

## Parte III

# Grafos



## Capítulo 7

# Grafos: introducción y definiciones

Los grafos permiten modelar infinidad de sistemas del mundo real en los que diferentes elementos de un conjunto están relacionados entre sí: ciudades conectadas por carreteras, proyectos divididos en tareas que dependen unas de otras, relaciones familiares en diagramas genealógicos, aeropuertos conectados por vuelos directos, etc. En muchos de estos sistemas surge la necesidad de resolver ciertos problemas que, en esencia, son instancias de problemas prototípicos que podemos formular en el terreno abstracto de los grafos. Tiene interés, por tanto, encontrar algoritmos eficientes para la resolución de estos problemas.

Un par de ejemplos ayudará a entender el planteamiento:

- Considérese el problema de encontrar la ruta por carretera que nos permita ir de una ciudad a otra con un menor consumo de combustible, o el de calcular la ruta que debe seguir un repartidor de pizzas para llegar cuanto antes al domicilio del cliente. Ambos son casos particulares de un mismo problema prototípico sobre grafos, el que conocemos como «problema del camino más corto».
- Dada la relación «ser amigo» en un grupo de personas o dada la relación «es un píxel vecino y del mismo color» en una imagen, el «problema del cálculo de las componentes conexas» de un grafo da solución a los respectivos problemas concretos de hallar los grupos de amigos o las zonas de una imagen que presentan el mismo color.

Resolver uno de los problemas prototípicos resuelve infinidad de problemas concretos.

Ante un problema concreto, la metodología de trabajo propuesta consiste en

- modelar el sistema real mediante un grafo y formular nuestro problema en términos de equivalencia con un problema prototípico sobre grafos,
- encontrar un método resolutivo para el problema prototípico sobre grafos (en muchos casos, disponible en la literatura),
- aplicar el método al grafo e interpretar la solución del problema prototípico en términos del problema original.

Este capítulo y los dos siguientes se dedican a presentar algunos conceptos relacionados con grafos, cuestiones relativas a su implementación y algunos de los algoritmos resolutivos para ciertos problemas fundamentales:

- el recorrido de los vértices de un grafo con diferentes criterios,
- el ordenamiento topológico de los vértices de un grafo,
- el cálculo de las componentes conexas de un grafo,
- el cálculo de la clausura transitiva de un grafo,

- el cálculo del árbol de recubrimiento mínimo de un grafo ponderado,
- la obtención del camino más corto entre dos vértices de un grafo ponderado,
- el cálculo del camino más corto entre un vértice y cualquier otro, esto es, del árbol de caminos más cortos,
- el cálculo de la distancia más corta entre todo par de vértices.

## 7.1. Grafos dirigidos

**Grafo:** *Graph*.

**Grafo dirigido:** *Directed graph*.

**Digrafo:** *Digraph*.

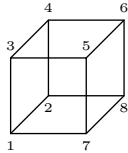
**Vértice:** *Vertex*.

**Nodo:** *Node*.

**Arista:** *Edge*.

**Arco:** *Arc*.

Usamos las letras  $V$  y  $E$  para referirnos a los conjuntos de vértices y aristas, respectivamente, por los términos ingleses «vertex» y «edge». Los términos vienen de considerar que los polígonos y poliedros describen grafos. Considera un cubo con sus vértices numerados como éste:



Hay *aristas* conectando, por ejemplo, los *vértices* 1 y 2 o 4 y 6.

Si necesitas dibujar grafos, puedes hacerlo con la ayuda de cualquier programa de dibujo. No obstante, hay una herramienta que automatiza el proceso: se le suministra una descripción del grafo en un fichero de texto y proporciona un fichero postscript (o en otro formato) con un diagrama que lo representa. El programa se llama *dot*, fue desarrollado por investigadores de AT&T y forma parte del paquete *graphviz*. Muchas distribuciones Linux lo incorporan. Si no ocurre con la tuya, busca *graphviz* en Google.

**Sucesor:** *Successor*.

**Vértice adyacente:** *Adjacent vertex*.

**Predecesor:** *Predecessor*.

**Sumidero:** *Sink*.

**Fuente:** *Source*.

**Grado de salida:** *Out-degree*.

**Grado de entrada:** *In-degree*.

Un **grafo dirigido** o digrafo  $G$  es una tupla  $(V, E)$  en la que  $V$  es un conjunto de **vértices** o nodos y  $E \subseteq V \times V$  es un conjunto de pares ordenados llamados **aristas** o arcos. Nótese que  $E$  es una relación binaria sobre  $V$ .

He aquí un ejemplo de grafo:

$$V = \{0, 1, 2, 3, 4, 5\},$$

$$E = \{(0, 1), (0, 3), (1, 4), (2, 4), (2, 5), (3, 0), (3, 1), (4, 3), (5, 5)\}.$$

Podemos representar un grafo con un diagrama en el que cada vértice es un círculo (o una caja) y cada arista es una flecha que une dos vértices. La figura 7.1 muestra una representación del grafo anterior. En ella se aprecia que hay dos aristas uniendo los vértices 0 y 3: la arista  $(0, 3)$  y la arista  $(3, 0)$ . Son aristas distintas, pues el orden con el que aparecen los vértices es diferente (las aristas son pares *ordenados*).

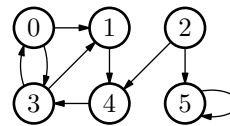


Figura 7.1: Grafo dirigido.

El primero de los vértices de una arista  $(u, v)$  es el **vértice de partida** u **origen** de la arista, y el segundo, el **vértice de llegada** o **destino**. Decimos que la arista sale, parte o emerge del primero y entra, llega o incide en el segundo. En el grafo del ejemplo, los vértices 2 y 5 están unidos por la arista  $(2, 5)$ . El vértice 2 es el vértice de partida de la arista  $(2, 5)$ , y el vértice 5 es el de llegada.

Dado un vértice  $u$ , los vértices  $v$  tales que  $(u, v) \in E$  se denominan **sucesores** de  $u$  o **vértices adyacentes** a  $u$ . Y dado un vértice  $v$ , los vértices  $u$  tales que  $(u, v) \in E$  se denominan **predecesores** de  $v$ . Un vértice sin sucesores es un **sumidero** y un vértice sin predecesores es una **fuentes**. El **grado de salida** de un vértice  $v$  es el número de sucesores que tiene, y el **grado de entrada**, el número de predecesores. Denotaremos con  $out\_degree(u)$  al grado de salida de un vértice  $u$  y con  $in\_degree(u)$  a su grado de entrada. En el grafo del ejemplo, el conjunto de sucesores del vértice 2 es  $\{4, 5\}$  y su conjunto de predecesores es el conjunto vacío. El grado de salida del vértice 2 es, pues, 2, y el de entrada es 0.

El **grado de entrada** (o **de salida**) de un grafo  $G = (V, E)$  es el mayor grado de entrada (o salida) de sus vértices:

$$in\_degree(G) = \max_{v \in V} in\_degree(v),$$

$$out\_degree(G) = \max_{u \in V} out\_degree(u).$$

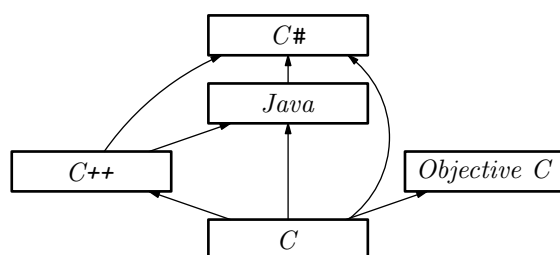
El grafo del ejemplo anterior tiene grados de entrada y salida idénticos; ambos valen 2.

El conjunto de vértices de un grafo no tiene por qué ser un subconjunto de los números naturales. He aquí un grafo cuyos vértices son lenguajes de programación.

$$V = \{C, C++, Java, C\#, Objective\ C\},$$

$$E = \{(C, C++), (C, Java), (C++, Java), (C, C\#), (Java, C\#), (C++, C\#), (C, Objective\ C)\}.$$

Las aristas de este grafo expresan la relación «fue tomado como base para»: «*C* fue tomado como base para *C++*», «*C* fue tomado como base para *Java*», ... La figura 7.2 muestra este grafo.



**Figura 7.2:** Grafo de lenguajes de programación. Cada arista une dos lenguajes e indica que un lenguaje (el vértice de partida) fue tomado como base para el diseño de otro (el vértice de llegada).

## 7.2. Grafos no dirigidos

Un **grafo no dirigido**  $G$  es un par  $(V, E)$  donde  $V$  es un conjunto de vértices y  $E$  es un conjunto de pares *no ordenados* de vértices distintos, es decir,  $E \subseteq \{\{u, v\} : u \neq v; u, v \in V\}$ . El conjunto  $E$  es, pues, una relación binaria simétrica sobre  $V$ .

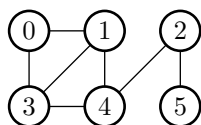
Consideremos, por ejemplo, un grafo en el que los vértices son personas y las aristas expresan la relación «es hermano de»: si  $u$  es hermano de  $v$ , entonces  $v$  es hermano de  $u$ . Se trata, pues, de un grafo no dirigido.

He aquí un ejemplo de grafo no dirigido:

$$V = \{0, 1, 2, 3, 4, 5\},$$

$$E = \{\{0, 1\}, \{0, 3\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 4\}\}.$$

En un grafo dirigido representamos gráficamente las aristas con flechas. En un grafo no dirigido las aristas son simples líneas (ver figura 7.3).



**Figura 7.3:** Grafo no dirigido.

En un grafo no dirigido, dos vértices unidos por una arista son **vértices adyacentes** y existe entre ellos, indistintamente, una relación de sucesión y precedencia. En el grafo de la figura 7.3, los vértices 2 y 5 son adyacentes y el conjunto de vértices adyacentes al vértice 4 es  $\{1, 2, 3\}$ . No tiene sentido hablar de grado de entrada o salida de un vértice  $v$ , pero sí de su **grado**, sin más, que es la talla del conjunto de vértices adyacentes a él y que denotamos con  $degree(v)$ . El grado de un grafo  $G = (V, E)$  es el grado del vértice con más vértices adyacentes:

$$degree(G) = \max_{u \in V} degree(u).$$

El grado del grafo de la figura 7.3 es 3.

Un grafo no dirigido puede considerarse un caso particular de grafo dirigido en el que  $(u, v) \in E$  implica  $(v, u) \in E$ . Así pues, cuando nos refiramos a un grafo, sin adjetivos, entenderemos que se trata de un grafo dirigido.

## 7.3. Grafos etiquetados y grafos ponderados

Un **grafo etiquetado** es un grafo  $G = (V, E)$  y una función  $f : E \rightarrow L$  que asocia a

**Grafo no dirigido:** *Undirected graph.*

Observa la notación de las aristas en un grafo no dirigido:  $\{u, v\}$  frente a  $(u, v)$ , que es como denotamos las aristas de un grafo dirigido. Las llaves indican que  $\{u, v\}$  es un conjunto de dos elementos y, por tanto, su orden no importa:  $\{u, v\}$  es equivalente a  $\{v, u\}$ .

**Grado:** *Degree.*

**Grafo etiquetado:** *Labeled graph.*

cada arista una *etiqueta*, es decir, un elemento de un conjunto dado  $L$ .

Consideremos, por ejemplo, el grafo dirigido y etiquetado  $G$  descrito por los siguientes conjuntos  $V$  y  $E$ :

$$\begin{aligned} V &= \{ \text{Pepe}, \text{María}, \text{Ana}, \text{Mar} \}, \\ E &= \{ (\text{Pepe}, \text{María}), (\text{Pepe}, \text{Ana}), (\text{Pepe}, \text{Mar}), (\text{María}, \text{Pepe}), (\text{María}, \text{Ana}), \\ &\quad (\text{Ana}, \text{Pepe}), (\text{Ana}, \text{María}), (\text{Ana}, \text{Mar}), (\text{Mar}, \text{Pepe}), (\text{Mar}, \text{Ana}) \}, \end{aligned}$$

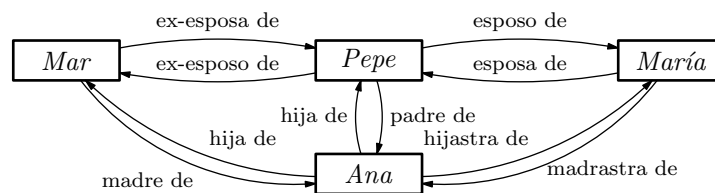
y el conjunto de etiquetas

$$L = \{ \text{esposo de, padre de, ex-esposo de, esposa de, madrastra de, hija de, hijastra de, hijo de, ex-esposa de, madre de} \}.$$

El grafo  $G$  y la función de etiquetado  $f : E \rightarrow L$  definida así:

$$\begin{aligned} f(\text{Pepe}, \text{María}) &= \text{esposo de}, & f(\text{Pepe}, \text{Ana}) &= \text{padre de}, \\ f(\text{Pepe}, \text{Mar}) &= \text{ex-esposo de}, & f(\text{María}, \text{Pepe}) &= \text{esposa de}, \\ f(\text{María}, \text{Ana}) &= \text{madrastra de}, & f(\text{Ana}, \text{Pepe}) &= \text{hija de}, \\ f(\text{Ana}, \text{María}) &= \text{hijastra de}, & f(\text{Ana}, \text{Mar}) &= \text{hija de}, \\ f(\text{Mar}, \text{Pepe}) &= \text{ex-esposa de}, & f(\text{Mar}, \text{Ana}) &= \text{madre de}. \end{aligned}$$

describe relaciones familiares entre cuatro personas. Cada etiqueta indica la relación entre las dos personas que enlaza (y la dirección de la arista importa). La figura 7.4 muestra una representación gráfica de  $G$  etiquetado con  $f$ .



**Figura 7.4:** Grafo dirigido y etiquetado. Los vértices son personas y las aristas están etiquetadas con vínculos familiares.

En ciertos problemas hemos de asociar un valor numérico a cada una de las aristas. Un grafo que representa, por ejemplo, ciudades conectadas por carreteras necesitará asociar a cada carretera (arista) su longitud en kilómetros si estamos interesados en efectuar cálculos de distancias entre pares de ciudades.

Un **grafo ponderado** es un caso particular de grafo etiquetado. Se describe con un grafo  $G = (V, E)$  y una función  $d : E \rightarrow \mathbb{R}$  que asigna un peso numérico a cada arista y a la que denominamos **función de ponderación**. Notaremos un grafo ponderado por  $d$  con  $G = (V, E, d)$ .

El grafo no dirigido definido como sigue modela un mapa de carreteras entre algunas de las principales ciudades de la isla de Mallorca.

$$\begin{aligned} V &= \{ \text{Alcúdia}, \text{Andratx}, \text{Artà}, \text{Calvià}, \text{Campos del Port}, \text{Capdepera}, \text{Inca}, \\ &\quad \text{Llucmajor}, \text{Manacor}, \text{Marratxí}, \text{Palma de Mallorca}, \text{Pollença}, \text{Santanyí}, \text{Sóller} \}, \\ E &= \{ \{ \text{Alcúdia}, \text{Artà} \}, \{ \text{Alcúdia}, \text{Inca} \}, \{ \text{Alcúdia}, \text{Pollença} \}, \{ \text{Andratx}, \text{Calvià} \}, \\ &\quad \{ \text{Andratx}, \text{Palma de Mallorca} \}, \{ \text{Andratx}, \text{Sóller} \}, \{ \text{Artà}, \text{Capdepera} \}, \\ &\quad \{ \text{Artà}, \text{Manacor} \}, \{ \text{Calvià}, \text{Palma de Mallorca} \}, \{ \text{Campos del Port}, \text{Llucmajor} \}, \\ &\quad \{ \text{Campos del Port}, \text{Santanyí} \}, \{ \text{Inca}, \text{Manacor} \}, \{ \text{Inca}, \text{Marratxí} \}, \\ &\quad \{ \text{Llucmajor}, \text{Palma de Mallorca} \}, \{ \text{Manacor}, \text{Santanyí} \}, \\ &\quad \{ \text{Marratxí}, \text{Palma de Mallorca} \}, \{ \text{Pollença}, \text{Sóller} \} \}. \end{aligned}$$

La siguiente función de ponderación asocia a cada conexión entre dos ciudades la dis-

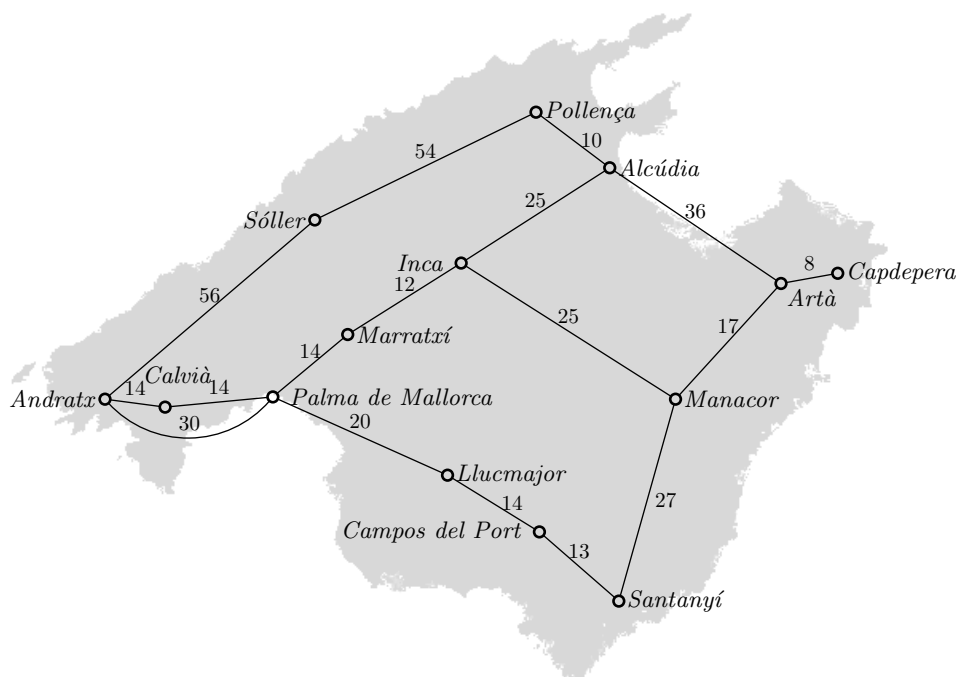
**Grafo ponderado:** *Weighted graph.*

**Función de ponderación:** *Weighting function.*

tancia kilométrica de la carretera que las une.

$$\begin{aligned}
 d(\text{Alcúdia}, \text{Artà}) &= 36, & d(\text{Alcúdia}, \text{Inca}) &= 25, \\
 d(\text{Alcúdia}, \text{Pollença}) &= 10, & d(\text{Andratx}, \text{Calvià}) &= 14, \\
 d(\text{Andratx}, \text{Palma de Mallorca}) &= 30, & d(\text{Andratx}, \text{Sóller}) &= 56, \\
 d(\text{Artà}, \text{Capdepera}) &= 8, & d(\text{Artà}, \text{Manacor}) &= 17, \\
 d(\text{Calvià}, \text{Palma de Mallorca}) &= 14, & d(\text{Campos del Port}, \text{Llucmajor}) &= 14, \\
 d(\text{Campos del Port}, \text{Santanyí}) &= 13, & d(\text{Inca}, \text{Manacor}) &= 25, \\
 d(\text{Inca}, \text{Marratxí}) &= 12, & d(\text{Llucmajor}, \text{Palma de Mallorca}) &= 20, \\
 d(\text{Manacor}, \text{Santanyí}) &= 27, & d(\text{Marratxí}, \text{Palma de Mallorca}) &= 14, \\
 d(\text{Pollença}, \text{Sóller}) &= 54.
 \end{aligned}$$

La figura 7.5 muestra una representación gráfica de  $G$ . Los pesos aparecen cerca de las respectivas aristas.



**Figura 7.5:** Grafo no dirigido y ponderado. El grafo modela un mapa de carreteras entre las principales ciudades de la isla de Mallorca. Cada arista es una conexión por carretera entre ciudades y el peso de la arista es su distancia en kilómetros. La zona noroeste de la isla es muy accidentada, lo que hace que las carreteras no sigan trayectorias rectas, aunque la idealización de la representación gráfica sugiera lo contrario.

Un **grafo** es **ponderado positivo** si los pesos son valores positivos o nulos, es decir, si  $d$  es una función  $d : E \rightarrow \mathbb{R}^{\geq 0}$ . Un mapa de carreteras, por ejemplo, puede modelarse con un grafo ponderado positivo.

Como encontraremos con frecuencia sistemas que, al modelarse con un grafo ponderado, asignan a cada arista una distancia, es frecuente llamar «distancia de una arista» a su peso. De hecho, normalmente usamos la letra  $d$  para la función de ponderación porque recuerda el término «distancia».

También es frecuente usar la letra  $w$ , por «weighth» (que significa peso) para la función de ponderación.

## 7.4. Caminos

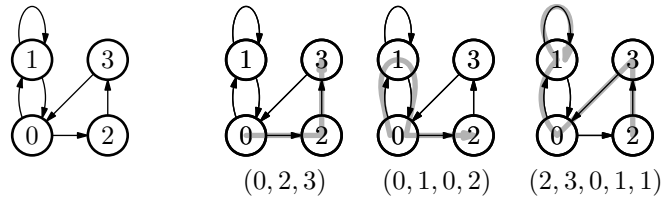
Un **camino** en un grafo  $G = (V, E)$  es una secuencia de vértices  $(v_1, v_2, \dots, v_n)$  tal que todo par de vértices consecutivos está unido por una arista de  $E$ , es decir,  $(v_i, v_{i+1})$  (o  $\{v_i, v_{i+1}\}$ , si el grafo es no dirigido) pertenece a  $E$  para todo  $i$  entre 1 y  $n-1$ . El vértice

**Camino:** *Path*.

**Longitud del camino:** *Path length*.

$v_1$  es el **vértice de partida** y  $v_n$  es el **vértice de llegada**. La **longitud o talla de un camino** es el número de aristas que lo componen. El camino  $(v_1, v_2, \dots, v_n)$  tiene longitud  $n - 1$ . Un caso particular de camino es el formado por un único vértice y tiene longitud 0.

En la figura 7.6 se muestra un grafo dirigido y algunas secuencias de vértices que constituyen caminos válidos.



**Figura 7.6:** A la izquierda se muestra un grafo y, a su derecha, tres caminos (en trazo gris).

Denotaremos el **conjunto de todos los caminos de un grafo entre un par de vértices**  $s$  y  $t$  con  $P_G(s, t)$ :

$$P_G(s, t) = \{(v_1, v_2, \dots, v_n) \mid v_1 = s; v_n = t; (v_i, v_{i+1}) \in E, 1 \leq i < n\}$$

Cuando  $G$  se deduzca del contexto, usaremos la notación  $P(s, t)$ .

Un camino  $(v_1, v_2, \dots, v_n)$  cuyos vértices de partida y llegada coinciden, es decir, tal que  $v_1 = v_n$ , recibe el nombre de **ciclo** o **bucle**. En el grafo de ejemplo de la figura 7.6, los caminos  $(0, 1, 0)$ ,  $(0, 2, 3, 0)$  o  $(0, 2, 3, 0, 1, 0, 1, 0)$  son ciclos. Un camino contiene un ciclo si un segmento suyo es un ciclo. Un camino es **simple** si no contiene ciclos, es decir, si no contiene vértices repetidos. Un camino que no es simple es un bucle o contiene un bucle. El camino  $(3, 0, 1, 0, 2)$ , por ejemplo, no es simple (el vértice 0 está repetido) y contiene un bucle:  $(0, 1, 0)$ .

