este capítulo estudiaremos el caso de grafo no-dirigido, pero todos los resultados pueden extenderse al caso dirigido fácilmente.

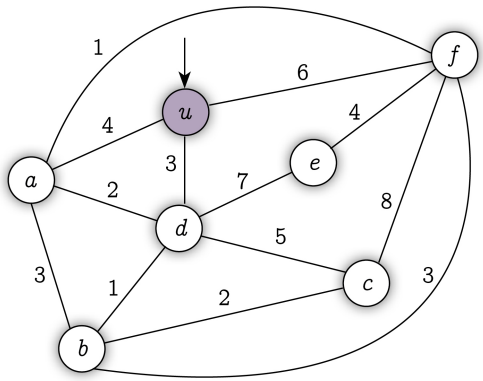


FIGURA 4.2. Grafo asociado a la ciudad anterior

Intentemos, por ejemplo, resolver este problema en el grafo de la Figura 4.2. ¿Cuánto nos demoramos desde la pizzería (vértice u) a un vértice cualquiera? Como vemos, incluso para un grafo tan simple como éste, hay muchos caminos de un vértice a otro, por lo que la estrategia de analizar todos los caminos posibles y escoger el más corto no es un buen algoritmo (o al menos, no es "eficiente").

Sin embargo, para salir de la pizzería, debemos salir por alguno de los tres vértices vecinos a ella (de costo 3, 4 y 6), por lo tanto, podemos asegurar que a donde vayamos, nos demoraremos al menos 3 minutos. Más aún, podemos estar seguros de que el camino más corto de u a d es de largo 3 (pues cualquier otro camino, tomará una distancia mayor que 4 o 6, dependiendo de si salimos por el vértice a o f, respectivamente). Recordemos que esto lo podemos saber gracias a que el costo de las aristas no es negativo.

¿Podemos repetir la idea anterior? Por ejemplo, digamos que queremos ir a otro vértice cualquiera (y recordemos que ya sabemos llegar de u a d). Entonces, el camino más corto de u a ese vértice, o bien, visita d y sale por un vecino de d, o sale por un vecino de u. Es decir, el camino visitará los vértices "conocidos" (vértices grises en la figura) y saldrá por algún vecino de éstos. Así, sabemos que el camino más corto de u a cualquier otro vértice, tiene largo al menos 4, que es el menor costo entre los caminos de u a un vecino no conocido, en este caso, u a d y luego b.

Denotemos por T el conjunto de vértices "conocidos" (vértices grises en la figura) y almacenemos en un vector L las distancias de nuestro vértice inicial u a cada vértice, usando sólo vértices grises. Entonces, el algoritmo que acabamos de hacer consiste en agregar el vértice de no-conocido, cuyo camino tenga el menor costo (en este caso b), verificamos los vecinos de v y actualizamos el valor del vector L para esos vértices, si es que la distancia a través del vértice v es menor. Por ejemplo, en la Figura 4.3,

agregamos el vértice b a T y actualizamos el costo de llegar a c de 8 a 6, ya que ahora este vértice está a distancia 2 de los vértices en T.

El algoritmo que acabamos de explicar, podemos aplicarlo recursivamente hasta incluir todos los vértices del grafo en T. Este algoritmo se conoce como "algoritmo de Dijkstra", en honor al matemático holandés Egsger Dijkstra quien lo describió en 1959.

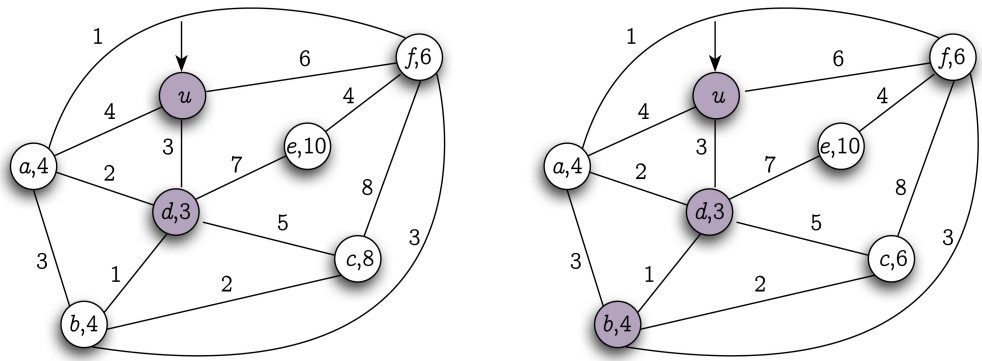


FIGURA 4.3. Grafo asociado a la ciudad anterior

Algoritmo 4.1 Algoritmo de Dijkstra

Entrada: $G = (V, E)$: grafo dirigido, $c : E \to \mathbb{R}^+$ costo de cada arista, u vértice inicial.

1: $L[u] \leftarrow 0$
2: $L[v] \leftarrow c(u, v)$, para todo v vecino de u.
3: $L[v] \leftarrow \infty$, para los vértices v restantes.
4: $T \leftarrow \{u\}$ /* conjunto de vértices que sabemos llegar */
5: **mientras** $T \neq V$
6: w vértice en $V \setminus T$ con el menor valor $L[w]$.
7: $T \leftarrow T \bigcup \{w\}$
8: **para** v vecino de T
9: **si** $L[v] > L[w] + c(v, w)$ **entonces**
10: $L[v] = L[w] + c(v, w)$
11: **fin**
12: **fin**
13: **fin**

Salida: $L[v]$ distancia más corta de u al vértice v $\forall v \in V$.

En la Figura 4.4 podemos ver la iteración final del Algoritmo de Dijkstra al aplicárselo a nuestro problema inicial. Notemos que, si bien sólo tenemos las distancias $L[v]$ desde el vértice u a cada vértice, es fácil reconstruir el camino que debemos utilizar para llegar de u a v en forma óptima. Por ejemplo, el camino más corto de u a e tiene costo 9, por lo que necesariamente tiene que llegar a él a través de f y así sucesivamente, hasta reconstruir el camino $u - a - f - e$.

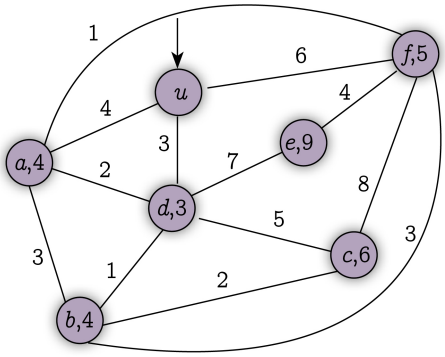


FIGURA 4.4. Resultado del Algoritmo de Dijkstra

También podemos ver en el ejemplo que, cuando un vértice v entra al conjunto T, el valor de $L[v]$ no varía en los siguientes pasos. Esta propiedad es la que demostremos para verificar que el algoritmo efectivamente calcula la distancia más corta desde u al resto de los vértices de G.

Demostración. Demostraremos que en cada paso se cumplen dos condiciones:

1. Para todo vértice $v \in T$, el valor de $L[v]$ es igual al largo del camino más corto desde u a v
2. Para todo vértice $v \notin T$, el valor de $L[v]$ es igual al largo del camino más corto desde u a v, pasando solamente por los vértices de T.

Estas propiedades las demostraremos por inducción en el tamaño de T. Podemos ver que si estas propiedades son verdaderas, en particular cuando T contenga todos los vértices del grafo V, significará que $L[v]$ son los valores que buscamos.

Probemos las propiedades anteriores. En el caso base ($|T| = 1$, es decir, $T = \{u\}$), se cumple gracias a las primeras tres líneas del algoritmo, donde se asignan los valores de L de forma de cumplir con las condiciones.

Probemos ahora la inducción. Supongamos que estamos en el paso 7 del algoritmo y vamos a agregar w a T. Por hipótesis de inducción, los vértices de T tienen el valor correcto de L, por lo que, para cumplir la primera condición sólo falta probar que $L[w]$ tiene el valor correcto. Supongamos que no se cumple, es decir, existe un camino P desde u a w de costo $K < L[v]$ (ver Figura 4.5). Por hipótesis de inducción, este

camino tiene que utilizar otros vértices que no estén en T. Sea z el primer vértice del camino P que no está en T. Por hipótesis de inducción, el costo del camino en P, desde u a z es $L[z]$ y por ser z un vértice intermedio en el camino más corto desde u a w, se debe cumplir que $L[z] < K$, luego, se concluye que $L[z] < L[v]$. Pero por otro lado, en la línea 6 elegimos a w como aquel vértice con menor valor en L que todos los otros vértices fuera de T, en particular el vértice z, por lo tanto, $L[w] \leq L[z]$, lo que nos lleva a una contradicción. Esto demuestra la primera condición enunciada al principio de la demostración. La segunda condición, se cumple gracias a las reasignaciones de la línea 10. □

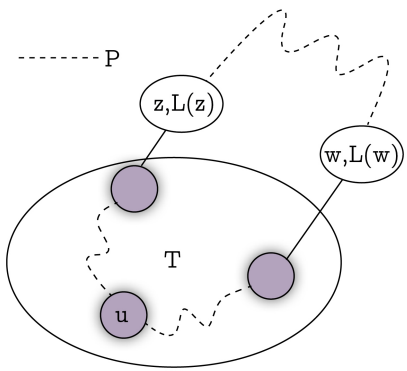


FIGURA 4.5. Demostración del algoritmo de Dijkstra

Estudiemos la complejidad del algoritmo de Dijkstra. El ciclo **mientras** (líneas 5 - 13) se repite $|V|-1$ veces y cada repetición del ciclo requiere encontrar el vértice fuera de T de menor distancia (línea 6) que se puede realizar en $\mathcal{O}(|V|)$ y la modificación de los vecinos (línea 10) que se realiza a lo más $|V|$ veces en cada iteración. Por lo tanto, la complejidad del algoritmo de Dijkstra es $\mathcal{O}(|V|^2)$.

Teorema 4.1. *El Algoritmo de Dijkstra corre en tiempo* $\mathcal{O}(|V|^2)$.

4.2 Camino más corto con costos negativos

El algoritmo de Dijkstra tiene un problema: los costos de las aristas tienen que ser positivos. Esto es un problema, pues en muchas aplicaciones los costos pueden ser negativos. Por ejemplo, pensemos que los caminos son canales con agua, por lo que el "costo" (por ejemplo, de energía) al atravesar una arista puede ser positivo (en contra de la corriente) o negativo (a favor de la corriente).

En el ejemplo de la Figura 4.6 hemos cambiado el costo de la arista c con f por -4. Podemos ver que la lógica del algoritmo de Dijkstra ya no es válida, ya que si

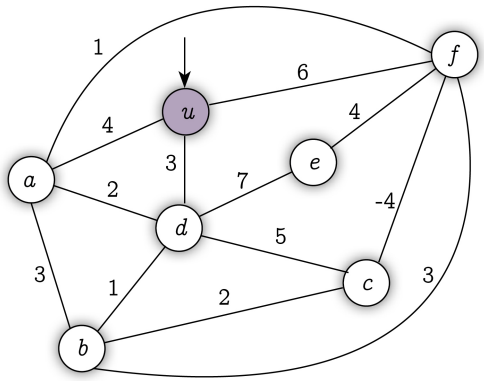


FIGURA 4.6. Ejemplo del problema

bien llegar u a f tiene largo 6, esto nos permitiría llegar a c con un camino de largo $6 - 4 = 2$, por lo que la demostración del teorema ya no es cierta.

Más aún, se puede dar el caso que el grafo contenga un ciclo, cuya suma de costos sea negativo. Esto es complicado, pues el concepto de camino más corto pierde sentido, ya que el camino más corto tiene costo $-\infty$, puesto que para cualquier camino de un vértice a otro, existe otro camino de costo menor, simplemente pasando suficientes veces por el ciclo de costo negativo.

Por todo esto, necesitamos un nuevo algoritmo que sea capaz de encontrar el camino más corto, aun si hay costos negativos, y si aparecen ciclos de costo negativos, darse cuenta de esto e indicarlos. El principal algoritmo para esto es conocido como Algoritmo de Ford-Bellman, desarrollado por Richard Bellman y Lester Ford.

La idea es similar a la de Dijkstra, almacenaremos en un vector L la distancia desde u al vértice, sin embargo, modificaremos la forma de actualizar este valor. En vez de ir agregando glotonamente vértices como lo hace Dijkstra, analizaremos cada arista del grafo para chequear si nos acerca o no a otros vértices. Repitiendo este procedimiento $|V| - 1$ veces, podemos estar seguro que el valor de L fue calculado correctamente.

En la Figura 4.7 podemos ver el resultado de ejecutar el Algoritmo de Ford-Bellman sobre el ejemplo de la Figura 4.6.

Adicionalmente, el algoritmo nos permite detectar ciclos negativos: si al terminar el algoritmo hay una arista (v, w) tal que $L[w] > L[v] + c(v, w)$ significa que la arista está en un ciclo de costo negativo, por lo que el problema del camino más corto no tiene sentido.

Probemos que el Algoritmo de Ford-Bellman efectivamente encuentra el camino más corto desde u a todos los vértices.

Algoritmo 4.2 Algoritmo de Ford-Bellman

Entrada: $G = (V, E)$: grafo dirigido, $c : E \to \mathbb{R}$ costo de cada arista, u vértice inicial.

1: $L[u] \leftarrow 0$
2: $L[v] \leftarrow \infty$, para todo $v \neq u$.
3: **para** $i = 1 \ldots |V| - 1$
4: **para** cada arista $(v, w) \in E$
5: **si** $L[w] > L[v] + c(v, w)$ **entonces**
6: $L[w] \leftarrow L[v] + c(v, w)$
7: **fin**
8: **fin**
9: **fin**

Salida: $L[v]$ distancia más corta de u al vértice v $\forall v \in V$.

