MANUAL

ALGORITMOS Y CONCEPTOS BÁSICOS

Suite informática de Teoría Algorítmica de Grafos

Graphvisualx

Índice general

Indice de figuras

Capítulo 1

Conceptos básicos

1.1. Generalidad

Definiremos un grafo como un sistema matemático abstracto. No obstante, para desarrollar el conocimiento de los mismos de forma intuitiva los representaremos mediante diagramas. A estos diagramas les daremos también el nombre de grafos, aun cuando los términos y definiciones no estén limitados únicamente a los grafos que pueden representarse mediante diagramas.
Un grafo es un conjunto de puntos y un conjunto de líneas donde cada línea une un punto con otro.

1.2. Definición formal

Llamaremos grafo, G, al par ordenado formado por un conjunto finito no vacío, V, y un conjunto, A, de pares no ordenados de elementos del mismo.

V es el conjunto de los vértices o nodos del grafo.

A será el conjunto de las aristas o arcos del grafo.

Utilizaremos la notación G = (V,A) para designar al grafo cuyos conjuntos de vértices y aristas son, respectivamente, V y A.

A cualquier arista de un grafo se le puede asociar una pareja de vértices del mismo. Si u y v son dos vértices de un grafo y la arista a esta asociada con este par, escribiremos $a = uv$.

Por ejemplo, si

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

y

$$A = \{v_1v_2, v_1v_3, v_1v_4, v_2v_4, v_2v_5\}$$

entonces el grafo $G = (V, A)$ tiene a v_1, v_2, v_3, v_4 y v_5 como vértices y sus aristas son $v_1v_2, v_1v_3, v_1v_4, v_2v_4$ y v_2v_5.

1.3. Tipos de grafos

1.3.1. Grafos lineales

Sea un grafo L_n , diremos que es un grafo lineal con n vértices ($n \geq 2$) si tiene n vértices (dos de grado 1 y el resto, si los hay, son de grado 2) y es isomorfo a:

Figura 1.1: Grafo lineal

1.3.2. Grafos circulares

Sea un grafo C_n , diremos que es un grafo circular con n vértices (todos de grado 2), si es $|n| \geq 3$:

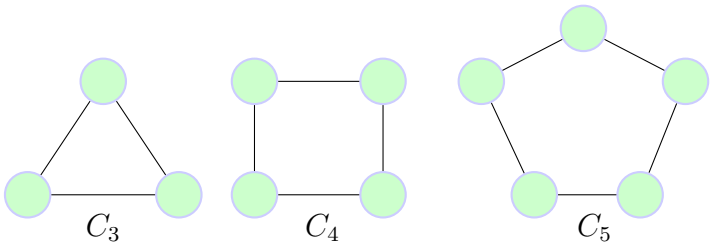


Figura 1.2: Grafo circular

1.3.3. Grafos completos

Se dice que un grafo es completo cuando todos sus vértices son adyacentes a todos los vértices del grafo, es decir, cuando cada par de vértices son los extremos de una arista. Notaremos por K_n los grafos completos de n vértices.

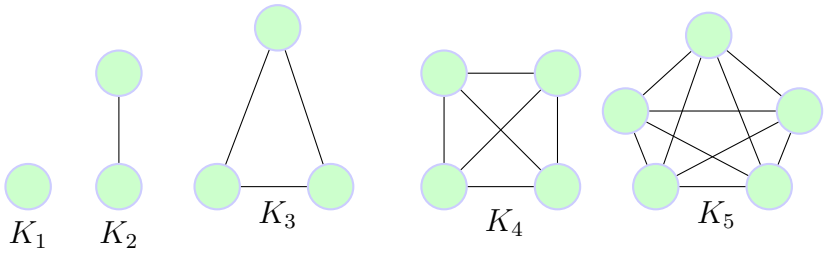


Figura 1.3: Grafo completo

1.3.4. Grafos vacíos

Si un grafo tiene n vértices y ninguna arista, estamos ante un grafo vacío.

1.3.5. Grafo regular

Un grafo se dice que es regular cuando todos sus vértices tienen el mismo grado.

1.3.6. Grafo complementario

Dado un grafo G con n vértices, llamaremos complemento de G (o complementario del grafo G), y lo notaremos por $\overline{G}$, al subgrafo de K_n formado por todos los vértices de G y las aristas que no están en G.

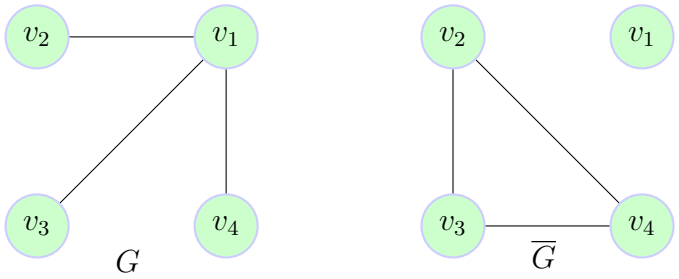


Figura 1.4: Grafo complementario

1.3.7. Grafos bipartitos

Los grafos bipartitos son aquellos que admiten una partición de sus vértices en dos conjuntos disjuntos $V = X \cup Y$, de manera que las aristas tienen un extremo en cada uno de estos conjuntos (van de vértices en X a vértices en Y).

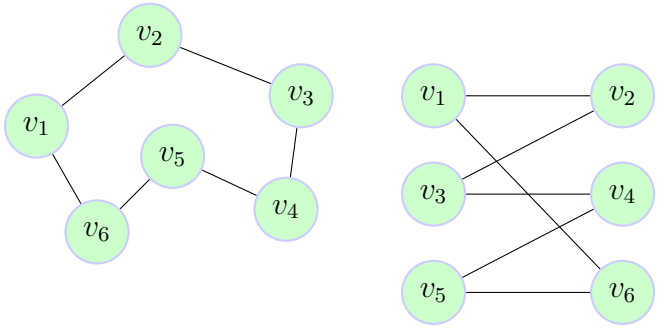


Figura 1.5: Dos realizaciones del mismo grafo bipartito

1.3.8. Multigrafos

Llamaremos de esta forma a los grafos en los que haya pares de vértices unidos por más de una arista.

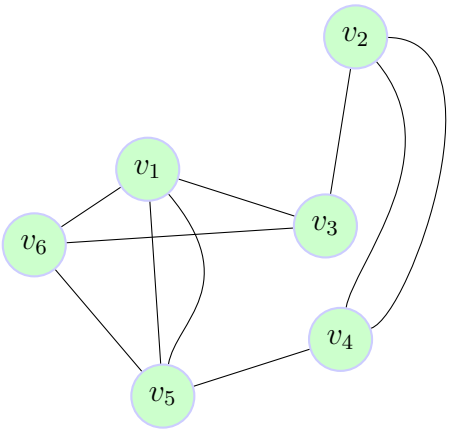


Figura 1.6: Multigrafo de 6 vértices

1.3.9. Pseudografo

Llamaremos pseudografo a los grafos en los que existan aristas cuyos extremos coincidan, es decir, aquellos en los que existan aristas que unan vértices consigo mismos. A tales aristas las llamaremos bucles o lazos.

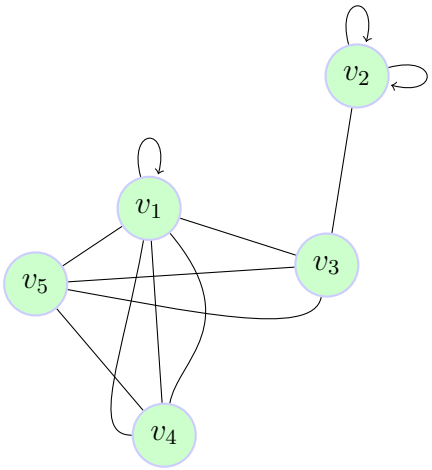


Figura 1.7: Pseudografo de 5 vértices

1.3.10. Digrafo

Es un grafo en el cual el conjunto de las aristas de A está formado por pares ordenados del conjunto de vértices V. Lo llamaremos también grafo dirigido.

Esto asigna un orden en los extremos de cada arista. Dicho orden se indica en el diagrama con una flecha y llamaremos origen o inicial al primer vértice de una arista y fin o terminal al segundo.

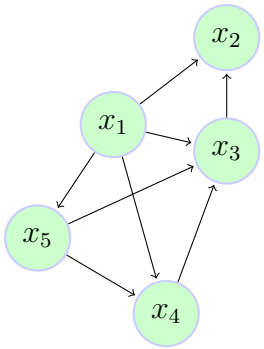


Figura 1.8: Digrafo de 5 vértices

1.4. Vértices adyacentes

Diremos que los vértices u y v son adyacentes, si existe una arista a tal que $a = uv$. A los vértices u y v los llamaremos extremos de la arista.

1.5. Vértices incidentes

Diremos que un vértice u es incidente con una arista a cuando u es extremo de la arista a.

1.6. Representación gráfica

Un grafo se representa mediante un diagrama en el cual a cada vértice le corresponde un punto y si dos vértices son adyacentes se unen sus puntos correspondientes mediante una línea.

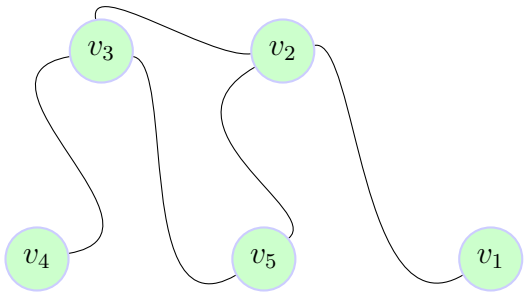


Figura 1.9: Representación gráfica de un grafo

En el grafo de la figura tiene como conjunto de vértices

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

siendo su conjunto de aristas,

$$A = \{v_1v_2, v_2v_3, v_2v_5, v_3v_4, v_3v_5\}$$

Vértices adyacentes: v_1 y v_2; v_2 y v_3; v_2 y v_5; v_3 y v_4; v_3 y v_5.
Vértices no adyacentes: v_1 y v_3; v_1 y v_4; v_2 y v_4; v_4 y v_5.

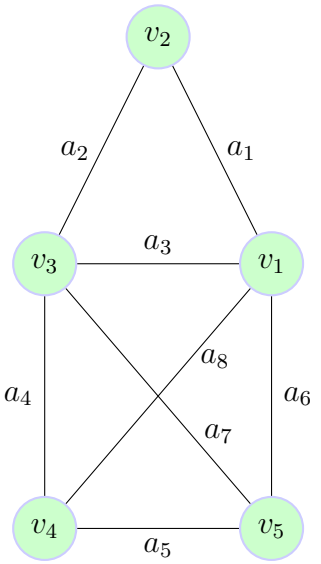
Hay diversas formas de representar un mismo grafo. La más geométrica ya la hemos utilizado: dando una representación gráfica de la estructura, representando los vértices

por puntos y las aristas por segmentos curvos o líneas (normalmente rectilíneos).

En aras de un tratamiento más mecanizado, que no involucre la realización gráfica de la estructura, se han ido sucediendo diversos procedimientos matriciales para representar las distintas estructuras, tales como la matriz de adyacencia, la matriz de incidencia y las listas de adyacencia.

1.6.1. Matriz de adyacencia

Una representación por matriz de adyacencia de un grafo es una matriz de V por V de valores booleanos. Si el grafo incluye una arista desde el nodo i hasta el nodo j, entonces matriz[i,j] = verdadero (1); en caso contrario matriz[i,j] = falso (0). En el caso de un grafo no dirigido, la matriz será necesariamente simétrica.