Dado que un camino está compuesto por una sucesión de aristas, en algunos sistemas modelados con grafos ponderados tiene interés el concepto de **distancia o peso de un camino**. En principio, definimos la distancia o peso de un camino en un grafo  $G = (V, E)$  ponderado por  $d : E \rightarrow \mathbb{R}$  así:<sup>1</sup>

$$D(v_1, v_2, \dots, v_n) = \sum_{1 \leq i < n} d(v_i, v_{i+1}).$$

En el grafo ponderado del mapa de carreteras de la isla de Mallorca (figura 7.5, página 155), el camino

*(Marratxí, Inca, Manacor, Artà)*

tiene longitud 3 y una distancia (o peso) de 54 kilómetros.

La forma en que componemos las distancias asociadas a cada una de las aristas para proporcionar la distancia de un camino no tiene por qué ser la suma. Es posible definir el peso de un camino como, por ejemplo, el producto de los pesos de sus aristas:

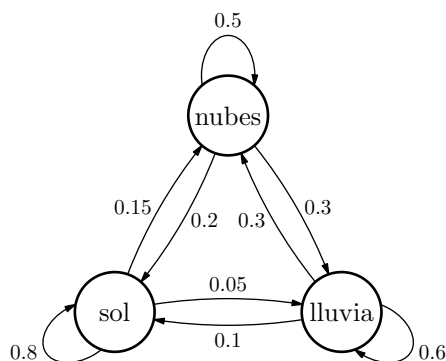
$$D(v_1, v_2, \dots, v_n) = \prod_{1 \leq i < n} d(v_i, v_{i+1}).$$

#### .....PROBLEMAS.....

► **105** No siempre hemos de sumar pesos para obtener la «distancia» de un camino. Esta figura ilustra un modelo de Markov para la predicción del tiempo que hará cada día.

<sup>1</sup>No debe confundirse la *longitud o talla* de un camino con su *distancia o peso*. Longitud y distancia se usan en este texto aquí con sentidos diferentes: la longitud es el número de aristas de un camino y la distancia es la suma de las distancias individuales asociadas a cada arista del camino.





La arista (sol, lluvia) tiene peso 0.05, valor que interpretamos como la probabilidad de pasar de un día soleado a uno lluvioso. Cada secuencia de vértices es un «camino» al que asociamos una probabilidad que no se calcula sumando las probabilidades (los pesos) de cada una de sus aristas, sino multiplicándolos.

Supongamos que hoy hace sol, ¿qué es más probable, la secuencia (sol, nubes, nubes, lluvia, nubes) o la secuencia (sol, sol, nubes, sol, sol, sol)?

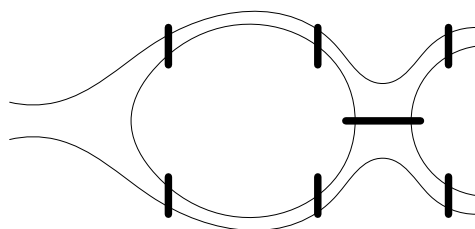
Un camino que visita todos los vértices del grafo exactamente una vez es un **camino hamiltoniano**. Un camino que visita todas las aristas de un grafo exactamente una vez es un **camino euleriano**.

**Camino hamiltoniano:**  
*Hamiltonian path.*

**Camino euleriano:** *Euler path.*

#### PROBLEMAS

► **106** La ciudad de Königsberg tiene dos islas donde el río Pregel se bifurca. Siete puentes interconectan las islas y las dos orillas del río tal y como muestra esta figura:



Este problema fue propuesto por Euler en 1736 y de él reciben nombre los caminos eulerianos.

¿Hay alguna forma de cruzar los siete puentes sin cruzar ninguno más de una vez?

## 7.5. Grafos acíclicos

Un grafo dirigido es **acíclico** si no contiene ciclos. Los grafos dirigidos acíclicos se conocen en la literatura con el término «**dag**». El grafo de los lenguajes de programación (figura 7.2, página 153), por ejemplo, es un grafo acíclico.

**Acíclico:** *Acyclic.*

Determinar si un grafo es o no acíclico presentará interés más adelante, ya que ciertos algoritmos sólo funcionan sobre grafos acíclicos. Por ejemplo, cuando deseemos resolver el «problema del camino más corto» en un grafo, deberemos seleccionar un algoritmo de un catálogo y el más eficiente de ellos sólo es aplicable si el grafo es acíclico.

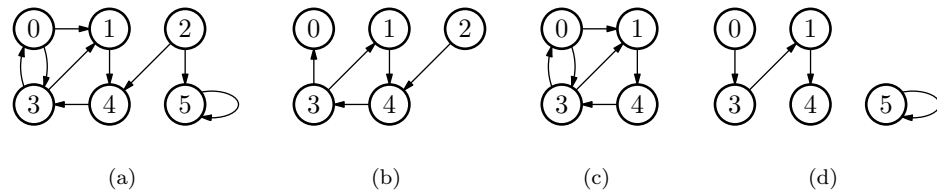
**Grafo dirigido acíclico (GDA):** *Directed acyclic graph (DAG).*

## 7.6. Subgrafos

Un grafo  $G' = (V', E')$  es un **subgrafo** de un grafo  $G = (V, E)$  si  $V' \subseteq V$  y  $E' \subseteq E$ . Dado un grafo  $G = (V, E)$  y un subconjunto  $V'$  de  $V$ , el **subgrafo de  $G$  inducido por  $V'$**  es  $G' = (V', \{(u, v) \in E \mid u, v \in V'\})$ . La figura 7.7 muestra (a) un grafo, (b) un subgrafo suyo y (c) el subgrafo inducido por un subconjunto de vértices. Del mismo modo podemos definir el subgrafo  $G' = (V', E')$  inducido por un conjunto de aristas  $E' \subseteq E$ , que es  $G' = (\{u \mid \exists v : (u, v) \in E'\} \cup \{v \mid \exists u : (u, v) \in E'\})$ . La figura 7.7 (d) muestra un subgrafo del grafo de la figura 7.7 (a) inducido por un subconjunto de aristas.

**Subgrafo:** *Subgraph.*

Un subgrafo completo, es decir, con todos sus vértices conectados a todos los demás vértices es un **clique**.



**Figura 7.7:** (a) Grafo dirigido  $G$ . (b) Un subgrafo de  $G$ . (c) El subgrafo de  $G$  inducido por los vértices  $\{0, 1, 3, 4\}$ . (d) El subgrafo de  $G$  inducido por las aristas  $\{(0, 3), (1, 4), (3, 1), (5, 5)\}$ .

### PROBLEMAS

► **107** El plan de estudios de un Máster en Patafísica es un conjunto de asignaturas organizadas en tres grupos «Ciencias Inexactas», «Inactividades Literarias» y «El Calendario Patafísico». Cada uno tiene cuatro asignaturas llamadas como el grupo correspondiente y seguidas de los números 1, 2, 3 y 4. La asignatura  $n$  de un grupo es incompatible con la asignatura  $n - 1$  del mismo grupo, para  $1 < n \leq 4$ . Por ejemplo, no se puede cursar «Ciencias Inexactas 3» hasta haber superado «Ciencias Inexactas 2». Existen dos incompatibilidades adicionales: no se puede cursar «Ciencias Inexactas 2» si no se superó «Inactividades Literarias 1» ni se puede cursar «El Calendario Patafísico 4» si no se superó «Inactividades Literarias 4». Se pide:

- Modelar con un grafo las asignaturas relacionadas por «ha de haberse superado para cursar».
- Determinar si se trata de un grafo cíclico o acíclico.
- Obtener el subgrafo inducido por las asignaturas de los dos primeros grupos.
- Si las asignaturas son anuales, ¿cuántos años son necesarios, al menos, para obtener el Máster en Patafísica? ¿Qué «camino» o «caminos» de asignaturas obliga a ello?

## 7.7. Conectividad

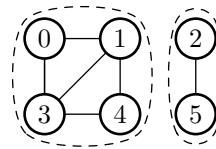
**Alcanzable:** *Reachable.*

**Conexo:** *Connected.*

**Componentes conexas:**  
*Connected component.*

Un vértice  $v$  es **alcanzable** desde un vértice  $u$  si existe un camino de  $u$  a  $v$ . Un **grafo no dirigido** es **conexo** si cualquier par de vértices está unido por un camino, es decir, si todo vértice es alcanzable desde cualquier otro. Las **componentes conexas** de un grafo no dirigido son los conjuntos de vértices alcanzables dos a dos.

En la figura 7.8 se muestra un ejemplo de grafo no dirigido y no conexo con dos componentes conexas.



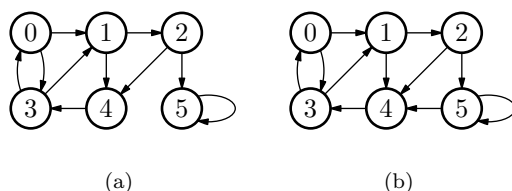
**Figura 7.8:** Grafo no dirigido y no conexo. El grafo contiene dos componentes conexas,  $\{0, 1, 3, 4\}$  y  $\{2, 5\}$ , que mostramos rodeadas por sendas líneas discontinuas.

**Débilmente conexo:** *Weakly connected.*

**Fuertemente conexo:**  
*Strongly connected.*

Un **grafo dirigido** es **débilmente conexo** si entre todo par de nodos  $u$  y  $v$  hay un camino que une  $u$  con  $v$  o  $v$  con  $u$ . Un **grafo dirigido** es **fuertemente conexo** si todo par de vértices es mutuamente alcanzable. Las **componentes fuertemente conexas** de un grafo dirigido son los conjuntos de vértices mutuamente alcanzables dos a dos.

En la figura 7.9 (a), aparece un grafo dirigido que no es fuertemente conexo: no es posible, por ejemplo, alcanzar el vértice 4 desde el vértice 5. El de la figura 7.9 (b) sí lo es.



**Figura 7.9:** (a) Grafo dirigido débilmente conexo. (b) Grafo dirigido fuertemente conexo.

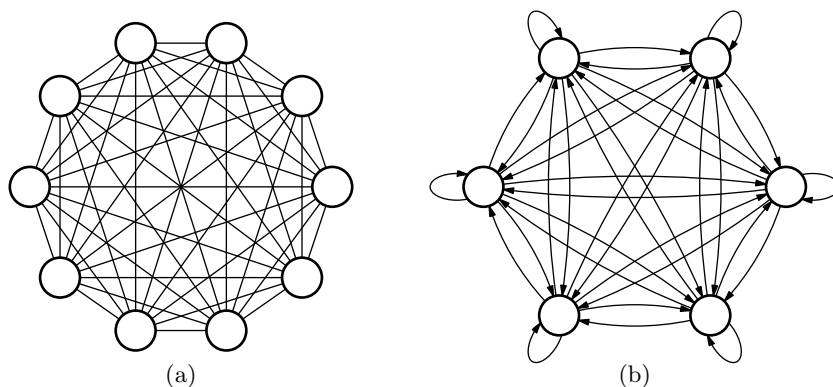
## 7.8. Densidad

Un **grafo** es **denso** si casi todo par de vértices está unido por una arista, es decir, si el número de aristas,  $|E|$ , es próximo a  $|V|^2$  en el caso de los grafos dirigidos y a  $|V| \cdot (|V| - 1)/2$  en el caso de los no dirigidos. Un grafo es **completo** si todo vértice está unido al resto de vértices. Un grafo es **disperso** cuando  $|E|$  es mucho menor que  $|V|^2$ .

**Denso:** *Dense.*

**Disperso:** *Sparse.*

En la figura 7.10 se muestra, a mano izquierda, un grafo completo no dirigido y, a mano derecha, un grafo completo dirigido.

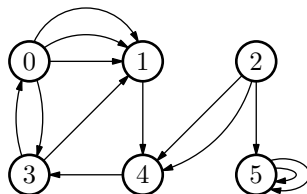


**Figura 7.10:** (a) Grafo no dirigido y completo. (b) Grafo dirigido y completo.

## 7.9. Multigrafos

Un **multigrafo** es un grafo en el que el conjunto de aristas es un multiconjunto, es decir, una colección en la que puede haber elementos repetidos. En un multigrafo puede existir, pues, más de una arista entre dos vértices. La figura 7.11 muestra un multigrafo.

**Multigraf:** *Multigraph.*



**Figura 7.11:** Multigrafo.

## 7.10. Algunos tipos especiales de grafo

Ciertos grafos presentan una estructura peculiar y nos interesa estudiarlos porque surgen de forma natural al modelar ciertos problemas y/o permiten obtener versiones especializadas (más eficientes) de ciertos algoritmos.

### 7.10.1. Grafos multietapa

**Grafo multietapa:** *Multistage graph.*

Un grafo  $G = (V, E)$  es multietapa si es posible particionar su conjunto de vértices  $V$  en  $k$  conjuntos, es decir  $V = V_1 \cup V_2 \cup \dots \cup V_k$ , donde  $V_i \cap V_j = \emptyset$  para todo  $i \neq j$ , y todas las aristas  $(u, v) \in E$  que tienen origen en una etapa  $i$  tienen destino en la etapa  $i + 1$ .

La figura 7.12 muestra una representación gráfica del grafo multietapa  $G = (V, E)$  donde:

$$V = \{1, 2, 3, 10, 11, 20, 21, 22, 23, 24, 30, 40, 41, 42\},$$

$$E = \{(0, 10), (0, 11), (1, 10), (2, 10), (2, 11), (10, 20), (10, 22), (10, 23), (10, 24), (11, 20), (11, 21), (11, 22), (20, 30), (21, 30), (23, 30), (30, 40), (30, 41), (30, 42)\}.$$

El grafo consta de 5 etapas:

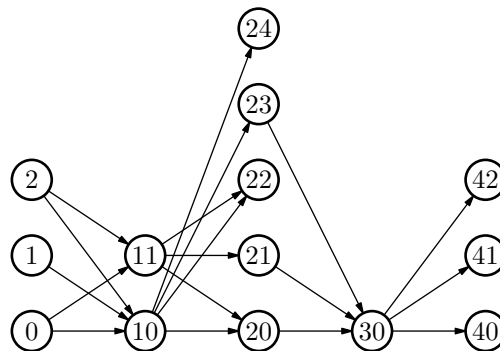
$$V_1 = \{1, 2, 3\},$$

$$V_2 = \{10, 11\},$$

$$V_3 = \{20, 21, 22, 23, 24\},$$

$$V_4 = \{30\},$$

$$V_5 = \{40, 41, 42\}.$$



**Figura 7.12:** Grafo multietapa. Los vértices se han representado de forma que cada columna corresponde a una etapa.

### 7.10.2. Árboles

Ya hemos estudiado los árboles en el contexto de las estructuras de datos. Un árbol es un caso particular de grafo ya que, a fin de cuentas, es un conjunto de nodos (vértices) vinculados entre sí por relaciones padre-hijo (aristas). Si en cada par de vértices relacionados tenemos en cuenta quién es padre y quién hijo, el grafo es dirigido; en caso contrario, no dirigido.

Un grafo dirigido es un árbol si todos los vértices excepto uno tienen un solo predecesor. El vértice que no tiene predecesor alguno se denomina raíz del árbol. La figura 7.13 (a) muestra un grafo dirigido arbóreo. La figura 7.13 (b) muestra el mismo grafo, pero de forma que hace más patente su estructura de árbol.

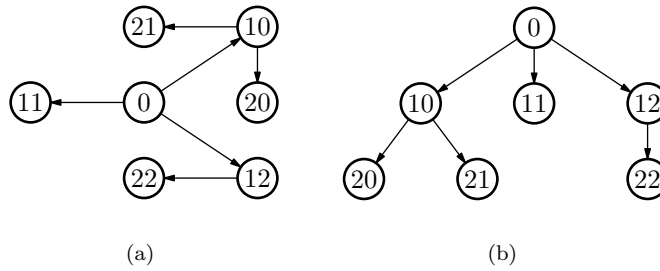
Un grafo no dirigido es un árbol si es conexo y acíclico. Una propiedad interesante de este tipo de grafos es que hay exactamente un camino sin ciclos entre cualquier par de vértices. La figura 7.14 (a) muestra un grafo dirigido arbóreo y la figura 7.14 (b) muestra el mismo grafo, pero haciendo manifiesta su estructura de árbol.

Una colección de árboles es un **bosque**. Un grafo no dirigido es un bosque si es acíclico.

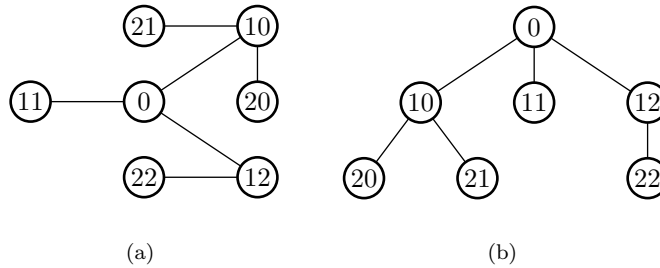
.....PROBLEMAS.....

► 108 ¿Es posible añadir una arista a un árbol de forma que el grafo resultante sea también un árbol?

.....



**Figura 7.13:** (a) Grafo dirigido con estructura de árbol. (b) Otra representación del mismo árbol en el que se advierte mejor que es, efectivamente, un árbol.



**Figura 7.14:** (a) Grafo no dirigido con estructura de árbol. (b) Otra representación del mismo árbol en el que se advierte mejor que es, efectivamente, un árbol.

### 7.10.3. Grafos euclídeos

Un grafo ponderado (dirigido o no dirigido) es **euclídeo** si cada vértice lleva asociado un punto en un espacio  $\mathbb{R}^n$  y la función de ponderación asigna a cada arista  $(u, v)$  el valor de la distancia euclídea entre los puntos asociados a  $u$  y  $v$ .

Podemos modelar el mapa de la figura 7.5 (página 155) con un grafo euclídeo si cada vértice mantiene las coordenadas geográficas de las ciudades respectivas y cada arista se pondera con la distancia que las separa. ¡Ojo!: entendemos por distancia la longitud del segmento de línea recta que las une, no la distancia real que recorre la correspondiente carretera: ésta última depende de los desniveles del terreno y de lo sinuoso del camino.

La figura 7.15 muestra gráficamente el grafo euclídeo no dirigido  $G = (V, E)$ , donde

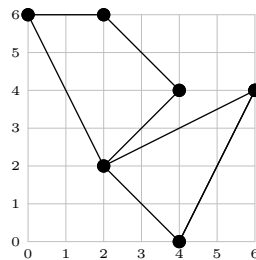
$$V = \{(0, 6), (2, 2), (2, 6), (4, 0), (4, 4), (6, 4)\},$$

$$E = \{\{(0, 6), (2, 2)\}, \{(0, 6), (2, 6)\}, \{(2, 2), (6, 0)\}, \{(2, 2), (4, 4)\}, \{(2, 2), (6, 4)\},$$

$$\{(2, 2), (4, 0)\}, \{(2, 6), (0, 6)\}, \{(2, 6), (4, 4)\}, \{(4, 0), (2, 2)\}, \{(4, 0), (6, 4)\},$$

$$\{(4, 4), (2, 6)\}, \{(4, 4), (2, 2)\}, \{(6, 4), (2, 2)\}, \{(6, 4), (4, 0)\}\}.$$

Nótese que cada vértice es un punto en  $\mathbb{R}^2$ . La función de ponderación es  $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ .



**Figura 7.15:** Un grafo euclídeo. Cada vértice se muestra como un punto en el plano coordenado. El peso de cada arista es la distancia entre los puntos que une.

**Grafo euclídeo:** *Euclidean graph.*



## Capítulo 8

# Implementación de grafos

Hemos de preocuparnos acerca de la implementación de los grafos para garantizar un consumo de memoria ajustado y la mayor eficiencia posible en el acceso a la información que almacenan. Generalmente nos preocupará la eficiencia con que podamos responder a las siguientes cuestiones:

- ¿Pertenece  $v$  al conjunto de vértices?
- ¿Pertenece  $(u, v)$  al conjunto de aristas?
- ¿Qué vértices son sucesores de un vértice  $u$ ?
- ¿Qué vértices son predecesores de un vértice  $v$ ?

En este capítulo presentamos diferentes clases para la implementación de los grafos, todas con un mismo interfaz.

- Un método constructor que admitirá un conjunto de vértices, un conjunto de aristas y la indicación de si el grafo es dirigido o no (será dirigido por defecto). El conjunto de vértices podrá suministrarse como secuencia (una tupla o lista, por ejemplo) o conjunto de valores (en este último caso, un objeto del tipo *Set* del módulo *sets*). El conjunto de aristas será una secuencia o conjunto de pares de vértices.
- Un atributo  $G.V$  (siendo  $G$  el grafo) que permite determinar si un elemento  $v$  pertenece al conjunto de vértices mediante la expresión  $v \text{ in } G.V$ .
- Un atributo  $G.E$  que permite determinar si un par de vértices  $(u, v)$  pertenece al conjunto de aristas mediante la expresión  $(u, v) \text{ in } G.E$ .
- Un método  $G.succs(u)$  que devuelve el conjunto de vértices sucesores de  $u$ .
- Y un método  $G.preds(v)$  que devuelve el conjunto de vértices predecesores de  $v$ .

Algunos grafos pueden considerarse estáticos, es decir, presentan conjuntos de vértices y aristas que no cambian nunca; otros, por contra, son modificables y permiten la adición o eliminación de vértices y/o aristas. Cuando convenga, pues, definiremos los siguientes métodos:

- $G.add\_vertex(v)$ : si  $v$  no está en  $G.V$ , lo añade;
- $G.remove\_vertex(v)$ : si  $v$  está en  $G.V$ , lo elimina de dicho conjunto y elimina, además, toda arista que parte o llega a  $v$ ;
- $G.add\_edge((u, v))$ : añade la arista  $(u, v)$  (un par de vértices) a  $G.E$  y, si  $u$  o  $v$  no pertenecen a  $G.V$ , también los añade;
- $G.remove\_edge((u, v))$ : elimina la arista  $(u, v)$  de  $G.E$ .

Los grafos no dirigidos se representarán como casos particulares de grafos dirigidos: cada arista no dirigida  $\{u, v\}$  se representará internamente con las dos aristas dirigidas  $(u, v)$  y  $(v, u)$ . Los métodos *succs* y *preds* devuelven, para un grafo no dirigido, el mismo conjunto: el de los vértices adyacentes.