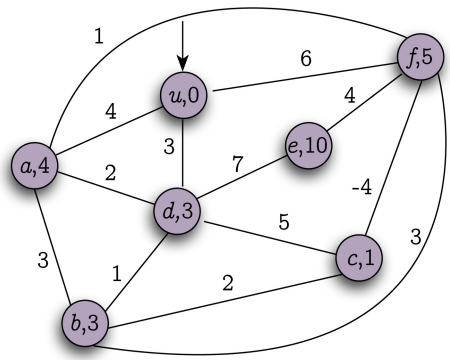


FIGURA 4.7. Resultado del algoritmo de Ford-Bellman

Demostración. Probaremos por inducción en el número de repeticiones del ciclo **para** (líneas 3-9) que se cumplen las siguientes propiedades:

1. Si $L[v] \neq \infty$, entonces $L[v]$ es igual al largo de algún camino de u a v.
2. Si hay un camino de u a v con a lo más i vértices, entonces $L[v]$ es a lo más el costo del camino más corto de u a v con a lo más i vértices.

En el caso base $i = 0$, las condiciones iniciales $L[u] = 0$ y $L[v] = \infty$ $\forall v \neq u$ aseguran que esto se cumpla.

Realicemos ahora la inducción. Para la primera propiedad, supongamos que estamos actualizando el costo de $L[w]$ por $L[v] + c(v, w)$. Por hipótesis de inducción, $L[v]$ es el costo de un camino de u a v, luego, $L[w]$ es el costo del camino que va de u a v y luego a w, por lo que la primera propiedad es cierta.

Probemos la segunda propiedad. Tomemos el camino más corto desde u a v con a lo más i aristas. Sea z el último vértice visitado por el camino, antes de llegar a v. Entonces, necesariamente este camino de u a z es el camino más corto, usando a lo más $i-1$ aristas. Por hipótesis de inducción, el valor de $L[z]$ es a lo más el costo de este camino de u a z, por lo que $L[z] + c(z, v)$ es a lo más el costo del camino más corto de u a v, usando a lo más i aristas, por lo que al final del i-ésimo ciclo **para**, el valor de $L[z]$ cumplirá la segunda condición.

Para terminar la demostración, notemos que cuando $i = |V|$, las propiedades nos aseguran que el valor de $L[v]$ es el costo del camino más corto de u a v, usando a lo más $|V|$ vértices, que es el valor que buscamos (a menos que existen ciclos negativos). □

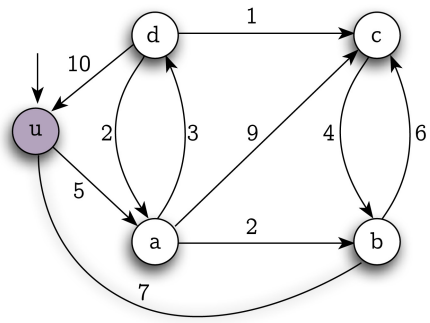
Estudiemos la eficiencia del Algoritmo de Ford-Bellman. El algoritmo incluye dos ciclos **para** anidados: el primero se repite $|V| - 1$ veces y el segundo, $|E|$ veces, por lo que la complejidad del algoritmo es $\mathcal{O}(|V| \cdot |E|)$.

Teorema 4.2. *El Algoritmo de Ford-Bellman corre en tiempo* $\mathcal{O}(|V| \cdot |E|)$.

Recordemos que el algoritmo de Dijkstra corre en tiempo $\mathcal{O}(|V|^2)$, es decir, es más eficiente. Debido a esto, para resolver el problema del camino más corto, comúnmente se utiliza el algoritmo de Dijkstra, a menos de que existan costos no-negativos en las aristas, en cuyo caso usamos Ford-Bellman.

4.3 Ejercicios

1. Encontrar los caminos más cortos para el grafo de la siguiente figura



2. En una empresa se instala una red de N computadores, todos conectados entre sí y además conectados a un supercomputador central (SC). Para testear la velocidad de la comunicación en la red, los técnicos deciden probar la eficiencia del sistema enviando N-1 mensajes en forma paralela (el SC tiene N procesadores), desde el SC hacia los demás equipos. Desafortunadamente, el SC está conectado solamente a uno de los equipos. Describa de qué manera se podría realizar el envío de mensajes, de tal forma que se minimice el tiempo entre el envío por parte del SC y la recepción por parte de un equipo (considere que el tiempo de demora del mensaje es $a_{i,j}$ entre el equipo i y j. El SC es el equipo 1).
3. Encuentre un ejemplo de un grafo con aristas de costo negativo (pero sin ciclos de costo negativo), donde Dijkstra entregue una solución incorrecta.
4. Suponga que tenemos aristas con costo negativos. Una solución sería sumar a cada arista una cantidad fija de forma de dejar todos los costos positivos y buscar el camino más corto ¿por qué esto no funciona? Busque un contraejemplo donde esto no funcione.
5. Un colegio debe reparar los pizarrones de sus salas regularmente, por lo cual está planificando esta reparación para los siguientes 7 años (2011 - 2017). La dirección del colegio ha indicado que cada pizarrón debe ser reparado después de 2 años de uso y antes de los 5 años de uso. El costo de esta reparación depende de cuantos años de uso tiene el pizarrón. La siguiente tabla muestra el costo de reparar un pizarrón, dependiendo de cuándo fue comprado y cuántos años de uso tiene:

	2 años uso	3 años uso	4 años uso
2011	3.800	4.100	6.800
2012	4.000	4.800	7.000
2013	4.200	5.100	7.200
2014	4.800	5.700	-
2015	5.700	-	-

Resuelva el problema usando Dijkstra. Es decir, construya un grafo apropiado, de forma que el camino más corto corresponda a la estrategia óptima de reparaciones que debe realizar el colegio.

Capítulo 5: Flujo en Redes: Problema de Flujo Máximo

En teoría de grafos, una *red* se entiende como un grafo dirigido, a cuyos arcos se les asigna un *flujo* y una *capacidad*, conceptos que serán formalmente definidos en este capítulo.

Este concepto es usualmente usado para modelar distintos problemas en ingeniería, como por ejemplo, el tráfico en una ciudad, las señales eléctricas en un circuito, las señales que se distribuyen a lo largo de la red telefónica, el transporte de gas, petróleo u otro fluido, entre otros.

En este capítulo definiremos formalmente nociones como flujo máximo, capacidad de arcos, corte, corte mínimo, y estableceremos importantes relaciones que existen entre ellas. Estas relaciones nos permiten resolver eficientemente ciertos problemas en flujo en redes, como por ejemplo, el de la determinación del flujo máximo en una red, el cual será estudiado en profundidad en este capítulo.

5.1 Definiciones básicas

En esta sección introduciremos algunas de las principales nociones que se usarán en el capítulo. Los conceptos básicos en teoría de grafos fueron definidos en el Capítulo 2 y se dan por conocidos.

A lo largo de este capítulo trabajaremos con grafos dirigidos y conexos $G = (V, A)$, salvo que se especifique explícitamente lo contrario. En este contexto, representaremos a un arco a en A de la forma $a = a_{ij} = (i, j)$, cuando éste vaya desde el vértice $i \in V$ al vértice $j \in V$. Esta notación será de suma utilidad cuando deseemos representar un problema de flujo en redes, usando optimización lineal[1].

Definición 5.1. Diremos que un grafo orientado $G = (V, A)$ es una *red*, si a cada arco $a_{ij} = (i, j) \in V$ le asociamos una cantidad real positiva f_{ij}, llamada *flujo* del arco a_{ij}. El vector $f = (f_{ij}) \in \mathbb{R}^{|A|}$ (con $(i, j) \in A$) es llamado *vector de flujos* o simplemente *flujo* del grafo G. A cada arco $a_{ij} \in V$ de una red $G = (V, A)$ también se le puede asociar una cota superior para los flujos f_{ij} respectivos. Esta cantidad positiva se denomina *capacidad* y se denota por c_{ij}. Con esto, para cada arco $a_{ij} \in V$, se tiene que:

$$0 \leq f_{ij} \leq c_{ij}. \tag{5.1}$$

El vector $c = (c_{ij}) \in \mathbb{R}^{|A|}$ (con $(i, j) \in A$) es llamado *vector de capacidades* o simplemente *capacidad* del grafo G. Permitiremos que $c_{ij} = \infty$, lo cual simplemente significa

[1] Ver monografía "Optimización Lineal: una mirada introductoria"

que dicho arco no tiene una cota superior para el flujo, es decir, *tiene capacidad no acotada*.[2]

El concepto de red aparece naturalmente en el modelamiento de problemas reales en ingeniería y otras áreas afines. Podemos, por ejemplo, pensar en el sistema de distribución de gas por cañerías en una ciudad. Aquí, cada casa de la ciudad puede representarse como un vértice $i \in V$ de un grafo $G = (V, A)$, cuyos arcos, $a \in A$, representan el sistema de cañerías que existe entre las casas para transportar el gas. El flujo f_{ij} representaría la cantidad de gas que está pasando por la cañería que une las casas i y j, cuya capacidad c_{ij} viene dada por el grosor de la cañería.

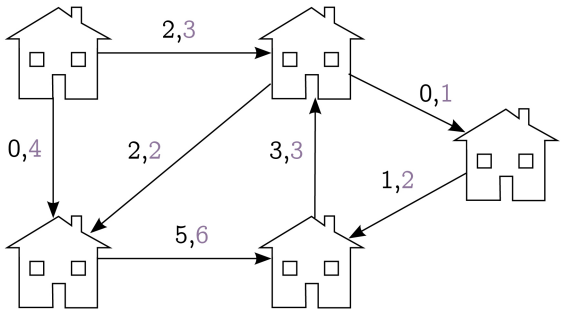


FIGURA 5.1. Transporte de gas en una ciudad, los flujos f_{ij} se presentan en negro y las capacidades c_{ij} en púrpura

Este ejemplo puede ser más realista si asociamos a cada casa una demanda de gas (por ejemplo, la demanda promedio mensual), que representamos por un cierto valor real *negativo* que asignaremos al vértice correspondiente a esa casa. Más aún, si las casas demandan gas, necesitamos empresas que "oferten" dicho gas, las cuales incorporaremos al grafo anterior como nuevos vértices. Asumiremos que existen cañerías que conectan estas empresas con algunas casas y que permiten, entonces, la distribución del gas, éstas serán representadas por nuevos arcos en el grafo. Cada empresa produce una cantidad de gas (producción promedio mensual) que se representa como un valor *positivo* asignado al vértice que la representa.

[2]Una extensión natural es permitir cotas inferiores $l = (l_{ij}) \in \mathbb{R}^{|A|}$ positivas, pero no necesariamente nulas. Los algoritmos y la teoría presentados en este capítulo son fácilmente adaptables a este cambio, pero la notación se hace un poco más complicada, por lo que no estudiaremos esta extensión.

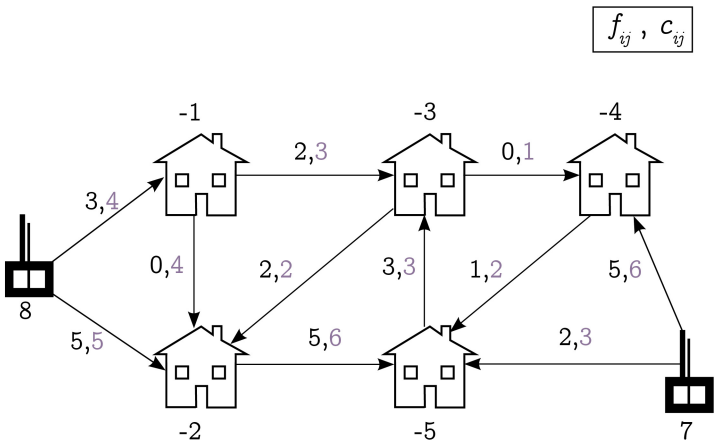


FIGURA 5.2. Transporte de gas en una ciudad, incluyendo ofertas y demandas

En este ejemplo, las casas son vértices demandantes y las empresas son vértices ofertantes, el valor que estos vértices tengan se llamará simplemente *oferta* y su caracterización como oferta o demanda vendrá dada por el signo que este valor tenga.

Formalmente esta cantidad se define como sigue:

Definición 5.2. En un red $G = (V, A)$, podemos asignar a cada vértice $i \in V$ un valor real o_i que será llamado *oferta del vértice* i. Si $o_i > 0$, diremos que este vértice es un vértice *ofertante*, si $o_i < 0$, diremos que es un vértice *demandante* y si $o_i = 0$, diremos que es un vértice *transiente* o *de paso*.

Para efectos de nuestra modelación asumiremos que un flujo f asociado a la red G, siempre debe satisfacer las ecuaciones de conservación:

$$\sum_{(i,j)\in A} f_{ij} - \sum_{(k,i)\in A} f_{ki} = o_i, \quad \forall i \in V. \tag{5.2}$$

Es decir, para cada vértice i, el flujo total que sale de i menos el flujo total que entra de i debe ser igual a su oferta o demanda según corresponda. Para enfatizar este hecho, en tal caso, diremos que el flujo f es *factible*.

Ejercicio 5.1. Verificar que el flujo $f = (f_{ij})$ de la red de la Figura 5.2 es factible.

Las ecuaciones (5.2) son conocidas como la *ley de Kirchhoff*, en honor al físico prusiano Gustav Robert Kirchhoff (1824 – 1887), nacido en Königsberg, Prusia del Este (ciudad que hoy es conocida como Kaliningrado y pertenece a Rusia), al igual que uno de los precursores de la teoría de grafos, Leonhard Euler (ver Capítulo 6). Kirchhoff estableció esta ley en 1845, en el contexto de circuitos eléctricos, mientras

aún era estudiante en la Universidad Albertus de Königsberg, lo que le permitió obtener los valores de la intensidad de corriente y potencial en cada punto de un circuito. Esta ley es una aplicación de la ley de conservación de la energía y sigue siendo utilizada en nuestro días.