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

Figura 1.10: Grafo con su representación en matriz de adyacencia

Con esta representación, es fácil ver si existe o no una arista entre dos nodos dados: buscar un valor en la matriz requiere un tiempo constante. Por otra parte, si deseamos examinar todos los nodos que estén conectados con algún nodo dado, tenemos que recorrer toda una fila completa de la matriz, en el caso de un grafo no dirigido, o bien tanto una fila completa como una columna completa en el caso de un grafo dirigido. Esto requiere un tiempo que está en $\Theta(V)$, el número de nodos que hay en el grafo, independientemente del número de aristas que entren o salgan de ese nodo particular. El espacio requerido para representar un grafo de esta manera es cuadrático con respecto al número de nodos.

La representación por matriz de adyacencia no es satisfactoria para la gran mayoría de grafos dispersos: necesitamos como mínimo para ello V^2 bits de almacenamiento y V^2 pasos sólo para la construcción de la matriz inicial representativa. En un grafo denso, donde el número de aristas (el número de 1 bit de la matriz) es proporcional a V^2, este coste puede ser aceptable, porque el tiempo proporcional a V^2 es el necesario para procesar las aristas sin importar la representación que se use en principio. En grafos dispersos, sin embargo, tan sólo la inicialización de la matriz podría ser el factor dominante en el tiempo de ejecución de un algoritmo. Por otra parte, tal vez ni siquiera habría suficiente espacio para la matriz de esa forma. Por ejemplo, podemos estar ante grafos con millones de vértices y decenas de millones de aristas, que sin embargo no tendrán que pagar el precio de reservar espacio en memoria para miles de millones de entradas con el valor 0 en la matriz de adyacencia.

Por otro lado, cuando es necesario procesar un grafo denso, las 0-entradas que representan ausencia de arista, incrementan nuestro espacio necesario sólo por un factor constante permitiéndonos determinar así, si cualquiera de las aristas está presente según la constante de tiempo. Si tenemos espacio disponible para almacenar una matriz de adyacencia y si V^2 es lo suficientemente pequeño como para ser representado en una cantidad insignificante de tiempo o para hacer uso de un algoritmo complejo que no requiera más de V^2 pasos para completarse, la representación mediante matriz de adyacencia puede ser el método de elección adecuado, sin importar lo denso que pueda ser el grafo.

1.6.2. Matriz de incidencia

La representación mediante matriz de incidencia de un grafo, es una matriz rectangular $v \times a$ de tantas filas como vértices $v = |V|$ tiene el grafo y tantas columnas como aristas $a = |A|$ hay. La entrada (i, j) consiste en un 1 si el vértice x_i es incidente con la arista e_j, siendo 0 en otro caso. Dado que cada arista incide en dos vértices (distintos), esta matriz verifica que cada columna contiene exactamente dos 1, siendo el resto de entradas 0. Toda matriz con esta propiedad es, de hecho, la matriz de incidencia de cierto grafo.

$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccccc} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Figura 1.11: Matriz de incidencia del grafo de la Figura 2.9

1.6.3. Listas de adyacencia

La representación estándar preferida para representar grafos que no sean densos es la representación mediante listas de adyacencia, donde hacemos un seguimiento de todos los vértices conectados a cada vértice en una lista enlazada que está asociada con ese vértice. En la lista de adyacencia se asocia a cada nodo i una lista formada por sus vecinos, esto es, una lista formada por aquellos nodos j tales que existe una arista de i a j (en el caso de un grafo dirigido) o entre i y j (en el caso de un grafo no dirigido). Si el número de aristas del grafo es pequeño, esta representación utiliza menos espacio que la dada anteriormente. También puede que sea posible en este caso examinar todos los vecinos de un modo dado en menos de V operaciones de análisis en el caso medio. Por otra parte, determinar si existe o no conexión directa entre dos nodos dados i y j nos obliga a recorrer la lista de vecinos del nodo i (y posiblemente también los del nodo j, en el caso de un grafo dirigido), lo cual es menos eficiente que buscar un valor booleano en una matriz.

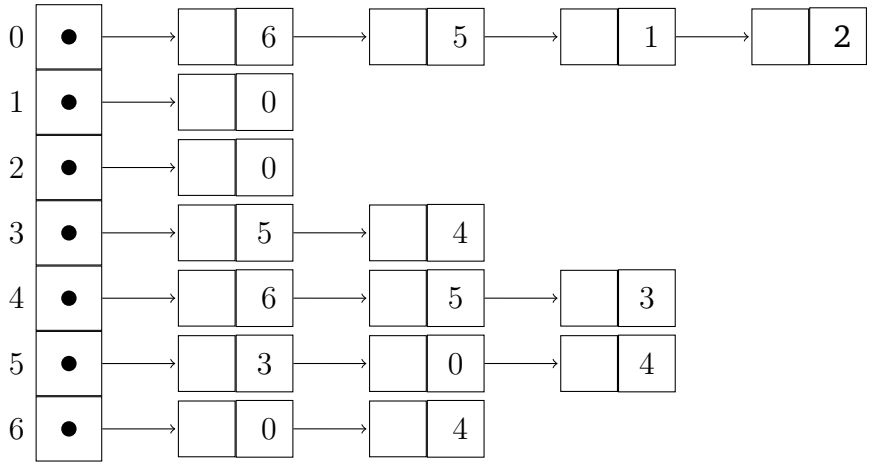


Figura 1.12: Listas de adyacencia

Para agregar una arista que conecte i e j, se añade j a la lista de adyacencia de i e i a la lista de adyacencia de j. De esta manera, podemos agregar nuevas aristas en tiempo constante, pero la cantidad total de espacio que utilizamos es proporcional al número de vértices más el número de aristas (en comparación con el número de vértices, que formarían un cuadrado en una matriz de adyacencia). Para los grafos no dirigidos, representaremos cada arista en dos lugares distintos: una arista que conecta con i e j se representa con nodos en ambas listas de adyacencia.

Con las listas de adyacencia, hay numerosas representaciones de un grafo dado incluso para una determinada numeración de vértices. No importa en qué orden aparezcan las aristas de las listas de adyacencia, ya que la estructura de la lista de adyacencia representará siempre al mismo grafo. Esta característica de las listas de adyacencia es

importante para saber porque el orden en que las aristas aparecen en las listas afecta, a su vez, al orden en que las aristas son procesadas por los algoritmos. Es decir, la estructura de las listas de adyacencia determina como se comportará el grafo para el cálculo de un algoritmo sobre él. A pesar de que un algoritmo debe producir una solución correcta sin importar que las aristas estén ordenadas en las listas de adyacencia, se podría llegar a dicha solución por diferentes secuencias de procesamiento del algoritmo sobre diferentes ordenamientos. Si un algoritmo no necesita examinar todas las aristas del grafo para su cálculo, este efecto podría afectar al tiempo de desarrollo del algoritmo. Y, si hay más de una solución correcta, ordenamientos de entrada de aristas diferentes pueden llevar a resultados de salida diferentes.

1.6.4. Principal ventaja de uso

La principal ventaja de la representación de listas de adyacencia sobre la representación en matriz de adyacencia es que siempre se utiliza el espacio proporcional a $A + V$, frente a V^2 en la matriz de adyacencia. La principal desventaja es que la comprobación de la existencia de arista entre vértices puede llevar un tiempo proporcional a V, a diferencia del tiempo constante de la matriz de adyacencia. En estas características, en consecuencia, radica la diferencia entre el uso de listas de adyacencia enlazadas y vectores que representen el conjunto de vértices incidentes con los otros vértices del conjunto.

1.7. Grados

1.7.1. Grado de un vértice

Llamaremos grado o valencia de un vértice al número de aristas que incidan en él.

Notaremos por $gr_G(v)$ al grado del vértice v en el grafo G y cuando no haya posibilidad de confusión notaremos, simplemente por $gr(v)$.

1.7.2. Vértice aislado

Un vértice de grado cero se denomina aislado.

1.7.3. Suma de los grados de un grafo

En cualquier grafo se verifica,

(a) *La suma de todos sus grados es igual al doble del número de sus aristas.*

(b) *El número de vértices de grado impar es par.*

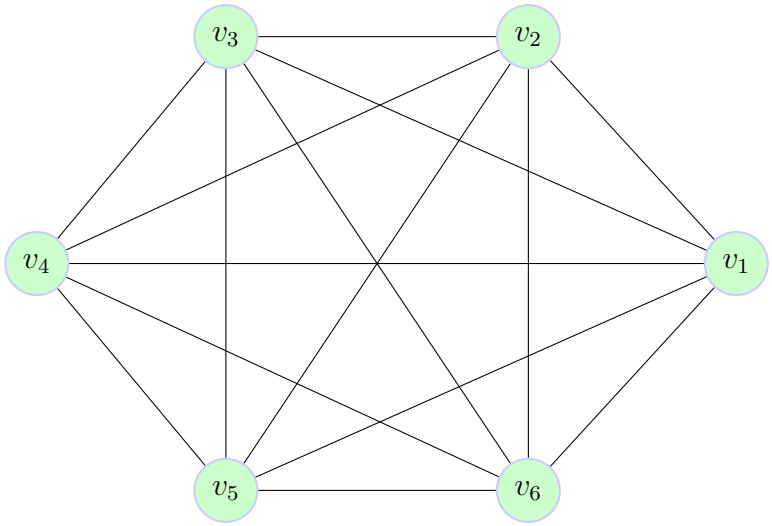


Figura 1.13: Grafo de ejemplo

Sea $G_1 = (V, A)$ siendo

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

y

$$A = \{v_1v_2, v_1v_3, v_1v_4, v_1v_5, v_1v_6, v_2v_3, v_2v_4, v_2v_5, v_2v_6, v_3v_4, v_3v_5, v_3v_6, v_4v_5, v_4v_6, v_5v_6\}$$

Entonces, $|A| = 15$ y $gr(v_i) = 5$, $i = 1, 2, 3, 4, 5, 6$, luego

$$\sum_{i=1}^{6} gr(v_i) = 30 = 2 \cdot 15{,}5 = 2|A|$$

Por otra parte, todos los vértices son de grado impar, luego su número (6) es par.

1.7.4. Grado de entrada y de salida

Si v es un vértice de un digrafo D, entonces su grado de entrada $gr_e(v)$ es el número de arcos en D de la forma uv y su grado de salida $gr_s(v)$ es el número de arcos en D de la forma vu.

1.8. Isomorfismo

1.8.1. Isomorfismo de grafos

Dos grafos $G_1 = (V_1, A_1)$ y $G_2 = (V_2, A_2)$ se dice que son isomorfos cuando existe una biyección entre los conjuntos de sus vértices que conserva la adyacencia. Si los grafos G_1 y G_2 son isomorfos, notaremos $G_1 \simeq G_2$.