En este capítulo incluimos numerosos programas que han de mostrar grafos para, informalmente, poner a prueba nuestras implementaciones. Este módulo incluye una función de utilidad para mostrar un grafo por pantalla:

Si usas el programa `dot` de `graphviz`, puede resultarte útil definir una función que escriba un grafo en un fichero siguiendo el formato que este programa puede leer. De ese modo tendrás la posibilidad de visualizar gráficamente las estructuras que implementamos en este capítulo.

```
show_graph.py
1 def show_graph(G):
2     print '%-12s|%-35s|%-29s' % ('Vértice', 'Sucesores', 'Predecesores')
3     print '-'*12 + '+' + '-'*35 + '+' + '-'*29
4     for v in G.V:
5         aux_succ = ''
6         for w in G.succs(v): aux_succ += str(w) + '␣'
7         aux_pred = ''
8         for u in G.preds(v): aux_pred += str(u) + '␣'
9         print '%-12s|%-35s|%-29s' % (v, aux_succ, aux_pred)
```

## 8.1. Grafos

Es este apartado vamos a presentar diferentes implementaciones de los grafos dirigidos. Cada una de ellas propone una forma diferente de organizar la información del conjunto de aristas  $E$ , ofreciendo así diferentes soluciones de compromiso entre el coste espacial y el coste temporal de cada una de las operaciones básicas de acceso: fundamentalmente, la determinación de la pertenencia de una arista a  $E$  y la obtención de la lista de vértices sucesores y predecesores de un vértice.

Consideraremos las siguientes implementaciones de  $E$ :

- Matriz de adyacencia.
- Listas de adyacencia.
- Vectores de adyacencia.
- Vectores de adyacencia ordenados.
- Conjuntos de adyacencia.
- Conjuntos de adyacencia con inversa.

Al construir un grafo indicaremos si éste es o no es dirigido. Una vez presentadas las implementaciones haremos un breve estudio comparativo.

### 8.1.1. Matriz de adyacencia

**Matriz de adyacencia:**  
*Adjacency matrix.*

Una **matriz de adyacencia** es una matriz (un vector bidimensional) indexada por vértices y que almacena valores booleanos. El par  $(u, v)$  es una arista del grafo si y sólo si la celda de la fila asociada a  $u$  y columna asociada a  $v$  vale *True*.

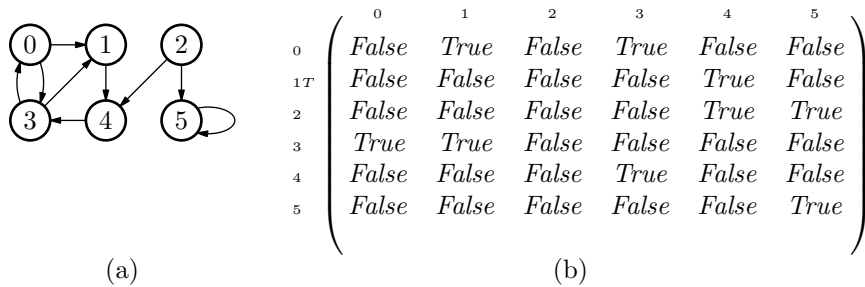
#### Conjunto de vértices en un rango $[0..n]$

Si el conjunto de vértices es un rango de números naturales consecutivos empezado en cero podemos usar los propios valores de los vértices como índices en el acceso a la matriz. La figura 8.1 ilustra la representación matricial del conjunto de aristas de un grafo dirigido con un conjunto de vértices como el descrito.

Empecemos por definir una clase para almacenar la matriz de adyacencia:

```
adjacency_matrix_graph1.py
1 class AdjacencyMatrix:
2     def __init__(self, n, E):
3         self.matrix = []
```





**Figura 8.1:** (a) Grafo dirigido. (b) Representación de su conjunto de aristas con una matriz de adyacencia. El valor *True* en la celda  $(i, j)$  de la matriz significa que hay una arista uniendo los vértices  $i$  y  $j$ . El valor *False* significa que no hay arista.

```

4     for i in range(n): self.matrix.append([False] * n)
5     for (u,v) in E: self.matrix[u][v] = True
6
7     def __contains__(self, (u, v)):
8         return self.matrix[u][v]
9
10    def __getitem__(self, (u,v)):
11        return self.matrix[u][v]
```

El constructor (`__init__`) de la matriz recibe el número de filas y columnas,  $n$ , y construye una matriz nula. El método `__contains__` permite efectuar consultas de pertenencia de una arista a la matriz. El método `__getitem__` se invoca cuando accedemos a una celda de la matriz.

He aquí una implementación de los métodos de una clase para grafos (constructor, acceso a sucesores y predecesores de un vértice) que usa la matriz de adyacencia:

```

adjacency_matrix_graph.1.py  adjacency_matrix_graph.1.py
13 class AdjacencyMatrixGraph:
14     def __init__(self, V=[], E=[], directed=True):
15         self.directed = directed
16         self.V = V
17         if directed: self.E = AdjacencyMatrix(len(self.V), E)
18         else: self.E = AdjacencyMatrix(len(self.V), E + [(v,u) for (u,v) in E])
19
20     def succs(self, u): # Devuelve los sucesores de u.
21         return [ v for v in self.V if self.E[u,v] ]
22
23     def preds(self, v): # Devuelve los predecesores de v.
24         return [ u for u in self.V if self.E[u,v] ]
```

En la figura 8.1 se muestra un grafo dirigido y la matriz de adyacencia que lo representa. Podemos codificarlo en Python y acceder a su información como muestra este programa:

```

test_adjacency_matrix_graph.1a.py  test_adjacency_matrix_graph.1a.py
1 from adjacency_matrix_graph_1 import AdjacencyMatrixGraph
2 from show_graph import show_graph
3
4 G = AdjacencyMatrixGraph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5),
5                                         (3,0), (3,1), (4,3), (5,5)])
6 print 'Vértices:', G.V
7 print 'Aristas:',
8 for u in G.V:
9     for v in G.V:
10        if (u,v) in G.E:
11            print '(%d,%d)' % (u,v),
```

```

12 print
13 print 'Sucesores de 0:', G.succs(0)
14 print 'Predecesores de 0:', G.preds(0)
15 print '¿Es 0 un vértice?:', 0 in G.V
16 print '¿Es 8 un vértice?:', 8 in G.V
17 print '¿Es (0,1) una arista?:', (0,1) in G.E
18 print '¿Es (0,0) una arista?:', (0,0) in G.E

```

```

Vértices: [0, 1, 2, 3, 4, 5]
Aristas: (0, 1) (0, 3) (1, 4) (2, 4) (2, 5) (3, 0) (3, 1) (4, 3) (5, 5)
Sucesores de 0: [1, 3]
Predecesores de 0: [3]
¿Es 0 un vértice?: True
¿Es 8 un vértice?: False
¿Es (0,1) una arista?: True
¿Es (0,0) una arista?: False

```

En adelante probaremos el interfaz de nuestros grafos con la función `show_graph`:

```

test_adjacency_matrix_1b.py test_adjacency_matrix_1b.py
1 from adjacency_matrix_graph_1 import AdjacencyMatrixGraph
2 from show_graph import show_graph
3
4 G = AdjacencyMatrixGraph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5),
5                                     (3,0), (3,1), (4,3), (5,5)])
6 show_graph(G)

```

Vértice	Sucesores	Predecesores
0	1 3	3
1	4	0 3
2	4 5	
3	0 1	0 4
4	3	1 2
5	5	2 5

El atributo `E` de `G` alberga esta matriz (lista de listas) Python:

```

[[False, True, False, True, False, False],
 [False, False, False, False, True, False],
 [False, False, False, False, True, True],
 [True, True, False, False, False, False],
 [False, False, False, True, False, False],
 [False, False, False, False, False, True]]

```

Ilustremos el modo de definición de grafos no dirigidos con un ejemplo: la codificación del grafo no dirigido de la figura 8.2.

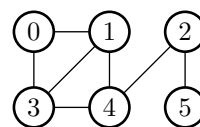


Figura 8.2: Grafo no dirigido.

```

test_adjacency_matrix_1c.py test_adjacency_matrix_1c.py
1 from adjacency_matrix_graph_1 import AdjacencyMatrixGraph
2 from show_graph import show_graph
3
4 G = AdjacencyMatrixGraph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5),
5                                     (3,0), (3,1), (4,3)], directed=False)
6 show_graph(G)

```

Vértice	Sucesores	Predecesores
0	1 3	1 3
1	0 3 4	0 3 4
2	4 5	4 5
3	0 1 4	0 1 4
4	1 2 3	1 2 3
5	2	2

Analicemos ahora la complejidad computacional de nuestra implementación. Hemos de destacar en primer lugar que el coste espacial es  $\Theta(|V|^2)$ . Se trata de un coste elevado cuando el grafo no es denso, es decir, cuando  $|E|$  es sensiblemente inferior a  $|V|^2$ . Hagamos ahora un análisis de la complejidad temporal de las diferentes operaciones que soporta este tipo de grafo. Sea  $G$  una instancia de *AdjacencyMatrixGraph*:

- El constructor recibe dos conjuntos:  $V$  y  $E$  (que pueden ser secuencias o conjuntos). El primero debe ser un rango de valores contiguos empezado en cero. La construcción de una instancia  $G$  de *AdjacencyMatrixGraph* requiere tiempo  $\Theta(|V|^2)$ , pues ése es el coste temporal de la reserva de memoria e inicialización de la matriz de adyacencia.
- Determinar si un vértice pertenece al conjunto de vértices es una operación  $\Theta(1)$  si se traduce en una mera consulta acerca de si un número es mayor o igual que 0 y menor que  $\text{len}(G.V)$ . (Si se efectúa evaluando una expresión de la forma  $u \text{ in } G.V$  es  $O(|V|)$ .)
- Saber si un par de vértices pertenece al conjunto de aristas,  $(u, v) \text{ in } G.E$ , es una operación  $\Theta(1)$ , pues sólo requiere consultar el valor de una celda de la matriz.
- El conjunto de sucesores de un vértice  $u$  requiere recorrer todos los vértices  $v$  de  $G.V$  para ver si  $(u, v)$  pertenece o no a  $G.E$ . El tiempo necesario es, pues,  $\Theta(|V|)$ .
- El conjunto de predecesores de un vértice  $v$  requiere recorrer todos los vértices  $u$  de  $G.V$  para ver si  $(u, v)$  pertenece o no a  $G.E$ . Este recorrido requiere tiempo  $\Theta(|V|)$ .

#### PROBLEMAS

- **109** Diseña una función que reciba un valor entero positivo  $n$  y devuelva un grafo no dirigido *completo* cuyo conjunto de vértices es  $V = \{0, 1, \dots, n-1\}$ . El conjunto de aristas debe representarse con una matriz de adyacencia.
- **110** Al instanciar un objeto de la clase *AdjacencyMatrixGraph* se construye internamente una matriz de dimensión  $|V| \times |V|$  que es simétrica y tiene diagonal principal nula. Modifica la clase para que represente un grafo no dirigido almacenando únicamente la región triangular superior de la matriz (sin la diagonal principal). Al representar el grafo de la figura 8.2 se almacenará así su matriz de adyacencia:

```
[[True, False, True, False, False ],
 [ False, True, True, False ],
  [ False, True, True ],
   [ True, False ],
    [ False ]]
```

### Conjunto de vértices arbitrario

La implementación que hemos presentado sólo resulta útil si el conjunto de vértices es un rango de naturales contiguos y empezado en cero. Encontraremos con frecuencia grafos en los que los vértices son elementos de un conjunto arbitrario: cadenas, tuplas, etc. Podemos tratar con estos conjuntos si mantenemos una tabla (un diccionario) que asocie a cada elemento un valor entero: el número de fila/columna que le corresponde en la matriz de adyacencia:

La única restricción que impone Python es que los vértices sean objetos inmutables (escalares, tuplas, cadenas...), pues han de poder usarse como índices en un diccionario.

```

adjacency_matrix_graph.2.py adjacency_matrix_graph.2.py
1 class AdjacencyMatrix:
2     def __init__(self, V, E):
3         self.index = {}
4         i = 0
5         for v in V:
6             self.index[v] = i
7             i += 1
8         self.matrix = []
9         for i in range(len(V)): self.matrix.append([False] * len(V))
10        for (u,v) in E: self.matrix[self.index[u]][self.index[v]] = True
11
12    def __contains__(self, (u, v)):
13        return self.matrix[self.index[u]][self.index[v]]
14
15    def __setitem__(self, (u,v), value):
16        self.matrix[self.index[u]][self.index[v]] = value
17
18    def __getitem__(self, (u,v)):
19        return self.matrix[self.index[u]][self.index[v]]
20
21    class AdjacencyMatrixGraph:
22        def __init__(self, V=[], E=[], directed=True):
23            self.directed = directed
24            self.V = V
25            if directed: self.E = AdjacencyMatrix(V, E)
26            else: self.E = AdjacencyMatrix(V, E + [(v,u) for (u,v) in E])
27
28        def succs(self, u):
29            return [ v for v in self.V if self.E[u, v] ]
30
31        def preds(self, v):
32            return [ u for u in self.V if self.E[u, v] ]

```

En este caso, el constructor de la matriz (método `__init__`) recibe el conjunto de vértices (una secuencia o conjunto de valores).

He aquí cómo codificar el grafo de los lenguajes de programación (véase la figura 8.3):

```

test_adjacency_matrix_graph.2.py test_adjacency_matrix_graph.2.py
1 from adjacency_matrix_graph_2 import AdjacencyMatrixGraph
2 from show_graph import show_graph
3
4 G = AdjacencyMatrixGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                           E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                               ('C', 'Objective_C'), ('C++', 'Java'),
7                               ('C++', 'C#'), ('Java', 'C#')])
8 show_graph(G)

```

Vértice	Sucesores	Predecesores
C	C++ Java C# Objective C	
C++	Java C#	C
Java	C#	C C++
C#		C C++ Java
Objective C		C

Resulta farragoso tener que trabajar explícitamente con la tabla *index* cada vez que se desea acceder al índice de un vértice. Python ofrece una solución más elegante: usar directamente un diccionario de diccionarios para mantener la matriz *E*.

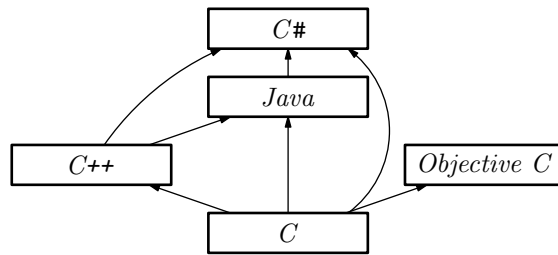


Figura 8.3: Grafo de los lenguajes de programación

```

adjacency_matrix_graph.py
1 class AdjacencyMatrix:
2     def __init__(self, V, E):
3         self.matrix = {}
4         for u in V:
5             self.matrix[u] = {}
6             for v in V:
7                 self.matrix[u][v] = False
8             for (u,v) in E: self.matrix[u][v] = True
9
10    def __contains__(self, (u, v)):
11        return self.matrix[u][v]
12
13    def __getitem__(self, (u,v)):
14        return self.matrix[u][v]
15
16    class AdjacencyMatrixGraph:
17        def __init__(self, V=[], E=[], directed=True):
18            self.directed = directed
19            self.V = V
20            if directed: self.E = AdjacencyMatrix(V, E)
21            else: self.E = AdjacencyMatrix(V, E + [(v,u) for (u,v) in E])
22
23        def succs(self, u):
24            return [ v for v in self.V if self.E[u,v] ]
25
26        def preds(self, v):
27            return [ u for u in self.V if self.E[u,v] ]

```

```

test_adjacency_matrix_graph.a.py
1 from adjacency_matrix_graph import AdjacencyMatrixGraph
2 from show_graph import show_graph
3
4 G = AdjacencyMatrixGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                           E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                               ('C', 'Objective_C'), ('C++', 'Java'),
7                               ('C++', 'C#'), ('Java', 'C#')])
8 show_graph(G)

```

Vértice	Sucesores	Predecesores
C	C++ Java C# Objective C	
C++	Java C#	C
Java	C#	C C++
C#		C C++ Java
Objective C		C

Internamente, el conjunto de aristas del grafo  $G$  se representa con un diccionario de diccionarios como éste:

```
{'C'      : {'C':0, 'C++':1, 'Java':1, 'C#':1, 'Objective_C':1},
 'C++'    : {'C':0, 'C++':0, 'Java':1, 'C#':1, 'Objective_C':0},
 'Java'   : {'C':0, 'C++':0, 'Java':0, 'C#':1, 'Objective_C':0},
 'C#'     : {'C':0, 'C++':0, 'Java':0, 'C#':0, 'Objective_C':0},
 'Objective_C': {'C':0, 'C++':0, 'Java':0, 'C#':0, 'Objective_C':0}}
```

El conjunto de aristas no tiene por qué ser una lista. Podemos suministrar, por ejemplo, un conjunto:

```
test_adjacency_matrix_graph.b.py test_adjacency_matrix_graph.b.py
1 from adjacency_matrix_graph import AdjacencyMatrixGraph
2 from show_graph import show_graph
3 from sets import Set
4
5 G = AdjacencyMatrixGraph(V=Set(['C', 'C++', 'Java', 'C#', 'Objective_C']),
6                           E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
7                               ('C', 'Objective_C'), ('C++', 'Java'),
8                               ('C++', 'C#'), ('Java', 'C#')])
9 show_graph(G)
```

Vértice	Sucesores	Predecesores
C#		C Java C++
Objective C		C
C	C# Objective C Java C++	
Java	C#	C C++
C++	C# Java	C

El coste espacial es el mismo de la versión anterior:  $O(|V|^2)$ . Para efectuar el análisis de complejidad temporal vamos a asumir que las operaciones de acceso y modificación a los elementos de un diccionario son  $\Theta(1)$ . Sea  $G$  una instancia de *AdjacencyMatrixGraph*:

- El constructor se ejecuta en tiempo  $\Theta(|V|^2)$ , que es el coste temporal de la reserva de memoria e inicialización de la matriz de adyacencia (diccionario de diccionarios).
- Determinar si un vértice pertenece al conjunto de vértices,  $u \text{ in } G.V$ , es una operación dependiente de la implementación de  $G.V$ . Por ejemplo, si es una lista, se ejecuta en tiempo  $\Theta(|V|)$  y si es un conjunto (una instancia de la clase *Set*), en tiempo  $\Theta(1)$ .
- Determinar si un par de vértices pertenece al conjunto de aristas mediante la evaluación de la expresión  $(u, v) \text{ in } G.E$  es una operación  $\Theta(1)$ .
- El conjunto de sucesores de un vértice  $u$  se obtiene en tiempo  $\Theta(|V|)$ .
- El conjunto de predecesores de un vértice  $v$  se calcula en tiempo  $\Theta(|V|)$ .

.....PROBLEMAS.....

► 111 Diseña una función que reciba una lista de vértices y devuelva un grafo no dirigido completo. El grafo debe representarse con una matriz de adyacencia.

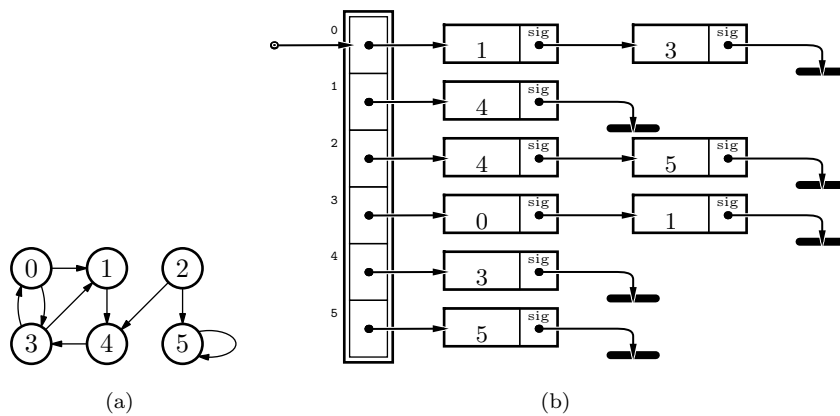
.....

### 8.1.2. Listas de adyacencia

**Listas de adyacencia:**  
*Adjacency lists.*

Una forma alternativa de representar un grafo es mediante listas de adyacencia. Consiste en definir un vector que asocia a cada vértice una lista enlazada con sus sucesores. Vamos a estudiar con detalle, únicamente, el caso en el que el conjunto de vértices es un rango  $[0..n]$ . (El caso general se propone como ejercicio.)

La figura 8.4 muestra un grafo y su implementación con listas de adyacencia.



**Figura 8.4:** (a) Grafo dirigido. (b) Representación del grafo con listas (enlazadas) de adyacencia.

`linked_adjacency_lists_graph.py` `linked_adjacency_lists_graph.py`

```

1 from linkedlist import LinkedList
2
3 class AdjacencyLists:
4     def __init__(self, V, E):
5         self.lists = [None] * len(V)
6         for v in V: self.lists[v] = LinkedList()
7         for (u, v) in E: self.lists[u].insert_head(v)
8
9     def __contains__(self, (u, v)):
10        node = self.lists[u].head
11        while node:
12            if v == node.item: return True
13            node = node.next
14        return False
15
16    def succs_to_array(self, u):
17        length = 0
18        node = self.lists[u].head
19        while node:
20            length += 1
21            node = node.next
22        succs = [None] * length
23        i = 0
24        node = self.lists[u].head
25        while node:
26            succs[i] = node.item
27            node = node.next
28            i += 1
29        return succs
30
31    def preds_to_array(self, v):
32        length = 0
33        for u in range(len(self.lists)):
34            if (u, v) in self: length += 1
35        preds = [None] * length
36        i = 0
37        for u in range(len(self.lists)):
38            if (u, v) in self:
39                preds[i] = u

```

```

40         i += 1
41     return preds
42
43 class AdjacencyListsGraph:
44     def __init__(self, V=[], E=[], directed=True):
45         self.directed = directed
46         self.V = V
47         if directed: self.E = AdjacencyLists(V, E)
48         else: self.E = AdjacencyLists(V, E + [(v,u) for (u,v) in E])
49
50     def succs(self, u):
51         return self.E.succs_to_array(u)
52
53     def preds(self, v):
54         return self.E.preds_to_array(v)

```

```

test_linked_adjacency_lists_graph test_linked_adjacency_lists_graph.py
1 from linked_adjacency_lists_graph import AdjacencyListsGraph
2 from show_graph import show_graph
3
4 G = AdjacencyListsGraph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5),
5                                     (3,0), (3,1), (4,3), (5,5)])
6 show_graph(G)

```

Vértice	Sucesores	Predecesores
0	3 1	3
1	4	0 3
2	5 4	
3	1 0	0 4
4	3	1 2
5	5	2 5

El coste espacial de esta implementación es  $O(|V| + |E|)$ , sensiblemente inferior al de las implementaciones basadas en la matriz de adyacencia. El coste de cada una de las operaciones se indica a continuación.

- El constructor se ejecuta en tiempo  $\Theta(|V| + |E|)$ .
- Determinar si un vértice pertenece al conjunto de vértices,  $u$  in  $G.V$ , es una operación dependiente de la implementación de  $G.V$ .
- Determinar si un par de vértices pertenece al conjunto de aristas,  $(u,v)$  in  $G.E$ , es una operación  $O(out\_degree(u))$ , es decir,  $O(|V|)$ .
- El conjunto de sucesores de un vértice  $u$  se obtiene en tiempo  $\Theta(out\_degree(u))$ . (o sea,  $\Theta(|V|)$ ).
- El conjunto de predecesores de un vértice  $v$  se calcula en tiempo  $O(|V|^2)$ .

.....PROBLEMAS.....

► **112** Diseña una clase para la implementación de grafos con listas enlazadas de adyacencia que permita trabajar con conjuntos arbitrarios de vértices. Analiza la complejidad computacional de cada uno de sus métodos.