Observación 5.1. Para que exista un flujo f que satisfaga las ecuaciones de flujo (5.2), debe necesariamente cumplirse que $\sum_{i\in V} o_i = 0$, es decir, que la oferta total (que son valores de signo positivo) sea igual que la demanda total (valores de signo negativo). Sin embargo, sin perdida de generalidad, uno puede también tener que $\sum_{i\in V} o_i \geq 0$, es decir, que la oferta sea mayor o igual que la demanda total. Para esto, basta incluir un vértice artificial $\tilde{v}$ que demande el valor $\sum_{i\in V} o_i \geq 0$ y cuyos grados entrantes y salientes sean $|V|$ y 0, respectivamente (es decir, todos los vértices envíen un arco a $\tilde{v}$ y que ningún arco comience desde $\tilde{v}$), para obtener una nueva red que sí satisface que $\sum_{i\in V} o_i = 0$.

En caso que la red $G = (V, A)$ tenga además capacidades $c = (c_{ij})$ asociadas, un flujo factible f debe también cumplir, para cada arco (i, j) en A, las restricciones sobre las capacidades dadas en (5.1).

Las nociones aquí definidas nos permitirán trabajar en la próxima Sección 5.2 con un problema particular de flujo en redes.

5.2 Problema de flujo máximo

Consideremos una red $G = (V, A)$ con capacidad $c = (c_{ij})$ y cuyo flujo es denotado por $f = (f_{ij})$. Para el problema de flujo máximo impondremos que existen sólo dos vértices en V, cuya oferta es distinta de cero. Llamemos a estos vértices *origen* y *destino*, y denotémoslos por las letras o y d, respectivamente. Así, las ecuaciones de flujo (5.2) nos dicen que:

$$\sum_{(o,j)\in A} f_{oj} - \sum_{(k,o)\in A} f_{ko} = o_o = -o_d$$

$$\text{y} \quad \sum_{(i,j)\in A} f_{ij} - \sum_{(k,i)\in A} f_{ki} = 0, \quad \forall i \in V \setminus \{o, d\}.$$

Por convención supondremos que o es el vértice ofertante ($o_o > 0$) y en consecuencia d será el vértice demandante ($o_d < 0$). Luego, la red anterior modela el envío de flujo desde el vértice origen o al vértice destino d (por esta razón estos vértices son también llamados fuente y pozo, respectivamente). En esta sección estudiaremos el problema de maximizar este envío, es decir, encontrar un flujo f factible para G, tal que o_o sea máximo (o equivalentemente o_d sea mínimo). Notemos que el flujo f y la cantidad o_o serán las variables de nuestro problema, pero o_o queda únicamente determinado por f, debido a la ecuación de flujo (5.2) del vértice o.

Este problema puede ser descrito como el siguiente problema de optimización lineal[3]:

$$o_o^* = \text{Maximizar} \quad o_o \tag{5.3}$$

sujeto a las ecuaciones conservación de flujo

$$\begin{cases} \sum_{(o,j)\in A} f_{oj} - \sum_{(k,o)\in A} f_{ko} = o_o, \\ \sum_{(d,j)\in A} f_{dj} - \sum_{(k,d)\in A} f_{kd} = -o_o, \\ \sum_{(i,j)\in A} f_{ij} - \sum_{(k,i)\in A} f_{ki} = 0, \quad \forall i \in V \setminus \{o,d\}, \\ 0 \le f_{ij} \le c_{ij}, \qquad \forall (i,j) \in A. \end{cases} \tag{5.4}$$

Aquí hemos denotado por o_o^* al valor óptimal del problema de optimización lineal (5.3)-(5.4). Este problema podría ser resuelto con algoritmos usados para resolver problemas de optimización lineal, como por ejemplo SIMPLEX[4]. Sin embargo, nuestra idea es atacar este problema aprovechando la estructura de grafo que lo define. Esto nos permitirá obtener algoritmos más eficientes, en términos de complejidad computacional (ver Sección 5.2.3), que los que podrían obtenerse de la optimización lineal. Para esto, los siguientes conceptos nos serán de gran utilidad.

Definición 5.3. Para $G = (V, A)$ una red, diremos que $\tilde{A} \subseteq A$ es un *corte* de la red G, si el grafo $G \setminus \tilde{A}$ tiene dos componentes conexas $G_1 = (V_1, A_1)$ y $G_2 = (V_2, A_2)$, tales que $o \in V_1$, $d \in V_2$, V_1 y V_2 forma una partición de V, y A_1, A_2 y $\tilde{A}$ forman una partición de A. Usualmente, denotaremos al conjunto de vértices V_1 por $\tilde{V}$ y lo llamaremos *conjunto de vértices generado* por el corte $\tilde{A}$. Notemos que su complemento $\tilde{V}^c$ coincide con V_2.

Por ejemplo, en la Figura 5.3 se representan tres diferentes cortes para un mismo grafo, definidos por los arcos con líneas discontinuas.

En el primer corte de la Figura 5.3 se tiene que el conjunto de vértices generado por el corte viene dado por $\tilde{V} = \{o\}$ y $\tilde{V}^c = V \setminus \tilde{V} = V \setminus \{o\}$.

Ejercicio 5.2. Demostrar que podemos definir de manera equivalente un corte $\tilde{A} \subseteq A$ de $G = (V, A)$ como el subconjunto de arcos tales que: 1) si se eliminan de A, no

[3]Ver monografía "Optimización Lineal: una mirada introductoria".

[4] Hemos mencionado en el Capítulo 1 que el algoritmo SIMPLEX, para resolver un problema de optimización lineal, tiene complejidad *exponencial* en el tamaño de los datos, lo cual resulta en teoría menos eficiente que el algoritmo de Ford-Fulkerson, presentado en la Sección 5.2.2. Sin embargo, existen algoritmos polinomiales que resuelven problemas de optimización lineal (por ejemplo, algoritmos de punto interior) que son comparables en términos de eficiencia computacional a los aquí mostrados.

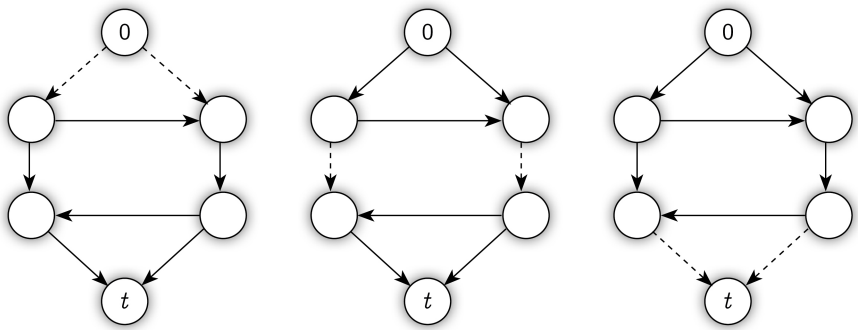


FIGURA 5.3. Ejemplos de cortes

existe una cadena[5] de G que conecte el vértice origen o con el vértice destino d, y 2) si eliminamos un arco $\tilde{a}$ de $\tilde{A}$, la propiedad 1) deja de cumplirse.

La identificación de un corte $\tilde{A} \subseteq A$ con el subconjunto de vértices $\tilde{V}$ que genera, nos permite relacionar la noción de corte con el problema de flujo máximo como se ve en el siguiente resultado.

Proposición 5.4. *Sea $G = (V, A)$ una red. Entonces, para cualquier corte $\tilde{A}$ de G y para cualquier flujo factible f asociado a G, se tiene que:*

$$o_o = \sum_{i \in \tilde{V},\, j \in \tilde{V}^c :\, (i,j) \in A} f_{ij} - \sum_{i \in \tilde{V}^c,\, j \in \tilde{V} :\, (i,j) \in A} f_{ij},$$

donde $\tilde{V}$ es el subconjunto de vértices que genera el corte $\tilde{A}$.

Demostración. Como f es un flujo factible de (5.4) tenemos que

$$\begin{aligned} o_o &= \sum_{(o,j) \in A} f_{oj} - \sum_{(k,o) \in A} f_{ko} \\ &= \sum_{(o,j) \in A} f_{oj} - \sum_{(k,o) \in A} f_{ko} + \sum_{i \in \tilde{V} \setminus \{o\}} \left(\sum_{(i,j) \in A} f_{ij} - \sum_{(k,i) \in A} f_{ki} \right), \end{aligned}$$

pues $\sum_{(i,j) \in A} f_{ij} - \sum_{(k,i) \in A} f_{ki} = 0$, para todo vértice de paso i (es decir, para todo i distinto de o o de d). Agrupando las sumatorias del lado derecho obtenemos que

$$o_o = \sum_{i \in \tilde{V}} \left(\sum_{(i,j) \in A} f_{ij} - \sum_{(k,i) \in A} f_{ki} \right) = \sum_{i \in \tilde{V}} \sum_{(i,j) \in A} f_{ij} - \sum_{i \in \tilde{V}} \sum_{(j,i) \in A} f_{ji}, \tag{5.5}$$

[5]Consultar la definición en el Capítulo 2.

donde en la última igualdad realizamos un cambio de índices de k a j y separamos las sumatorias. Ahora, dado que

$$\sum_{i\in\tilde{V}}\sum_{(i,j)\in A} f_{ij} = \sum_{i\in\tilde{V}, j\in\tilde{V}:\,(i,j)\in A} f_{ij} + \sum_{i\in\tilde{V}, j\in\tilde{V}^c:\,(i,j)\in A} f_{ij}$$

y

$$\sum_{i\in\tilde{V}}\sum_{(i,j)\in A} f_{ji} = \sum_{i\in\tilde{V}, j\in\tilde{V}:\,(i,j)\in A} f_{ji} + \sum_{i\in\tilde{V}, j\in\tilde{V}^c:\,(i,j)\in A} f_{ji},$$

de la igualdad $\sum_{i\in\tilde{V}, j\in\tilde{V}:\,(i,j)\in A} f_{ij} = \sum_{i\in\tilde{V}, j\in\tilde{V}:\,(i,j)\in A} f_{ji}$ se deduce que

$$\begin{aligned}\sum_{i\in\tilde{V}}\sum_{(i,j)\in A} f_{ij} - \sum_{i\in\tilde{V}}\sum_{(j,i)\in A} f_{ji} &= \sum_{i\in\tilde{V},\, j\in\tilde{V}^c:\,(i,j)\in A} f_{ij} - \sum_{i\in\tilde{V}, j\in\tilde{V}^c:\,(i,j)\in A} f_{ji} \\ &= \sum_{i\in\tilde{V},\, j\in\tilde{V}^c:\,(i,j)\in A} f_{ij} - \sum_{i\in\tilde{V}^c,\, j\in\tilde{V}:\,(i,j)\in A} f_{ij},\end{aligned}$$

y la proposición queda demostrada gracias a (5.5). □

La propiedad anterior nos dice que o_o corresponde al flujo desde $\tilde{V}$ a $\tilde{V}^c$ menos el flujo de $\tilde{V}^c$ a $\tilde{V}$. Por lo tanto, un corte $\tilde{A}$ nos permite caracterizar el valor del flujo enviado desde el vértice origen o al vértice destino d únicamente en función de los flujos en los arcos que definen el corte.

Ejemplo 5.1. Para ilustrar la Propiedad 5.4, asignemos un flujo unitario ($f_{ij} = 1$) a todos los arcos del grafo de la Figura 5.3 con excepción de los 2 arcos horizontales, obteniendo que $o_o = 2$. Notemos que para cualquiera de los tres cortes de la Figura 5.3 tenemos que $\sum_{i\in\tilde{V},\, j\in\tilde{V}^c:\,(i,j)\in A} f_{ij} - \sum_{i\in\tilde{V}^c,\, j\in\tilde{V}:\,(i,j)\in A} f_{ij} = 2 - 0 = 2$, corroborando el resultado obtenido en la Proposición 5.4.

Definición 5.5. La *capacidad* de un corte se define como la suma de las capacidades de los arcos que lo conforman y que "siguen el sentido" definido desde el vértice origen o al vértice destino d. Es decir, si denotamos la capacidad del corte $\tilde{A}$ por $c(\tilde{A})$ y si $\tilde{V}$ es el subconjunto de vértices que genera este corte $\tilde{A}$, la capacidad del corte se calcula:

$$c(\tilde{A}) = \sum_{i\in\tilde{V},\, j\in\tilde{V}^c:\,(i,j)\in A} c_{ij}.$$

Por ejemplo, si en el primer ejemplo de la Figura 5.3 asignamos las capacidades 1 al arco que va desde el vértice o al vértice del lado izquierdo y 3 al arco que va desde al vértice o al vértice del lado derecho, se tendrá que la capacidad de dicho corte será igual a 1+3=4.

La noción de capacidad de un corte está muy ligada con la de flujo máximo de una red, es esto lo que nos permitirá derivar algoritmos eficientes que aprovechen la estructura del problema de flujo máximo. La primera relación entre estas dos nociones es establecida en la siguiente proposición.

Proposición 5.6. *Sea $G = (V, A)$ una red. Para cualquier corte $\tilde{A}$ de G y para cualquier flujo factible f asociado a G, se tiene que o_o es siempre menor o igual que la capacidad de $\tilde{A}$. En particular, el valor óptimal o_o^* del problema de flujo máximo en G (es decir, el problema* (5.3)-(5.4)*) satisface que:*

$$o_o^* \leq \min\{c(\tilde{A}) : \tilde{A} \text{ es un corte de } G\}. \tag{5.6}$$

Demostración. De la Proposición 5.4 se obtiene

$$o_o = \sum_{i \in \tilde{V},\, j \in \tilde{V}^c :\, (i,j) \in A} f_{ij} - \sum_{i \in \tilde{V}^c,\, j \in \tilde{V} :\, (i,j) \in A} f_{ij},$$

donde $\tilde{V}$ es el subconjunto de vértices que genera el corte $\tilde{A}$. Por otro lado, como f es un flujo factible, debe cumplirse que $0 \leq f_{ij} \leq c_{ij}$, lo que junto a la definición capacidad de un corte nos lleva a las desigualdades:

$$\begin{aligned} o_o &\leq \sum_{i \in \tilde{V},\, j \in \tilde{V}^c :\, (i,j) \in A} c_{ij} - \sum_{i \in \tilde{V}^c,\, j \in \tilde{V} :\, (i,j) \in A} f_{ij} \\ &\leq \sum_{i \in \tilde{V},\, j \in \tilde{V}^c :\, (i,j) \in A} c_{ij} = c(\tilde{A}), \end{aligned}$$

lo que prueba la primera parte de la proposición.