Según la definición anterior,

$$G_1 \simeq G_2 \iff \exists f : V_1 \to V_2 : \begin{cases} f \text{ es biyectiva} \\ uv \in A_1 \iff f(u)(v) \in A_2; \forall u, v \in V_1 \end{cases}$$

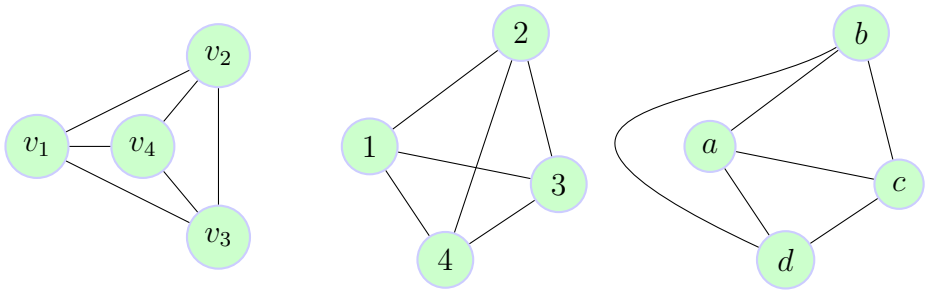


Figura 1.14: Tres grafos isomorfos entre sí

1.9. Subgrafos

1.9.1. Definición

Un subgrafo de un grafo $G = (V(G)), A(G))$ es un grafo $H = (V(H), A(H))$ tal que $V(H) \subseteq V(G)$ y $A(H) \subseteq A(G)$

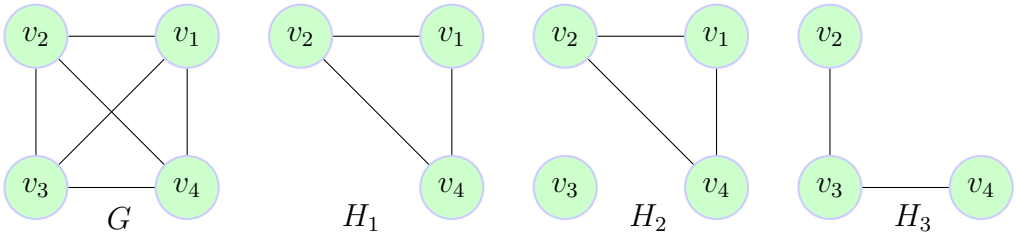


Figura 1.15: Un grafo G y tres de sus subgrafos

$$G = \{v_1, v_2, v_3, v_4\}, \{v_1v_2, v_1v_3, v_1v_4, v_2v_3, v_2v_4, v_3v_4\}$$
$$H_1 = \{v_1, v_2, v_4\}, \{v_1v_2, v_1v_4, v_2v_4\}$$
$$H_2 = \{v_1, v_2, v_3, v_4\}, \{v_1v_2, v_1v_4, v_2v_4\}$$
$$H_3 = \{v_2, v_3, v_4\}, \{v_2v_3, v_3v_4\}$$

1.9.2. Eliminación de aristas

Si a es una arista del grafo G, entonces el subgrafo $G \setminus \{a\}$ es el grafo que se obtiene de G eliminando la arista a.
En general, escribiremos $G \setminus \{a_1, a_2, \ldots, a_k\}$ para denominar al subgrafo que se obtiene de G eliminando las aristas $a_1, a_2, \ldots, a_k$.

1.9.3. Eliminación de vértices

Si v es un vértice del grafo G, entonces el subgrafo $G \setminus \{v\}$ es el grafo que se obtiene de G eliminando el vértice v.
En general, escribiremos $G \setminus \{v_1, v_2, \ldots, v_k\}$ para denominar al subgrafo que se obtiene de G eliminando los vértices $v_1, v_2, \ldots, v_k$.

1.10. Caminos y ciclos

1.10.1. Camino

Sea G un grafo o un multigrafo. Un camino en G es una sucesión donde se alternan vértices y aristas, comenzando y terminando con vértices y en el que cada arista es incidente con los dos vértices que la preceden y la siguen.

Un camino que une los vértices v_1 y v_n sería:

$$v_1, v_1v_2, v_2, v_2v_3, \ldots, v_{n-1}, v_{n-1}v_n, v_n$$

Si se trata de un grafo (no un multigrafo) este camino también puede especificarse simplemente por la sucesión de sus vértices, $v_1, v_2, v_3, \ldots, v_{n-1}, v_n$ y lo representamos por:

$$\gamma = \langle v_1, v_2, v_3, \ldots, v_{n-1}, v_n \rangle$$

A los vértices v_1 y v_n se les denomina extremos del camino. Suele decirse también que el camino conecta v_1 con v_n o que va de v_1 a v_n. La longitud del camino es el número $n - 1$ de aristas que contiene.
Un camino es simple si en la sucesión de vértices no hay ninguno repetido.

1.10.2. Ciclo

Sea G un grafo o un multigrafo. Un ciclo en G es un camino en el que sus extremos coinciden.

El ciclo será simple si no hay, además del primero y el último, ningún otro vértice repetido.

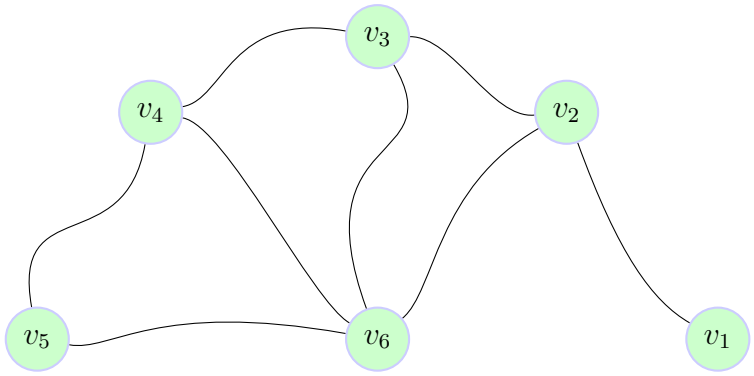


Figura 1.16: Caminos y Ciclos

$\gamma = \langle v_1,\ v_2,\ v_6,\ v_3,\ v_4,\ v_6,\ v_5 \rangle$ es un camino.

$\gamma = \langle v_1,\ v_2,\ v_3,\ v_4 \rangle$ es un camino simple ya que no hay ningún vértice repetido.

$\gamma = \langle v_1,\ v_2,\ v_6,\ v_5,\ v_4,\ v_6,\ v_2,\ v_1 \rangle$ es un ciclo.

$\gamma = \langle v_2,\ v_3,\ v_4,\ v_5,\ v_6,\ v_2 \rangle$ es un ciclo simple ya que se repiten, únicamente, los vértices primero y último.

Un grafo acíclico no contiene ningún ciclo. Los árboles están conectados con los grafos no dirigidos acíclicos. Los árboles son los grafos más simples, y sus estructuras intrínsecamente recursivas, ya que la eliminación o corte de cualesquieras aristas deja como resultado dos pequeños árboles.
A los grafos dirigidos acíclicos se les llaman DAG. Estos surgen de forma natural en los problemas de horarios, donde una arista dirigida (x, y) indica que la actividad de x debe ocurrir antes que y. Una operación llamada *ordenación topológica* ordena los vértices de un DAG según restricciones de precedencia en la ejecución o procesamiento de dichos elementos. La ordenación topológica es típicamente el primer paso para cualquier algoritmo sobre un DAG.

Nota. Se muestra una aproximación de comprobación de Aciclicidad para grafo dirigidos.

1.11. Conexión en grafos

Una de las propiedades más elementales de las que puede gozar cualquier grafo es que sea conexo.

1.11.1. Vértices conectados

Dos vértices de un grafo se dice que están conectados cuando existe un camino entre ambos. Es decir,

$$u \text{ y } v \text{ están conectados} \iff \exists \mu = \langle u, v \rangle$$

μ es un camino que une al vértice u con el v.

1.11.2. Grafos conexos

Un grafo se dice que es conexo si cada par de sus vértices están conectados. Es decir,

$$G \text{ es conexo} \iff \forall u, v : \exists \mu = \langle u, v \rangle$$

En caso contrario, diremos que G es un grafo desconexo.

Grafo conexo G_1 Grafo desconexo G_2

Figura 1.17: Ejemplos de grafos conexos

1.11.3. Proposición

Dado un grafo, la relación "estar conectado con" definida en el conjunto de sus vértices es una relación de equivalencia.

Demostración

Sea el grafo $G = (V, A)$ y definimos el conjunto V de sus vértices la siguiente relación

$$u \mathcal{R} v \iff u \text{ está conectado con } v$$

Veamos que esta relación es de equivalencia.

(a) *Reflexividad*. Sea u cualquiera de V. Entonces, el camino $\mu = \langle u, u \rangle$ conecta u con u, luego

$$\forall u \in V \; ; u \mathcal{R} u$$

es decir $\mathcal{R}$ es reflexiva.

(b) *Simetría*. Sea u y v dos elementos cualesquiera de V. Entonces,

$$u\mathcal{R}v \iff \exists\mu = \langle u, v\rangle \Longrightarrow \exists\mu' = \langle u, v\rangle \iff v\mathcal{R}u$$

luego,

$$\forall u, v \in V; u\mathcal{R}v \Longrightarrow v\mathcal{R}u$$

o sea, $\mathcal{R}$ es simétrica.

(c) *Transitividad*. Si u, v y w son tres vértices cualesquiera de G, entonces

$$\left.\begin{array}{l} u\mathcal{R}v \iff \exists\mu_1 = \langle u, v\rangle \\ v\mathcal{R}w \iff \exists\mu_2 = \langle v, w\rangle \end{array}\right\} \Longrightarrow \exists\mu = \langle u, w\rangle \iff u\mathcal{R}w$$

Bastaría, pues, con unir los caminos μ_1 y μ_2. Por lo tanto

$$\forall u, v, w; u\mathcal{R}v \wedge v\mathcal{R}w \Longrightarrow u\mathcal{R}w$$

es decir, $\mathcal{R}$ es transitiva.

1.11.4. Componentes conexas de un grafo

Dado un grafo $G = (V, A)$, las clases de equivalencia en el conjunto de sus vértices, V, por la relación de equivalencia "estar conectado con" reciben el nombre de componentes conexas de G.

Obsérvese que de esta forma un grafo no conexo G puede ser "partido" por la relación anterior en subgrafos conexos que son las citadas componentes conexas de G.

Ejemplo. El conjunto de vértices del grafo G_2 de la figura 2.15 es

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

y si consideramos en él la relación de equivalencia definida en la proposición anterior, las clases de equivalencia serán

$$[v_1] = \{v_1, v_2\} = [v_2]$$

$$[v_3] = \{v_3, v_4, v_5\} = [v_4] = [v_5]$$

Por lo tanto, el grafo G_2 tiene dos componentes conexas que son los subgrafos H_1 y H_2 cuyos conjuntos de vértices son $[v_1]$ y $[v_3]$, es decir,

$$H_1 = (\{v_1, v_2\}, \{v_1v_2\})$$

$$H_2 = (\{v_3, v_4, v_5\}, \{v_3v_4, v_3v_5, v_4v_5\})$$

Ejemplo. Demostraremos que en un grafo conexo $G = (V, A)$ se verifica: $|V| - 1 \leq |A|$

Solución
Utilizaremos la inducción sobre el número de vértices de G.
Paso básico. Si $|V| = 1$, entonces $|A| = 0$, luego

$$|V| - 1 = 1 - 1 = 0 = |A|$$

Paso inductivo. Supongamos que la desigualdad es cierta para $|V| = p$ con $p > 1$ y veamos que también es cierta para $|V| = p + 1$.

En efecto, sea u un vértice cualquier de G. Como el número de vértices, p, es mayor que 1, habrá otro vértice, v en G distinto de u y, al ser G conexo, deberá existir, al menos, un camino entre u y v, luego $gr(v) \geq 1$.

- Si $gr(u) = 1$ y a es la única arista que tiene a u como extremo, entonces el grafo

  $$(V \setminus \{u\}, A \setminus \{a\})$$

  es conexo y tiene p vértices. Por la hipótesis de inducción,

  $$|V \setminus \{u\}| - 1 \leq |A \setminus \{a\}|$$

  es decir,

  $$|V| - 2 \leq |A| - 1$$

  de donde,

  $$|V| - 1 \leq |A|$$

- Si $gr(u) > 1$, $\forall u \in V$, entonces por el teorema 2.9.4

  $$2|V| \leq \text{Suma de los grados de los vértices de } G = 2|A|$$

  o sea, $|V| \leq |A|$, de aquí que

  $$|V| - 1 < |A|$$

  Por el *primer principio de inducción matemática*,

  $$|V| - 1 \leq |A|$$

1.11.5. Puntos de corte

Dado un grafo conexo $G = (V, A)$, un vértice u de G se llama punto de corte cuando el subgrafo G_u cuyos vértices son los de $V \setminus \{u\}$ y cuyas aristas son todas las de A cuyos vértices están en $V \setminus \{u\}$ no es conexo.