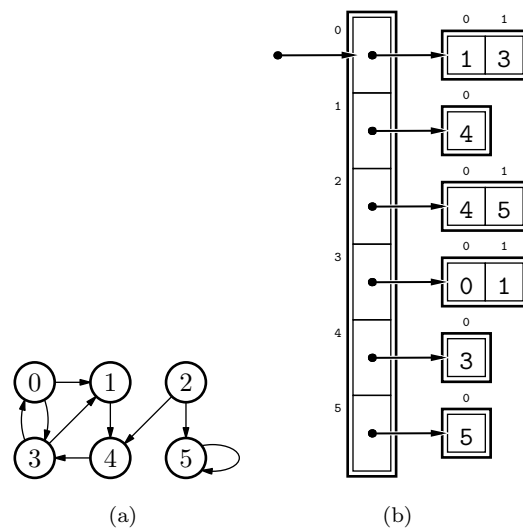


Figura 8.5: (a) Grafo dirigido. (b) Representación del grafo con vectores de adyacencia.

### 8.1.3. Vectores de adyacencia

Podemos usar vectores (listas Python) en lugar de listas enlazadas. La figura 8.5 ilustra esta representación alternativa. Una implementación con listas enlazadas resulta más farragosa que una equivalente con vectores, así que normalmente usaremos la segunda. Téngase en cuenta, no obstante, que si el grafo va a ser modificado, una lista enlazada puede ser ventajosa frente a un vector.

En aras de la brevedad, vamos a considerar únicamente la implementación para conjuntos de vértices arbitrarios. Dejamos el caso particular en el que  $V = [0..n]$  como ejercicio para el lector.

```
adjacency_arrays_graph.py adjacency_arrays_graph.py
1 class AdjacencyArrays:
2     def __init__(self, V, E):
3         self.arrays = {}
4         for v in V: self.arrays[v] = []
5         for (u, v) in E: self.arrays[u].append(v)
6
7     def __contains__(self, (u, v)):
8         return v in self.arrays[u]
9
10    def __getitem__(self, u):
11        return self.arrays[u]
12
13    def append(self, (u, v)):
14        return self.arrays[u].append(v)
15
16    class AdjacencyArraysGraph:
17        def __init__(self, V=[], E=[], directed=True):
18            self.directed = directed
19            self.V = V
20            if directed: self.E = AdjacencyArrays(V, E)
21            else: self.E = AdjacencyArrays(V, E + [(v,u) for (u,v) in E])
22
23        def succs(self, u):
24            return self.E[u]
25
26        def preds(self, v):
```

```

27     if self.directed: return [ u for u in self.V if v in self.E[u] ]
28     else: return self.E[v]

```

El grafo dirigido de la figura 8.3 se representa así con listas de adyacencia:

```

test_adjacency_arrays_graph.a.py  test_adjacency_arrays_graph.a.py
1  from adjacency_arrays_graph import AdjacencyArraysGraph
2  from show_graph import show_graph
3
4  G = AdjacencyArraysGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                           E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                              ('C', 'Objective_C'), ('C++', 'Java'),
7                              ('C++', 'C#'), ('Java', 'C#')])
8  show_graph(G)

```

Vértice	Sucesores	Predecesores
C	C++ Java C# Objective C	
C++	Java C#	C
Java	C#	C C++
C#		C C++ Java
Objective C		C

El coste espacial de esta implementación es  $O(|V| + |E|)$ . El coste temporal de cada operación es:

- En la versión presentada, el constructor se ejecuta en tiempo  $O(|V|^2)$ , pero es posible diseñar una versión diferente cuyo tiempo de ejecución sea  $\Theta(|V| + |E|)$ .
- Determinar si un vértice pertenece al conjunto de vértices,  $u \in G.V$ , es una operación dependiente de la implementación de  $G.V$ .
- Determinar si un par de vértices pertenece al conjunto de aristas,  $(u, v) \in G.E$ , es una operación  $O(out\_degree(u))$ , es decir,  $O(|V|)$ .
- El conjunto de sucesores de un vértice  $u$  se obtiene en tiempo  $\Theta(1)$ . Nótese que el conjunto sucesores devuelto es una mera referencia al vector correspondiente. Recorrerlo total o parcialmente es  $O(out\_degree(u))$ , pero obtener la lista es  $\Theta(1)$ .
- El conjunto de predecesores de un vértice  $v$  se calcula en tiempo  $O(|E|)$ .

Se puede apreciar que el tiempo de ejecución de todas las operaciones es idéntico al de las listas de adyacencia.

.....PROBLEMAS.....

► **113** Es posible reducir el coste temporal de la construcción de un grafo (método `__init__`) a  $\Theta(|V| + |E|)$ . Diseña un constructor con este coste temporal.

► **114** Diseña una versión de `AdjacencyArraysGraph` que implemente eficientemente grafos en los que los vértices forman un intervalo  $[0..n]$ . Analiza su coste computacional.

#### 8.1.4. Vectores de adyacencia ordenados

Podemos reducir significativamente el coste de determinar la pertenencia de una arista al conjunto de aristas si existe una relación de orden entre los elementos de  $V$  y mantenemos los vectores de adyacencia ordenados, pues realizar búsquedas dicotómicas en cada vector es una operación  $O(\lg out\_degree(G))$ :

```

adjacency_arrays_graph.py  adjacency_arrays_graph.py
30 from bisect import bisect
31
32 class SortedAdjacencyArrays:

```

La función `bisect` del módulo del mismo nombre efectúa la búsqueda binaria en un vector. Consulta el manual de referencia de las librerías de Python para averiguar los detalles acerca de su uso.

```

33 def __init__(self, V, E):
34     self.arrays = {}
35     E.sort()
36     for u in V: self.arrays[u] = [ v for (w,v) in E if u == w ]
37
38 def __contains__(self, (u, v)):
39     index = bisect(self.arrays[u], v)
40     if index > 0: return self.arrays[u][index-1] == v
41     return False
42
43 def __getitem__(self, u):
44     return self.arrays[u]
45
46 class SortedAdjacencyArraysGraph:
47     def __init__(self, V=[], E=[], directed=True):
48         self.directed = directed
49         self.V = V
50         if directed: self.E = SortedAdjacencyArrays(V, E)
51         else: self.E = SortedAdjacencyArrays(V, E + [(v,u) for (u,v) in E])
52
53     def succs(self, u):
54         return self.E[u]
55
56     def preds(self, v):
57         return [ u for u in self.V if v in self.E[u] ]

```

```

test_adjacency_arrays_graph_b.py test_adjacency_arrays_graph_b.py
1 from adjacency_arrays_graph import SortedAdjacencyArraysGraph
2 from show_graph import show_graph
3
4 G = SortedAdjacencyArraysGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                                E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                                  ('C', 'Objective_C'), ('C++', 'Java'),
7                                  ('C++', 'C#'), ('Java', 'C#')])
8 show_graph(G)

```

Vértice	Sucesores	Predecesores
C	C# C++ Java Objective C	
C++	C# Java	C
Java	C#	C C++
C#		C C++ Java
Objective C		C

El coste espacial de esta implementación es  $O(|V| + |E|)$ . Recogemos ahora el coste temporal de las operaciones que hemos modificado con respecto a los vectores de adyacencia no ordenados.

- El constructor se ejecuta en tiempo  $\Theta(|V|^2 + |E| \lg |E|)$ , aunque es posible ofrecer una implementación alternativa  $\Theta(|V| + |E| \lg |E|)$ .
- Determinar si un par de vértices pertenece al conjunto de aristas,  $(u, v)$  in  $G.E$ , es una operación  $O(\lg \text{out\_degree}(u))$ , es decir,  $O(\lg |V|)$ .

### 8.1.5. Conjuntos de adyacencia

Aún es posible reducir más el coste (esperado) de la determinación de pertenencia de una arista a  $G.E$ : usando tablas de dispersión (tipo *Set* del módulo *sets*) para representar los conjuntos de sucesores.

adjacency\_sets\_graph.py

adjacency\_sets\_graph.py

```

1 from sets import Set
2
3 class AdjacencySets:
4     def __init__(self, V, E):
5         self.sets = {}
6         for u in V: self.sets[u] = Set([ v for (w,v) in E if u == w ])
7
8     def __contains__(self, (u, v)):
9         return v in self.sets[u]
10
11     def __getitem__(self, u):
12         return self.sets[u]
13
14 class AdjacencySetsGraph:
15     def __init__(self, V=[], E=[], directed=True):
16         self.directed = directed
17         self.V = V
18         if directed: self.E = AdjacencySets(V, E)
19         else: self.E = AdjacencySets(V, E + [(v,u) for (u,v) in E])
20
21     def succs(self, u):
22         return self.E[u]
23
24     def preds(self, v):
25         if self.directed: return [ u for u in self.V if v in self.E[u] ]
26         else: return self.E[v]

```

test\_adjacency\_sets\_graph.py

test\_adjacency\_sets\_graph.py

```

1 from adjacency_sets_graph import AdjacencySetsGraph
2 from show_graph import show_graph
3
4 G = AdjacencySetsGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                        E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                          ('C', 'Objective_C'), ('C++', 'Java'),
7                          ('C++', 'C#'), ('Java', 'C#')])
8 show_graph(G)

```

Vértice	Sucesores	Predecesores
C	C# Objective C Java C++	
C++	C# Java	C
Java	C#	C C++
C#		C C++ Java
Objective C		C

El coste espacial de esta implementación es  $O(|V| + |E|)$ . El coste temporal (esperado) de las operaciones se detalla a continuación.

- El constructor se ejecuta en tiempo  $\Theta(|V| + |E|)$ .
- Determinar si un vértice pertenece al conjunto de vértices,  $u$  in  $G.V$ , es una operación dependiente de coste de la implementación de  $G.V$ .
- Determinar si un par de vértices pertenece al conjunto de aristas,  $(u, v)$  in  $G.E$ , es una operación de  $\Theta(1)$ .
- El conjunto de sucesores de un vértice  $u$  se obtiene en tiempo  $\Theta(1)$ . (Nótese que el conjunto de sucesores devuelto es una mera referencia al conjunto correspondiente.)
- El conjunto de predecesores de un vértice  $v$  se calcula en tiempo  $\Theta(|V|)$ .

### 8.1.6. Conjuntos de adyacencia con inversa

La obtención de los predecesores de un vértice es una operación relativamente costosa en grafos dirigidos. Podemos reducir su complejidad temporal a costa de un aumento de la ocupación espacial si, además de almacenar explícitamente el conjunto de aristas  $E$ , almacenamos su «inversa»  $E^{-1} = \{(v, u) \mid (u, v) \in E\}$ :

```
inv_adjacency_sets_graph.py
1 from adjacency_sets_graph import AdjacencySets
2
3 class InvAdjacencySetsGraph:
4     def __init__(self, V=[], E=[], directed=True):
5         self.directed = directed
6         self.V = V
7         if directed:
8             self.E = AdjacencySets(V, E)
9             self.EInv = AdjacencySets(V, [(v,u) for (u,v) in E])
10        else:
11            self.E = AdjacencySets(V, E + [(v,u) for (u,v) in E])
12
13    def succs(self, u):
14        return self.E[u]
15
16    def preds(self, v):
17        if self.directed: return self.EInv[v]
18        else: return self.E[v]
```

```
test_inv_adjacency_sets_graph.py
1 from inv_adjacency_sets_graph import InvAdjacencySetsGraph
2 from show_graph import show_graph
3
4 G = InvAdjacencySetsGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                           E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                              ('C', 'Objective_C'), ('C++', 'Java'),
7                              ('C++', 'C#'), ('Java', 'C#')])
8 show_graph(G)
```

Vértice	Sucesores	Predecesores
C	C# Objective C Java C++	
C++	C# Java	C
Java	C#	C C++
C#		C Java C++
Objective C		C

La única operación cuyo coste temporal se ve afectado es el acceso a los predecesores de un vértice, que pasa a ser una operación  $\Theta(1)$ .

### 8.1.7. Un estudio comparativo y algunas conclusiones

Hemos considerado numerosas implementaciones diferentes para los grafos. No podemos inclinarnos por una sola de ellas. En según qué circunstancias de uso, una puede resultar claramente preferible.

La tabla 8.1 resume el consume espacial de cada una de las implementaciones consideradas. Si nos preocupa el consumo de memoria y el grafo es disperso, por ejemplo, la matrices de adyacencia no parecen una buena elección. No obstante, su gran eficiencia para determinar la pertenencia de una arista al grafo podría compensar este problema cuando el grafo se usa en el contexto de algún algoritmo que efectúa esta operación numerosas veces, especialmente si el grafo es denso. La tabla 8.2 muestra el tiempo

necesario para construir un grafo con cada implementación a partir de un conjunto con sus vértices y un conjunto que especifica sus aristas. Nótese que mantener ordenados los vectores de adyacencia acelera la determinación de pertenencia de una arista si el número de sucesores es elevado. Otra opción que hemos de considerar es si deseamos almacenar vectores de adyacencia inversa, esto es, una representación explícita de los predecesores de cada vértice, pues duplica el consumo de memoria y sólo resulta conveniente si nuestro algoritmo accede efectivamente a los predecesores de un vértice.

	Espacio
Matriz de adyacencia	$\Theta( V ^2)$
Listas de adyacencia	$\Theta( V  +  E )$
Vectores de adyacencia	$\Theta( V  +  E )$
Vectores de adyacencia ordenados	$\Theta( V  +  E )$
Conjuntos de adyacencia	$\Theta( V  +  E )$
Conjuntos de adyacencia con inversa	$\Theta( V  +  E )$

**Tabla 8.1:** Coste espacial de las diferentes implementaciones.

	$G = \text{Graph}(V, E)$	$(u, v)$ in $G.E$
Matriz de ady.	$\Theta( V ^2)$	$\Theta(1)$
Listas de ady.	$\Theta( V  +  E )$	$\Theta(\text{out\_degree}(u))$
Vectores de ady.	$\Theta( V  +  E )$	$\Theta(\text{out\_degree}(u))$
Vectores de ady. ordenados	$\Theta( V  +  E )$	$\Theta(\lg \text{out\_degree}(u))$
Conjuntos de ady.	$\Theta( V  +  E )$	$\Theta(1)$
Conjuntos de ady. con inversa	$\Theta( V  +  E )$	$\Theta(1)$

**Tabla 8.2:** Coste temporal de la construcción del grafo y de determinación de pertenencia de un par de vértices al conjunto de aristas. (Los tiempos de las implementaciones con conjuntos son tiempos esperados.)

Si descartamos las matrices de adyacencia, hemos de escoger entre listas (enlazadas) de adyacencia y vectores de adyacencia. La tabla 8.3 recoge el coste de acceder a los sucesores y predecesores de un vértice. La gestión de las listas enlazadas comporta una mayor lentitud (en la práctica) en operaciones de recorrido de sucesores/predecesores y un mayor consumo de memoria por la gestión de punteros.

	$G.\text{succs}(u)$	$G.\text{preds}(v)$
Matriz de ady.	$\Theta( V )$	$\Theta( V )$
Listas de ady.	$\Theta(1)$	$\Theta( E )$
Vectores de ady.	$\Theta(1)$	$\Theta( E )$
Vectores de ady. ordenados	$\Theta(1)$	$\Theta( V  \lg  V )$
Conjuntos de ady.	$\Theta(1)$	$\Theta( E )$
Conjuntos de ady. con inversa	$\Theta(1)$	$\Theta(1)$

**Tabla 8.3:** Coste temporal del acceso a sucesores y predecesores. El tiempo de acceso a sucesores/predecesores considera únicamente el acceso a la lista, vector o conjunto de sucesores/predecesores, y no su recorrido. (Los tiempos de las implementaciones con conjuntos son tiempos esperados.)

Salvo que digamos lo contrario, usaremos en adelante los conjuntos de adyacencia con inversa. Suponen un compromiso razonable entre consumo de memoria (es proporcional al número de aristas) y velocidad de ejecución (ofrece tiempos esperados constantes).

En los algoritmos del siguiente capítulo usaremos una clase *Graph* que no es sino la clase *InvAdjacencySetsGraph*:

```

graph.py
1 from inv_adjacency_sets_graph import InvAdjacencySetsGraph
2
3 Graph = InvAdjacencySetsGraph

```

.....PROBLEMAS.....

- **115** Rellena tablas similares para las operaciones de adición/eliminación de vértices y aristas.
- **116** Implementa una nueva clase para representar grafos que almacene tanto una matriz de adyacencia como listas de adyacencia (para sucesores y predecesores) desordenadas. ¿Con qué coste podemos efectuar cada una de las operaciones? ¿Qué complejidad espacial presenta la nueva estructura de datos?
- **117** Piensa en cómo se implementarían los métodos de acceso convencionales en una nueva clase para representar grafos que almacenase tanto una matriz de adyacencia como listas de adyacencia (para sucesores y predecesores) *ordenadas*. ¿Con qué coste podemos determinar la pertenencia de una arista a  $E$  y obtener la relación de sucesores y predecesores de un vértice cualquiera?
- .....

## 8.2. Grafos mutables

Algunos algoritmos construyen grafos mediante la adición o borrado, paso a paso, de vértices y aristas. Vamos a enriquecer el interfaz de los grafos con métodos que permitan efectuar estas operaciones. En lugar de presentar una implementación de éstas sobre cada una de las clases definidas en la anterior sección, nos limitaremos a hacerlo sobre la clase que utilizaremos como implementación por defecto para los grafos: los conjuntos de adyacencia con inversa.

```


inv_adjacency_sets_graph.py
1 from sets import Set
2
3 class AdjacencySets:
4     def __init__(self, V, E):
5         self.sets = {}
6         for u in V: self.sets[u] = Set([ v for (w,v) in E if u == w ])
7
8     def __contains__(self, (u, v)):
9         return v in self.sets[u]
10
11     def __getitem__(self, u):
12         return self.sets[u]
13
14     def __delitem__(self, v):
15         del self.sets[v]
16
17     def add_set(self, v):
18         self.sets[v] = Set()
19
20 class InvAdjacencySetsGraph:
21     def __init__(self, V=[], E=[], directed=True):
22         self.directed = directed
23         self.V = V
24         if directed:
25             self.E = AdjacencySets(V, E)
26             self.EInv = AdjacencySets(V, [(v,u) for (u,v) in E])
27         else:
28             self.E = AdjacencySets(V, E + [(v,u) for (u,v) in E])
29
30     def succs(self, u):
31         return self.E[u]
32
33     def preds(self, v):
34         if self.directed: return self.EInv[v]

```

```

35     else: return self.E[v]
36
37     def add_vertex(self, v):
38         if v not in self.V:
39             try: self.V.append(v)
40             except AttributeError: self.V.add(v)
41             self.E.add_set(v)
42             if self.directed: self.EInv.add_set(v)
43
44     def remove_vertex(self, v):
45         if v in self.V:
46             try: del self.V[self.V.index(v)]
47             except TypeError: self.V.discard(v)
48             del self.E[v]
49             for u in self.V: self.E[u].discard(v)
50             if self.directed:
51                 del self.EInv[v]
52                 for u in self.V: self.EInv[u].discard(v)
53
54     def add_edge(self, (u,v)):
55         self.add_vertex(u)
56         self.add_vertex(v)
57         if v not in self.E[u]:
58             self.E[u].add(v)
59             if self.directed: self.EInv[v].add(u)
60             else: self.E[v].add(u)
61
62     def remove_edge(self, (u,v)):
63         if u in self.V and v in self.V and v in self.E[u]:
64             self.E[u].discard(v)
65             if self.directed: self.EInv[v].discard(u)
66             else: self.E[v].discard(u)

```

 test\_adjacency\_sets\_graph\_b.py

test\_adjacency\_sets\_graph\_b.py

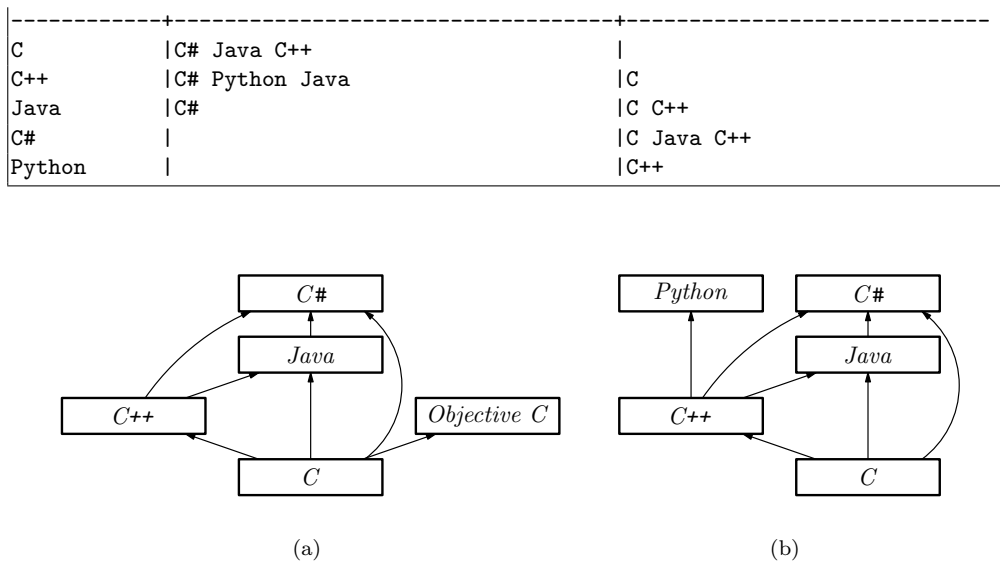
```

1 from inv_adjacency_sets_graph import InvAdjacencySetsGraph
2 from show_graph import show_graph
3
4 G = InvAdjacencySetsGraph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5                             E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'),
6                                 ('C', 'Objective_C'), ('C++', 'Java'),
7                                 ('C++', 'C#'), ('Java', 'C#')])
8 show_graph(G)
9
10 G.add_vertex('Python')
11 G.add_edge('C++', 'Python')
12 G.remove_edge('C', 'Objective_C')
13 G.remove_vertex('Objective_C')
14 print
15 show_graph(G)

```

Vértice	Sucesores	Predecesores
C	C# Objective C Java C++	
C++	C# Java	C
Java	C#	C C++
C#		C Java C++
Objective C		C
Vértice	Sucesores	Predecesores





**Figura 8.6:** (a) Grafo de lenguajes de programación original. (b) Grafo obtenido a partir del anterior mediante la adición y supresión de algunos vértices y aristas.

### PROBLEMAS

- **118** Implementa las operaciones de adición y supresión de vértices y aristas en las diferentes implementaciones de grafos mostradas en la sección 8.1. Analiza sus respectivos costes temporales.
- **119** El *cuadrado* de un grafo dirigido  $G = (V, E)$  se define como  $G^2 = (V, E')$  donde  $(u, w) \in E'$  si y sólo si para algún  $v \in V$  se cumple que  $(u, v) \in E$  y  $(v, w) \in E$ . La siguiente función recibe un grafo  $G$  y devuelve el grafo  $G^2$ . Ambos grafos están implementados con matrices de adyacencia.

```
squared_graph.py
1 from graph import Graph
2
3 def squared_graph(G):
4     Gsquared = Graph(V=[v for v in G.V])
5     for u in G.V:
6         for v in G.succs(u):
7             for w in G.succs(v):
8                 Gsquared.add_edge( (u,w) )
9     return Gsquared
```

```
test_squared_graph.py
1 from graph import Graph
2 from squared_graph import squared_graph
3 from show_graph import show_graph
4
5 G = Graph(V=range(6),
6           E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1), (4,3), (5,5)])
7 G2 = squared_graph(G)
8
9 print 'Grafo G:'
10 show_graph(G)
11
12 print
13 print 'Grafo G al cuadrado:'
14 show_graph(G2)
```

Grafo G:		
Vértice	Sucesores	Predecesores
0	1 3	3
1	4	0 3
2	4 5	
3	0 1	0 4
4	3	1 2
5	5	2 5

Grafo G al cuadrado:		
Vértice	Sucesores	Predecesores
0	0 1 4	0 4
1	3	0 3 4
2	3 5	
3	1 3 4	1 2 3
4	0 1	0 3
5	5	2 5

Calcula el coste temporal de la función *squared\_graph*, justificándolo adecuadamente.