Finalmente, la desigualdad (5.6) es consecuencia directa de lo anterior. □

La proposición anterior nos entrega una cota superior para el valor óptimo o_o^* del problema de flujo máximo. En el resto de la sección nos dedicaremos a demostrar que esta cota superior es, en efecto, igual a este valor.

Sea C una cadena de G. Sabemos de la Definición 2.14 de cadena que, si $a = (i, j)$ es uno de los arcos que forman esta cadena, entonces se tiene que $(i, j) \in A$ o bien $(j, i) \in A$. En el primer caso hemos dicho que a es un *arco hacia adelante*, mientras en el segundo caso hemos dicho que a es un *arco hacia atrás* o *en reversa*. Con esto en mente podemos definir lo siguiente:

Definición 5.7. Sean $G = (V, A)$ una red con capacidad c y flujo factible f. Un *camino aumentador* (o *cadena* aumentadora) es una cadena C entre el vértice origen o y el vértice destino d tal que, para todo arco $a_k = (i_k, i_{k+1})$ en C se satisface:

- Si a_k es una arco hacia adelante, entonces $\Delta_k = c_{i_k i_{k+1}} - f_{i_k i_{k+1}} > 0$.
- Si a_k es una arco hacia atrás, entonces $\Delta_k = f_{i_k i_{k+1}} > 0$.

Si C es un camino aumentador formado por los arcos a_k, con $k = 0, ..., l$, podemos definir la cantidad $\Delta = \min_k \Delta_k > 0$. Luego, cada arco a_k que satisface $\Delta_k = \Delta$ es llamado *cuello de botella* de C.

Ejemplo 5.2. En la red de la Figura 5.4 los valores f y c que acompañan a los arcos, representan el flujo y capacidad, respectivamente, de cada arco. Hemos dibujado además en púrpura a los arcos que forman el camino aumentador $C = (1, 3, 4, 6)$. Para este camino aumentador, el valor Δ es igual a 1 y los cuellos de botellas son los arcos

$(1,3)$ y $(4,6)$. Notemos además que este camino aumentador no es un camino, pues el arco $(3,4)$ es un arco hacia atrás.

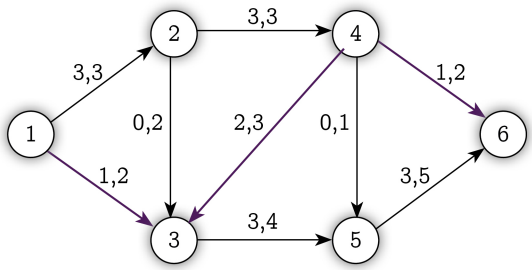


FIGURA 5.4. Ejemplo de camino aumentador y cuello de botella

Notemos que si existe un camino aumentador C para una red G con flujo factible f, entonces podemos aumentar el valor del flujo total enviado desde o a d en un valor Δ, construyendo un nuevo flujo factible f' asociado a G como sigue:

Para todo arco a_k que no está en C el flujo no cambia, es decir, $f'_{a_k} = f_{a_k}$. Para cada arco a_k de C calculamos el nuevo flujo f'_{a_k} de la siguiente manera:

(5.7a) Si a_k es una arco hacia adelante, entonces $f'_{a_k} = f_{a_k} + \Delta$,

(5.7b) y si a_k es una arco hacia atrás, entonces $f'_{a_k} = f_{a_k} - \Delta$.

En efecto, los cambios producidos por (5.7) respetan las ecuaciones de flujo (5.2) (se deja al lector verificar esto en detalle) y, debido a la definición de Δ, el nuevo flujo f' también respeta las cotas de capacidad (5.1), por lo tanto, el flujo f' satisface (5.4), es decir, es un flujo factible para G. Por otro lado, si a_0 es el primer arco del camino aumentador C, entonces, si este es un arco hacia adelante, es decir, $a_0 = (o,i)$ para cierto $i \in V$, entonces $f'_{oi} = f_{oi} + \Delta$, en caso contrario, es decir, $a_0 = (i,o)$ para cierto $i \in V$, entonces $f'_{io} = f_{io} - \Delta$. En ambos casos, el flujo total que oferta el vértice o, dado por $o_o = \sum_{(o,j)\in A} f_{oj} - \sum_{(k,o)\in A} f_{ko}$, aumentó en un valor Δ.

Ilustremos este procedimiento a partir del camino aumentador de la Figura 5.4. Para los arcos cuellos de botella $(1,3)$ y $(4,6)$, ambos arcos hacia adelante, su flujo aumenta en 1, obteniendo un flujo igual a 2, mientras que para el arco hacia atrás $(4,3)$ su flujo se reduce en 1, obteniendo un flujo igual a 1. El flujo del resto de los arcos no cambia. El nuevo flujo f' se muestra en la Figura 5.5.

Dado que la existencia de caminos aumentadores nos permite siempre encontrar un nuevo flujo factible que aumenta el flujo total enviado desde el vértice origen o (es decir, aumenta o_o), es natural pensar que en caso de no existir flujos aumentadores

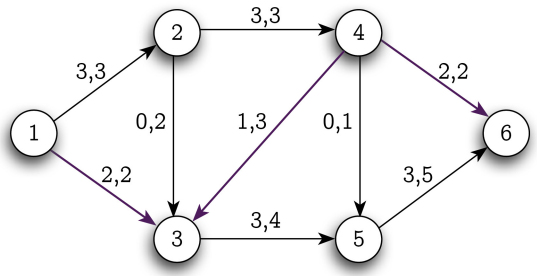


FIGURA 5.5. Calculo del nuevo flujo f'

nos encontramos frente a un flujo que maximiza esta cantidad, es decir, frente a una solución del problema de flujo máximo (5.3)-(5.4). Esta intuición es formalmente demostrada en el siguiente teorema.

Teorema 5.8. *Si para una red $G = (V, N)$ y un flujo factible f asociado, no existen caminos aumentadores, entonces f resuelve el problema de flujo máximo* (5.3)-(5.4).

Demostración. Sean G una red, f un flujo factible asociado a G y o_o el flujo enviado desde o para el flujo f, para los cuales no existen caminos aumentadores. Gracias a la Proposición 5.6 nos basta exhibir un corte $\tilde{A}$ de G, cuya capacidad sea igual a o_o. Para este propósito, definiremos el siguiente procedimiento de etiquetamiento de los vértices en V:

- Primero, etiquetamos (o marcamos) el vértice origen o.
- Luego, recorremos los arcos de G y etiquetamos sus vértices como sigue:
- Si para un arco $a_{ij} = (i, j) \in A$, i está etiquetado y j no está etiquetado, entonces, si $f_{ij} < c_{ij}$, etiquetamos j.
- Si para un arco $a_{ij} = (i, j) \in A$, j está etiquetado e i no está etiquetado, entonces, si $f_{ij} > 0$, etiquetamos i.
- Repetimos los dos últimos pasos hasta etiquetar tantos vértices de V como sea posible.

Se muestra fácilmente que este proceso etiquetará el vértice destino d si y solamente si existe un camino aumentador. Luego, según nuestros supuestos, d no es etiquetado al aplicar este procedimiento. Así, si denotamos por $\tilde{V}$ al subconjunto de vértices de V que han sido etiquetados, vemos que éste define un corte $\tilde{A}$ compuesto por todos los arcos $a_{ij} = (i, j) \in A$ que satisfacen que $i \in \tilde{V}$ y $j \in \tilde{V}^c$, o bien, $j \in \tilde{V}$ e $i \in \tilde{V}^c$. Del proceso de etiquetado deducimos que para el primer caso necesariamente se tiene que $f_{ij} = c_{ij}$, mientras que para el segundo se cumple que $f_{ij} = 0$. Con esto, de la Proposición 5.4 se deduce

$$o_o = \sum_{i \in \tilde{V},\, j \in \tilde{V}^c :\, (i,j) \in A} f_{ij} - \sum_{i \in \tilde{V}^c,\, j \in \tilde{V} :\, (i,j) \in A} f_{ij} = \sum_{i \in \tilde{V},\, j \in \tilde{V}^c :\, (i,j) \in A} c_{ij} = c(\tilde{A}).$$

Hemos encontrado entonces un corte $\tilde{A}$ de G con capacidad igual a o_o y, por lo tanto se ha demostrado el teorema. □

La demostración del Teorema 5.8 nos ha revelado que cuando el flujo f es una solución del problema de flujo máximo (5.3)-(5.4) (es decir, para el cual no existen caminos aumentadores), podemos siempre encontrar un corte $\tilde{A}$ tal que $o_o = c(\tilde{A})$. Lo que implica la deseada igualdad en (5.6). Este resultado, conocido como *teorema de flujo máximo–corte mínimo* (o *max-flow, min-cut* en inglés), se debe a los matemáticos Lester Randolph Ford Jr y Delbert Ray Fulkerson (ver **[10]**) y es uno de los resultados más importantes en teoría de grafos.

Teorema 5.9. *Para una red $G = (V, A)$ siempre se tiene que la máxima cantidad de flujo que se puede enviar desde el vértice origen o al vértice destino d, es igual a la capacidad mínima entre todos los cortes de G. Es decir:*

$$o_o^* = \min\{c(\tilde{A}) : \tilde{A} \text{ es un corte de } G\}, \tag{5.8}$$

donde o_o^* es el valor óptimal del problema de flujo máximo en G (i.e. (5.3)-(5.4)*).*

Ejercicio 5.3. Calcule el problema dual[6] del problema de flujo máximo (5.3)-(5.4) e interprete el Teorema 5.9, usando la teoría de dualidad para la optimización lineal.

5.2.1 Aplicaciones

El problema de flujo máximo nos permite modelar cualquier problema donde se considera sólo una fuente ofertante y un centro demandante de un cierto recurso y donde dicha fuente desea enviar la mayor cantidad del recurso (que se representará por los flujos) a través de una red que une estos dos puntos. Por ejemplo, podemos considerar una simplificación del problema de transporte de gas en una ciudad (ver Figura 5.2), donde existe sólo una empresa productora de gas y una casa demandante de éste. Para que este problema tenga sentido práctico, podemos pensar que dicho vértice demandante es una ciudad con una alta necesidad de gas como Santiago y el vértice ofertante representa a un único productor de gas para dicha ciudad como, en el caso de Santiago, Argentina. Finalmente, la red representa los distintos gasoductos que unen el país trasandino con Santiago.

Otro ejemplo del mismo tipo, es el modelamiento del flujo vehicular en la hora *punta* de la mañana en Santiago. Por ejemplo, podemos caracterizar el vértice demandante como el centro de Santiago, donde se concentra la mayor cantidad de empleos, y el vértice ofertante como una comuna con una gran población que en su mayoría trabaja en el centro de Santiago, por ejemplo, La Florida (Ver Figura 5.6). Luego, las principales calles que permiten ir desde La Florida a Santiago Centro se describirían como arcos con sus distintas capacidades, pasando por vértices de paso que permitan tomar más de una ruta entre el origen y destino.

Así, este simple problema permitiría realizar una estimación del flujo vehicular máximo desde La Florida hacia el centro de Santiago, así como la distribución de

[6]Ver monografía "Optimización Lineal: una mirada introductoria".

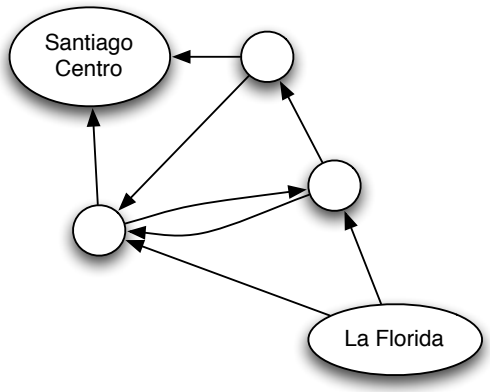


FIGURA 5.6. Dibujo flujo vehicular Santiago

éste entre las principales calles y avenidas entre estas dos comunas, considerando las capacidades que estas vías tienen. Esto podría, por ejemplo, permitir al Ministerio de Vivienda y Urbanismo regular las políticas de construcción, restringiendo el negocio inmobiliario en las zonas aledañas a las calles donde el flujo vehicular real es cercano al dado por flujo máximo y, al mismo tiempo, incentivándolo en las zonas donde no se da esta situación.

Ejercicio 5.4. Construya un grafo donde el vértice origen sea la comuna o distrito donde se encuentra su casa y el vértice destino sea la comuna o distrito donde se encuentra su lugar de trabajo o su universidad, y los arcos y vértices intermedios representen las distintas formas de llegar del origen al destino (por ejemplo, en el caso de la Figura 5.6, podemos considerar tomar un bus o un auto vía Vicuña Mackena, tomar metro línea 5, etc., y las distintas combinaciones entre estas alternativas). Luego, estime las capacidades máximas de estas rutas, en términos de personas transportadas por hora, tomando en cuenta el tamaño de las rutas, las frecuencias y capacidades de los distintos medios de transporte, etc., ¿qué representa, en este caso, la solución el problema de flujo máximo obtenido?

5.2.2 Algoritmo de Ford-Fulkerson

Las definiciones y resultados de la sección anterior nos permiten elaborar un algoritmo que resuelve el problema de flujo máximo, aprovechando la estructura de grafos del problema y no la estructura de optimización lineal del problema (5.3)-(5.4) (ver Nota 4 al pie de la página 89). Este algoritmo, introducido a mediados de los años 50 por Ford y Fulkerson, es el primero y uno de los más conocidos para resolver este problema.