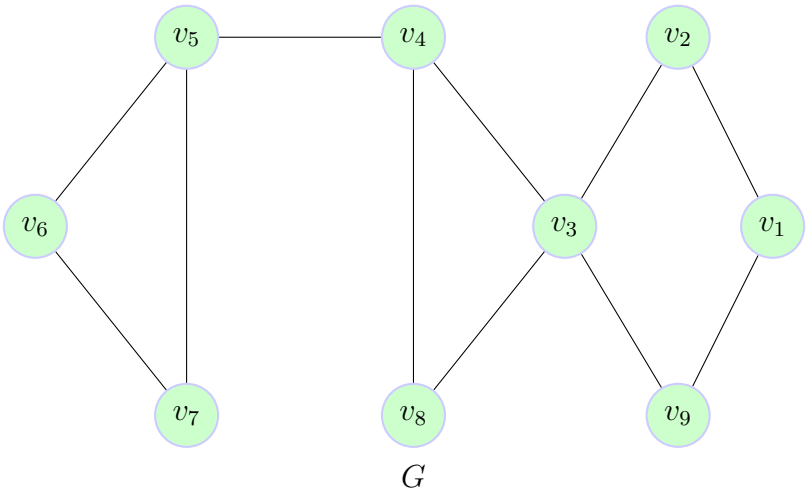
Una consecuencia de esto es que uno de los conjuntos de corte produce exactamente dos componentes.

A veces es útil para indicar un conjunto límite fijado por la partición de los vértices que induce. Si V denota al conjunto de vértices de G y si P es el subconjunto de vértices en una componente de G inducida por el conjunto o punto de corte, entonces el límite del conjunto se puede especificar por $(P, \overline{P})$ donde $\overline{P} = V - P$.

Nota. El algoritmo implementado verifica si existe algún punto de corte en el grafo. Hay otra versión que muestra dichos puntos de cortes del grafo.

1.11.6. Puentes

Dado un grafo conexo $G = (V, A)$, a cualquier arista "a" de G tal que el grafo $(V, A \setminus \{a\})$ no sea conexo, lo llamaremos puente.



G

Puntos de corte. Los vértices v_3, v_4 y v_5 ya que en la figura existen puntos que no pueden conectarse a través de ningún camino.

Puentes. El único puente que existe en el grafo propuesto es la arista v_4v_5 ya que en el grafo resultante existen vértices que no están conectados, es decir, no es conexo.

Si $G = (V, A)$ es un grafo conexo y a es una arista puente de G, entonces $(V, A \setminus \{a\})$ tiene exactamente dos componentes conexas.

Demostración

Llamemos v y w a los vértices que son extremos de la arista a. Y dividamos los vértices de G en dos clases:

1. El conjunto de vértices V_1, formado por aquellos para los que existe un camino que los conecta con v *sin usar* la arista a (esto es, sin pasar por el vértice w). Entre estos está, por supuesto, el propio vértice v.

2. El conjunto de vértices V_2 de los vértices que *necesariamente* han de usar la arista a para conectarse a v. Entre ello está w.

Que esto es una partición de los vértices de G es obvio. Pero, además, la intersección de V_1 y V_2 es vacía: si un vértice $x \in V$ estuviera en V_1 y en V_2 a la vez, a no sería arista puente, porque podríamos quitarla sin que se desconectara el grafo. Si ahora formamos el grafo $G = (V, A \setminus \{a\})$, sus dos componentes conexas son, precisamente, los vértices de V_1 (y sus aristas) y los vértices de V_2 (y sus aristas).

1.12. Estructura de partición

Un partición de un conjunto C de elementos de un cierto tipo es un conjunto de subconjuntos disjuntos cuya unión es el conjunto total C.
Esta estructura se ha adaptado, en este caso, para representar particiones de elementos del tipo Arista cuyo representación interna es de la forma:

$$Arista\ a\ =\ \langle x, y, z\rangle : \begin{cases} x \text{ el vértice origen} \\ y \text{ el vértice destino} \\ z \text{ el coste de la arista} \end{cases}$$

Una relación de equivalencia sobre los elementos de un conjunto C define una partición C y, viceversa, cualquier partición de C define una relación de equivalencia sobre sus elementos, de tal forma que cada miembro de la partición es una clase de equivalencia. Así pues, para cada subconjunto o clase podemos elegir cualquier elemento como representante canónico de todos sus miembros.

A continuación se da la especificación del TAD que representará la estructura de partición de nuestro sistema:

La estructura de datos alternativa que se ha usado para la implementación de ciertas operaciones del TAD ha sido la de Bosque de Árboles.

La causa por la que operación *Unión()* no se ejecuta en un tiempo constante es que para cada elemento se almacena el representante de su clase y esto obliga a modificar el representante de todos los elementos de uno de los conjuntos unidos. Podemos cambiar la estructura de datos para conseguir que la unión sea $O(1)$, pero a costa de empeorar el tiempo de ejecución de la operación *Encontrar()*, ya que no existe una estructura de

datos que permita ejecutar simultáneamente estas dos operaciones en un tiempo constante.

1.13. Disperso frente a denso

Se define la densidad de un grafo como el promedio del grado de los vértices, o $2A \setminus V$. Un grafo *denso* es un grafo cuyo promedio de grado de vértices es proporcional a V; por otro lado un grafo disperso es un grafo cuyo complementario es denso. En otras palabras, consideramos que un grafo es denso si A es proporcional a V^2 sino será disperso. Esta definición asintótica no es necesariamente significativa para un grafo en particular, pero la distinción es clara en general: Un grafo que cuenta con miles de vértices y miles de decenas de aristas es claramente disperso, y un grafo que tiene cientos de vértices y millones de aristas es claramente denso. Podríamos contemplar un grafo disperso con miles de millones de vértices, pero un grafo denso con miles de millones de vértices tendrá un numero abrumador de aristas.

Saber si un grafo es disperso o denso, es generalmente un factor clave en la selección de un algoritmo eficiente para procesar el grafo. Por ejemplo, para un problema en concreto, podríamos desarrollar un algoritmo que tomara alrededor de V^2 y otro que tomara alrededor de $A \lg A$ pasos. Estas fórmulas nos dicen que el segundo algoritmo sería mejor para grafos dispersos, mientras que el primero sería preferible para grafos densos. Por ejemplo, un grafo denso con millones de aristas podría tener sólo unos miles de vértices: en este caso V^2 y A serían comparables en valor, y el algoritmo de V^2 pasos sería 20 veces más rápido que el algoritmo de $A \lg A$ pasos. Por otro lado, un grafo disperso con millones de aristas también tendría millones de vértices, por lo que el algoritmo de $A \lg A$ pasos podría ser millones de veces más rápido que el algoritmo de V^2 pasos. Podríamos hacer concesiones específicas sobre la base del análisis de estas fórmulas con más detalle, pero por lo general es suficiente en la práctica usar los términos disperso y denso de manera informal para ayudarnos a comprender las características fundamentales de rendimiento.

Capítulo 2

Algoritmos de grafos

2.1. Caminos y ciclos de Euler

En 1736, Leonhard Euler publicó un trabajo[1] en el cual resolvería el que ha venido a llamarse "problema de los puentes de Königsberg". La ciudad Königsberg (llamada Kaliningrado, durante la época soviética) estaba dividida por el río Pregel en cuatro zonas: las dos orillas (A y C), la isla llamada Kneiphof (B) y la parte comprendida entre las dos bifurcaciones del río. En aquel entonces había siete puentes comunicando las distintas zonas tal y como se observa en la figura. Parece ser que uno de los pasatiempos de los ciudadanos de Königsberg consistía en buscar un recorrido que travesase cada puente exactamente una vez. Se pensaba que tal paseo no era posible debido a los múltiples intentos fallidos pero que nadie había podido demostrar tal imposibilidad. Fue Euler quién demostró que la existencia de un tal recorrido requeriría, en general, que a lo sumo dos de las zonas estuvieran unidas con el resto mediante un número impar de puentes, cada una de ellas. Para probarlo, Euler reemplazó, en primer lugar, el mapa de la ciudad por un diagrama más simple, donde se incluye la información más relevante del problema, y, en segundo lugar, dio un razonamiento que sobrepasa el diagrama particular.

[1] Solutio problematis ad geometriam situs pertinentis (1741) (La solución a un problema relativo a la geometría de la posición).

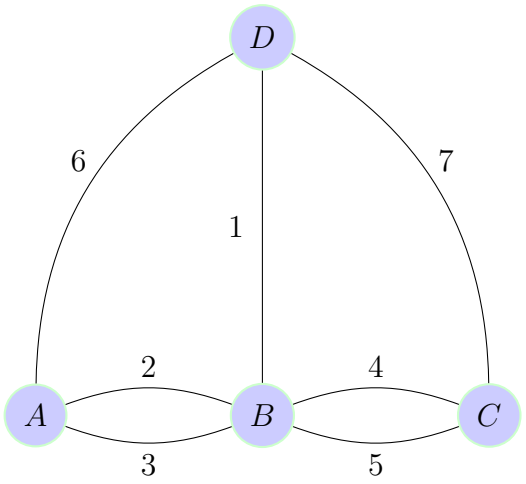


Figura 2.1: Problema de los puentes de Königsberg

Cabe mencionar la siguiente información:

- Euler no llegó a dibujar el diagrama de puntos y líneas que hoy conocemos como grafo (multigrafo) asociado al problema de los puentes de Königsberg.
- Euler enunció, pero no demostró, que la condición necesaria dada por él también era suficiente. Su demostración se debe a Hierholzer (1873), quien al parecer desconocía el trabajo anterior de Euler. Hierholzer utiliza la expresión "sistema de líneas entrelazadas" para referirse a lo que hoy conocemos por grafo.

La principal aplicación fue la siguiente:

- El problema del cartero chino (Kwan Mei-Ko (1960)). Dicho problema se emplea para hallar la ruta óptima para un gps, etc. Además dicho problema tiene un algoritmo (algoritmo de Edmonds) de resolución en tiempo polinómico. En la base de dicho algoritmo se encuentra la observación de Goodman y Hedetniemi respecto a la equivalencia entre encontrar un recorrido de cartero de peso mínimo y la búsqueda de un conjunto de aristas de peso total mínimo cuya duplicación de lugar a un multigrafo Euleriano.

2.1.1. Ciclo de Euler

Un ciclo de un grafo o multigrafo se dice de Euler si pasa todos los vértices recorriendo cada arista exactamente una vez.

2.1.2. Grafo euleriano

Un grafo que admita un ciclo de Euler se denomina grafo Euleriano.

2.1.3. Primer lema

Una condición necesaria para que un grafo o multigrafo sea Euleriano es que todos sus vértices sean de grado par.

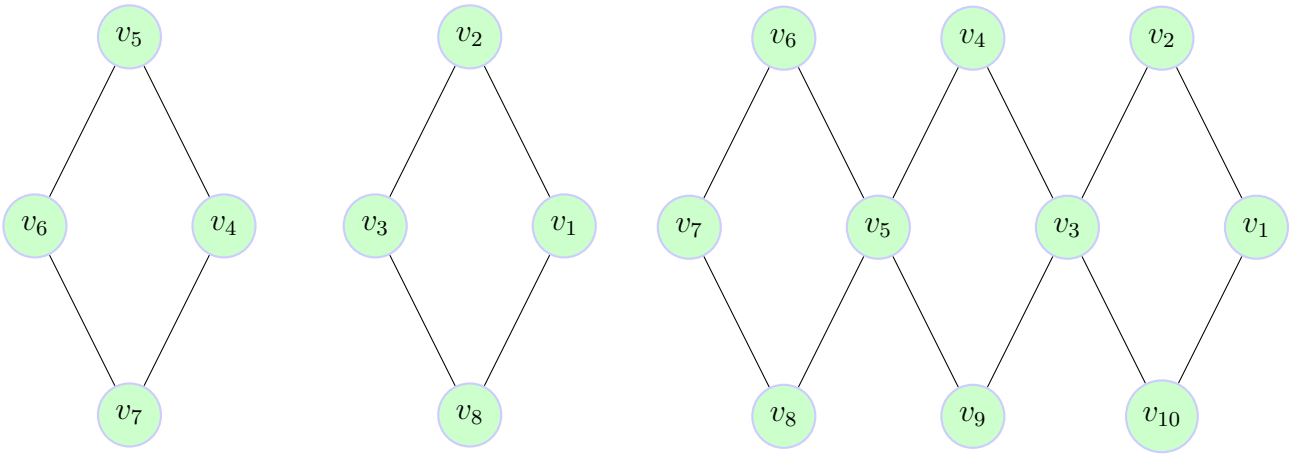
Demostración

En efecto, supongamos que G es un grafo Euleriano, es decir, supongamos que existe un ciclo de Euler, γ, en G. Sea v un vértice cualquiera de G. Veamos que tiene grado par.