► **120** El grafo *traspuesto* de un grafo dirigido  $G = (V, E)$  es un grafo  $G^t = (V, E^t)$  donde  $(u, v) \in E^t$  si y sólo si  $(v, u) \in E$ . Escribe un programa que construya el grafo traspuesto  $G^t$  a partir de otro grafo  $G$ . Calcula el coste temporal del algoritmo en función de la implementación escogida para  $G$  y  $G^t$ :

- Matriz de adyacencia.
- Vectores de adyacencia.
- Conjuntos de adyacencia con inversa.

► **121** Dados dos grafos  $G_1 = (V, E_1)$  y  $G_2 = (V, E_2)$  con el mismo conjunto de vértices, se define la *suma* de dos grafos como un grafo  $G'$  que tiene una arista  $(u, v)$  si  $(u, v) \in E_1$  o  $(u, v) \in E_2$ . Diseña una función que reciba dos grafos con el mismo conjunto de vértices y devuelva la suma de los dos. Calcula el coste temporal del algoritmo en función de la implementación escogida para los grafos:

- Matriz de adyacencia.
- Vectores de adyacencia.
- Conjuntos de adyacencia con inversa.

► **122** Dados dos grafos  $G_1$  y  $G_2$  con el mismo conjunto de vértices, se define la *diferencia* de  $G_1$  y  $G_2$  como un grafo  $G'$  que tiene una arista  $(u, v)$  si está en  $G_1$  pero no en  $G_2$ . Diseña una función que reciba dos grafos con el mismo conjunto de vértices y devuelva su diferencia. Calcula el coste temporal del algoritmo en función de la implementación escogida para los grafos:

- Matriz de adyacencia.
- Vectores de adyacencia.
- Conjuntos de adyacencia con inversa.

Unión: *Join*.

► **123** Dados dos grafos  $G_1 = (V_1, E_1)$  y  $G_2 = (V_2, E_2)$  se define la *unión* de  $G_1$  y  $G_2$  como el grafo  $G' = (V_1 \cup V_2, E_1 \cup E_2)$ . Diseña una función que calcule la unión de dos grafos. Calcula el coste temporal del algoritmo en función de la implementación escogida para los grafos:

- Matriz de adyacencia.
- Vectores de adyacencia.

c) Conjuntos de adyacencia con inversa.

► **124** El *producto* de dos grafos  $G_1 = (V_1, E_1)$  y  $G_2 = (V_2, E_2)$  es un grafo  $G'$  cuyo conjunto de vértices es el producto cartesiano  $V_1 \times V_2$ . Los vértices de  $G'$  son, pues, pares de la forma  $(v_1, v_2)$  donde  $v_1 \in V_1$  y  $v_2 \in V_2$ . Dos vértices  $(u_1, u_2)$  y  $(v_1, v_2)$  forman una arista en  $G$  si  $(u_1, v_1) \in E_1$  y  $(u_2, v_2) \in E_2$ . Calcula el coste temporal del algoritmo en función de la implementación escogida para los grafos:

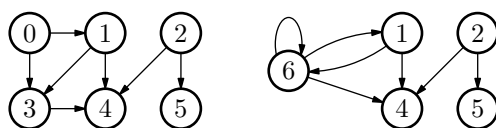
a) Matriz de adyacencia.

b) Vectores de adyacencia.

c) Conjuntos de adyacencia con inversa.

► **125** La operación de *contracción* de una arista  $(u, v)$  transforma un grafo  $G$  en un grafo  $G'$  en el que los vértices  $u$  y  $v$  se sustituyen por un único vértice  $w$  tal que todo vértice predecesor de  $u$  o  $v$  en  $G$  es predecesor de  $w$  en  $G'$  y todo vértice sucesor de  $u$  o  $v$  en  $G$  es sucesor de  $w$  en  $G'$ .

A la izquierda se muestra un grafo y a la derecha el resultado de contraer la arista  $(0, 3)$ :



Diseña una función que reciba un grafo  $G$  y una arista y devuelva el grafo resultante de contraer dicha arista en  $G$ . Calcula el coste temporal del algoritmo en función de la implementación escogida para los grafos:

a) Matriz de adyacencia.

b) Vectores de adyacencia.

c) Conjuntos de adyacencia con inversa.

► **126** Dados dos grafos  $G_1 = (V, E_1)$  y  $G_2 = (V, E_2)$ , se desea obtener un nuevo grafo  $G = (V, E)$  que sea la *diferencia simétrica* de  $G_1$  y  $G_2$ , tal que el conjunto de aristas  $E$  contenga las aristas de  $E_1$  que no estén en  $E_2$ , junto con las de  $E_2$  que no estén en  $E_1$ . Escribe funciones que construyan el grafo resultante  $G$ , considerando que los grafos se implementan como:

a) Matriz de adyacencia.

b) Listas enlazadas de adyacencia.

c) Vectores de adyacencia.

d) Conjuntos de adyacencia con inversa.

Analiza el coste espacial y temporal de cada función.

► **127** Se dice que un grafo  $G = (V, E)$  es *bipartido* si:

**Bipartido:** *Bipartite*.

- Existen un par de conjuntos  $V_1, V_2$  tales que  $V$  es la unión de  $V_1$  y  $V_2$ , y la intersección de  $V_1$  y  $V_2$  es vacía.
- Para toda arista  $(u, v)$  de  $E$ ,  $u$  está en  $V_1$ , y  $v$  está en  $V_2$ .

Se debe escribir un algoritmo que diga si un grafo  $G$ , del que se supone que sus vértices están numerados de 1 a  $n$ , es bipartido. Estima el coste temporal para esta implementación del grafo:

a) Matriz de adyacencia.

b) Listas enlazadas de adyacencia.

c) Vectores de adyacencia.

d) Conjuntos de adyacencia con inversa.

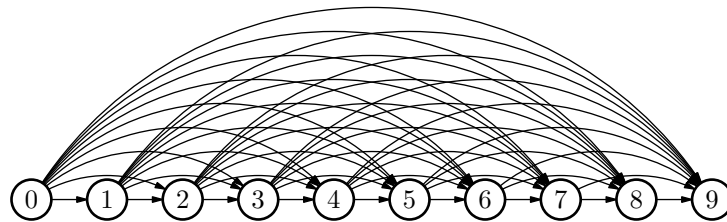
► **128** El *complemento* de un grafo no dirigido  $G = (V, E)$  es un grafo no dirigido  $G' = (V, E')$  en el que dos vértices son adyacentes si y sólo si no lo son en  $G$ . Calcula el coste temporal del algoritmo en función de si representas los grafos con:

- a) Una matriz de adyacencia.
- b) Vectores de adyacencia.
- c) Conjuntos de adyacencia con inversa.

### 8.3. Grafos con representación implícita de la información

Las representaciones de grafos que hemos considerado mantienen en memoria, explícitamente, un conjunto de vértices y otro de aristas (de diferentes formas y con diferentes prestaciones). Sin embargo, algunos grafos poseen una estructura tal que resulta innecesario almacenar en memoria ambos conjuntos. En su lugar, se pueden deducir y calcular «al vuelo». Ilustraremos este comentario con un par de ejemplos.

Consideraremos un grafo  $G$  como el de la figura 8.7 cuyo conjunto de vértices es  $\{0, 1, 2, \dots, 9\}$  y tal que todo vértice  $i$ , para  $0 \leq i < 9$ , está conectado con todos los vértices  $j$  tales que  $i < j < 10$ .



**Figura 8.7:** Un grafo dirigido con estructura que puede representarse sin necesidad de explicitar  $V$  y  $E$ .

Esta clase define el grafo  $G$ :

```

1 class MyGraph:
2     def __getattr__(self, attr):
3         if attr == 'V': return range(10)
4         elif attr == 'E': return [(u,v) for u in range(10) for v in range(u+1,10)]
5
6     def succs(self, u):
7         return range(u+1, 10)
8
9     def preds(self, v):
10        return range(0, v)

```

Ya que no almacenamos los vértices y aristas explícitamente, hemos añadido un método `__getattr__` que da acceso a unos atributos virtuales  $V$  y  $E$ . Al acceder a  $V$  o  $E$  se crea, respectivamente, un vector de vértices o un vector de aristas explícitos. Téngase en cuenta que, aunque resulte posible determinar la pertenencia de un vértice a  $V$  con una expresión como  $v \text{ in } G.V$ , resulta costoso en tanto que se invoca este método de acceso de  $G.V$  y se construye explícitamente un vector de 10 elementos.

Podríamos incluso generalizar este tipo de grafos para que el número de vértices se nos indique en el momento de la instanciación:

```

1 class MyGeneralGraph:
2     def __init__(self, n):

```

```

3     self.n = n
4
5     def __getattr__(self, attr):
6         if attr == 'V': return range(self.n)
7         elif attr == 'E': return [(u,v) for u in range(self.n) for v in range(u+1, self.n)]
8
9     def succs(self, u):
10        return range(u+1, self.n)
11
12    def preds(self, v):
13        return range(0, v)

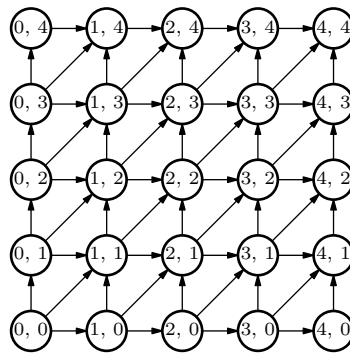
```

### PROBLEMAS

► 129 ¿Qué coste presenta cada uno de los métodos de la clase *MyGeneralGraph*?

Otro tipo de grafo estructurado de tal forma que no se precisa almacenar explícitamente sus vértices y aristas es el denominado grafo mallado, en malla o retícula. La figura 8.8 muestra un caso particular de grafo mallado.

**Malla:** *Grid*.



**Figura 8.8:** Un ejemplo de grafo mallado.

Y esta es su implementación (a la que hemos añadido código para efectuar un par de comprobaciones):

```

mygrid.py
1 from show_graph import show_graph
2
3 class MyGrid:
4     def __getattr__(self, attr):
5         if attr == 'V':
6             return [(i,j) for i in range(5) for j in range(5)]
7         elif attr == 'E':
8             return [((i,j), (i+1,j+1)) for i in range(4) for j in range(4)] + \
9                    [((i,j), (i+1,j)) for i in range(4) for j in range(5)] + \
10                   [((i,j), (i,j+1)) for i in range(5) for j in range(4)]
11
12    def succs(self, u):
13        i, j = u
14        s = []
15        if i < 4: s.append((i+1, j))
16        if j < 4: s.append((i, j+1))
17        if i < 4 and j < 4: s.append((i+1, j+1))
18        return s
19
20    def preds(self, v):
21        i, j = v
22        p = []

```

```

23     if i > 0: p.append((i-1, j))
24     if j > 0: p.append((i, j-1))
25     if i > 0 and j > 0: p.append((i-1, j-1))
26     return p
27
28 G = MyGrid()
29 show_graph(G)

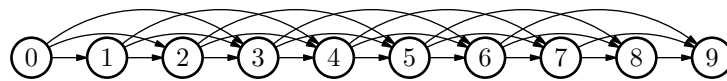
```

Vértice	Sucesores	Predecesores
(0, 0)	(1, 0) (0, 1) (1, 1)	
(0, 1)	(1, 1) (0, 2) (1, 2)	(0, 0)
(0, 2)	(1, 2) (0, 3) (1, 3)	(0, 1)
(0, 3)	(1, 3) (0, 4) (1, 4)	(0, 2)
(0, 4)	(1, 4)	(0, 3)
(1, 0)	(2, 0) (1, 1) (2, 1)	(0, 0)
(1, 1)	(2, 1) (1, 2) (2, 2)	(0, 1) (1, 0) (0, 0)
(1, 2)	(2, 2) (1, 3) (2, 3)	(0, 2) (1, 1) (0, 1)
(1, 3)	(2, 3) (1, 4) (2, 4)	(0, 3) (1, 2) (0, 2)
(1, 4)	(2, 4)	(0, 4) (1, 3) (0, 3)
(2, 0)	(3, 0) (2, 1) (3, 1)	(1, 0)
(2, 1)	(3, 1) (2, 2) (3, 2)	(1, 1) (2, 0) (1, 0)
(2, 2)	(3, 2) (2, 3) (3, 3)	(1, 2) (2, 1) (1, 1)
(2, 3)	(3, 3) (2, 4) (3, 4)	(1, 3) (2, 2) (1, 2)
(2, 4)	(3, 4)	(1, 4) (2, 3) (1, 3)
(3, 0)	(4, 0) (3, 1) (4, 1)	(2, 0)
(3, 1)	(4, 1) (3, 2) (4, 2)	(2, 1) (3, 0) (2, 0)
(3, 2)	(4, 2) (3, 3) (4, 3)	(2, 2) (3, 1) (2, 1)
(3, 3)	(4, 3) (3, 4) (4, 4)	(2, 3) (3, 2) (2, 2)
(3, 4)	(4, 4)	(2, 4) (3, 3) (2, 3)
(4, 0)	(4, 1)	(3, 0)
(4, 1)	(4, 2)	(3, 1) (4, 0) (3, 0)
(4, 2)	(4, 3)	(3, 2) (4, 1) (3, 1)
(4, 3)	(4, 4)	(3, 3) (4, 2) (3, 2)
(4, 4)		(3, 4) (4, 3) (3, 3)

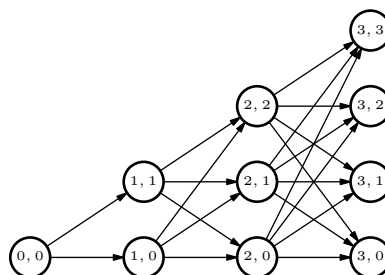
### .....PROBLEMAS.....

► **130** Diseña una clase que permita definir grafos mallados como el anterior dados el número de filas y el número de columnas del grafo.

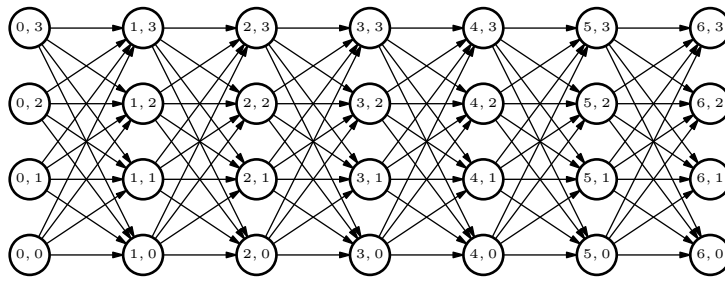
► **131** Diseña una clase que permita definir grafos con esta estructura:



► **132** Diseña una clase que, dado un número de columnas, permita definir grafos multietapa con esta estructura:



► **133** Diseña una clase que, dado un número de columnas, permita definir grafos multietapa con esta estructura:



## 8.4. Grafos ponderados

Recordemos que un grafo ponderado es un grafo  $G = (V, E)$  al que asociamos una función  $d : E \rightarrow \mathbb{R}$ . Es posible asociar más de una función de ponderación a un mismo grafo y obtener así diferentes grafos ponderados que comparten un mismo «núcleo». Pensemos, por ejemplo, en un mapa de carreteras. Podemos ponderar cada carretera con su longitud en kilómetros o con el tiempo medio que requiere recorrerla.

Para facilitar el trabajo con diferentes funciones de ponderación hemos optado por separar éstas de la definición del propio grafo. Aquellos algoritmos que manipulen grafos ponderados recibirán dos datos de entrada separados: el grafo y la función de ponderación. Así, un algoritmo que calcule el camino más corto entre dos ciudades podrá devolver el camino que recorre menos kilómetros o el camino que puede recorrerse en menor tiempo (pueden no coincidir) en función de si suministramos una función de ponderación u otra.

Una implementación naïf de una función de ponderación considera explícitamente todo par de vértices unidos por una arista. La función de ponderación del grafo de la figura 8.9, que representamos con una matriz de adyacencia, se puede codificar así:

La longitud de una carretera y el tiempo necesario para recorrerla pueden ser bien distintos. Téngase en cuenta que la segunda no es sólo función de la longitud de la carretera, pues depende además de la calidad del firme, del número de carriles, de la congestión del tráfico, etc.

test\_weighted\_graph.a.py

test\_weighted\_graph.a.py

```
1 from graph import Graph
2
3 G = Graph(V=range(6),
4           E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1), (4,3), (5,5)])
5
6 def d(u, v):
7     if u == 0:
8         if v == 1: return 4
9         elif v == 3: return 4
10    elif u == 1:
11        if v == 4:
12            return 1
13    elif u == 2:
14        if v == 4: return 0
15        if v == 5: return 2
16    elif u == 3:
17        if v == 0: return 1
18        elif v == 1: return 4
19    elif u == 4:
20        if v == 3: return 1
21    elif u == 5:
22        if v == 5: return 2
23    raise KeyError, '(%d,%d) has no weight' % (u,v)
24
25 for u in G.V:
26     print 'Aristas que parten de %s y sus pesos:' % u,
27     for v in G.succs(u):
28         print '(%s,%s): %s' % (u, v, d(u,v)),
29     print
```

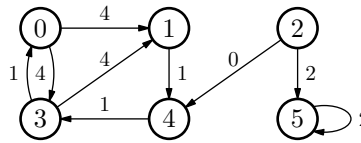


Figura 8.9: Grafo dirigido y ponderado.

Este es el resultado de ejecutar el programa:

```

Aristas que parten de 0 y sus pesos: (0,1): 4.    (0,3): 4.
Aristas que parten de 1 y sus pesos: (1,4): 1.
Aristas que parten de 2 y sus pesos: (2,4): 0.    (2,5): 2.
Aristas que parten de 3 y sus pesos: (3,0): 1.    (3,1): 4.
Aristas que parten de 4 y sus pesos: (4,3): 1.
Aristas que parten de 5 y sus pesos: (5,5): 2.

```

Codificar así la función de ponderación resulta farragoso. Acceder a un peso requiere, además, tiempo  $O(|V|)$ . Resulta sencillo implementar la función de ponderación de forma que su tiempo de ejecución sea  $\Theta(1)$ : basta con implementar directamente la matriz.

```

test_weighted_graph.b.py
test_weighted_graph.b.py
1 from graph import Graph
2
3 G = Graph(V=range(6),
4           E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1), (4,3), (5,5)])
5
6 dist_matrix = [[None, 4,    None, 4,    None, None ],
7                [None, None, None, None, 1,    None ],
8                [None, None, None, None, 0,    2    ],
9                [1,    4,    None, None, None, None ],
10               [None, None, None, 1,    None, None ],
11               [None, None, None, None, None, 2    ]]
12
13 def d(u, v):
14     return dist_matrix[u][v]
15
16 print 'Vértices:', G.V
17 for u in G.V:
18     print 'Aristas que parten de %s y sus pesos: ' % u,
19     for v in G.succs(u):
20         print '(%s,%s): %s' % (u, v, d(u,v)),
21     print

```

```

Vértices: [0, 1, 2, 3, 4, 5]
Aristas que parten de 0 y sus pesos: (0,1): 4.    (0,3): 4.
Aristas que parten de 1 y sus pesos: (1,4): 1.
Aristas que parten de 2 y sus pesos: (2,4): 0.    (2,5): 2.
Aristas que parten de 3 y sus pesos: (3,0): 1.    (3,1): 4.
Aristas que parten de 4 y sus pesos: (4,3): 1.
Aristas que parten de 5 y sus pesos: (5,5): 2.

```

Nótese que hemos codificado con *None* la ausencia de peso entre un par de vértices. Una alternativa hubiera sido codificar el «peso de las aristas sin peso», valga la expresión, con un valor «infinito» (o un valor suficientemente grande) o un valor negativo (si los pesos válidos son positivos), pero no con el valor cero, pues hay aristas cuyo peso es, precisamente, cero.

En lugar de usar una matriz podemos codificar la asociación entre aristas y pesos con un diccionario. De este modo nos ahorramos el problema de codificar «pesos de aristas sin peso».



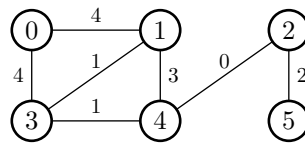


Figura 8.10: Grafo no dirigido y ponderado.

```

test_weighted_graph.c.py  test_weighted_graph.c.py
1 from graph import Graph
2
3 G = Graph(V=range(6),
4           E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1), (4,3), (5,5)])
5
6 dist_table = {(0,1):4, (0,3):4, (1,4):1, (2,4):0, (2,5):2,
7               (3,0):1, (3,1):4, (4,3):1, (5,5):2}
8
9 def d(u, v):
10     return dist_table[u,v]
11
12 print 'Vértices:', G.V
13 for u in G.V:
14     print 'Aristas que parten de %s y sus pesos:' % u,
15     for v in G.succs(u): print '(%s,%s): %s' % (u, v, d(u,v)),
16     print

```

```

Vértices: [0, 1, 2, 3, 4, 5]
Aristas que parten de 0 y sus pesos: (0, 1): 4.    (0, 3): 4.
Aristas que parten de 1 y sus pesos: (1, 4): 1.
Aristas que parten de 2 y sus pesos: (2, 4): 0.    (2, 5): 2.
Aristas que parten de 3 y sus pesos: (3, 0): 1.    (3, 1): 4.
Aristas que parten de 4 y sus pesos: (4, 3): 1.
Aristas que parten de 5 y sus pesos: (5, 5): 2.

```

El grafo de la figura 8.10 muestra un grafo no dirigido y ponderado que se codifica así con la clase que hemos definido:

```

test_weighted_graph.e.py  test_weighted_graph.e.py
1 from graph import Graph
2
3 G = Graph(V=range(6),
4           E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1), (4,3)],
5           directed=False)
6
7 dist = {(0,1):4, (0,3):4, (1,3):1, (1,4):3, (2,4):0, (2,5):2, (3,4):1}
8
9 def d(u, v):
10     if (u,v) in dist: return dist[u,v]
11     return dist[v,u]
12
13 print 'Vértices:', G.V
14 for u in G.V:
15     print 'Aristas que parten de %s y sus pesos:' % u,
16     for v in G.succs(u):
17         print '(%s,%s): %s' % (u, v, d(u,v)),
18     print

```

```

Vértices: [0, 1, 2, 3, 4, 5]
Aristas que parten de 0 y sus pesos: (0,1): 4.    (0,3): 4.
Aristas que parten de 1 y sus pesos: (1,0): 4.    (1,3): 1.    (1,4): 3.