La idea fundamental del algoritmo es la utilización de caminos aumentadores. Dejando por el momento de lado el proceso para encontrar dicho camino aumentador, podemos esquematizar el algoritmo en los siguientes pasos:

- Se comienza con un flujo factible f. Por ejemplo, el flujo nulo: $f_{ij} = 0$, para todo $(i, j) \in A$. Fijar el contador $k = 0$.
- Se encuentra un camino aumentador C^k para la red G y el flujo f^k.
- Luego, se calcula f^{k+1}, usando la actualización (5.7).
- Repitiendo los dos últimos pasos se genera una secuencia de flujos factibles $\{f^k\}$, para los cuales el flujo total enviado desde el vértice origen va en aumento (i.e. $o_o^{k+1} \geq o_o^k$).

Este algoritmo funciona (es decir, nos permite obtener un flujo f^* solución del problema de flujo máximo) debido a que: 1) si el algoritmo no encuentra un camino aumentador en la iteración k, entonces, gracias al Teorema 5.8, el actual flujo f^k es solución del problema de flujo máximo, y 2) como consecuencia del Teorema 5.9 y del proceso (5.7) usado para actualizar los flujos, se necesitan a lo más $\min\{c(\tilde{A}) : \tilde{A}$ es un corte de $G\}$ iteraciones[7].

Abordemos ahora la obtención de un camino aumentador para una red G y flujo factible asociado f. Para este propósito se trabajará con un grafo auxiliar $G' = (V', A')$, llamado *grafo residual*, con la propiedad que los caminos entre o y d en este grafo G' coincidirán con un camino aumentador para la red G y flujo f. Por lo tanto, encontrar un camino aumentado se reducirá a encontrar un camino de o y d, lo cual puede ser realizado de varias maneras y es eficiente desde el punto de vista de su complejidad computacional (por ejemplo, utilizando lo visto en el Capítulo 2).

El algoritmo para construir el grafo residual G', a partir de la red $G = (V, A)$ y flujo factible asociado f, es descrito a continuación:

- Los vértices V' de G' serán los mismos que los de G, es decir, $V' = V$.
- Comenzaremos sin arcos en G', es decir, $A' = \emptyset$. Luego, recorreremos todos los arcos $a_{ij} = (i, j)$ de G e incorporaremos arcos a G' según los criterios:
- Si $f_{ij} < c_{ij}$, entonces el arco (i, j) se incorpora a A' con capacidad $c'_{ij} = f_{ij} - c_{ij}$.
- Si $f_{ij} > 0$, entonces el arco (j, i) se incorpora a A' con capacidad $c'_{ij} = f_{ij}$.

Por construcción del grafo residual G', un camino aumentador C en G para el flujo f, corresponde a un camino C' entre o y d en G', y que los valores Δ_k, dados en la Definición 5.7, corresponden a las capacidades de los arcos que forman tal camino C'. Así, el valor $\Delta = \min_k \Delta_k$ también determina el "cuello de botella" para el flujo que se puede enviar desde o a d en el grafo residual G'.

[7]En realidad, esto se puede asegurar sólo si las capacidades c_{ij} son números enteros, pues cada actualización del flujo debe al menos aumentar o_o en una unidad de flujo. En general, se ve que deberían haber, a lo más, $\min\{c(\tilde{A}) : \tilde{A}$ es un corte de $G\}/\min\{c_{ij} : (i, j) \in A, c_{ij} \neq 0\}$ iteraciones.

Observemos también que los criterios, dados en el tercer y cuarto ítem, para incorporar arcos a G' no son excluyentes, por lo que un arco a_{ij} en A, puede generar dos arcos en A' entre los mismos vértices, pero en sentidos opuestos y con capacidades que no serán necesariamente las mismas. En tal caso, este arco $a_{ij} = (i, j)$ puede ser usado por un camino aumentador tanto hacia adelante, como hacia atrás.

Ejemplo 5.3. En la Figura 5.7 aparece al lado izquierdo una red $G = (V, A)$, donde en sus arcos se especifica flujo y capacidad, respectivamente, y al lado derecho el grafo residual $G' = (V', A')$, obtenido por el algoritmo.

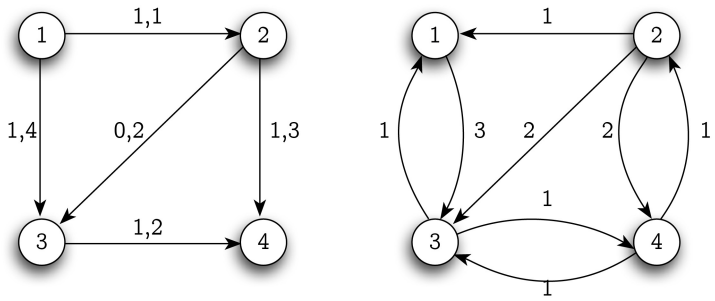


FIGURA 5.7. Ejemplo grafo residual

Ejercicio 5.5. Encuentre el grafo residual de la red de la Figura 5.8, donde los valores que aparecen en cada arco corresponden a flujo/capacidad y el vértice de origen es 1 y el destino es 5.

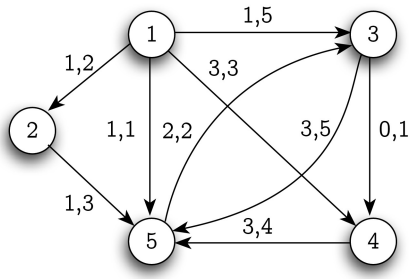


FIGURA 5.8. Calcule el grafo residual

Este algoritmo se escribe en pseudo-lenguaje de programación (ver Sección 1.1.1) como sigue:

Algoritmo 5.1 Construcción del grafo residual $G' = (V', A')$

Entrada: Red $G = (V, A)$ (con capacidad c) y flujo factible f
1: $A' \leftarrow \emptyset$ /* En un comienzo G' no tiene arcos */
2: **para** $a_{ij} = (i, j) \in A$
3: **si** $f_{ij} < c_{ij}$ **entonces**
4: $A' \leftarrow A' \cup \{a_{ij}\}$
5: $c'_{ij} = c_{ij} - f_{ij}$ /* Cálculo de la capacidad del arco a_{ij} en G' */
6: **fin**
7: **si** $f_{ij} > 0$ **entonces**
8: $A' \leftarrow A' \cup \{(j, i)\}$ /* Se adiciona el arco en el sentido inverso */
9: $c'_{ij} = f_{ij}$
10: **fin**
11: **fin**
Salida: $G' = (V, A')$(con capacidad c') /* G y G' tienen los mismos vértices */

Antes de mostrar con un ejemplo el funcionamiento del algoritmo de Ford-Fulkerson, escribámoslo usando pseudo-lenguaje de programación.

Algoritmo 5.2 Algoritmo de Ford-Fulkerson

Entrada: $G = (V, A)$ (con capacidad c)
1: $f_a \leftarrow 0$ para todo $a \in A$ /* Inicializar el flujo */
2: $G' \leftarrow$ Grafo_residual(G, f)
3: $o^*_o \leftarrow 0$ /* Inicializar el valor óptimal del flujo máximo */
4: **mientras** Exista un camino $C' = (a'_k)$ entre o y d en G'
5: $\Delta \leftarrow \min_k c'(a'_k)$ /* Δ corresponde al máximo flujo que se puede enviar desde o a d, usando el camino C' en G' */
6: **para** $a'_k = (i_k, i_{k+1}) \in C'$
7: **si** $a'_k \in A$ **entonces**
8: $f_{a'_k} \leftarrow f_{a'_k} + \Delta$ /* a'_k es un arco hacia adelante en el camino aumentador C correspondiente a C' */
9: **sino**
10: $f_{i_{k+1}i_k} \leftarrow f_{i_{k+1}i_k} - \Delta$ /* a'_k es un arco hacia atrás en C */
11: **fin**
12: **fin**
13: $G' \leftarrow$ Grafo_residual(G, f)
14: $o^*_o \leftarrow o^*_o + \Delta$
15: **fin**
Salida: f flujo óptimal, o^*_o valor óptimal

Observación 5.2. Notemos que, dado que el flujo inicial f es nulo, el primer grafo residual G' calculado en la línea 2 del algoritmo nos entrega el mismo grafo G.

Observación 5.3. La sentencia "**si**" definida entre las líneas 7 y 11, corresponden a la actualización de flujo introducida en (5.7).

Ejemplo 5.4. Ilustremos el algoritmo de Ford-Fulkerson con el grafo G de la Figura 5.9. Asignemos a cada arco una capacidad igual a 1 y encontremos el flujo máximo del vértice 1 al 4.

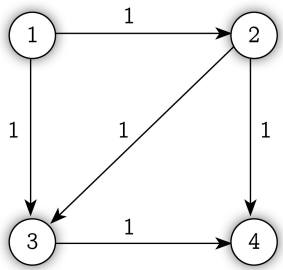


FIGURA 5.9. Ejemplo algoritmo de Ford-Fulkerson

Se inicializa el flujo f en cero. El grafo residual G' calculado en la línea 2 es igual al grafo original (ver Observación 5.2), por lo que se omite su gráfico.

Tomemos el camino $C' = (1, 2, 3, 4)$ en G'. En la línea 5 del algoritmo de Ford-Fulkerson se obtiene que el valor de Δ es 1. Luego, después de la asignación de flujo de las líneas 6 a 12, el flujo f queda:

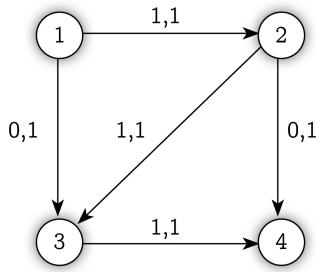


FIGURA 5.10. Asignación de flujo en la primera iteración

donde en los arcos se especifica flujo y capacidad, respectivamente.

En la línea 13 del algoritmo de Ford-Fulkerson se utiliza el Algoritmo 5.1 para obtener el grafo residual asociado a la red y flujo anterior, obteniendo el grafo de la Figura 5.11 (donde en cada arco se ha especificado la capacidad c' de los arcos en A').

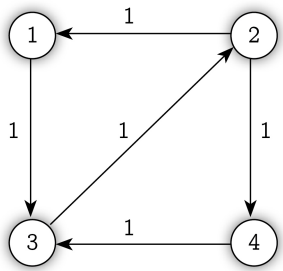


FIGURA 5.11. Asignación de flujo en la segunda iteración

Hemos terminado la primera iteración del ciclo "**mientras**" definido entre las líneas 4 y 15. En una segunda iteración de este ciclo notamos que el único camino posible entre o y d en G' está dado por $C' = (1, 3, 2, 4)$. El valor de Δ, calculado en la línea 5, es igual a 1. Luego, la asignación de flujo de las líneas 6 a 12 nos entregan un flujo f como aparece en la Figura 5.13.

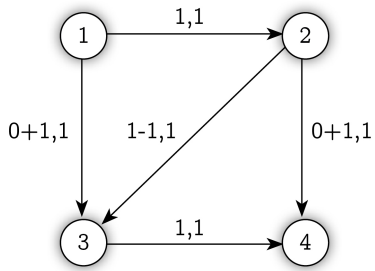


FIGURA 5.12. Flujo óptimo

Para este grafo no existen caminos aumentadores. Esto se puede ver calculando una vez más el grafo residual G' y notando que no existen caminos entre o y d en G' (esto se deja como ejercicio al lector). Por lo tanto, el flujo dado en la Figura 5.13 representa la solución del problema de flujo máximo para el grafo G dado en la Figura 5.9, cuyo valor óptimal es 2.

Otra manera de verificar que este flujo es efectivamente óptimo es a través del Teorema 5.9 de flujo máximo-corte mínimo. En un grafo de tamaño pequeño, como lo es G, podemos calcular por inspección la capacidad de corte mínima, la que en este caso resulta ser igual a 2 (posibles cortes son $\{(1,2),(1,3)\}$ o $\{(2,4),(3,4)\}$). Así, como el flujo de la Figura 5.13 envía desde el vértice origen o un flujo total igual a $o_o = 2$, esté necesariamente debe ser óptimo.

Ejercicio 5.6. Resuelva, usando el algoritmo de Ford-Fulkerson, el problema planteado en el Ejercicio 5.4.

Ejercicio 5.7. Encontrar el flujo máximo para el grafo de la Figura 5.13 (en los arcos aparecen las capacidades).

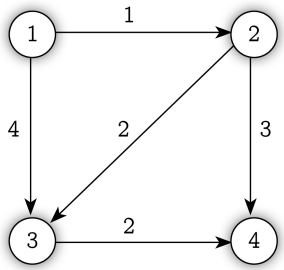


FIGURA 5.13. Encuentre el flujo máximo

Observación 5.4. Una ligera variante del problema de flujo máximo consiste en considerar, además de capacidades superiores, capacidades inferiores en los arcos que puedan tomar valores estrictamente positivos. Modificando ligeramente las definiciones dadas en este capítulo se puede reproducir la teoría desarrollada para obtener los mismos resultados. Más aún, el algoritmo de Ford-Fulkerson, introducido en esta sección, es establecido con mínimas diferencias. La única complicación importante, asociada a esta variante, es la determinación de un flujo factible inicial asociado al grafo en el algoritmo de Ford-Fulkerson. (Recordemos que en esta sección simplemente usamos el flujo nulo). Un algoritmo para la determinación de un flujo factible inicial para esta variante del problema escapa de los objetivos del curso, pero puede ser consultado, por ejemplo, en el libro **[16]**.

5.2.3 Breve discusión sobre la complejidad del algoritmo de Ford-Fulkerson

El algoritmo que hemos presentado en este capítulo (que, como hemos mencionado anteriormente, fue desarrollado por Ford y Fulkerson en el artículo **[10]** a mediados de los años 50) puede ser muy ineficiente en términos de complejidad computacional.