- Si v no es el primer vértice de γ, cada una de las veces que el ciclo pase por v entrará y saldrá por dos aristas distintas de la vez anterior, luego contribuirá con 2 al grado de v.
- Si v es el primer vértice de γ, el ciclo γ contribuye con 2 al grado de v en cada una de las "visitas" que se realicen a v, salvo en la primera y en la última en la que añade 1 cada vez.

Por lo tanto, en cualquier caso, el grado de v es par.

Nota.



(a) Grafo no Euleriano (b) Grafo Euleriano

El grafo de la figura (a) nos muestra que la condición no es suficiente, es decir, existen grafos con todos los vértices de grado par y, sin embargo, no son eulerianos. Obsérvese que si *conectamos* el grafo, entonces si es euleriano (apartado (b) de la figura). En efecto, el ciclo

$$\gamma = \langle v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_5, v_9, v_3, v_4, v_{10}, v_1 \rangle$$

es de Euler.

Nota. En el Primer Lema hemos visto que

Si G es un Grafo Euleriano, entonces todos sus vértices son de grado par.

de donde negando ambos miembros, y teniendo en cuenta la equivalencia lógica entre una proposición condicional y su contrarrecíproca, tendremos

Si existe algún vértice de grado impar, entonces G no es Euleriano.

es decir, si en un Grafo G existe, al menos, un vértice de grado impar, entonces no es Euleriano.

2.1.4. Camino de Euler

Se dice que un camino de un grafo o multigrafo es de Euler si pasa por todos los vértices del mismo, recorriendo cada arista del mismo exactamente una vez.

Obsérvese que un *camino de Euler* en un grafo G puede entenderse también como una forma de dibujar el grafo sin levantar el lápiz del papel y sin pintar dos veces la misma arista.

Nota. El algoritmo implementado para el Camino de Euler comprueba la existencia de dicho camino o no. Hay otra versión que muestra en la salida estándar del sistema uno de los caminos posibles hallados.

2.1.5. Segundo lema

Una condición necesaria para que un grafo o multigrafo admita un camino de Euler es que el número de vértices de grado impar sea 2 o ninguno.

Demostración

Sea $G = (V, A)$ un grafo con un camino de Euler $\gamma = \langle u, u_1, u_2, \ldots, u_p, v\rangle$.
Tomamos un punto w que no pertenezca a V y sea $G' = (V', A')$ un grafo tal que

$$V' = V \cup \{w\}$$

$$A' = A \cup \{uw, vw\}$$

es decir, el grafo obtenido añadiendo el nuevo punto como vértice al grafo original y las dos aristas adyacentes al mismo y a los extremos u y v.
El ciclo

$$\langle w, u, u_1, \ldots, u_p, v, w\rangle$$

es de Euler en G', de aquí que G' sea un grafo euleriano y aplicando el *primer lema*, tengamos que todos sus vértices son de grado par.
Pues bien, si x es cualquier vértice de G distinto de u y de v, entonces

$$gr_G(x) = gr_{G'}(x)$$

luego el grado de x en el grafo G es par. Por otra parte,

$$gr_G(u) = gr_{G'}(u) - 1 \Longrightarrow gr_G(u) \text{ es impar}$$

y

$$gr_G(v) = gr_{G'}(v) - 1 \Longrightarrow gr_G(v) \text{ es impar}$$

luego los únicos dos vértices de grado impar son u y v.

Nota. En el segundo lema, hemos visto que

> *"Si G es un grafo con un camino de Euler, entonces el número de vértices de grado impar es 2 o ninguno"*

Si ahora negamos ambos miembros, y tenemos en cuenta la equivalencia lógica entre una proposición condicional y su contrarrecíproca, tendremos

> *"Si el número de vértices de grado impar es distinto de 2, entonces G no tiene ningún camino de Euler"*

2.1.6. Tercer lema

Si G es un grafo en el que todos sus vértices tienen grado par, entonces para cada par de vértices adyacentes de G, puede encontrarse un ciclo que contiene a la arista que forman ambos.

Demostración

Sean u y v dos vértices adyacentes de G y sea γ un camino que comienza en u y continúa por la arista uv.

Cada vez que γ llega a un vértice w distinto de u, continuamos el camino por una arista que no esté en γ, si w es igual a u damos por terminado el proceso. Dado que los grados de los vértices son pares por hipótesis, cada vez que el camino γ pasa por un vértice utiliza dos aristas con un extremo en el mismo. Como el número de aristas y el de vértices es finito, el camino γ acaba por volver a u y γ es, según la construcción hecha, un ciclo.

2.2. Grafos ponderados

2.2.1. Caminos más cortos desde un vértice: Dijkstra

El algoritmo de Dijkstra, aplicado a un (di)grafo ponderado con pesos no negativos (no funciona para pesos negativos, en cuyo caso hay que utilizar otro procedimiento llamada algoritmo (BELLMAN-FORD)[2.2.2], se fundamenta en algo evidente: si se pretende encontrar la distancia más corta desde un punto A a otros puntos, y en un momento determinado se sabe la distancia más corta desde A a un punto D, entonces se puede tratar de mejorar las distancias parciales conocidas desde A a los vértices adyacentes a D comparándolas con las distancias que se obtienen yendo desde A a D y desde D a estos vértices, tal como ilustra la siguiente Figura.

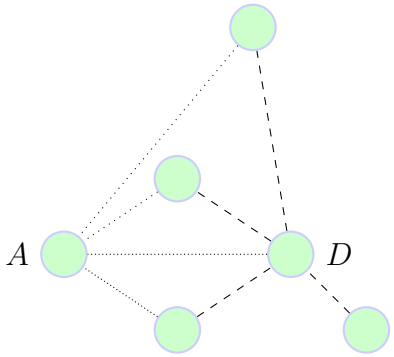


Figura 2.2: Se cotejan las distancias desde el vértice base D

La cristalización matemática de esta idea consiste en lo siguiente. Se define una función $l(x)$ que a cada vértice x le asocie la distancia parcial conocida desde el vértice origen prefijado (llamémoslo A, por comodidad) hasta el propio vértice x. En cada etapa, cuando se haya determinado la distancia exacta de A a un vértice D, el cual ejerce las labores de nueva base o pivote, se actualizan los valores $l(y)$ para vértices y adyacentes a D según la fórmula $l(y) = \min\{l(y), l(D) + peso(\{D, y\})\}$; de manera que si el camino desde A hasta D más la arista que va de D a y resulta más económico (esto es, más corto) que el que antes se conocía yendo desde A hasta y sin pasar por D (distancia que marcaba el valor $l(y)$), se actualizaba el valor de $l(y)$ según el nuevo camino. A la hora de poder reconstruir los caminos que dan las distancias más cortas desde A, es necesario asimismo guardar la arista $\{D, y\}$ por la que se llega al vértice y.

En definitiva, el algoritmo de Dijkstra funciona así:

1. Se toma el vértice A desde el que se van a hallar las distancias más cortas a los restantes vértices.
2. Se inicializan los valores $l(x)$ a infinito, para $x \neq A$, y $l(A) = 0$.

3. Se toma como primer vértice base (o pivote) a A, que será asimismo el vértice raíz del árbol recubridor que dará las distancias más cortas hasta A.

4. Ahora, utilizando A como vértice base, se actualizan los valores $l(x)$ de aquellos vértices adyacentes a A según la fórmula $l(x) = \min\{l(x), l(A) + peso(\{A, x\})\}$.

5. Se toma como nuevo vértice base uno de entre aquellos cuyas distancias a A sea mínima, y se almacena la arista por la que se ha llegado a dicho vértice.

6. Sucesivamente, utilizando el vértice base D correspondiente, se actualizan los valores $l(x)$ de aquellos vértices adyacentes a D según la fórmula $l(x) = \min\{l(x), l(D) + peso(\{D, x\})\}$; de manera que se puede elegir al nuevo vértice base y la arista mediante la cual se ha llegado al mismo.

7. El proceso termina cuando ya se ha calculado las distancias más cortas a todos los vértices, requiriendo tantas etapas como vértices tiene el grafo menos 1.

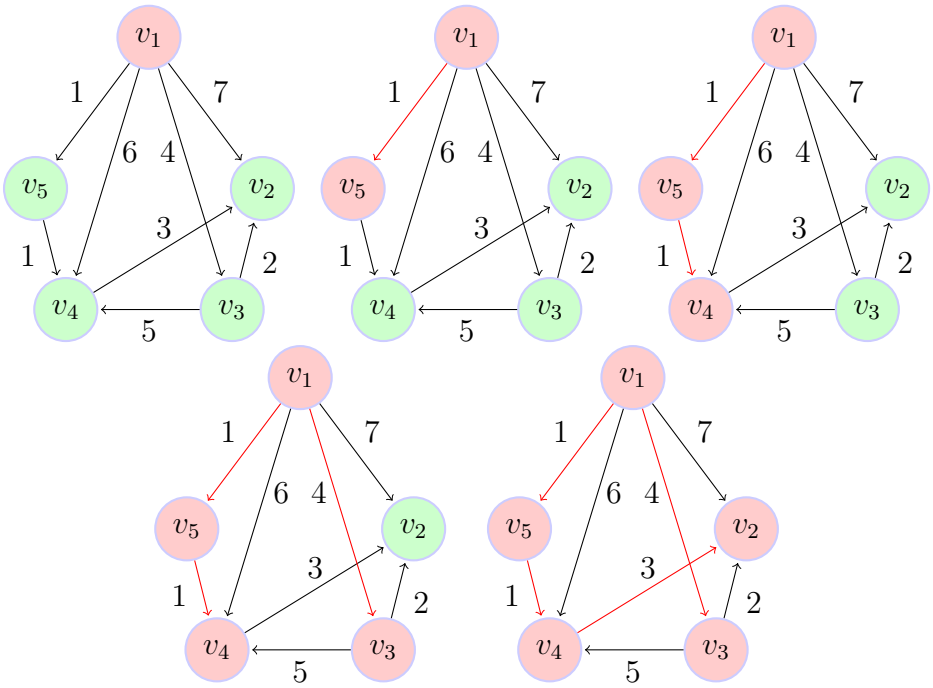


Figura 2.3: Un grafo dirigido ponderado y los pasos que realiza el algoritmo de Dijkstra en el grafo. En cada paso se colorean los nuevos vértices y las nuevas aristas procesadas.

2.2.2. Caminos más cortos desde un vértice: Bellman-Ford

El algoritmo de Bellman-Ford resuelve el problema del camino más corto posible desde un vértice origen de un grafo en el caso general de que los pesos de las aristas puedan ser negativos. Dado un grafo ponderado y dirigido $G = (V, A)$ con un vértice origen s y la función de coste o ponderación $w : A \to \mathbb{R}$, el algoritmo de Bellman-Ford devuelve

un valor booleano que indica si existe o no un ciclo de coste negativo que fuera accesible desde el vértice origen. Si es que existe un ciclo, el algoritmo indica que no existe solución. Si no hay ciclo de este tipo, el algoritmo produce los caminos más cortos asociados a los pesos.