```

Aristas que parten de 2 y sus pesos:	(2,4): 0.	(2,5): 2.
Aristas que parten de 3 y sus pesos:	(3,0): 4.	(3,1): 1.    (3,4): 1.
Aristas que parten de 4 y sus pesos:	(4,1): 3.	(4,2): 0.    (4,3): 1.
Aristas que parten de 5 y sus pesos:	(5,2): 2.	

Obsérvese cómo hemos codificado la función  $d$  para que no sea necesario almacenar en la tabla el peso de  $(v, u)$  si ya se ha almacenado el peso de  $(u, v)$ .

Los métodos de codificación de la función de ponderación son independientes del modo en que se codifica el propio grafo: no importa si usamos matrices o listas de adyacencia. Consideremos un ejemplo más: un grafo ponderado (el de la figura 7.5) implementado con listas de adyacencia.

```

mallorca.py
mallorca.py
1 from graph import Graph
2
3 Mallorca = Graph(V=['Alcúdia', 'Andratx', 'Artà', 'Calvià', 'Campos_del_Port',
4                    'Capdepera', 'Inca', 'Llucmajor', 'Manacor', 'Marratxí',
5                    'Palma_de_Mallorca', 'Pollença', 'Santanyí', 'Sóller'],
6                    E=[('Alcúdia', 'Artà'), ('Alcúdia', 'Inca'),
7                      ('Alcúdia', 'Pollença'), ('Andratx', 'Calvià'),
8                      ('Andratx', 'Palma_de_Mallorca'), ('Andratx', 'Sóller'),
9                      ('Artà', 'Capdepera'), ('Artà', 'Manacor'),
10                     ('Calvià', 'Palma_de_Mallorca'),
11                     ('Campos_del_Port', 'Llucmajor'),
12                     ('Campos_del_Port', 'Santanyí'), ('Inca', 'Manacor'),
13                     ('Inca', 'Marratxí'), ('Llucmajor', 'Palma_de_Mallorca'),
14                     ('Manacor', 'Santanyí'), ('Marratxí', 'Palma_de_Mallorca'),
15                     ('Pollença', 'Sóller')],
16                     directed=False)
17
18 km = {('Alcúdia', 'Artà')           : 36,
19       ('Alcúdia', 'Inca')           : 25,
20       ('Alcúdia', 'Pollença')       : 10,
21       ('Andratx', 'Calvià')         : 14,
22       ('Andratx', 'Palma_de_Mallorca') : 30,
23       ('Andratx', 'Sóller')         : 56,
24       ('Artà', 'Capdepera')         : 8,
25       ('Artà', 'Manacor')           : 17,
26       ('Calvià', 'Palma_de_Mallorca') : 14,
27       ('Campos_del_Port', 'Llucmajor') : 14,
28       ('Campos_del_Port', 'Santanyí') : 13,
29       ('Inca', 'Manacor')           : 25,
30       ('Inca', 'Marratxí')          : 12,
31       ('Llucmajor', 'Palma_de_Mallorca') : 20,
32       ('Manacor', 'Santanyí')       : 27,
33       ('Marratxí', 'Palma_de_Mallorca') : 14,
34       ('Pollença', 'Sóller')        : 54}
35
36 def d(u,v):
37     if (u,v) in km: return km[u,v]
38     else: return km[v,u]

```

```

test_weighted_graph_f.py
1 from mallorca import Mallorca, d
2
3 print 'Vértices:', G.V
4 for u in G.V:
5     print 'Aristas que parten de %s y sus pesos:' % u
6     for v in G.succs(u):
7         print '    (%s, %s): %s.' % (u, v, d(u,v))

```

```
python2.3: can't open file 'test_adjacency_lists_f.py'
```

### 8.4.1. Un caso especial: los grafos euclídeos

Merece consideración aparte el caso de los grafos euclídeos. La función de distancia es, en su caso, la distancia euclídea entre dos puntos. No es necesario, pues, almacenar una tabla con las distancias entre todo par de nodos: podemos efectuar el cálculo «al vuelo».

El grafo euclídeo de la figura 7.15 (página 161) se puede codificar así:

```
test.euclidean_graph.py test.euclidean_graph.py
1 from graph import Graph
2 from math import sqrt
3
4 G = Graph(V=[(0,6), (2,2), (2,6), (4,4), (4,0), (6,4)],
5           E=[((0,6),(2,2)), ((0,6),(2,6)), ((2,2),(4,4)), ((2,2),(4,0)),
6             ((2,2),(6,4)), ((2,6),(4,4)), ((4,0),(6,4))],
7           directed=False)
8
9 def d(u, v):
10     return sqrt((u[0]-v[0])**2 + (u[1]-v[1])**2)
11
12 print 'Vértices:', G.V
13 for u in G.V:
14     print 'Aristas que parten de %s y sus distancias:' % (u,)
15     for v in G.succs(u):
16         print '%s, %s): %4.2f' % (u, v, d(u,v))
```

```
Vértices: [(0, 6), (2, 2), (2, 6), (4, 4), (4, 0), (6, 4)]
```

```
Aristas que parten de (0, 6) y sus distancias:
```

```
((0, 6),(2, 6)): 2.00.
```

```
((0, 6),(2, 2)): 4.47.
```

```
Aristas que parten de (2, 2) y sus distancias:
```

```
((2, 2),(0, 6)): 4.47.
```

```
((2, 2),(4, 4)): 2.83.
```

```
((2, 2),(6, 4)): 4.47.
```

```
((2, 2),(4, 0)): 2.83.
```

```
Aristas que parten de (2, 6) y sus distancias:
```

```
((2, 6),(4, 4)): 2.83.
```

```
((2, 6),(0, 6)): 2.00.
```

```
Aristas que parten de (4, 4) y sus distancias:
```

```
((4, 4),(2, 6)): 2.83.
```

```
((4, 4),(2, 2)): 2.83.
```

```
Aristas que parten de (4, 0) y sus distancias:
```

```
((4, 0),(6, 4)): 4.47.
```

```
((4, 0),(2, 2)): 2.83.
```

```
Aristas que parten de (6, 4) y sus distancias:
```

```
((6, 4),(4, 0)): 4.47.
```

```
((6, 4),(2, 2)): 4.47.
```

## 8.5. Lectura/escritura de grafos

En el siguiente capítulo ilustraremos el funcionamiento de algunos algoritmos con grafos que especificamos con ficheros de texto. No hay una forma estándar de codificar la información de un grafo, así que tendremos que diseñar rutinas específicas para su lectura y escritura en fichero.

Ilustremos el problema con un grafo concreto: un mapa de ciudades de la península ibérica (en el que cada ciudad lleva asociadas sus coordenadas en el plano) conectadas por carreteras. Cada arista viene ponderada por la distancia euclídea que separa a las ciudades que conecta. El grafo ponderado en sí se especifica en un único fichero llamado

iberia. Cada línea del fichero contiene tres campos separados: ciudad de origen, ciudad de destino y distancia (euclídea) que los separa.

```

1 A_Coruña:Pontedeume:20.904
2 Abrantes:Castelo_Branco:67.303
3 Adanero:San_Rafael:39.312
4 Aguilar_de_Campo6:Burgos:56.790
5 Aguilar_de_Campo6:Reinosa:22.853
6 Alacant:Santa_Pola:9.234
7 Albacete:Almansa:57.257
8 Albacete:La_Roda:27.547
9 Albacete:Ruidera:77.504
10 Albocàsser:Xert:18.468
11 Albocàsser:L'Alcora:23.946
12 Albufeira:Ourique:53.544
13 Alcala_de_Henarés:Guadalajara:31.348
:
590 Ávila:Peñaranda_de_Bracamonte:48.620
591 Ávila:Adanero:35.523
592 Ávila:Arévalo:54.040
593 Écija:Lucena:62.371
594 Évora:Montemor-o-Novo:24.167
595 Évora:Beja:55.480
596 Évora:Grândula:59.931
597 Úbeda:Linares:25.871

```

La figura 8.11 muestra gráficamente la información almacenada en estos dos ficheros.

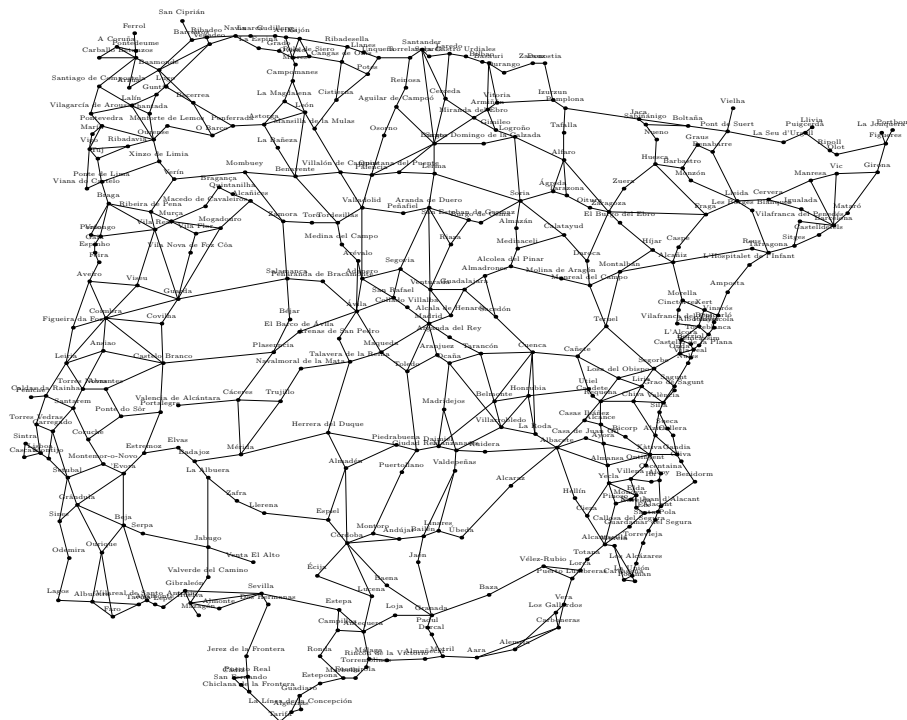


Figura 8.11: Mapa de ciudades y carreteras de la península ibérica.

Las coordenadas de cada ciudad se encuentran disponibles en otro fichero llamado iberia\_coords. Los datos que contiene resultan útiles para producir representaciones gráficas similares a la de la figura 8.11.

```

iberia_coords
iberia_coords
1 Aara:51.100:357.700
2 Abrantes:-346.020:70.080
3 Adanero:-71.540:-48.180
4 Aguilar_de_Campo6:-35.040:-232.140
5 Alacant:245.280:201.480
6 Albacete:137.240:132.860
7 Albocàsser:289.080:2.920
8 Albufeira:-360.620:297.840
:
370 Zaragoza:192.720:-119.720
371 Zarauz:110.960:-278.860
372 Zuera:208.780:-144.540
373 Ágreda:132.860:-141.620
374 Ávila:-77.380:-13.140
375 Écija:-119.720:270.100
376 Évora:-327.040:160.600
377 Úbeda:35.040:229.220

```

Diseñemos una rutina capaz de leer el grafo en cuestión como grafo no dirigido y ponderado en el que la distancia entre dos ciudades conectadas por carretera es la distancia euclídea. La función de lectura del grafo devuelve dos objetos: el grafo en sí y la función de ponderación.

```

load_graph.py
load_graph.py
1 from graph import Graph
2 from math import sqrt
3 from sets import Set
4
5 def load_weighted_graph(filename):
6     G = Graph(V=Set(), E=[], directed=False)
7     dist = {}
8     for line in file(filename):
9         cityA, cityB, aux = line.strip().split(':')
10        G.add_vertex( cityA )
11        G.add_vertex( cityB )
12        G.add_edge((cityA, cityB))
13        dist[cityA, cityB] = dist[cityB, cityA] = float(aux)
14
15    def d(A, B, coords=dist):
16        return dist[A,B]
17
18    return G, d

```

```

test_load_graph.py
test_load_graph.py
1 from load_graph import load_weighted_graph
2
3 G, d = load_weighted_graph('iberia')
4 print 'Ciudades conectadas a Castelló de la Plana',
5 print 'y distancia euclídea (aproximada):'
6 for v in G.succs('Castelló de la Plana'):
7     print '%s: %.3f' % (v, d('Castelló de la Plana', v))

```

```

Ciudades conectadas a Castelló de la Plana y distancia euclídea (aproximada):
Borriol: 6.529
Onda: 15.243
Benicàssim: 7.300
Vila-real: 9.234

```

## .....PROBLEMAS.....

- **134** Diseña una función que reciba un grafo y almacene en un fichero la información relativa a sus aristas como pares de vértices separados por el carácter «:».
- **135** Diseña una función que reciba un grafo ponderado (un grafo y una función de ponderación de las aristas) y almacene en un fichero la información relativa a sus aristas y pesos respectivos como tripletes «**origen:destino:peso**». Almacena el mapa de carreteras de Mallorca (véase la figura 7.5) en ese formato.
- .....

## Capítulo 9

# Algoritmos sobre grafos

En este capítulo presentamos algunos algoritmos sobre grafos. Nos limitamos a estudiar algunos de los considerados más relevantes o que resultarán instrumentales en otros capítulos.

- Exploración de grafos: procedimientos para visitar todos los vértices de un grafo en cierto orden inducido por las aristas y, si conviene, ejecutar una acción sobre cada uno de ellos:
  - Exploración por primero en anchura.
  - Exploración por primero en profundidad.
- Ordenación topológica (para grafos acíclicos).
- Componentes conexas.
- Clausura transitiva de un grafo (algoritmo de Warshall).
- Árbol de recubrimiento de coste mínimo (algoritmo de Kruskal).
- Camino más corto entre un par de vértices de un grafo o entre un vértice y todos los demás:
  - En grafos acíclicos.
  - En grafos ponderados positivos (algoritmo de Dijkstra).
  - En grafos sin bucles (algoritmo de Bellman-Ford).
- Camino más corto entre todo par de vértices (algoritmo de Floyd).

Al final del capítulo se glosan otros problemas de interés y se propone al estudiante una serie de ejercicios o problemas relacionados. Es aconsejable recurrir a la abundante bibliografía disponible para estudiar con detalle la solución de aquellos problemas que más puedan interesar al lector.

### 9.1. Exploración de grafos

Los primeros algoritmos sobre grafos que vamos a estudiar solucionan el problema de recorrer o explorar todos los vértices de un grafo a partir de sus conjuntos de aristas. No parecen tener gran interés por sí mismos, pero son instrumentales para el diseño de otros algoritmos.

Empezaremos por considerar el recorrido o exploración de un tipo particular de grafos, los árboles, para entender mejor las propiedades de las técnicas de recorrido aplicadas a grafos en general.

La técnicas de recorrido de grafos serán útiles cuando estudiemos la estrategia de «vuelta atrás» (backtracking).

### 9.1.1. Recorrido de árboles

El objetivo del recorrido de un árbol es visitar cada uno de sus vértices, considerando de algún modo las relaciones padre-hijo, para efectuar algún proceso sobre ellos. Los diferentes recorridos proporcionan órdenes de visita distintos. Algunos de los órdenes de visita resultan interesantes para efectuar ciertos cálculos sobre el árbol en su totalidad.

Estudiaremos cuatro tipos de recorrido de los árboles:

**Recorrido por niveles:**  
*Level-order traversal.*

**Recorrido por primero en profundidad:** *Depth-first traversal.*

**Recorrido en preorden:**  
*Preorder traversal.*

**Recorrido en postorden:**  
*Postorder traversal.*

**Recorrido en inorden:**  
*In-order traversal.*

- Recorrido **por niveles** o por primero en anchura.
- Recorrido **por primero en profundidad**. Hay tres variantes de este tipo de recorrido:
  - Recorrido en **preorden**.
  - Recorrido en **postorden**.
  - Recorrido en **inorden**, limitado a árboles binarios.

Como sólo pretendemos ilustrar los órdenes de recorrido con árboles para, más adelante, ilustrar esta misma actividad con grafos generales, nos limitaremos a un procesamiento de los vértices extremadamente sencillo: mostrarlos por pantalla.

#### Recorrido por niveles

El recorrido por niveles parte de la raíz y visita los vértices en orden de profundidad creciente. Primero visita la raíz; luego, todos sus hijos; luego, todos los nietos (primero los hijos del primer hijo, después los del segundo, y así sucesivamente); luego, todos los bisnietos (primero los del primer nieto, después los del segundo, etc.); y así sucesivamente. La figura 9.1 muestra un árbol y la figura 9.4 ilustra un recorrido por primero en profundidad.

El algoritmo se apoya en el uso de una cola FIFO para determinar el orden de visita de los vértices.

```

tree_traversal.py
1 from fifo import FIFO
2
3 def level_order_traversal(G, s): # G debe ser un árbol (codificado como un grafo).
4     Q = FIFO()
5     Q.append(s)
6     while not Q.is_empty():
7         u = Q.pop()
8         print u, # Procesa el vértice u.
9         for v in G.succs(u):
10            Q.append(v)

```

Probemos el algoritmo con el árbol de la figura 9.1.

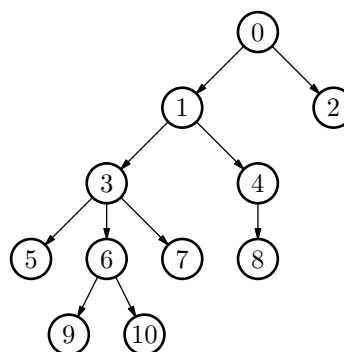


Figura 9.1: Un árbol.



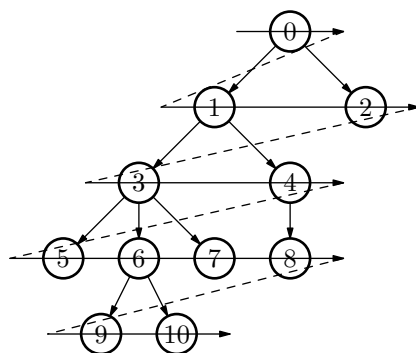


Figura 9.2: Recorrido por niveles del árbol de la figura 9.1.

test\_tree\_traversal.a.py

test\_tree\_traversal.a.py

```
1 from tree_traversal import level_order_traversal
2 from graph import Graph
3
4 G = Graph(V=range(11), E=[(0,1), (0,2), (1,3), (1,4), (3,5), (3,6), (3,7),
5                           (4,8), (6,9), (6,10)])
6 level_order_traversal(G, 0)
```

0 1 2 3 4 5 6 7 8 9 10

Nótese que el orden de visita coincide con la numeración utilizada en los vértices, que era, precisamente, una numeración por niveles (asignamos números sucesivos de arriba a abajo y de izquierda a derecha).

#### PROBLEMAS

► **136** Diseña una versión de la función de recorrido por niveles que explore un árbol representado mediante una lista de listas. El árbol de la figura 9.1 se suministrará a la función codificado así: [0, [1, [3, [5], [6, [9], [10]], [7]], [4, [8]]], [2]].

### Recorrido por primero en profundidad

El recorrido por primero en profundidad se entiende bien en términos recursivos. Consiste en visitar la raíz y, a continuación, disparar un recorrido por primero en profundidad sobre cada uno de los hijos.

tree\_traversal.py

tree\_traversal.py

```
12 def recursive_depth_first_traversal(G, s): # G es un árbol (un grafo).
13     print s, # Procesa el vértice s.
14     for v in G.succs(s): # Y visita a los subárboles asociados a cada hijo.
15         recursive_depth_first_traversal(G, v)
```

En el fondo, se trata de un algoritmo «divide y vencerás»: para resolver el problema «mostrar los nodos de un árbol», muestro la raíz y resuelvo el problema «mostrar los nodos de un árbol» en cada uno de sus subárboles.

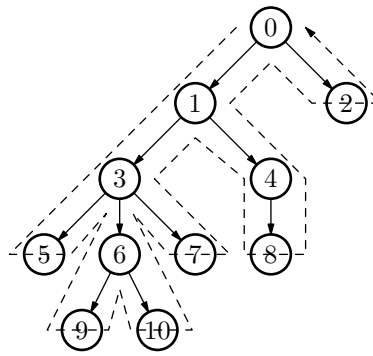
test\_tree\_traversal.b.py

test\_tree\_traversal.b.py

```
1 from tree_traversal import recursive_depth_first_traversal
2 from graph import Graph
3
4 G = Graph(V=range(11), E=[(0,1), (0,2), (1,3), (1,4), (3,5), (3,6), (3,7),
5                           (4,8), (6,9), (6,10)])
6 recursive_depth_first_traversal(G, 0)
```

0 1 3 5 6 9 10 7 4 8 2

Es posible diseñar una versión iterativa usando una cola LIFO que, en cierto sentido, simula la pila de llamadas a función:



**Figura 9.3:** Recorrido por primero en profundidad del árbol de la figura 9.1.

tree\_traversal.py

tree\_traversal.py

```

17 from lifo import LIFO
18
19 def iterative_depth_first_traversal(G, s): # G es un árbol (un grafo).
20     Q = LIFO()
21     Q.push(s)
22     while not Q.is_empty():
23         u = Q.pop()
24         print u, # Procesa el vértice u.
25         succs = list(G.succs(u))
26         succs.reverse()
27         for v in succs:
28             Q.push(v)

```

test\_tree\_traversal.c.py

test\_tree\_traversal.c.py

```

1 from tree_traversal import iterative_depth_first_traversal
2 from graph import Graph
3
4 G = Graph(V=range(11), E=[(0,1), (0,2), (1,3), (1,4), (3,5), (3,6), (3,7),
5                             (4,8), (6,9), (6,10)])
6 iterative_depth_first_traversal(G, 0)

```

0 1 3 5 6 9 10 7 4 8 2

Hemos dicho antes que hay tres tipos diferentes de recorrido por primero en profundidad: dos, los recorridos en preorden y postorden, son de aplicación a cualquier tipo de árbol y el tercero, recorrido en inorden, sólo es aplicable a los árboles binarios. Sólo se diferencian en el instante en que efectúan el procesamiento de los nodos. Presentamos algoritmos para los tres tipos de recorrido, pero sólo en su versión recursiva. (Nótese que *recursive\_depth\_first\_traversal* es el recorrido en preorden).

tree\_traversal.py

tree\_traversal.py

```

30 def preorder_traversal(G, s): # G es un árbol (un grafo).
31     print s, # Procesa el vértice s.
32     for v in G.succs(s):
33         preorder_traversal(G, v)
34
35 def postorder_traversal(G, s): # G es un árbol (un grafo).
36     for v in G.succs(s):
37         postorder_traversal(G, v)
38     print s, # Procesa el vértice s.
39
40 def in_order_traversal(G, s): # G es un árbol binario (un grafo).
41     if len(G.succs(s)) > 0:

```

```

42  in_order_traversal(G, G.succs(s)[0]) # Visita el subárbol izquierdo.
43  print s, # Procesa el vértice s.
44  in_order_traversal(G, G.succs(s)[1]) # Visita el subárbol derecho.