Para ver esto, consideraremos (a lo largo de toda esta sección) una red $G = (V, A)$, con capacidad c, y denotaremos por n y m las cardinalidades de V y A, respectivamente. También denotaremos por $c_{\max}$ la capacidad máxima de los arcos en A, i.e. $c_{\max} = \max_{(i,j)\in A} c_{ij}$. Para simplicidad del análisis supondremos que las capacidades c_{ij} de G son números enteros (ver nota número 7 al pie de la página 97).

Proposición 5.10. *El Algoritmo 5.2 de Ford-Fulkerson encuentra una solución del problema de flujo máximo* (5.3)-(5.4) *en a lo más* $O(mnc_{max})$ *operaciones.*

Demostración. Primero, notemos que el Algoritmo 5.1, que nos permite construir el grafo residual G', realiza a lo más $O(m)$ operaciones[8] (pues recorre sólo una vez el conjunto de arcos A), entonces las asignaciones de las líneas 1 y 2 realizan a lo más $O(m)$ operaciones, mientras que en la línea 3 se realiza sólo una (que corresponde a una asignación). Así, la complejidad total del algoritmo de Ford-Fulkerson dependerá fuertemente de la complejidad del ciclo "**mientras**" definido entre las líneas 4 y 15 de este algoritmo.

Hemos establecido en la página 97 que el algoritmo de Ford-Fulkerson necesita a lo más $\min\{c(\tilde{A}) : \tilde{A}$ es un corte de $G\}$ iteraciones de este ciclo "**mientras**" para encontrar un flujo máximo. Notemos que esta cantidad se puede acotar por $nc_{\max}$. En efecto, para todo corte $\tilde{A}$ de G y para todo arco $a \in \tilde{A}$ se tiene que $c_a \leq c_{\max}$. Entonces, si denotamos por $\tilde{V}$ el subconjunto de vértices que genera el corte $\tilde{A}$, se tiene que:

$$c(\tilde{A}) = \sum_{i\in\tilde{V},\, j\in\tilde{V}^c:\,(i,j)\in A} c_{ij} \leq |\tilde{V}|c_{\max} \leq nc_{\max}.$$

Procedamos ahora a calcular la complejidad del ciclo "**mientras**" de las líneas 4 a 15. Dentro de cada ciclo se realiza: 1) en la línea 5 a lo más $|C'|$ operaciones, 2) en el ciclo "**para**" de las líneas 6 a 12 a lo más $|C'|$ operaciones también, y 3) en las líneas 13 y 14 a lo más m y 1 operaciones, respectivamente. Por lo tanto, al interior de este ciclo "**mientras**" se realizan a lo más $2|C'| + m + 1 = O(m)$ operaciones (pues claramente $|C'| \leq m$). De está forma, el algoritmo de Ford-Fulkerson realiza a lo más $O(m) + nc_{\max}O(m) = O(mnc_{\max})$ operaciones (hemos usado la primera regla del Teorema 1.5). □

Como el valor $c_{\max}$ no depende del tamaño del problema (es decir, de n y m), este resultado no es del todo satisfactorio en términos de complejidad computacional. En efecto, si $c_{\max}$ fuese para todas las redes que deseamos resolver una constante o al menos un valor acotado superiormente por una constante (i.e. $c_{\max} = O(1)$), la complejidad del algoritmo es simplemente $O(mn)$ lo que es considerado para este problema como algo bastante bueno, pero nada impide que $c_{\max}$ aumente su valor a medida que n y/o m aumentan su valor. Por ejemplo, $c_{\max}$ podría ser una función del tipo exponencial de n y m, digamos $c_{\max} = 2^{n+m}$, en tal caso, el algoritmo de Ford-Fulkerson tendría una complejidad $O(mnc_{\max}) = O(mn2^{n+m}) = O(4^{n+m})$ (hemos

[8]Consultar la Definición 1.3 para la notación *gran o*: $O(\cdot)$

usado la segunda regla del Teorema 1.5), es decir exponencial, lo que hace inviable su resolución para valores altos de n o m.

Ésta "mala" eficiencia del algoritmo de Ford-Fulkerson depende fuertemente de la manera que elegimos el camino aumentador. Por esto, se han desarrollado variantes del algoritmo aquí presentado que mejoran notablemente la eficiencia de éste, a tal punto de transformarlo en un algoritmo *polinomial* (ver Tabla 1.4 en el Capítulo 1). Una pequeña variante es la propuesta por J. Edmonds y R.M. Karp en su artículo [**8**] del año 1972, donde el camino C' entre o y d en el grafo residual G' es elegido usando el *camino de costo mínimo* (o el camino más corto), interpretando las capacidades como costos. Se demuestra que esta variante necesita a lo más $\frac{1}{2}mn$ caminos aumentadores para encontrar la solución del problema de flujo máximo, obteniendo así un algoritmo que resuelve el problema en a lo más $O(nm^2)$ iteraciones. El lector puede consultar este resultado en el libro de A. Gibbons [**13**, Teorema 4.4].

Existen también otros algoritmos que resuelven el problema de flujo máximo basándose en la idea de caminos aumentadores de Ford y Fulkerson. Si bien estas variantes del algoritmo escapan a los objetivos de este texto, podemos mencionar, por ejemplo, el artículo [**7**] de Dinits, en el año 1970, donde se desarrolla una variante que realiza a lo más $O(n^2m)$ operaciones para resolver el problema de flujo máximo y el artículo [**11**] de Gabow, del año 1985, que muestra otra variante del algoritmo que necesita a lo más $O(nm\log(c_{\max}))$ operaciones para estos efectos.

5.3 Ejercicios

1. Usted posee celulares en las tres compañías de telefonía móvil del país: Caro, Ntel y Radiofónica. Su plan en la compañía Caro le permite llamar 100 minutos al mes para teléfonos Caro, 50 minutos para teléfonos Ntel y 10 minutos para teléfonos Radiofónica. El de la compañía Ntel le permite llamar 100 minutos al mes para teléfonos Caro y 150 minutos para teléfonos Ntel. Finalmente, su plan en Radiofónica le permite llamar 300 minutos sólo para teléfonos de la misma compañía. Plantee un problema de flujo máximo que le permite calcular cuánto es la máxima cantidad de minutos que puede utilizar al mes.

   Indicación: Naturalmente aparecerán 3 vértices destino. Para escribir el problema, usando un sólo vértice destino, introduzca un vértice destino artificial y 3 arcos entre los vértices destino originales y éste nuevo vértice destino, todos con capacidades infinitas (o sumamente grande).
2. Sea $G = (V, A)$ una red y $A' \subseteq A$ un subconjunto de arcos tal que, para la red $G \setminus A' = (V, A \setminus A')$ (con las mismas capacidades de G) no existe un flujo factible estrictamente positivo. Pruebe que necesariamente A' contiene un corte de G.
3. Sean $\tilde{A}$ y $\tilde{A}'$ cortes de capacidad mínima para la red G (es decir, ambos resuelven el problema $\min\{c(\tilde{A}) : \tilde{A} \text{ es un corte de } G\}$), y $\tilde{V}$ y $\tilde{V}'$ los subconjuntos de vértices que generan. Pruebe que los cortes que generan los vértices $\tilde{V} \cup \tilde{V}'$ y $\tilde{V} \cap \tilde{V}'$ también son de capacidad mínima.
4. Considere el problema de flujo máximo para la red de la Figura 5.14 (donde 1 es el vértice origen y 4 el vértice destino). La solución de este problema se obtiene cuando los flujos son $f_{1,2} = 2$, $f_{1,3} = 2$, $f_{2,3} = 1$, $f_{3,2} = 0$, $f_{2,4} = 1$ y $f_{3,4} = 3$. Verifique esto encontrando un corte de capacidad mínima asociado a este flujo.

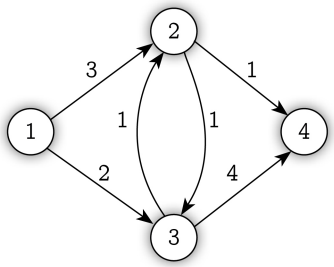


FIGURA 5.14. Los valores que aparecen en los arcos indican sus capacidades

5. Resuelva ahora el problema de flujo máximo para la red de la Figura 5.14, usando el algoritmo de Ford-Fulkerson. Para esto, comience con el flujo nulo y precise en cada iteración del algoritmo el grafo residual, el flujo asignado a cada arco y la cadena aumentadora elegida.

6. Agreguemos ahora a la red de la Figura 5.14 un nuevo vértice de paso, denotado por 5 y los arcos $(1, 5)$, $(3, 5)$ y $(5, 4)$. A partir de la solución dada en el Ejercicio 5, encuentre por inspección la nueva solución del problema. Demuestre, usando el teorema de flujo máximo – corte mínimo, que el flujo encontrado por usted es efectivamente una solución.
7. Resuelva el problema de flujo en redes planteado en el Ejercicio 1, usando el algoritmo de Ford-Fulkerson, ¿cuánto es la cantidad máxima de minutos que puede hablar al mes? Responda la misma pregunta si decide eliminar su plan de la compañía Ntel.
8. Considere la red de oleoductos de la Figura 5.15 que conecta una planta de extracción petrolera con una refinería, donde en cada arco se especifica la capacidad máxima de cada tramo del oleoducto (medida en miles de metros cúbicos por hora: $1000 * m^3/hr$).

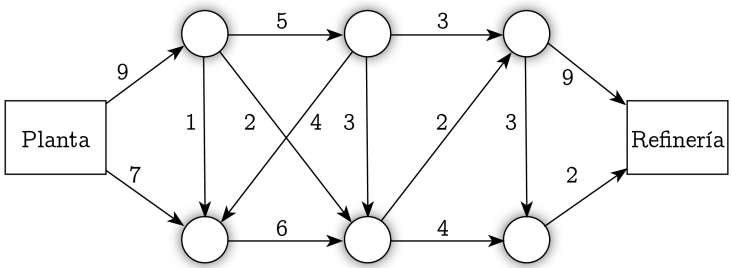


FIGURA 5.15. Red de oleoductos con sus capacidades

Calcule la mayor cantidad que se puede enviar de la plataforma a la refinería y determine qué tramos de oleoductos deben utilizarse.

9. Encuentre el flujo máximo, usando el algoritmo de Ford-Fulkerson, de la siguiente red (en los arcos aparecen sus capacidades):

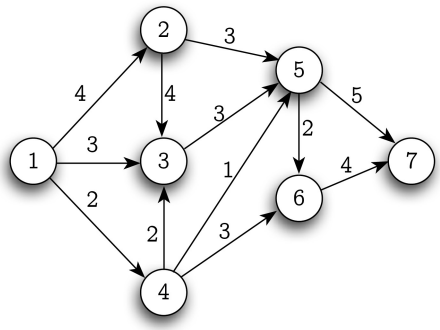


FIGURA 5.16. Encuentre el flujo máximo

10. Considere una red G y un flujo máximo f. Describa un algoritmo que permita encontrar un arco a de G tal que, si se aumenta la capacidad c_a de a, se puede también aumentar el valor del flujo máximo f. Este arco a, ¿siempre debe existir? Usando lo aprendido en la Sección 1.3, calcule la complejidad del algoritmo propuesto.

Capítulo 6: Ciclos

6.1 Ciclos Eulerianos

Durante el siglo XIII, en la ciudad de Königsberg, Prusia (actualmente Kaliningrado, Rusia), el río Pregel cruzaba la ciudad dejando dos pequeñas islas, que se conectan a tierra a través de siete puentes (ver Figura 6.1). En ese tiempo, se cuestionaba si era posible o no recorrer cada uno de los puentes y volver al punto inicial, pasando sobre cada puente exactamente una vez.

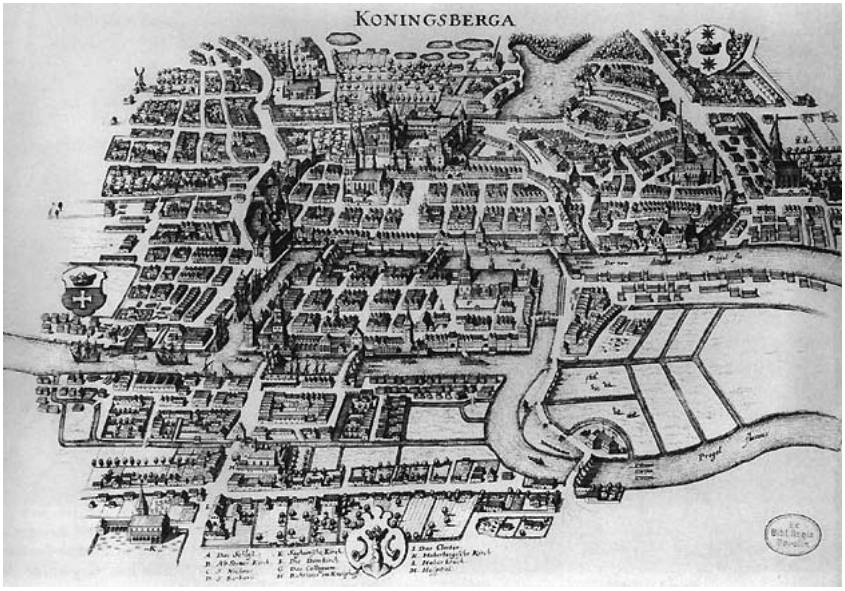


FIGURA 6.1. Los puentes de Königsberg en 1736

Modelemos este problema como si fuera un grafo. Representemos cada pedazo de tierra como un vértice y cada puente como un arista que une dos vértices. Notemos que este grafo es en realidad un multigrafo, pues contiene dos aristas distintas conectando un mismo par de vértices. Sin embargo, esto podemos solucionarlo agregando un vértice artificial en cada arista, obteniendo el grafo simple de la Figura 6.2.