El algoritmo relaja las aristas que va procesando, disminuyendo progresivamente la estimación del peso del vértice a través de los caminos más cortos posibles desde s a cada vértice $v \in V$ hasta que se logre el camino real de peso más corto. El algoritmo devuelve **true** si y sólo si el grafo no contiene ciclos de peso negativo que sean accesibles desde el vértice origen.

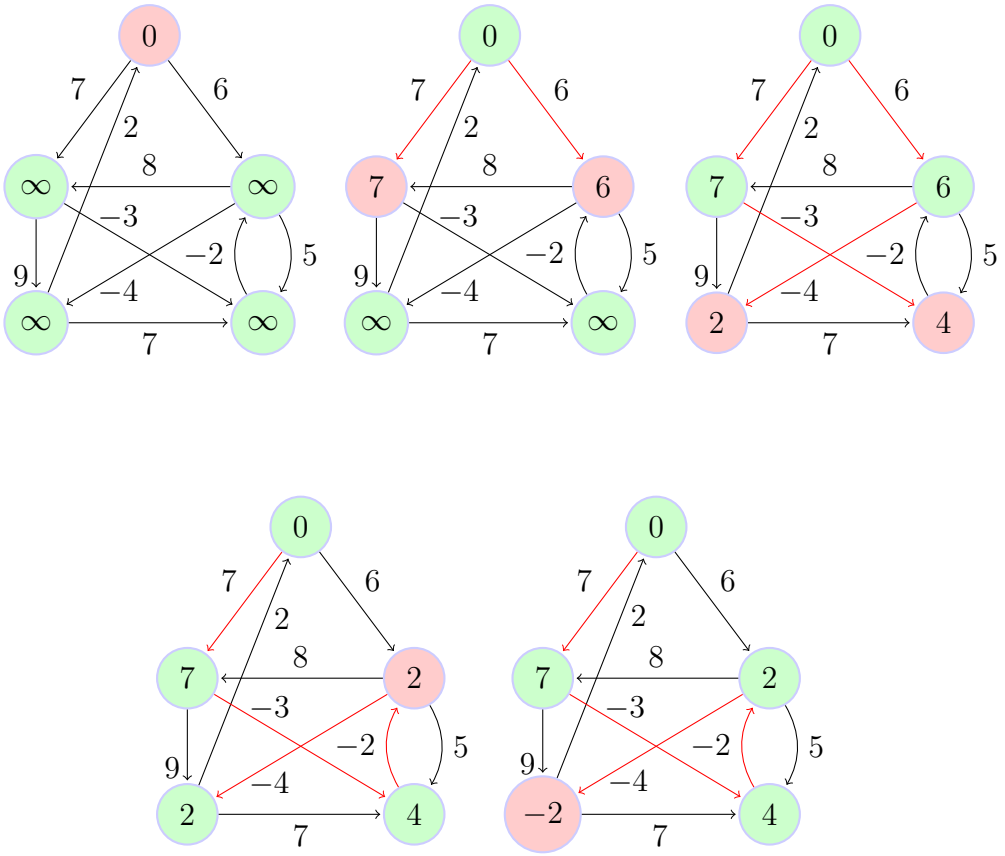


Figura 2.4: La ejecución del algoritmo de Bellman-Ford.

2.2.3. Caminos más cortos: Floyd

Supóngase que se tiene un grafo no dirigido $G = (V, A)$ en el cual cada arista tiene un peso no negativo. El problema es encontrar el camino de longitud más corta entre v y w para cada par ordenado de vértices (v, w)

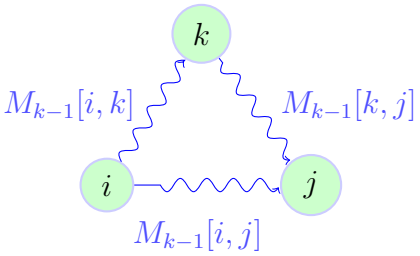
Podría resolverse este problema por medio del algoritmo de Dijkstra, tomando por turno cada vértice como vértice origen, pero una forma más directa de solución es mediante el algoritmo creado por R. W. Floyd. Por conveniencia se supone que los vértices en V están numerados en la forma $v_1, v_2, \ldots, v_n$. El algoritmo de Floyd usa una matriz M de dimensión $V \times V$ en la que se calculan las longitudes de los caminos más cortos. Inicialmente se hace $M[i, j] = A(i, j)$, $\forall i \neq j$. Si no existe una arista que vaya de i a j, se supone que $M[i, j] = \infty$.

Después, se realizarán n iteraciones para la matriz M para hallar los posibles caminos más cortos tomando como referencia todos los vértices entre si. Al final de la k-ésima iteración, $M[i, j]$ tendrá por valor la longitud más pequeña de cualquier camino que vaya desde el vértice i hasta el vértice j y que no pase por un vértice con número mayor que k. Esto es, i y j, los vértices extremos del camino, pueden ser cualquier vértice,

pero todo vértice intermedio debe ser menor o igual que k.

En la k-ésima iteración se aplica la siguiente fórmula para calcular M.

$$M_k[i,j] = \min \begin{cases} M_{k-1}[i,j] \\ M_{k-1}[i,k] \ + \ M_{k-1}[k,j] \end{cases}$$



El subíndice k denota el valor de la matriz M después de la k-ésima iteración; no indica la existencia de n matrices distintas.
Para obtener el valor de $M_K[i,j]$, se compara $M_{k-1}[i,j]$, que es el costo de ir de i a j sin pasar por k o cualquier otro vértice con numeración mayor, con $M_{k-1}[i,k] + M_{k-1}[k,j]$, que es el costo de ir primero de i a k y después de k a j, sin pasar a través de un vértice con un coste mayor que k. Si el paso del vértice k produce un camino mas económico que el $M_{k-1}[i,j]$, se elige este costo para $M_k[i,j]$. Además como $M_k[i,k] = M_{k-1}[i,k]$ y $M_k[k,j] = M_{k-1}[k,j]$, ninguna entrada con cualquier subíndice igual a k cambia durante la k-ésima iteración.

2.2.4. Existencia de caminos: Warshall

Supóngase que la matriz de costes M es sólo la matriz de adyacencia para el grafo dirigido dado. Esto es, $M[i,j] = 1$ si hay un camino de longitud igual o mayor que 1 de i a j, y 0 en otro caso. Al resultado del algoritmo de Warshall se le conoce a menudo como cierre transitivo de la matriz de adyacencia.

Demostremos como se calcula el cierre transitivo por inducción sobre k.

Después de la primera iteración del bucle, la matriz tiene un 1 en la fila i y la columna j si y sólo si existen los respectivos caminos $i-j$ o $i-0-j$. Esto nos lleva a la hipótesis inductiva siguiente: La iteración k-ésima del bucle establece el bit de la fila i y la columna j de la matriz a 1 si y sólo si existe un camino dirigido desde i a j en el grafo que no incluya ningún vértice con un índice (o coste) mayor que k (excepto, posiblemente, los extremos i y j). Como se acaba de argumentar, la condición es cierta cuando k es 0, después de la primera iteración del bucle. Suponiendo que es cierto para la iteración k-ésima del bucle, hay un camino de i a j que no incluye ningún vértice con índices superiores a $k+1$ si y sólo si:

(i) hay un camino de i a j que no incluye ningún vértice con índices mayores que k, en cuyo caso $M[i,j]$ se encuentra en una iteración anterior del camino (por la hipótesis inductiva); o

(ii) hay un camino de de i para $k+1$ y un camino de $k+1$ a j, ninguno de los cuales incluye cualquier otro vértice con índices (o costes) mayores que k (excepto los extremos), en cuyo caso $M[i,k+1]$ y $M[k+1,j]$ se establecieron con anterioridad a 1 (por hipótesis), por lo que el bucle interno establece $M[i,j]$.

2.2.5. Árboles de expansión mínimos

Sea $G=(V,A)$ un grafo conexo no dirigido en donde V es el conjunto de nodos o vértices y A es el conjunto de aristas. Cada arista posee una coste no negativo. El problema consiste en hallar un subconjunto C de las aristas de G tal que utilizando solamente las aristas de C, todos los vértices deben quedar conectados, y además la suma de las longitudes de las aristas de C debe ser tan pequeña como sea posible. Dado que C es conexo, debe existir al menos una solución. Si C tiene aristas de coste 0, pueden existir varias soluciones cuyo coste total sea el mínimo, pero que tengan números distintos de aristas. En este caso, dadas dos soluciones de igual longitud total, preferimos la que contenga menos aristas. Incluso con esta condición el problema puede tener varias soluciones diferentes y de igual valor. El problema, entonces, consiste en hallar un subconjunto C de las aristas cuyo coste total sea el menor posible.

Sea $G'=(V,C)$ el grafo parcial formado por los vértices de G y las aristas de C, y supongamos que en V hay n vértices. Un grafo conexo con n nodos debe tener al menos $n-1$ aristas, así que éste es el número mínimo de aristas que puede haber en C. Por otra parte, un grafo con n nodos y más de $n-1$ aristas contiene al menos un ciclo. Por tanto, si G' es conexo y C tiene más de $n-1$ aristas, se puede eliminar al menos una arista sin desconectar G', siempre y cuando seleccionemos una aristas que forme parte de un ciclo. Esto hará o bien que disminuya el coste total de las aristas de C, o bien que el coste total quede intacto (si hemos eliminado una arista de coste 0) a la vez que disminuye el número de aristas que hay en C. En ambos casos, la nueva solución es preferible a la anterior. Por tanto, un conjunto C con n o más aristas no puede ser óptimo. Se sigue que C debe tener exactamente $n-1$ aristas, y como G' es conexo, tiene que ser un árbol.

El grafo G' se denomina Árbol de Expansión Mínimo para el grafo G. Este problema tiene muchas aplicaciones. Por ejemplo, supongamos que los nodos de G representan ciudades, y sea el coste de una arista $a=ij$ el coste de tender una línea telefónica desde i hasta j. Entonces un Árbol de Expansión Mínimo de G se corresponde con la red más barata posible para dar servicio a todas las ciudades en cuestión, siempre y cuando sólo se puedan utilizar conexiones directas entre ciudades (en otras palabras, siempre y cuando no se permita establecer centrales telefónicas en el campo, entre las ciudades). Relajar esta condición es equivalente a permitir que se añadan vértices adicionales, auxiliares, a G para obtener soluciones más baratas.

2.2.6. Árboles de expansión mínimo: Kruskal

El conjunto de aristas C está vacío inicialmente. A medida que progresa el algoritmo, se van añadiendo aristas a C. Mientras no haya encontrado una solución, el grafo parcial formado por los vértices de G y las aristas de C consta de varios componentes conexos. (Inicialmente, cuando C está vacío, cada nodo de G forma una componente conexa distinta trivial). Los elementos de C que se incluyen en una componente conexa dada forman un Árbol de Expansión Mínimo para vértices de esta componente. Al final del algoritmo, sólo queda una componente conexa, así que C es un Árbol de Expansión Mínimo para todos los vértices de G.

Para construir componentes conexas más y más grandes, examinamos las aristas de G por orden creciente de costes. Si una arista une a dos vértices de componentes conexas distintas, se lo añadimos a C. En consecuencia, las dos componentes conexas forman ahora una única componente. En caso contrario, se rechaza la arista: une a dos vértices de la misma componente conexa, y por tanto no se puede añadir a C sin formar un ciclo (porque las aristas de C forman un árbol para cada componente). El algoritmo se detiene cuando sólo queda una componente conexa.