```

test\_tree\_traversal.d.py

test\_tree\_traversal.d.py

```

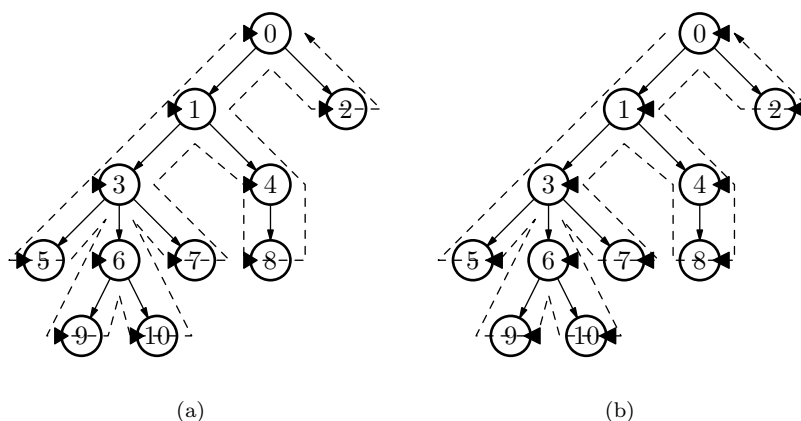
1  from tree_traversal import preorder_traversal, postorder_traversal
2  from graph import Graph
3
4  G = Graph(V=range(11), E=[(0,1), (0,2), (1,3), (1,4), (3,5), (3,6), (3,7),
5                             (4,8), (6,9), (6,10)])
6  print 'Recorrido en preorden:',
7  preorder_traversal(G, 0)
8  print
9  print 'Recorrido en postorden:',
10 postorder_traversal(G, 0)

```

```

Recorrido en preorden: 0 1 3 5 6 9 10 7 4 8 2
Recorrido en postorden: 5 9 10 6 7 3 8 4 1 2 0

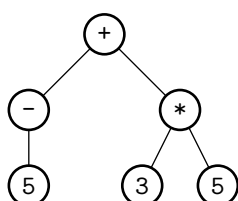
```



**Figura 9.4:** Recorrido en (a) preorden y (b) postorden del árbol de la figura 9.1. Siguiendo las líneas de trazo discontinuo se puede ver el orden de procesado de los vértices: los triángulos marcan los instantes en que se procesan los respectivos vértices.

#### PROBLEMAS

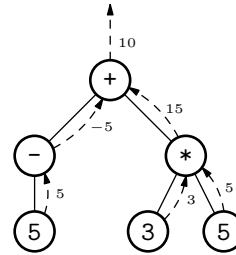
- **137** Haz una traza de *in\_order\_traversal* (véase el programa `tree_traversal.py`) para un árbol binario completo con 7 vértices.
- **138** Diseña versiones de los tres recorridos en profundidad (preorden, postorden e inorden) para árboles codificados mediante listas de listas.
- **139** El recorrido en postorden encuentra aplicación en la evaluación de expresiones aritméticas por parte de los intérpretes para lenguajes de programación. Una expresión aritmética puede representarse mediante un árbol. La expresión « $-5 + 3 * 5$ », por ejemplo, se puede representar con un árbol como éste:



El analizador del intérprete se encarga de «traducir» la secuencia de caracteres que forma la expresión en el correspondiente árbol que la representa. Nosotros partimos ya del árbol construido y sólo nos preocupa su evaluación. Para ello, podemos efectuar un recorrido del árbol en postorden con el cálculo y devolución de un valor (el resultado de evaluar un subárbol) como acción asociada a la visita de cada nodo. El procesamiento de cada nodo del árbol es éste:

- Si se trata de un nodo que alberga un valor numérico, se devuelve dicho valor.
- Si se trata de un nodo etiquetado con una operación, se obtiene el valor de cada uno de los hijos, se operan éstos de acuerdo con la etiqueta del nodo y se devuelve el valor numérico del resultado.

En esta figura se muestra el efecto del recorrido sobre el árbol del ejemplo:



Esta codificación de las expresiones aritméticas es muy similar a la propia del lenguaje de programación Lisp. La expresión «-5 + 3 \* 5» se codifica en Lisp con esta lista: «(+ (- 5) (\* 3 5))».

La lista de listas con la que representaremos un árbol como el de la figura es `[ "+", [ "-", [ 5 ], [ "*", [ 3 ], [ 5 ] ] ]`. El resultado de evaluar la expresión que representa ese árbol es el valor 10.

Diseña una función que reciba un árbol codificado como lista de listas y devuelva el resultado de evaluar la expresión que representa.

► **140** Disponemos de un lenguaje ensamblador para un ordenador que efectúa operaciones aritméticas con una pila. Las instrucciones para apilar y desapilar son **PUSH** *valor* y **POP**, respectivamente. Las siguientes instrucciones toman dos elementos de la pila, les aplican una operación (la cima de la pila es el operando derecho y el elemento que hay debajo de la pila es el operando izquierdo) y dejan el resultado en la pila: **ADD** (suma), **SUB** (resta), **MUL** y (producto) **DIV** (división). Una operación adicional, **CHSGN**, cambia el signo del elemento de la cima de la pila.

Diseña una función que reciba un árbol que codifica una expresión aritmética y muestre por pantalla las instrucciones de ensamblador que permiten evaluarla en el computador.

Por ejemplo, si le suministramos este árbol, `[ "+", [ "-", [ 5 ], [ "*", [ 3 ], [ 5 ] ] ]`, mostrará por pantalla el siguiente texto:

```
PUSH 5
CHSGN
PUSH 3
PUSH 5
MUL
SUM
.....
```

### 9.1.2. Exploración por primero en anchura en grafos cualesquiera

Veamos ahora cómo explorar los vértices de un grafo cualquiera. Empezaremos por la exploración por primero en anchura, que es similar al recorrido por niveles de los nodos de un árbol.

La exploración por primero en anchura o, simplemente, en anchura, propone el recorrido de los vértices de un grafo alcanzables desde un vértice dado al que denominamos vértice inicial. Sigue la siguiente estrategia:

1. Se marca como visitado el vértice que nos suministran y se inserta en una cola FIFO.

**Exploración por primero en anchura:** *Breadth-first search (BFS)*.

2. Se repite el siguiente procedimiento hasta que la cola esté vacía:

- 2.1. Se extrae el primer vértice  $u$  de la cola y se consideran todos sus sucesores. Si un sucesor  $v$  no había sido visitado aún, se marca como visitado y se añade a la cola FIFO.

Nótese por qué decimos que trata de una generalización de recorrido por niveles en un árbol: primero se visita el vértice de partida (en el árbol sería la raíz); luego, sus sucesores (en el árbol, sus hijos); luego, los sucesores de los sucesores (en el árbol, los nietos); y así sucesivamente. La dificultad que plantean los grafos generales es que, al poderse llegar a un mismo vértice desde otro por más de un camino, necesitamos «marcar» aquellos que ya han sido explorados para no visitarlos más de una vez.

Si no hay un orden establecido entre los sucesores de un vértice no habrá un único recorrido posible el efectuar la exploración por primero en anchura.

### Una implementación básica

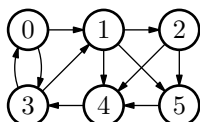
Nuestra primera implementación del método requiere que se suministre un grafo (una instancia de cualquier clase con el interfaz propuesto para un grafo) y un vértice a partir del cual iniciar la búsqueda:

```

1 from fifo import FIFO
2 from sets import Set
3
4 def breadth_first_search(G, s):
5     Q = FIFO()
6     visited = Set([s])
7     Q.append(s)
8     while not Q.is_empty():
9         u = Q.pop()
10        for v in G.succs(u):
11            if v not in visited:
12                visited.add(v)
13                Q.append(v)
```

#### PROBLEMAS

► 141 Haz una traza de *breadth\_first\_search* al invocarse sobre el vértice 0 de este grafo:



Indica el orden con el que ingresan los vértices en el conjunto *visited*.

El único efecto de esta función es marcar los vértices visitados como tales, pero al no devolver resultado alguno (ni modificar variable alguna o mostrar algo por pantalla), resulta ineficaz. Resultaría interesante poder pasar una función que el procedimiento invoque sobre cada vértice en el momento en que éste sea extraído de la cola.



bfs1.py

bfs1.py

```

1 from fifo import FIFO
2 from sets import Set
3
4 def breadth_first_search(G, s, action):
5     Q = FIFO()
6     visited = Set([s])
7     Q.append(s)
8     while not Q.is_empty():
9         u = Q.pop()
```

```

10     action(u)
11     for v in G.succs(u):
12         if v not in visited:
13             visited.add(v)
14             Q.append(v)

```

La función *action* tiene un solo parámetro: el vértice que se procesa. Podemos poner a prueba nuestra función con este sencillo programa:

```

test_bfs1.py
1 from fifo import FIFO
2 from graph import Graph
3 from bfs1 import breadth_first_search
4
5 G = Graph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1),
6                           (4,3), (5,5)])
7
8 def show_node(u):
9     print "Visitando el vértice", u
10
11 breadth_first_search(G, 0, show_node)

```

El programa efectúa una prueba con el grafo de la figura 9.5.

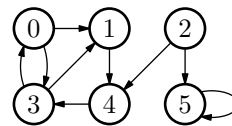


Figura 9.5: Un grafo dirigido.

Éste es el resultado de ejecutar `test_bfs1.py`:

```

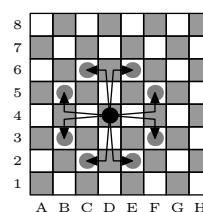
Visitando el vértice 0
Visitando el vértice 1
Visitando el vértice 3
Visitando el vértice 4

```

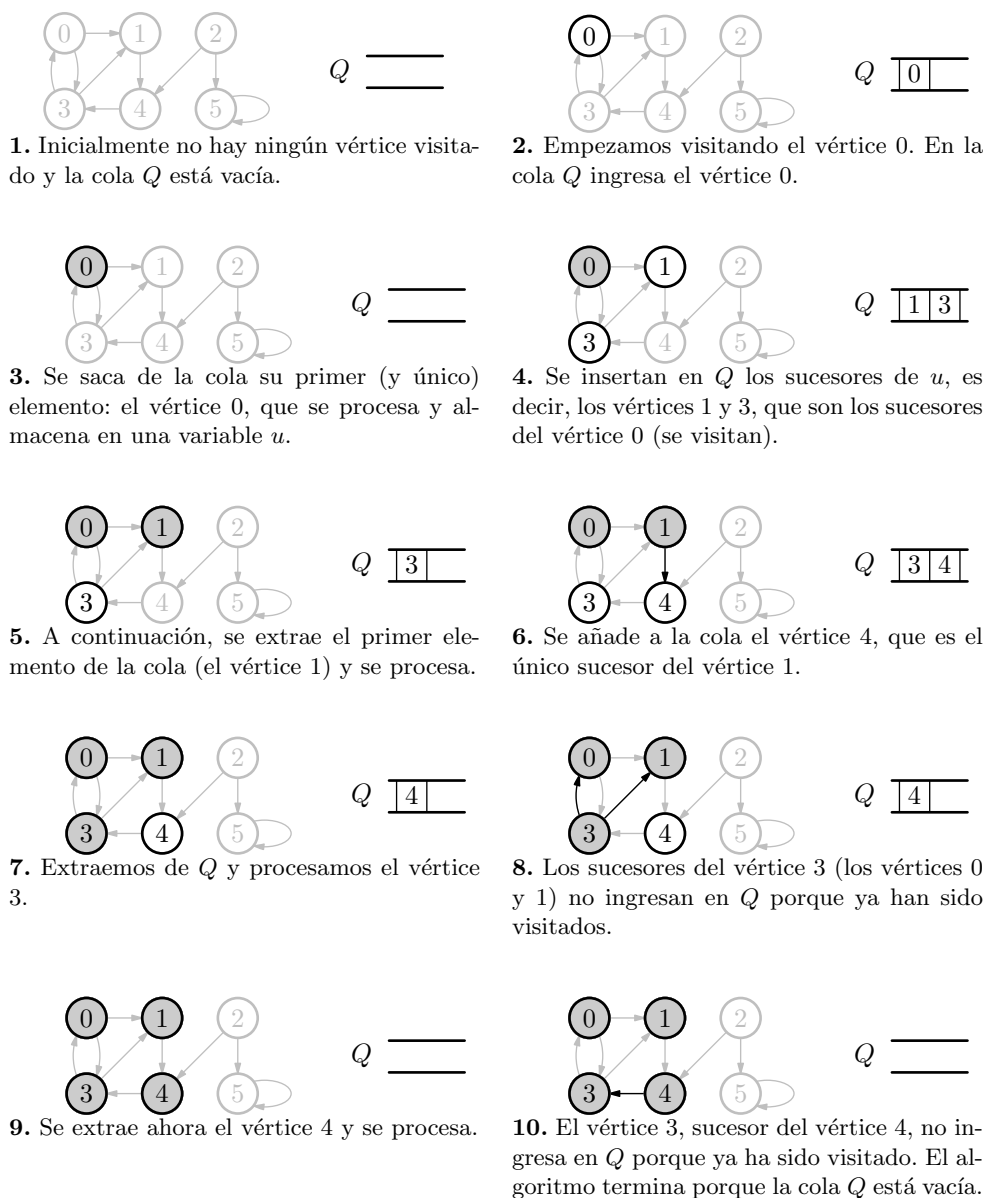
Veamos cómo funciona el algoritmo. Durante la ejecución, los vértices se pueden dividir en tres conjuntos: intactos, visitados y procesados. En la figura 9.6, que muestra gráficamente una traza paso a paso, dibujamos los vértices intactos con trazo gris, los visitados con trazo continuo y los procesados con trazo continuo y relleno gris. En cada una de las figuras se muestra también el estado actual de la cola *Q*.

.....PROBLEMAS.....

► 142 El caballo efectúa un curioso movimiento en el juego de ajedrez. En la siguiente figura se marcan todos los escaques alcanzables desde el escaque D4, que son E6, F5, F3, E2, C2, B3, B5 y C6.



Diseña un programa que permita responder a las siguientes preguntas mostrando el conjunto de casillas alcanzables desde una casilla determinada:

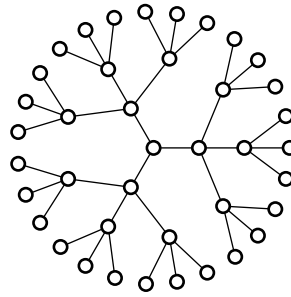


**Figura 9.6:** Trazo de la exploración por primero en anchura sobre el grafo de la figura 9.5 empezando en el vértice 0.

- ¿Es posible alcanzar cualquier casilla del tablero desde cualquier otra con un caballo?
- ¿Qué casillas podemos alcanzar con un peón blanco que parte del escaque D1?
- ¿Podemos alcanzar cualquier casilla de un tablero de  $3 \times 100$  con un caballo ubicado inicialmente en el escaque A1? ¿Y en un tablero de  $2 \times 100$ ?

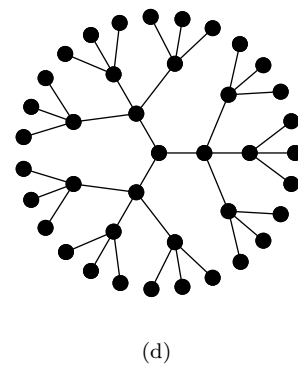
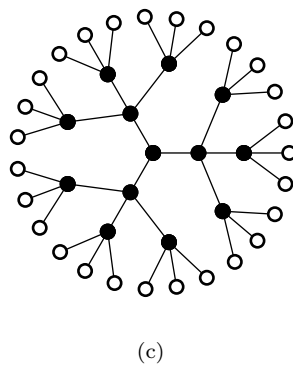
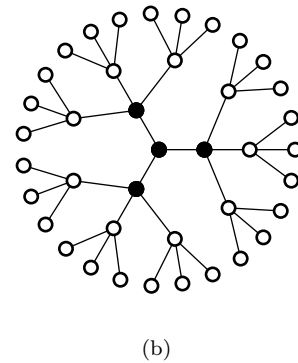
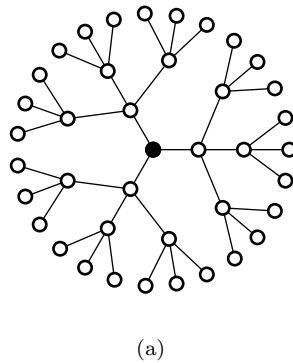
La figura 9.7 muestra un grafo sobre el que presentamos una traza del algoritmo de exploración en anchura. Hemos dispuesto los vértices en circunferencias concéntricas alrededor del vértice en el que vamos a iniciar una exploración.

Analicemos el vértice de la exploración en los instantes de la ejecución que se recogen en la figura 9.8. Inicialmente se visita el vértice central (figura 9.8 (a)). Como todos sus vértices adyacentes se consideran a continuación e ingresan en la cola, serán los siguientes vértices en salir de ella. Así pues, los siguientes vértices en ser visitados son los que se encuentran separados por una arista del vértice central (figura 9.8 (b)). En la



**Figura 9.7:** Un grafo no dirigido en el que los vértices se han dispuesto en circunferencias concéntricas según la longitud que los separa del vértice central.

cola habrán ingresado entonces todos los vértices adyacentes a estos vértices, es decir, habrán ingresado todos los vértices separados del central por dos aristas (figura 9.8 (c)).

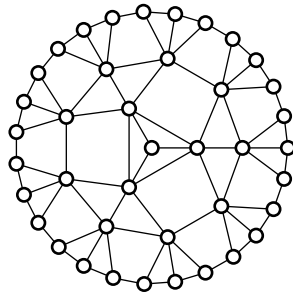


**Figura 9.8:** Trazo de la exploración por primero en anchura sobre el grafo de la figura 9.7. (a) Iniciamos un recorrido en anchura en el vértice central, (b) se visitan primero los vértices alcanzables desde él (primera circunferencia); a continuación, (c) los alcanzables desde los vértices marcados en la anterior figura; y, finalmente, (d) los vértices exteriores.

#### PROBLEMAS

► **143** Haz una traza del algoritmo de recorrido por primero en anchura en el grafo que mostramos a continuación. Empieza por el vértice central.





### Un algoritmo que visita todos los vértices

Es fácil advertir que el algoritmo no necesariamente visita todos los vértices del grafo: sólo recorre los vértices alcanzables desde el vértice inicial.

Los vértices 2 y 5 del grafo de la figura 9.5 no se visitan si iniciamos el recorrido en los vértices 0, 1, 3 o 4. Modificando ligeramente la función *breadth\_first\_search*, podemos usarla desde otra que sí efectúa un recorrido por todos los vértices del grafo:

```
bfs.py
1 from fifo import FIFO
2 from sets import Set
3
4 def breadth_first_search(G, s, action, visited = Set()):
5     Q = FIFO()
6     visited.add(s)
7     Q.append(s)
8     while not Q.is_empty():
9         u = Q.pop()
10        action(u)
11        for v in G.succs(u):
12            if v not in visited:
13                visited.add(v)
14                Q.append(v)
15
16 def full_breadth_first_search(G, action):
17     visited = Set()
18     for u in G.V:
19         if u not in visited:
20             breadth_first_search(G, u, action, visited)
```

```
test_bfs.py
1 from fifo import FIFO
2 from graph import Graph
3 from bfs import breadth_first_search, full_breadth_first_search
4
5 G = Graph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1),
6                          (4,3), (5,5)])
7
8 def show_node(u):
9     print "Visitando el vértice", u
10
11 print "Resultado de breadth_first_search sobre 0"
12 breadth_first_search(G, 0, show_node)
13
14 print
15 print "Resultado de full_breadth_first_search"
16 full_breadth_first_search(G, show_node)
```

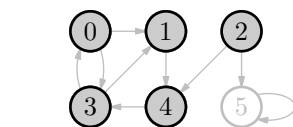
```

Resultado de breadth_first_search sobre 0
Visitando el vértice 0
Visitando el vértice 1
Visitando el vértice 3
Visitando el vértice 4

Resultado de full_breadth_first_search
Visitando el vértice 0
Visitando el vértice 1
Visitando el vértice 3
Visitando el vértice 4
Visitando el vértice 2
Visitando el vértice 5

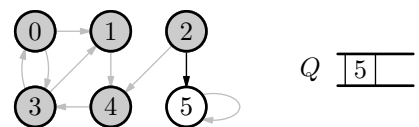
```

La función *full\_breadth\_first\_search* selecciona inicialmente un vértice al azar. Supongamos que selecciona el vértice 0. Si hacemos una traza paso a paso, llegaremos a la misma situación descrita en el décimo paso de la traza mostrada en la figura 9.6. El bucle de la línea 18 de *bfs.py* recorre el resto de vértices y descarta los ya visitados. Así pues, selecciona a continuación un vértice no visitado cualquiera. En la figura 9.9 se completa la traza.



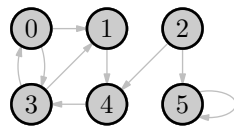
11. Supongamos que se selecciona ahora el vértice 2. Al llamar a la función *breadth\_first\_search*(*G*, 2, *visited*) se visita y procesa el vértice 2.

$Q =$



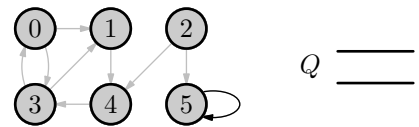
12. Ingresa en *Q* el vértice 5, pero no el vértice 4.

$Q = \boxed{5}$



13. El siguiente paso consiste, pues, en extraer de *Q* el vértice 5 y procesarlo.

$Q =$



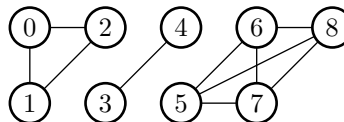
14. El único sucesor del vértice 5 es el mismo vértice 5, pero no ingresa en *Q* porque ya ha sido visitado.

$Q =$

**Figura 9.9:** Traza de la exploración de todos los vértices por primero en anchura sobre el grafo de la figura 9.5. Esta figura muestra los pasos que se dan después de los que se ilustran en la figura 9.6.

#### PROBLEMAS

► 144 Haz una traza del algoritmo *full\_breadth\_first\_search* sobre este grafo no dirigido:



#### Análisis de complejidad

El coste del algoritmo depende de la representación del grafo. Si el grafo tiene  $|V|$  vértices y  $|E|$  aristas y se representa mediante listas de adyacencia, el coste temporal es  $O(|V| + |E|)$ : se visitan todos los vértices una sola vez y para cada uno de ellos se recorren todas las aristas que parten de él. Si el grafo se representa mediante una matriz de adyacencia, el coste temporal pasa a ser  $O(|V|^2)$ , pues el cálculo de los sucesores

requiere tiempo  $O(|V|)$ . El espacio necesario es  $O(|V|)$ , pues la pila  $Q$  puede llegar a almacenar  $|V| - 1$  vértices.

#### PROBLEMAS

► **145** Escribe una función que determine si un grafo no dirigido es conexo mediante una exploración en anchura de sus vértices. ¿Cuál es el coste temporal del algoritmo?

► **146** El recorrido por primero en profundidad permite detectar si un grafo presenta ciclos. Implementa una función *detecta\_ciclos* que devuelva *True* si el grafo contiene ciclos y *False* en caso contrario. ¿Qué complejidad computacional tiene?

La detección de ciclos es un problema de gran interés práctico para, por ejemplo, detectar situaciones de bloqueo mutuo (*deadlock*), es decir, situaciones en las que una tarea espera a la finalización de otra que, directa o indirectamente depende de la primera.