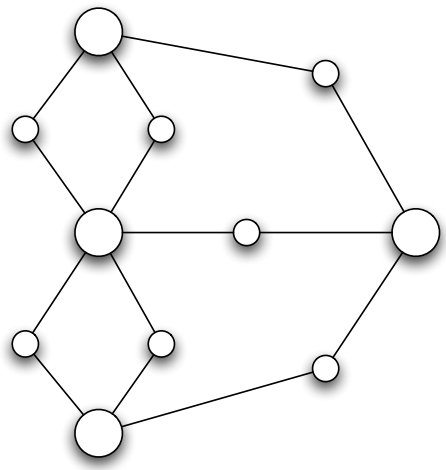


FIGURA 6.2. Representación en grafo de los puentes de Königsberg

Por lo tanto, si deseamos buscar un camino que vuelva al punto de partida que recorra cada uno de los puentes una sola vez, en nuestro grafo lo que deseamos encontrar es un *ciclo* que recorra cada una de las aristas del grafo exactamente una vez, es decir, un *ciclo Euleriano*.

Definición 6.1. Dado un grafo G, un ciclo Euleriano es un ciclo que visita cada una de las aristas del grafo exactamente una vez.

Posiblemente usted ya ha intentado construir un ciclo con esta características en la Figura 6.1 y seguramente no ha podido encontrar una solución. Esto se debe a que tal ciclo no existe para este grafo. Efectivamente, podemos clasificar aquellos grafos para los cuales existe un ciclo Euleriano y a estos grafos los llamaremos *grafos Eulerianos*.

Fue Leonhard Euler en 1736 quien logró resolver el problema de los puentes de Königsberg, obteniendo una caracterización de aquellos grafos que son Eulerianos (y por esto, recibieron el nombre de grafos **Eulerianos**). Esta condición es muy simple de verificar y sólo requiere contar el número de aristas incidentes a cada vértice.

Teorema 6.2 (Euler, 1736)**.** *Un grafo conexo es Euleriano si y sólo si cada vértice tiene grado par.*

Demostremos el teorema anterior. Una de las implicancias es sencilla: cada vez que entro a un vértice por una arista, necesito otra para salir de ese vértice, por lo que, si existe un ciclo Euleriano, entonces los vértices tienen que tener grado par.

La demostración de la otra implicancia se basa en el siguiente argumento algorítmico: supongamos que tenemos un ciclo, pero que no necesariamente visita todos los arcos. Si uno "remueve" los arcos de este ciclo, quedan subgrafos conexos que también poseen en cada vértice grado par, por lo que podemos construir nuevos ciclos en estos subgrafos e "insertarlos" al ciclo anterior, obteniendo así un ciclo más largo. Repitiendo este argumento podemos construir un ciclo que visite todas las aristas, es decir, un ciclo Euleriano.

Ejercicio 6.1.

1. Escriba el algoritmo anterior como pseudo-código.
2. Formalice la demostración anterior, usando argumentos de inducción.

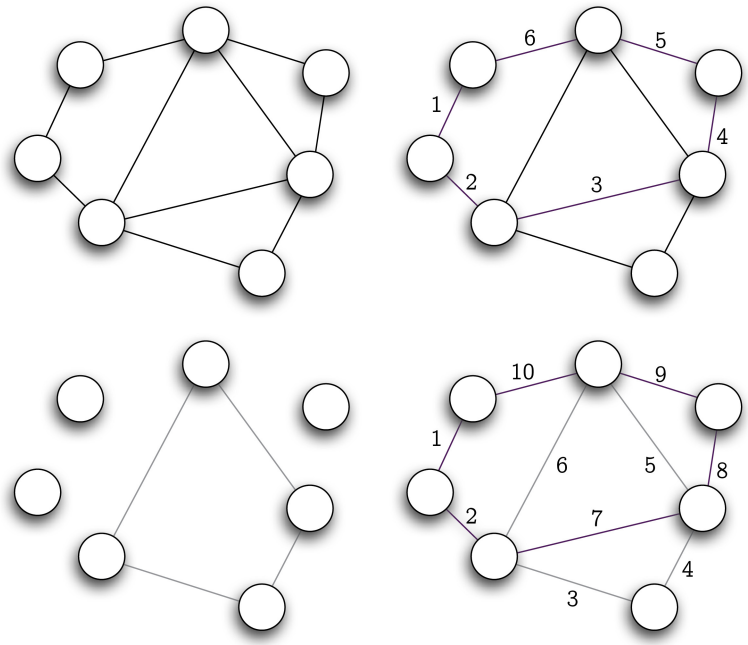


FIGURA 6.3. Ejemplo de la construcción de un ciclo Euleriano

En el ejemplo de la Figura 6.3, podemos empezar con el ciclo púrpura de largo 6, remover las aristas de este ciclo, buscar un nuevo ciclo (de color gris) e insertarlo dentro del ciclo púrpura, obteniendo así un ciclo Euleriano.

Notemos que el argumento anterior nos entrega además un algoritmo para construir un ciclo Euleriano en un grafo, conocido como el Algoritmo de Hierholzer[**15**]. Este algoritmo puede ser implementado de forma que visite cada arista a lo más dos veces, por lo que la complejidad de este algoritmo es $\mathcal{O}(m)$, donde m es la cantidad de aristas del grafo.

Ejercicio 6.2. Pruebe usando el pseudo-código, escrito en el ejercicio anterior, que la complejidad del algoritmo es $\mathcal{O}(m)$.

6.2 Ciclos Hamiltonianos

En el mismo problema anterior, tal vez lo que deseamos no es recorrer todos los puentes, sino que todas las "zonas" de Königsberg. Es decir, en el grafo que modela la ciudad deseamos encontrar un ciclo, pero esta vez le pediremos al ciclo que visite todos los **vértices** del grafo, pasando sólo una vez por cada uno. A este tipo de ciclo lo llamaremos *ciclos Hamiltonianos*.

El nombre de este tipo de ciclo viene de un famoso "juego" inventado por William Hamilton en 1857 y comercializado en Europa bajo el nombre "The Icosian Game". El problema era encontrar en el grafo de la Figura 6.4, un ciclo Hamiltoniano, es decir, un ciclo que visite cada uno de los vértices exactamente una vez.

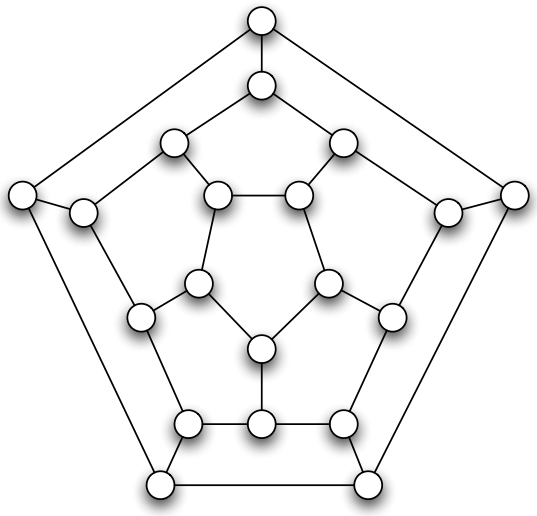


FIGURA 6.4. El grafo del "Icosian Game"

Definición 6.3. Dado un grafo G, un ciclo Hamiltoniano es un ciclo que visita cada uno de los vértices del grafo exactamente una vez.

Notemos la similaridad entre la definición de ciclos Eulerianos (ciclo que visita todas las aristas exactamente una vez) y un ciclo Hamiltoniano (ciclo que visita todos los vértices exactamente una vez). De la misma forma que antes, si para un grafo existe un ciclo Hamiltoniano, diremos que es un *grafo Hamiltoniano*. Al contrario de los grafos Eulerianos, el poder caracterizar si un grafo es Hamiltoniano o no es un problema "difícil".

Pero, ¿qué significa que un problema sea difícil? Efectivamente, el intentar definir esta noción de "dificultad" ha originado una nueva área de la ciencia llamada "complejidad computacional", cuyos inicios se remontan a los años 1950 y que ha dado origen a algunos de los problemas abiertos más famosos de la matemática. Retomaremos este tema en la siguiente sección.

El ejemplo más simple de un grafo Hamiltoniano es cuando el grafo es un ciclo, pues él mismo es un ciclo que recorre todos los vértices. Por otro lado, si un grafo es Hamiltoniano y le agrego una arista, el grafo resultante sigue siendo Hamiltoniano. En particular, este argumento permite probar que los grafos completos son Hamiltonianos.

Proposición 6.4. *Si $G = K_n$ es el grafo completo de n vértices (con $n > 2$), entonces G es Hamiltoniano.*

Esto responde también a otra idea intuitiva: mientras más aristas tiene un grafo de n vértices, es más posible que contenga un ciclo Hamiltoniano. Esta idea, permitió a G.A. Dirac probar uno de los primeros resultados en grafos Hamiltonianos.

Teorema 6.5 (Dirac, 1952)**.** *Todo grafo con n vértices ($n \geq 3$) y grado mínimo $\delta \geq \frac{n}{2}$ es Hamiltoniano.*

Demostración. El primer paso es probar que el grafo G es conexo. Dejamos este paso como un ejercicio para el lector.

Sea $P = x_0 \dots x_k$ el camino más largo G. Notemos que los vértices x_0 y x_k tienen a todos sus vecinos en P, pues si no, podríamos encontrar un camino más largo que P. Es decir, en P hay al menos $\frac{n}{2}$ vecinos de x_0 y al menos $\frac{n}{2}$ vecinos de x_k. Por el principio del palomar, necesariamente hay un vértice x_i vecino de x_k y un vértice x_{i+1} vecino de x_0, obteniendo el subgrafo de la Figura 6.5.

Notemos que podemos construir un ciclo C que recorre todos los vértices de P: $C = x_0 x_{i+1} x_{i+1} \dots x_k x_i x_{i-1} \dots x_1 x_0$. Más aún, este ciclo es un ciclo Hamiltoniano de G. Supongamos que esto no es cierto, es decir, que hay vértices del grafo G que no están en C. Como el grafo es conexo, alguno de estos vértices tiene que estar conectado a algún vértice de C, pero esto es una contradicción, pues significa que uniendo C a ese vértice, podemos construir un camino de largo $k+1$, lo que no es posible, pues P era el camino de mayor largo del grafo. □

La mejor caracterización que se conoce hasta hoy para grafos Hamiltonianos es una extensión del teorema anterior y fue descubierta por A. Bondy y V. Chvátal en 1972. Ella utiliza el concepto de la *clausura* de un grafo G, que se obtiene a partir G agregando una arista entre aquellos pares de vértices (u, v) tales que $\delta(u) + \delta(v) \geq n$.

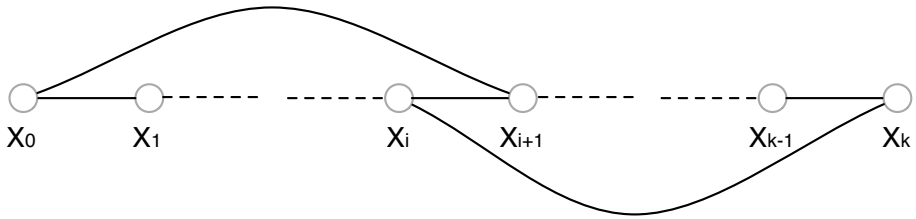


FIGURA 6.5. Demostración del teorema de Dirac

Teorema 6.6 (Bondy-Chvátal, 1972)**.** *Un grafo es Hamiltoniano, si y sólo si su grafo clausura es Hamiltoniano*

Notemos que el Teorema 6.5 está contenido dentro de esta caracterización, pues si el grado mínimo de un grafo es mayor o igual que $\frac{n}{2}$, entonces la clausura de este grafo es el grafo completo, el cual es Hamiltoniano como probamos anteriormente.

Pero volvamos a nuestro problema inicial: el Teorema 6.6 entrega una caracterización de Hamiltonianos basado en que su clausura sea también un grafo Hamiltoniano. ¿Podemos caracterizar a un grafo Hamiltoniano directamente? o dicho de otra forma, ¿podemos crear un algoritmo que encuentre un ciclo Hamiltoniano en un grafo?

En principio sí podemos hacerlo, es cosa de probar con todos los ciclos posibles sobre el grafo y verificar si ellos son Hamiltonianos. Esto, sin embargo, tiene complejidad exponencial, pues para un grafo general hay una cantidad exponencial de posibles ciclos. Esto lo hace imposible de llevar a la práctica, por los argumentos del primer capítulo de esta monografía.

Por lo tanto, la pregunta correcta es ¿existe un algoritmo de orden polinomial para encontrar un ciclo Hamiltoniano en un grafo?

Esta pregunta aún no tiene respuesta. No se sabe si existe o no un algoritmo con tales características y se cree que no es posible, pero esto no ha podido ser probado. Notemos que si bien no podemos encontrar un algoritmo polinomial para encontrar un ciclo Hamiltoniano, nosotros sí podemos "verificar" si un ciclo es o no Hamiltoniano en tiempo polinomial (basta con verificar si pasa por cada vértice exactamente una vez).

6.3 Las clases $\mathcal{P}$ y $\mathcal{NP}$

Para terminar esta monografía, estudiemos tal vez el problema más importante de las ciencias de la computación: si $\mathcal{P} = \mathcal{NP}$.

Supongamos que tenemos un problema de decisión, cuya respuesta es "sí" o "no", por ejemplo, si un grafo dado tiene un ciclo Euleriano.

La clase de todos aquellos problemas de decisión que podemos responder en tiempo polinomial, es lo que se conoce como $\mathcal{P}$. Aquellos problemas que no sabemos

responder en tiempo polinomial, pero que dada una solución podemos verificarla en tiempo polinomial, es lo que se conoce como $\mathcal{NP}$.

Por ejemplo, decidir si un grafo es Euleriano, esta en $\mathcal{P}$, pues basta con verificar si el grado de cada vértice del grafo es par o no. Por otro lado, decidir si un grafo es Hamiltoniano, está en $\mathcal{NP}$, aun cuando no sabemos responder la pregunta, si alguien nos entrega un ciclo del grafo, nosotros podemos verificar si este es Hamiltoniano o no en tiempo polinomial.