Para implementar el algoritmo, tenemos que manejar un cierto número de conjuntos, a saber, los vértices de cada componente conexa. Es preciso efectuar rápidamente dos operaciones: $Encontrar(x)$, que nos dice en qué componente conexa se encuentra el vértice x, y $Union(A, B)$, para fusionar dos componentes conexas. Por tanto, utilizamos estructuras de partición. Además de la nueva estructura auxiliar para el trabajo con las distintas componentes conexas que se pueden generar en la ejecución del algoritmo se ha introducido también un algoritmo de ordenación rápida (Quicksort), que para este caso servirá para ordenar las aristas de las distintas componentes de trabajo de menos a mayor coste por arista de sus respectivos conjuntos de aristas de la componente.

Algoritmo de Ordenación Rápida: Quicksort

Quicksort es un método de divide y vencerás para el ordenamiento de ejemplares. Funciona mediante la partición de un array en dos partes, para a continuación ordenar las dos partes de manera independiente. Como se verá, la posición exacta de la partición depende del orden inicial de los elementos para la entrada de ejemplares. La clave del método es el proceso de partición, que reordena el array para realizar las siguientes tres condiciones:

- El elemento $vector\_kruskal[i]$ está en su lugar definitivo en el array para algunos i.
- Ninguno de los elementos de $vector\_kruskal[l], \ldots, vector\_kruskal[i-1]$ es mayor que $vector\_kruskal[i]$.
- Ninguno de los elementos de $vector\_kruskal[i+1], \ldots, vector\_kruskal[y]$ es menor que $vector\_kruskal[y]$.

Utilizaremos la siguiente estrategia general para la aplicación de la partición. En primer lugar, elegimos una forma arbitraria para el $vector\_kruskal[y]$ que será el *elemento de división* - aquel que irá colocado en la última posición. A continuación, se recorre desde el extremo izquierdo del array hasta que encontramos un elemento mayor que el elemento de división, y luego recorremos desde el extremo derecho del array hasta que encontramos un ejemplar de valor menor que el elemento de división. Los dos elementos que se han encontrado en la primera iteración, estarán fuera de la partición original y por lo tanto habrá que intercambiar sus posiciones. Continuando de esta manera, nos aseguramos de que no hay elementos del array en la izquierda que sean mayores que el elemento de división, y de que no hay elementos del array en la derecha que sean menores que el elemento de división.

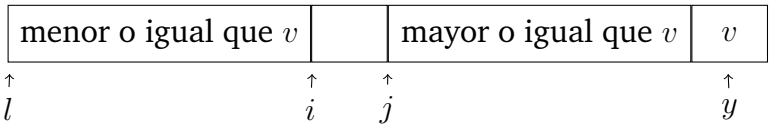


Figura 2.5: Estructura de partición para Quicksort

En este caso, v se referirá al valor del elemento de división (coste de la arista), i representará el puntero que apunta a la zona izquierda del array, y j para el puntero derecho. Como se indica en el diagrama, lo mejor es detener la exploración a la izquierda de los elementos mayores o iguales que el elemento de división y de igual forma con los elemento de la derecha que sean menores o iguales que el elemento de división, a pesar de que esta política pueda crear intercambios de elementos iguales al elemento de división. Cuando se crucen los punteros, todo lo que tenemos que hacer para completar el proceso de partición será el intercambio de $vector\_kruskal[y]$ con el elemento más a la izquierda de la parte derecha del array (el elemento apuntado por el puntero de la izquierda).

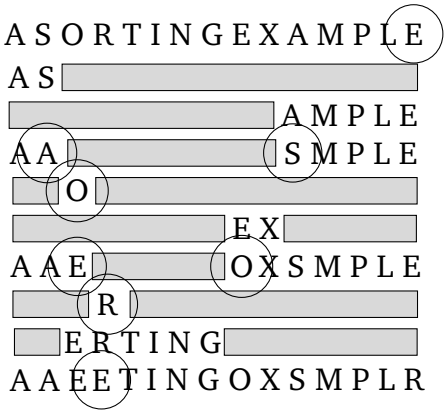


Figura 2.6: Particionamiento en Quicksort

2.2.7. Árboles de expansión mínimo: Prim

En el algoritmo de Kruskal, la función de selección escoge las aristas por orden creciente de costes, sin preocuparse demasiado por su conexión con las aristas seleccionadas anteriormente, salvo que se tiene cuidado para no formar nunca un ciclo. El resultado es un bosque de árboles que crece al azar, hasta que finalmente todas las componentes del bosque se fusionan en un único árbol. En el algoritmo de Prim, por otro parte, el Árbol de Expansión Mínimo crece de forma natural, comenzando por una raíz arbitraria. En cada fase se añade una nueva rama al árbol ya construido; el algoritmo se detiene cuando se han alcanzado todos los nodos.

Sea B un conjunto de vértices, y sea C un conjunto de aristas. Inicialmente, B contiene un único vértice arbitrario, y C está vacío. En cada paso, el algoritmo de Prim busca la arista más corta posible $\{u, v\}$ tal que $u \in B$ y $v \in V \setminus B$. Entonces añade v a B y $\{u, v\}$ a C. Se prosigue mientras $B \neq V$.

2.3. Búsqueda en grafos

A menudo para aprenden las propiedades estructurales de un grafo, se tienen que examinar todos sus vértices y aristas. La determinación de algunas propiedades simples de un grafo - por ejemplo el número de los grados de los vértices - es fácil si sólo examinamos cada arista (en cualquier orden establecido). Muchas propiedades de los grafos se establecen con sus recorridos o caminos, así que es natural hallar una forma para aprender dichas propiedades de enrutamiento desde un vértice hacia sus aristas y viceversa.

La Búsqueda en Grafos de esta manera es equivalente a una exploración de un laberinto. En concreto, las calles o pasadizos corresponden con las aristas del grafo y los puntos donde las calles se cruzan en el laberinto a los vértices del grafo. Cuando un programa cambia el valor de una variable desde el vértice v al vértice w a causa de una arista vw, lo vemos de una forma equivalente a que la persona en el laberinto se ha movido desde v hasta w.

La Búsqueda en Profundidad (DFS) es un algoritmo clásico y versátil que se utiliza para resolver la conectividad y muchos otros problemas de procesamiento de grafos. Los algoritmos básicos admiten dos implementaciones simples: una de ellas es recursiva y la otra usa una estructura de tipo LIFO (una pila). Sustituyendo la pila con una estructura FIFO (una cola) se obtiene otro algoritmo clásico, la Búsqueda en Anchura (BFS), que se utiliza para resolver otros tipos de problemas de procesamiento de grafos en relación con los caminos más cortos.

2.3.1. Búsqueda en profundidad (Depth-First Search)

Sea $G = (V, A)$ un grafo no dirigido formado por todos aquellos vértices que deseamos visitar. Supongamos que de alguna manera es posible marcar un vértice para mostrar que ya ha sido visitado.

Para efectuar un recorrido en *profundidad* del grafo, se selecciona cualquier vértice $v \in V$ como punto de partida. Se marca este vértice para mostrar que ya ha sido visitado. A continuación, si hay un nodo adyacente a v que no haya sido visitado todavía, se toma este vértice como nuevo punto de partida y se invoca recursivamente al procedimiento de recorrido en profundidad. Al volver de la llamada recursiva, si hay otro vértice adyacente a v que no haya sido visitado, se toma este vértice como punto de partida siguiente, se vuelve a llamar recursivamente al procedimiento, y así sucesivamente. Cuando están marcados todos los vértices adyacentes a v, el recorrido que comenzara en v ha finalizado. Si queda algún vértice de G que no haya sido visitado, tomamos cualesquiera de ellos como nuevo punto de partida, y volvemos a invocar el procedimiento. Se sigue así hasta que estén marcados todos los vértices de G.

El algoritmo se llama de búsqueda o recorrido en profundidad porque inicia tantas llamadas recursivas como sea posible antes de volver de una llamada. La recursión sólo se detiene cuando la exploración del grafo se ve bloqueada y no puede proseguir. En ese momento, la recursión "retrocede" para que sea posible estudiar posibilidades alternativas a niveles más elevados. Si el grafo corresponde a un juego, se puede pensar intuitivamente que esto es una búsqueda que explora el resultado de una estrategia particular con tantas jugadas anticipadas como sea posible, antes de explorar los alrededores para ver qué tácticas alternativas pudieran estar disponibles.

Nota. Búsqueda en Profundidad para grafos dirigidos. El algoritmo es esencialmente el mismo que para los grafos no dirigidos; la diferencia reside en la interpretación de la palabra "adyacente". En un grafo dirigido, el vértice w es adyacente al vértice v si existe la arista dirigida (v, w). Si existe (v, w) pero (w, v) no existe, entonces w es adyacente a v, pero v no es adyacente a w.

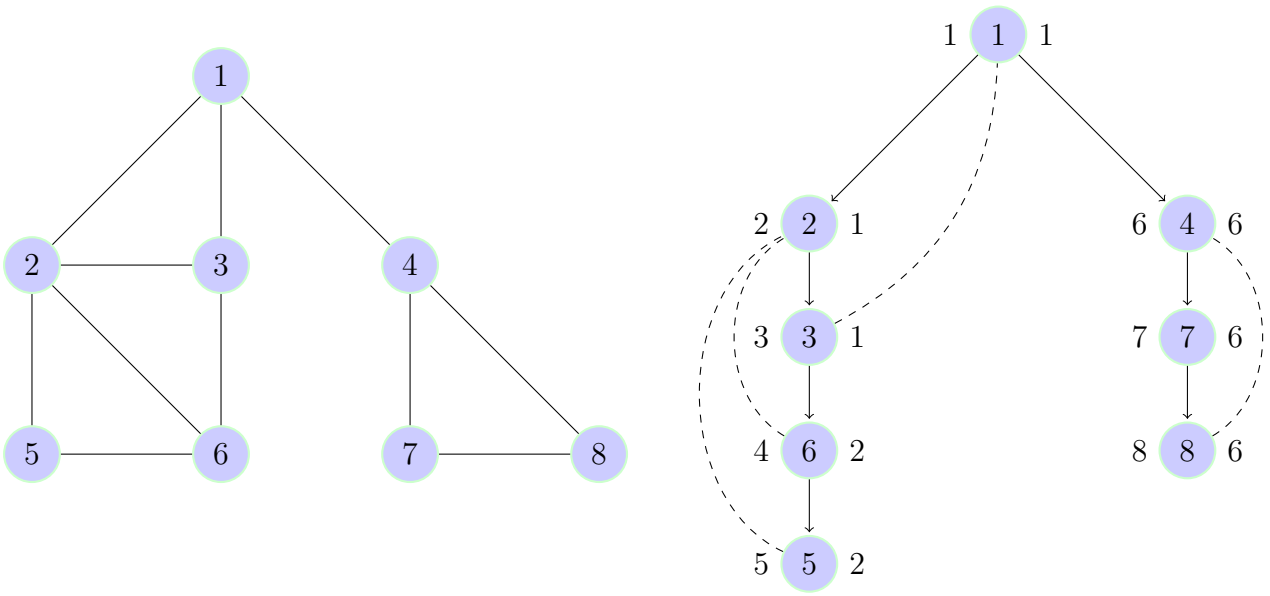


Figura 2.7: Un grafo no dirigido y su árbol de recorrido en profundidad.

2.3.2. Búsqueda en anchura (Breadth-First Search)

Cuando un recorrido en profundidad llega a un vértice v, intenta a continuación visitar algún vecino de v, después algún vecino del vecino, y así sucesivamente. Cuando un recorrido en anchura llega a algún vértice v, por otra parte, primero visita todos los vecinos de v. Sólo examina vértices más alejados después de haber hecho esto. A diferencia del recorrido en profundidad, el recorrido en anchura no es naturalmente recursivo.