Dada la definición anterior, si un problema está en $\mathcal{P}$, entonces está en $\mathcal{NP}$, es decir, $\mathcal{P} \subseteq \mathcal{NP}$. La otra inclusión ha dado origen a uno de los problemas más famosos de la matemática actual:

$$\boxed{\text{¿ } \mathcal{P} = \mathcal{NP} \text{ ?}}$$

La pregunta $\mathcal{P} = \mathcal{NP}$ es uno de los siete "problemas del milenio", para los cuales el Instituto de Matemáticas Clay ha ofrecido 1 millón de dolares a aquel que pueda probar que $\mathcal{P} = \mathcal{NP}$ o $\mathcal{P} \neq \mathcal{NP}$. En palabras, la pregunta es:

> *S*i para un problema de decisión ("sí" o "no") soy capaz de verificar una respuesta rápidamente (en tiempo polinomial), ¿puedo calcular una respuesta rápidamente?

La mayoría de los matemáticos cree que estas clases no son iguales, sin embargo, si usted encuentra un algoritmo polinomial que decida si existe o no un ciclo Hamiltoniano en un grafo cualquiera, ¡¡habrá probado que $\mathcal{P} = \mathcal{NP}$!!.

Para terminar, presentamos una lista de algunos problemas de la teoría de grafos que se encuentran en $\mathcal{P}$ y en $\mathcal{NP}$ relacionados con los temas vistos en esta monografía.

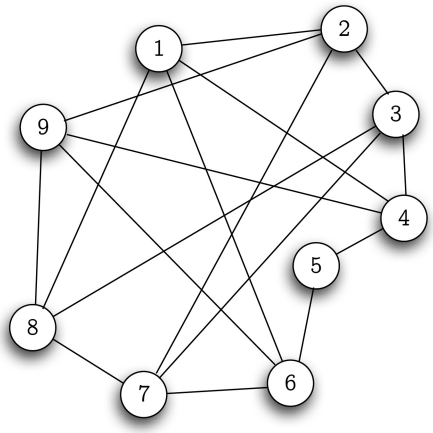
en $\mathcal{P}$	en $\mathcal{NP}$
Encontrar un ciclo Euleriano en G.	Encontrar un ciclo Hamiltoniano en G.
Encontrar el subgrafo sin ciclos de mayor tamaño en G.	Encontrar el subgrafo completo de mayor tamaño en G.
Encontrar un árbol generador de G.	Encontrar un árbol generador de G de grado máximo K.
Encontrar el camino más corto entre dos vértices.	Encontrar el camino más largo entre dos vértices.
Encontrar el flujo máximo de una red	Dado un costo por cada arco, encontrar el flujo de costo mínimo de una red.

Tabla 6.1. Algunos problemas en P y NP

6.4 Ejercicios

1. Calcule cuántos ciclos Eulerianos tiene el grafo completo K_4.
2. Sea el siguiente algoritmo para construir un ciclo Euleriano: partiendo de un vértice v_0, continuar por una arista incidente aún no visitada y repita esto hasta que no queden aristas incidentes sin visitar.
   - Encuentre un grafo donde este algoritmo termine con un ciclo Euleriano.
   - Encuentre un grafo donde este algoritmo no termine con un ciclo Euleriano.
   - Pruebe que en un grafo Euleriano, este algoritmo partiendo de v_0 termina con un ciclo Euleriano si y sólo si todo ciclo contiene a v_0.
3. El *grafo línea* de G es un grafo donde hay un vértice por cada arista de G y una arista entre dos vértices, si y sólo si las correspondientes aristas en G son incidentes a un mismo vértice. Pruebe que, si G es Euleriano, entonces el grafo línea de G es Euleriano y Hamiltoniano.
4. Pruebe que si G es Hamiltoniano, entonces el grafo línea de G es Hamiltoniano.
5. Dé un ejemplo de un grafo no-Hamiltoniano con 10 vértices tal que, para cada par de vértices no adyacentes u y v, se tiene que $\delta(u) + \delta(v) \geq 9$.
6. Encuentre los valores de n para los que el grafo completo K_n sea Euleriano.
7. Encuentre los valores de n para los que el grafo completo K_n sea Hamiltoniano.
8. Dibuje dos grafos de 10 vértices y 13 aristas: uno que sea Euleriano, pero no Hamiltoniano, y el otro que sea Hamiltoniano, pero no Euleriano.
9. En el juego Dominó, dos fichas pueden ir adyacentes por lados que tengan el mismo número. ¿Es posible ordenar todos los dominós en un círculo, respetando la regla anterior? Modele este problema, usando un grafo donde los vértices son las números y las aristas son los dominós. ¿Qué tipo de ciclo estamos buscando? ¿se puede modelar como un ciclo Hamiltoniano sobre otro grafo similar?
10. Javiera va a invitar a comer a seis amigos: Alvaro, Bernardo, Claudio, Eduardo, Eric y Marcos. Javiera tiene buenas relaciones con todos ellos, sin embargo:
    - Claudio estuvo de cumpleaños y Eric no lo saludó.
    - Alvaro es hincha de Colo Colo, Eduardo de la U. de Chile y Claudio de Everton, por lo que siempre que están juntos empiezan a discutir.
    - Marcos le debe dinero a Bernardo.
    - Eric le rayó el auto a Alvaro y no le ha dicho aún.
    - Bernardo acaba de imitar a Eric, lo que lo enojó muchísimo.
    - Marcos y Alvaro tienen diferencias políticas irreconciliables.

    A pesar de todo esto, Javiera quiere sentar a sus amigos alrededor de una mesa redonda, cuidando de que cada invitado quede al lado de alguien con quien tenga buenas relaciones. ¿Es esto posible?
11. En un paseo de curso a un parque nacional, Verónica y Carmen desean recorrer los lugares de interés del parque, usando los senderos mostrados en la siguiente figura:



Carmen es una turista práctica, por lo que desea recorrer todos los lugares de interés, sin repetir ninguno o volver a su lugar inicial. Verónica por el contrario es aventurera y le gustaría recorrer todos los senderos del bosque exactamente una vez, sin importar si pasa dos veces por un mismo lugar de interés.

- ¿Es posible encontrar una ruta para Verónica? Si es posible, encuéntrela.
- ¿Es posible encontrar una ruta para Carmen? Si es posible, encuéntrela.

Bibliografía

[1] Aho, A.V., Ullman, J.D. *Foundations of computer science*. Principles of Computer Science Series. Computer Science Press, Nueva York, 1992.

[2] Aldous, J.D., Wilson, R. *Graphs and Applications: An Introductory Approach*. Springer-Verlag, 2003.

[3] Bang-Jensen, J., Gutin, G. *Digraphs: theory, algorithms and applications*. Springer-Verlag, 2002.

[4] Bollobás, B. *Modern graph theory*, volume 184 of *Graduate Texts in Mathematics*. Springer-Verlag, Nueva York, 1998.

[5] Bonnans, J.F., Gilbert, J.C., Lemaréchal, C., Sagastizábal, C. *Numerical Optimization: theoretical and numerical aspects*. Universitext. Springer-Verlag, Berlin, 2004.

[6] Diestel, R. *Graph Theory*. Springer-Verlag, 2000.

[7] Dinits, E.A. Algorithms for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970.

[8] Edmonds, J., Karp, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. In *Combinatorial Structures and their Applications (Proc. Calgary Internat. Conf., Calgary, Alta., 1969)*, pages 93–96. Gordon and Breach, New York, 1970.

[9] Erdös, P., Szekeres, G. *A combinatorial problem in geometry*. Compositio Math., 2:463–470, 1935.

[10] Ford, L.R., Fulkerson, D.R. *Maximal flow through a network*. Canad. J. Math., 8:399–404, 1956.

[11] Gabow, N. *Scaling algorithms for network problems*. J. Comput. System Sci., 31(2):148–168, 1985. Special issue: Twenty-fourth annual symposium on the foundations of computer science (Tucson, Ariz., 1983).

[12] Chartrand, G., Lesniak, L. *Graphs and Digraphs*. Chapman & Hall/CRC, 2004.

[13] Gibbons, A. *Algorithmic graph theory*. Cambridge University Press, Cambridge, 1985.

[14] Harary, F. *Graph theory*. Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London, 1969.

[15] Hierholzer, C. *Über die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechnung zu umfahren*. Mathematische Annalen, 6:30–32, 1873.

[16] Ortiz, C., Varas, S., Vera, J. *Optimización y Modelos para la gestión*. Dolmen Ediciones, 2000.

[17] Rosen, K. *Discrete Mathematics and its applications*. McGraw-Hill, 2006.

Índice de figuras

1.1. Lista $(b_1, b_2, ..., b_n)$ justo antes de la iteración i-ésima del algoritmo selección 27
1.2. Llamadas y salidas realizadas por la rutina factorial(4) 31
1.3. División y fusión recursiva 35
1.4. Algoritmo glotón usado para dar un vuelto de $760. 37
1.5. Crecimiento de funciones del ejemplo 40
1.6. Índices i y j utilizados en el Algoritmo 1.4 43

2.1. Ejemplo de un grafo 50
2.2. Ejemplo de un grafo dirigido 51
2.3. Otras representaciones gráficas del grafo G 51
2.4. Grafos isomorfos 52
2.5. Ejemplo de un grafo conexo y uno que no lo es 54
2.6. Diferencia entre un grafo y un conjunto de vértices. Izquierda: G, Derecha: $G \setminus \{2\}$ 55
2.7. Diferencia entre un grafo y un conjunto de aristas. Izquierda: G, Derecha: $G \setminus \{(1,2),(2,4)\}$ 56
2.8. Suma entre un grafo y una arista. Izquierda: G, Derecha: $G + (2,5)$ 56
2.9. Ejemplo de un árbol 57
2.10. Ejemplo de un grafo bipartito y un no-bipartito 62
2.11. Ejemplo del grafo bipartito-completo $K_{3,2}$ 63
2.12. Ejemplo de un grafo dirigido y su versión no-dirigida 65
2.13. En púrpura se describe un camino y un ciclo, respectivamente 65

3.1. Grafo de los posibles caminos entre los pueblos. El número al lado de cada arista corresponde al costo de construcción de ese camino 69
3.2. Dos soluciones al problema anterior 70
3.3. Pasos del Algoritmo de Prim 71
3.4. Pasos del Algoritmo de Kruskal (comparar con Figura 3.3) 72

4.1. Ejemplo de una ciudad 75
4.2. Grafo asociado a la ciudad anterior 76
4.3. Grafo asociado a la ciudad anterior 77
4.4. Resultado del Algoritmo de Dijkstra 78
4.5. Demostración del algoritmo de Dijkstra 79
4.6. Ejemplo del problema 80
4.7. Resultado del algoritmo de Ford-Bellman 81

5.1. Transporte de gas en una ciudad, los flujos f_{ij} se presentan en negro y las capacidades c_{ij} en púrpura 86
5.2. Transporte de gas en una ciudad, incluyendo ofertas y demandas 87
5.3. Ejemplos de cortes 90
5.4. Ejemplo de camino aumentador y cuello de botella 93
5.5. Calculo del nuevo flujo f' 94
5.6. Dibujo flujo vehicular Santiago 96
5.7. Ejemplo grafo residual 98
5.8. Calcule el grafo residual 98
5.9. Ejemplo algoritmo de Ford-Fulkerson 100
5.10. Asignación de flujo en la primera iteración 100
5.11. Asignación de flujo en la segunda iteración 101
5.12. Flujo óptimo 101
5.13. Encuentre el flujo máximo 102
5.14. Los valores que aparecen en los arcos indican sus capacidades 105
5.15. Red de oleoductos con sus capacidades 106
5.16. Encuentre el flujo máximo 107

6.1. Los puentes de Königsberg en 1736 109
6.2. Representación en grafo de los puentes de Königsberg 110
6.3. Ejemplo de la construcción de un ciclo Euleriano 111
6.4. El grafo del "Icosian Game" 112
6.5. Demostración del teorema de Dirac 114

Índice de Términos

$\delta(v)$, 52

algoritmo, 21
    Dijkstra, 77
    Ford-Bellman, 81
    Ford-Fulkerson, 99
    Hierholzer, 111
    Kruskal, 72
    Prim, 71
algoritmo, 21
árbol, 57
    generador, 67
arco, 50
arista
    incidente, 52
aristas, 49

bosque, 67
bucles, 49

cadena, 65
camino, 53
    aumentador, 92
    más corto, 69
capacidad, 85
ciclo, 54
    Euleriano, 110
    Hamiltoniano, 112
clase
    $\mathcal{NP}$, 115
    $\mathcal{P}$, 114
complejidad, 37, 38
componentes conexas, 54
corte, 89
cuello de botella, 92

Egsger Dijkstra, 77

flujo, 85
    factible, 87
Frank P. Ramsey, 60

G.A. Dirac, 113
grafo, 49
    árbol, 57
    bipartito, 61
        completo, 63
    complemento, 56
    completo, 58
    conexo, 54
    dirigido, 50
    Euleriano, 110
    Hamiltoniano, 113
    isomorfo, 52
    no-dirigido, 49
    residual, 97
    suma, 55
    tamaño, 50
Gustav Robert Kirchhoff, 87

hojas, 57

Joseph Kruskal, 72

lema del apretón de manos, 53
Leonhard Euler, 88, 110
Lester Ford, 80
Ley de Kirchhoff, 87

números de Ramsey, 60
nodos, 49

operaciones sobre grafos, 55
ordenamiento de listas, 26

Problema
de flujo máximo, 88
pseudo-lenguaje de programación, 22

Red, 85
Richard Bellman, 80
Robert C. Prim, 70

subgrafo, 53

teorema de flujo máximo–corte mínimo, 95
tipos de algoritmos: iteracion, induccion y recursion, 23

vértice, 49
aislado, 52
demandante, 87
grado, 52
ofertante, 87
transiente, 87

William Hamilton, 112