Supongamos que queremos encontrar un camino más corto entre dos vértices específicos de un grafo - un camino que conecte a los vértices con la peculiaridad de que no exista otro camino con esos vértices y menos aristas. El método clásico para llevar a cabo esta tarea, llamado Búsqueda en Anchura (Breadth-First search), es también la base de numerosos algoritmos para el procesamiento de grafos. La Búsqueda en Profundidad nos ofrece poca ayuda en la resolución de este problema, ya que el orden en el que navega a través del grafo no tiene relación con el objetivo de encontrar los caminos más cortos. Por el contrario, la Búsqueda en Anchura se basa en este cometido. Para encontrar un camino más corto desde v hasta w, que comenzará en v hasta w comprobando entre todos los vértices que podemos llegar siguiendo una de las aristas, luego de comprobar todos los vértices que se pueden recorrer de esos extremos y así sucesivamente.

Cuando llegamos a un punto de la búsqueda en el grafo en donde tenemos que atravesar más de una arista, se elige una y almacenamos las demás sin procesar para un análisis posterior. En la Búsqueda en Profundidad, se suele utilizar la inserción de elementos en una pila (que es gestionada por una función o estructura auxiliar a la función de búsqueda recursiva) para este propósito. Usando la regla LIFO que caracteriza a las

pilas para explorar pasadizos o pasajes que estén muy próximos en un laberinto: Para ello elegimos, de las caminos que aún no hayan sido revisados, aquella que sea la más recientemente visitada. En la Búsqueda por Anchura, se explorarán los vértices según su ponderación o coste desde el punto de partida del camino o análisis. Para un laberinto, hacer esta búsqueda en este orden puede requerir un equipo entero de búsqueda; pero dentro de un programa de ordenador es más fácil de solucionar. Simplemente utilizaremos una estructura de tipo FIFO (cola) en lugar de una LIFO (pila).

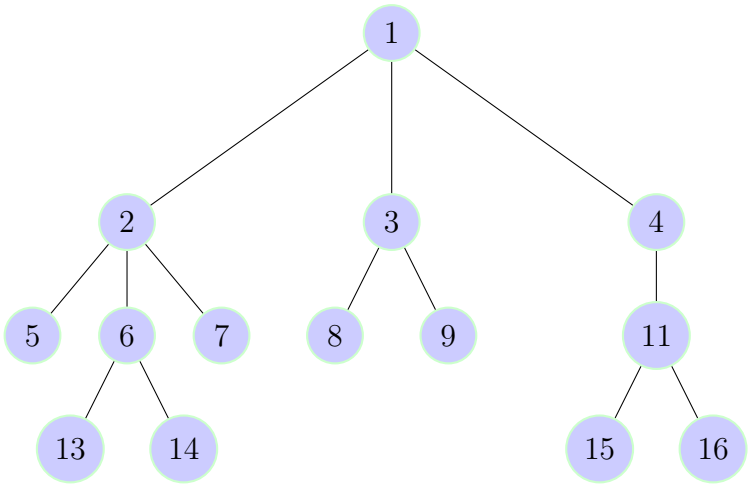


Figura 2.8: Un árbol de búsqueda en anchura. Los nodos son numerados en el orden en que serán procesados por el algoritmo. El hijo de un nodo se visita de izquierda a derecha.

2.4. Coloración

En ocasiones, un problema se puede modelar distribuyendo los vértices o las aristas de un cierto grafo en paquetes distintos, de manera que vértices (o aristas, en su caso) de un mismo paquete no sean adyacentes (respectivamente, incidentes). Si se da un mismo color a los elementos de un mismo paquete, y se emplean colores distintos para cada paquete, se obtiene lo que se da en llamar un vértice (arista) coloración. La idea es encontrar el menor número de paquetes (esto es, colores), para distribuir los elementos (vértices o aristas, según sea el caso).

Un planteamiento general que se ajusta a esta circunstancia es el llamado *problema de incompatibilidades*, que consta de ciertos elementos (modelados por los vértices de un grafo) y relaciones de incompatibilidad entre ellos (las cuales determinan las aristas del grafo). El grafo resultante se denomina *grafo de incompatibilidades*. El distribuir los elementos en el menor número de agrupaciones posible de forma que elementos incompatibles no compartan una misma agrupación, se traduce al instante en conseguir una vértice coloración (i.e.[2] una asignación de colores a los vértices de manera que vértices adyacentes tengan colores distintos).

Como ejemplo, se puede considerar el problema de diseñar un calendario a doble vuelta para una competición deportiva en forma de liguilla de n equipos, en la que todos se han de enfrentar con todos dos veces. Este problema admite una primera simplificación, en la que basta diseñar un calendario a una sola vuelta, para reproducirlo en la segunda

[2]Del Latín i.e. “id est” Significado: “Esto es”, “Es decir”

vuelta cambiando los papeles de los equipos que juegan como locales y como visitantes.

Si se modela el problema como el grafo G cuyos vértices son los n equipos y las aristas entre ellos representan los partidos, evidentemente se tiene que $G = K_n$, grafo completo de n vértices. Organizar el calendario se traduce en repartir los partidos (aristas) en jornadas (colores), de manera que un mismo equipo (vértice) no juegue más de una vez en cada jornada (no aparezca como extremo en más de una arista). En definitiva, se trata de colorear las aristas de K_n utilizando el menor número de colores, de manera que aristas con el mismo color representan partidos que se pueden jugar en la misma jornada.

Este problema admite ser modelado de otra forma: considérese el grafo H cuyos vértices son los partidos a jugar, y cuyas aristas relacionan partidos incompatibles, en tanto en cuanto implican a un mismo equipo. Resolver el problema consiste ahora en encontrar una vértice coloración de H.

2.4.1. Vértice coloraciones

Se denomina vértice coloración de un grafo $G = (V, A)$ a una asignación $c : V \to \mathbb{N}$ que asocie a cada vértice x_i un color $c_i \in \mathbb{N}$, de manera que a vértices adyacentes correspondan colores distintos: si $\{x_i, x_j\} \in A$ entonces $c_i \neq c_j$. Si la coloración consta de una paleta de k colores, entonces se habla simplemente de k-coloración.

Dado un grafo $G = (V, A)$, siempre existe un valor umbral k para el cual G admite una vértice coloración con una paleta de k colores, pero no una $(k-1)$-coloración. Es decir, k es el menor número de colores con los que se puede obtener una vértice coloración de G. Este valor se conoce como *número cromático* de G, y se denota en la forma $\chi(G) = k$.

Determinar cuál es el número cromático de un grafo es en general un problema de una envergadura considerable: **no hay** un procedimiento que dé una respuesta en tiempo razonable, para un grafo genérico dado.

Ni que decir tiene que el número cromático de un grafo no conexo consiste en el mayor de entre los números cromáticos de sus componentes conexas, razón por la cual nos centraremos en el estudio de coloraciones sobres grafos conexos.

Con normalidad, se tiende a acotar el número cromático tanto inferior como superiormente, $s \leq \chi(G) \leq t$, de manera que progresivamente se afinan estas cotas hasta llevarlas a coincidir, $s = \chi(G) = t$, momento en el cual queda completamente determinado $\chi(G)$.

Sin mucho esfuerzo, es fácil asegurar de entrada que $1 \leq \chi(G) \leq v$ para grafos G de v vértices. Más aún, $\chi(G) = 1$ sólo en el caso de grafos G vacíos, mientras que $\chi(G) = v$ en un grafo de v vértices sólo si $G = K_v$.

El **teorema de Brooks** perfila un poco más la cota superior para $\chi(G)$ en grafos conexos, en función de la valencia (grado) máxima:

$$\Delta = \max_{x \in V} gr_G(x) : \begin{cases} \chi(K_n) = n = \Delta + 1 \text{ y } \chi(C_{2n+1}) = 3 = \Delta + 1 \\ \text{En otro caso, } \chi(G) \leq \Delta \end{cases}$$

No obstante, esta cota puede ser bastante rudimentaria (piénsese en un grafo estrella S_m de m puntas, con $m = 1000$ por ejemplo: se tiene que $\chi(S_m) = 2$, mientras que $\Delta = m = 1000$).

Para obtener cotas inferiores, es frecuente buscar subgrafos $H \subseteq G$ de los cuales se conozca $\chi(H)$, toda vez que $\chi(H) \leq \chi(G)$ para cualquier $H \subseteq G$.

Para obtener cotas superiores, se suele realizar coloraciones concretas, de manera que $\chi(G)$ siempre es menor o igual que el número de colores utilizados.

A la hora de realizar coloraciones, es útil el llamado *algoritmo voraz*, el cual, progresando sobre una ordenación prefijada de los vértices, procede a colorearlos asignando a cada vértice el primer color libre (i.e. el primer color no utilizado en los vértices a él adyacentes previamente coloreados).

Aunque este algoritmo no tiene por qué devolver el número cromático de G, con normalidad devuelve un número razonablemente pequeño de colores. No obstante, siempre se puede aplicar varias veces el algoritmo a ordenaciones distintas de los vértices, para ver si eventualmente el número de colores desciende. Una aplicación exhaustiva del algoritmo a las $V!$ ordenaciones posibles en un grafo de V vértices resolvería efectivamente el problema; la "única" dificultad estriba en que $V!$ aplicaciones del algoritmo resultan ya impracticables cuando V no es siquiera demasiado grande (nótese que $8! = 40320$).

2.5. Ordenación topológica

La Ordenación Topológica es la operación más importante en los grafos acíclicos dirigidos (DAG). Este ordena los vértices en una línea de tal manera que todas las aristas dirigidas vayan de izquierda a derecha. Tal ordenamiento no puede existir si el grafo contiene un ciclo dirigido.

Cada DAG tiene al menos una ordenación topológica. La importancia de la ordenación topológica es que ordena procesando cada vértice antes de cualquier sucesor que tenga. Supongamos que las aristas representan restricciones de prioridad, de modo que la arista (x, y) significa que el trabajo o proceso x debe hacerse antes del proceso y. Entonces, cualquier ordenación topológica define un programa de forma lícita. De hecho, puede haber muchas ordenaciones para un mismo DAG dado.

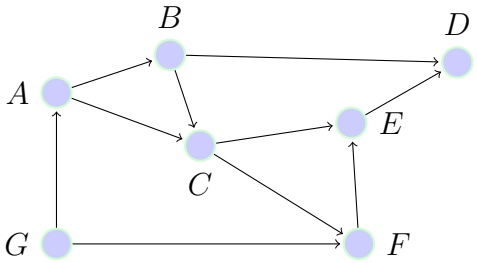


Figura 2.9: Un DAG con un solo orden topológico (G, A, B, C, F, E, D)

Pero las aplicaciones son más variadas. Supongamos que buscamos el camino más corto (o largo) desde x a y en un DAG. Ningún vértice aparece después de y en el orden topológico que puede contribuir a cualquiera de estos caminos, porque no habrá forma de volver a y. Podríamos procesar de forma correcta todos los vértices desde la izquierda a la derecha en el orden topológico, considerando el impacto de sus aristas exteriores, y sabiendo que se han analizado todo antes de que se necesite en otra operación. El orden topológico resulta muy útil para prácticamente cualquier problema algorítmico en grafos dirigidos.

El orden topológico puede realizarse de una manera eficiente usando la búsqueda en profundidad. Un grafo dirigido es un DAG si y sólo si no hay aristas de retroceso en él. Etiquetando los vértices en el orden inverso al que fueron procesados se encuentra una ordenación topológica de un DAG. ¿Porqué? Considere que sucede con cada arista dirigida $\{x, y\}$ cuando nos encontramos explorando el vértice x:

- Si no se ha pasado por y aún, entonces comenzaremos una búsqueda en profundidad en y antes de que podamos continuar con x. Por lo tanto, y es marcado como "procesado.ªntes que x lo sea, y x aparece antes que y en la ordenación topológica, como debería ser.
- Si se ha pasado por y pero no se ha marcado como "procesado", entonces $\{x, y\}$ es una arista de retroceso, lo cual no estaría permitido en un DAG.
- Si y se ha procesado, entonces habrá sido etiquetado antes que x. De tal forma que, x aparece antes que y en el orden topológico, como debería ser