### Una aplicación del recorrido por primero en anchura: cálculo del número de aristas que separan a dos vértices

Hemos visto que la exploración por primero en anchura recorre los vértices por niveles. Tras visitar el primer vértice, visita a todos los que se encuentran a una arista de distancia del primero. Sólo cuando los ha visitado todos, visita todos los vértices alcanzables desde el primero con dos aristas. Y así sucesivamente. Podemos aprovechar esta propiedad para diseñar un algoritmo que calcule el menor número de aristas que se deben recorrer para ir de un vértice  $s$  a un vértice  $t$ :

```
bfs_shortest_path_length.py
1 from fifo import FIFO
2 from sets import Set
3
4 def bfs_shortest_path_length(G, s, t):
5     Q = FIFO()
6     visited = Set([s])
7     length = {s: 0}
8     if s == t:
9         return 0
10    Q.append(s)
11    while not Q.is_empty():
12        u = Q.pop()
13        if u == t:
14            return length[t]
15        for v in G.succs(u):
16            if v not in visited:
17                visited.add(v)
18                length[v] = length[u] + 1
19                Q.append(v)
20        else:
21            if length[u] + 1 < length[v]:
22                length[v] = length[u] + 1
23    return None
```

Probemos el programa con el grafo de la figura 9.5:

```
test_bfs_shortest_path_length.py
1 from graph import Graph
2 from bfs_shortest_path_length import bfs_shortest_path_length
3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1),
5                           (4,3), (5,5)])
6
7 print 'Número de aristas entre 0 y 3:', bfs_shortest_path_length(G, 0, 3)
8 print 'Número de aristas entre 0 y 4:', bfs_shortest_path_length(G, 0, 4)
9 print 'Número de aristas entre 2 y 0:', bfs_shortest_path_length(G, 2, 0)
```

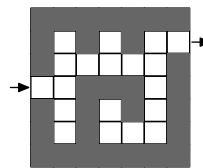
Número de aristas entre 0 y 3: 1
Número de aristas entre 0 y 4: 2
Número de aristas entre 2 y 0: 3

Más adelante estudiaremos un algoritmo para el problema del camino más corto en un grafo (no el de menor número de aristas, sino el de menor suma de pesos o distancias) que se inspira en la exploración por primero en anchura, pero que usa una cola de prioridad en lugar de una cola FIFO: el algoritmo de Dijkstra.

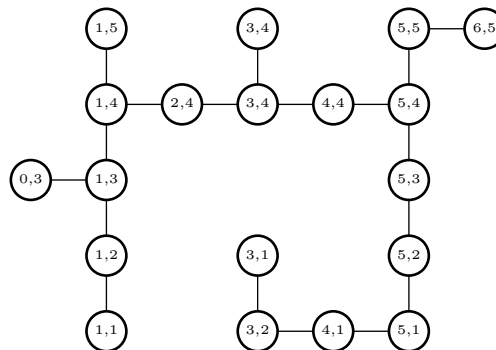
.....PROBLEMAS.....

► **147** Diseña un programa que calcule el menor número de ciudades que hemos de travesar para viajar entre ciudades cualesquiera del mapa de la península ibérica descrito en el fichero *iberia*.

► **148** Podemos representar un laberinto sobre un tablero matricial de  $n \times m$  casillas en el que cada casilla es pared, pasillo, entrada o salida y en el que la única entrada y la única salida se encuentran en los bordes del tablero. He aquí una ilustración de un laberinto que cumple estas condiciones:



Podemos modelar un laberinto con un grafo no dirigido en el que cada vértice es una casilla etiquetada con pasillo. Si dos casillas «pasillo» son vecinas, entonces hay una arista uniendo sus respectivos vértices. He aquí el grafo que modela el anterior laberinto:



Diseña:

- Una función que, dada una descripción de un laberinto como lista de cadenas devuelva un grafo  $G = (V, E)$  que lo represente en el que el vértice 0 corresponde a la casilla de entrada y el vértice  $|V| - 1$  corresponda a la casilla de salida. Cada carácter de las cadenas codificará el tipo de la casilla correspondiente (por ejemplo, 'x' es pared, '□' es pasillo, 'e' es entrada y 's' es salida). Por ejemplo, esta matriz modela el laberinto del ejemplo:

```
[ 'xxxxxxx',
  'x□x□x□s',
  'x□xxx□x',
  'e□x□x□x',
  'x□x□□□x',
  'xxxxxxx']
```

- Una función que detecte si el laberinto tiene salida.
- Una función que nos devuelva la longitud del camino más corto entre el vértice de entrada y el de salida.

- d) Una función que nos devuelva la secuencia de vértices que corresponde al camino más corto entre la entrada y la salida.

Debes analizar la complejidad computacional de cada uno de los algoritmos diseñados.

### 9.1.3. Exploración por primero en profundidad en grafos cualesquiera

La exploración por primero en profundidad también parte de un vértice inicial y explora todos los vértices alcanzables desde él. Puede formularse como un algoritmo recursivo que se invoca sobre un vértice  $u$ :

**Exploración por primero en profundidad:** *Depth-first search (DFS).*

1. Procesar el vértice  $u$  y marcarlo como visitado.
2. Para todo vértice  $v$  sucesor de  $u$ , si no ha sido visitado previamente, invocar este método recursivamente sobre él.

Al igual que ocurría con el recorrido por primero en anchura, no hay un único recorrido por primero en profundidad, pues no hay un orden definido entre los sucesores de un vértice.

#### Una implementación básica

Resulta inmediato codificar el procedimiento mediante una función recursiva:

```
dfs.py dfs.py
1 from sets import Set
2
3 def depth_first_search(G, u, action, visited = Set()):
4     action(u)
5     visited.add(u)
6     for v in G.succs(u):
7         if v not in visited:
8             depth_first_search(G, v, action, visited)
```

Esta primera versión del recorrido por primero en profundidad es una generalización del recorrido en preorden de un árbol: el vértice se procesa tan pronto se invoca el método *depth\_first\_search* sobre él, antes de iniciar la visita de sus sucesores. Una versión simétrica generaliza el recorrido en postorden procesando cada vértice únicamente cuando todos sus sucesores han sido procesados:

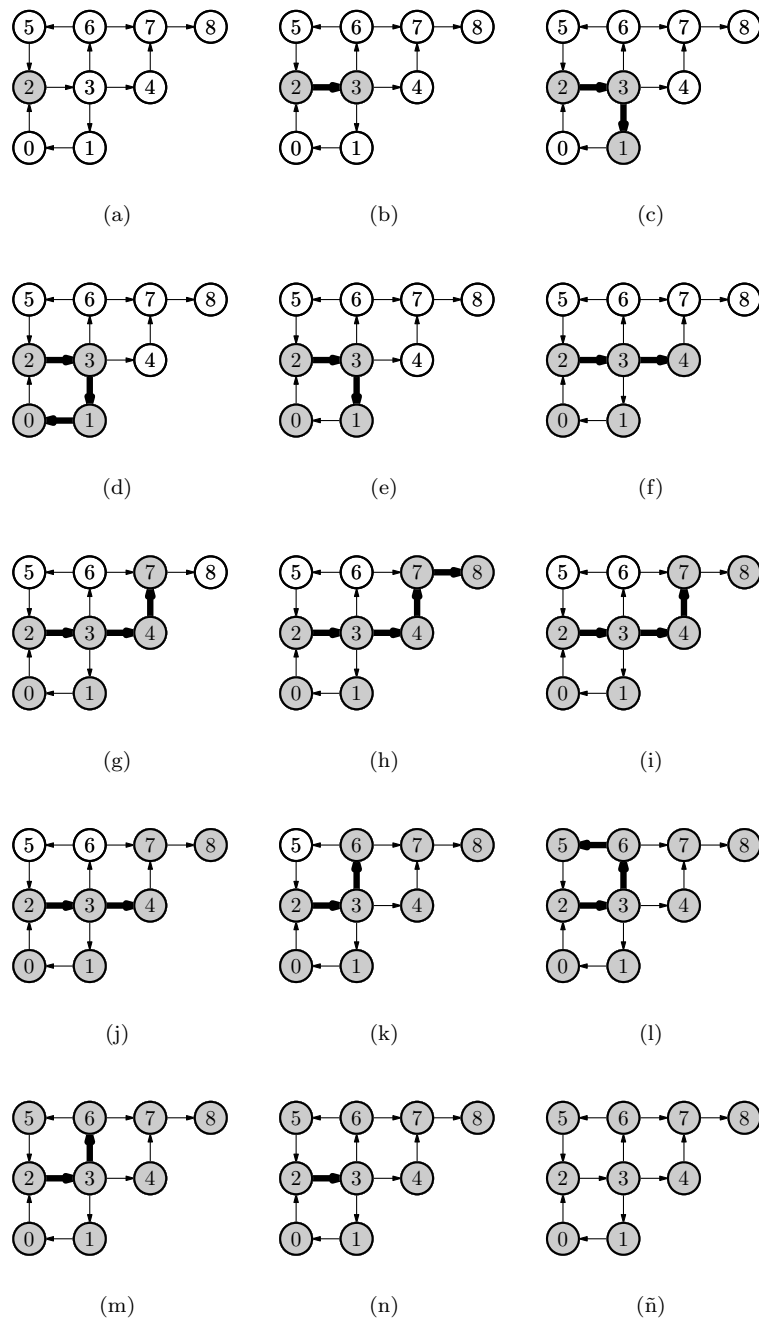
```
dfs.py dfs.py
10 def depth_first_search_post(G, u, action, visited = Set()):
11     visited.add(u)
12     for v in G.succs(u):
13         if v not in visited:
14             depth_first_search(G, v, action, visited)
15     action(u)
16
17 depth_first_search_post = depth_first_search
```

La exploración por primero en profundidad no tiene por qué implementarse mediante un algoritmo recursivo: podemos utilizar un algoritmo iterativo similar al presentado con *breadth\_first\_search*, pero usando una cola LIFO (una pila) en lugar de una cola FIFO. Esa cola LIFO simula la pila de llamadas recursivas.

La figura 9.10 ilustra gráficamente el orden de recorrido de vértices por primero en profundidad en un grafo.

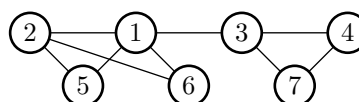
.....PROBLEMAS.....

- 149 Implementa una versión iterativa de la búsqueda por primero en profundidad.



**Figura 9.10:** Trazo de la exploración por primero en profundidad empezando en el vértice 2.

► **150** Haz una traza del algoritmo de exploración por primero en profundidad sobre este grafo no dirigido:



► **151** Diseña un programa que construya laberintos siguiendo una estrategia basada en la exploración por primero en profundidad. Codificaremos el laberinto de modo diferente al propuesto en el ejercicio 148. Usaremos una matriz en la que cada celda indica si hay pared al norte, al sur, al este y/o al oeste. He aquí un laberinto de este tipo:

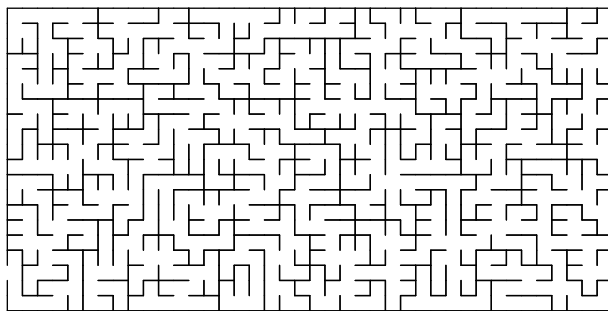


Si codificamos cada celda de la matriz con una cadena que indica qué muros están presentes ('n' para norte, 's' para sur, 'w' para oeste, y 'e' para este), el laberinto del ejemplo se describirá con esta matriz:

```
[['wn', 'nes', 'wns', 'ne'],
 ['ws', 'n', 'ne', 'w'],
 ['ns', 'e', 'wes', 'we'],
 ['wns', 's', 'ns', 'es']]
```

Inicialmente, todas las celdas de la matriz tienen cuatro muros. Inicia un recorrido *aleatorio* del laberinto (escoge las direcciones inexploradas al azar) por primero en profundidad desde una de las casillas exteriores. Ve destruyendo los muros que separan celdas conforme avanza la exploración. Al final, rompe los muros exteriores de dos casillas para que el laberinto tenga una entrada y una salida.

He aquí un grafo generado por el procedimiento descrito sobre una matriz de  $40 \times 20$  celdas:



Resulta más eficiente codificar con cuatro bits la existencia de paredes. El valor 15, que en binario es 1111, representaría una casilla con pared en las cuatro direcciones. Recuerda que puedes activar y desactivar bits con operaciones como  $\&$  y  $|$ . Y aún más eficiente resulta codificar en cada casilla la posible existencia de sólo dos muros: el muro este y el sur. Es posible porque, salvo en la casillas de los bordes oeste y norte, la existencia de un muro oeste se deduce de la existencia de un muro este en la casilla de su izquierda, y la existencia de un muro norte se deduce de la existencia de un muro sur en la casilla de encima.

### Un algoritmo que visita todos los vértices

Este algoritmo adolece del mismo problema que presentaba la primera versión del algoritmo de exploración por primero en anchura: sólo recorre los vértices alcanzables desde  $u$ . He aquí una función adicional que explora completamente el grafo por primero en profundidad:

```
dfs.py
19 def full_depth_first_search(G, action):
20     visited = Set()
21     for v in G.V:
22         if v not in visited:
23             depth_first_search(G, v, action, visited)
```

```
test dfs.py
1 from fifo import FIFO
2 from graph import Graph
3 from dfs import depth_first_search, full_depth_first_search
4
5 G = Graph(V=range(6), E=[(0,1), (0,3), (1,4), (2,4), (2,5), (3,0), (3,1),
6                        (4,3), (5,5)])
7
8 def show_node(u):
9     print "Visitando el vértice", u
10
11 print "Resultado de depth_first_search sobre 0"
12 depth_first_search(G, 0, show_node)
13
14 print "Resultado de full_depth_first_search"
15 full_depth_first_search(G, show_node)
```

```

Resultado de depth_first_search sobre 0
Visitando el vértice 0
Visitando el vértice 1
Visitando el vértice 4
Visitando el vértice 3
Resultado de full_depth_first_search
Visitando el vértice 0
Visitando el vértice 1
Visitando el vértice 4
Visitando el vértice 3
Visitando el vértice 2
Visitando el vértice 5

```

### Análisis de complejidad

El algoritmo de exploración por primero en profundidad es  $O(|V| + |E|)$  si el grafo se representa con listas o conjuntos de adyacencia y  $O(|V|^2)$  si se representa con una matriz de adyacencia: cada vértice del grafo es visitado una sola vez y para cada vértice visitado se recorren todas sus aristas de salida. Su complejidad espacial es  $O(|V|)$ : la pila de llamadas recursivas puede llegar a tener altura  $|V|$ .

.....PROBLEMAS.....

► **152** Haz una traza del algoritmo de exploración por primero en profundidad sobre el grafo de la figura 9.5:

- a) Empezando en el vértice 0.
- b) Empezando en el vértice 2.

► **153** Muestra ordenadamente los vértices visitados al explorar por primero en profundidad en el grafo de la figura 9.7:

- a) Empezando por el vértice central.
- b) Empezando por alguno de los vértices de la circunferencia exterior.

► **154** Resuelve el problema de determinar si un laberinto tiene salida o no (problema 148, apartado b) siguiendo un recorrido por primero en profundidad. ¿Qué ocurre en el mejor caso cuando se busca la salida del laberinto? ¿Y en el peor?

.....

## 9.2. Ordenación topológica

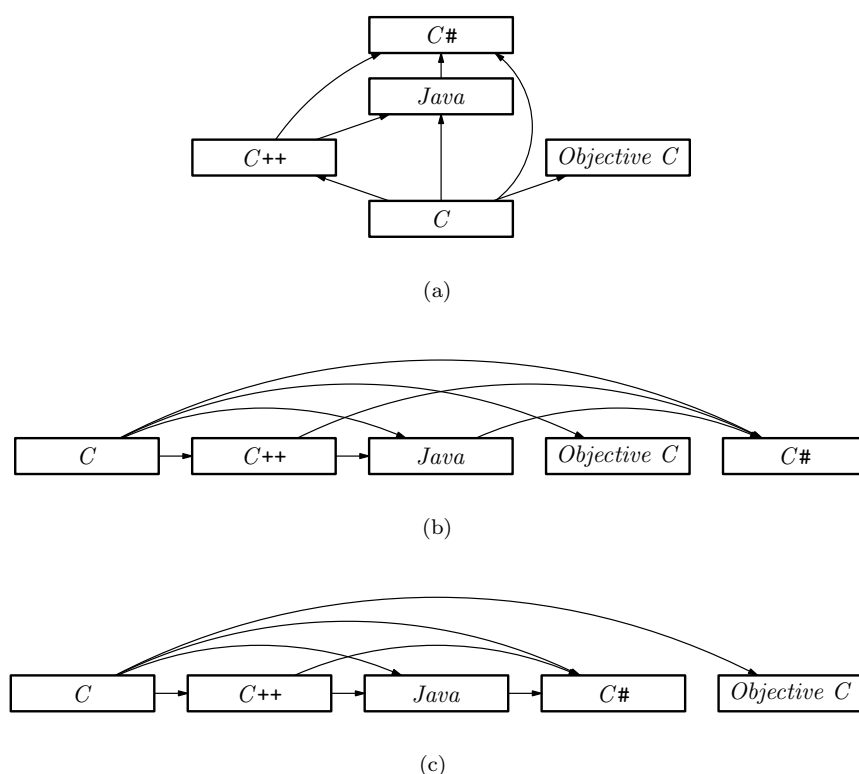
Los grafos acíclicos presentan una propiedad interesante: es posible encontrar un orden lineal de sus vértices tal que, para toda arista  $(u, v)$ , el vértice  $u$  precede al vértice  $v$ .

Consideremos el grafo acíclico de los lenguajes de programación. La figura 9.11 (a) muestra una posible representación de ese grafo. La representación mostrada en la figura 9.11 (b), dispone los vértices de izquierda a derecha en un orden que satisface la propiedad enunciada: las aristas siempre parten de un vértice más a la izquierda que el vértice de llegada. No tiene por qué haber un único ordenamiento que satisfaga esta propiedad. El ordenamiento de la figura 9.11 (c), por ejemplo, también la observa.

Si pensamos en que los vértices son actividades que se van a realizar y las aristas determinan relaciones de precedencia entre ellas (si  $(u, v)$  es una arista, hemos de llevar a cabo la actividad  $u$  después de completar la actividad  $v$ ) veremos que el ordenamiento topológico describe un orden de ejecución de tareas que no viola ninguna de las relaciones de precedencia.

Un **algoritmo de ordenación topológica** encuentra un orden como el descrito en un grafo acíclico. Resulta de ayuda pensar en términos de actividades y relaciones de precedencia para diseñar una estrategia de ordenación. La idea consiste en asignar a cada tarea  $v$  un «instante de finalización»  $f[v]$ , es decir, un valor numérico entre 1 y  $|V|$  que indica el instante de tiempo en que podemos ejecutarla.





**Figura 9.11:** Tres representaciones del grafo de los lenguajes de programación. (a) Representación arbitraria. (b) Representación que muestra los vértices ordenados topológicamente de izquierda a derecha. (c) Representación similar a la anterior, pero con un orden topológico distinto.

Si hay una relación  $(u, v)$ , la tarea  $v$  debe ejecutarse después de completar la tarea  $u$ , así que le corresponde un «instante de finalización» superior. Esto se puede conseguir si efectuamos un recorrido por primero en profundidad ejecutando una acción en postorden: la asignación del «instante de finalización». Eso sí, hemos de asignar «instantes de finalización» en orden decreciente. Este programa ilustra la idea:

```

topsort.py
1 from sets import Set
2
3 def topsort_from(G, u, visited, f):
4     global _counter
5     visited.add(u)
6     for v in G.succs(u):
7         if v not in visited:
8             topsort_from(G, v, visited, f)
9     _counter -= 1
10    f[_counter] = u
11
12 def topsort(G):
13     global _counter
14     visited = Set()
15     f = [None] * len(G.V)
16     _counter = len(G.V)
17     for v in G.V:
18         if v not in visited:
19             topsort_from(G, v, visited, f)
20     return f

```

```

test_topsort.py

```

```

1 from graph import Graph
2 from topsort import topsort
3
4 G = Graph(V=['C', 'C++', 'Java', 'C#', 'Objective_C'],
5           E=[('C', 'C++'), ('C', 'Java'), ('C', 'C#'), ('C', 'Objective_C'),
6             ('C++', 'Java'), ('C++', 'C#'), ('Java', 'C#')])
7
8 for u in topsort(G):
9     print u

```

```

C
C++
Java
Objective C
C#

```

**Teorema 9.1** La función *topsort* devuelve el conjunto de vértices de un grafo acíclico ordenado topológicamente.

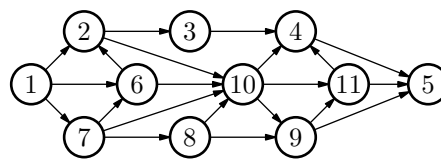
*Demostración.* Cada vértice se visita una sola vez y recibe, al finalizar la visita, un valor numérico determinado por un contador, así que cada vértice  $v$  recibe un valor numérico  $f[v]$  diferente. Basta con demostrar que si  $(u, v) \in E$ ,  $f[u] < f[v]$ . Si se invoca *topsort\_from* sobre  $u$ , recibirá un valor  $f[u]$  menor que  $f[v]$ , pues la visita a  $v$  finaliza antes que la visita a  $u$ . El único problema es, pues, que se llame a *topsort\_from* sobre  $v$  antes de que se produzca la llamada a la misma función sobre  $u$ . En tal caso,  $v$  también habrá completado su visita antes de que se efectúe la visita a  $u$  y tendrá, por tanto, un valor del «instante de finalización» mayor.  $\square$

La complejidad temporal de la función *topsort* es  $O(|V| + |E|)$ , pues no es más que una exploración por primero en profundidad del grafo completo ejecutando una acción (asignar un valor numérico) de coste temporal  $O(1)$  sobre cada vértice. La complejidad espacial es  $O(|V|)$ , pues el número de llamadas recursivas activas simultáneamente puede llegar a ser  $|V|$  en grafos con un camino que visita todos los vértices.

Nótese que *topsort* sólo funciona correctamente si el grafo es acíclico. Puede resultar necesario, pues, efectuar un preproceso que nos indique si el grafo presenta ciclos. Este preproceso puede basarse en un recorrido en profundidad de los vértices.

#### .....PROBLEMAS.....

► **155** Haz una traza del algoritmo de ordenación topológica sobre este grafo:



A continuación, dibuja el grafo con los vértices dispuestos a lo largo de un eje horizontal respetando el orden topológico.

► **156** ¿Cuántas ordenaciones topológicas diferentes admite un grafo con  $n$  vértices y ninguna arista?

► **157** ¿Qué ocurre si suministramos un grafo no conexo al algoritmo de ordenación topológica? Si el algoritmo desarrollado no visita todos los vértices, modifícalo para que lo haga.

► **158** Al planificar un proyecto hay tareas que dependen de la terminación de otras. Nos suministran un fichero en el que cada línea contiene una tarea (descrita por caracteres y guiones) y una secuencia de tareas que deben ejecutarse antes. Por ejemplo:

ejemplo\_tareas.py

```
1 alquilar-perforadora_estudio-mercado-perforadoras
```

```

2 obtener-licencia-empresa┐constituir-sociedad┐aportar-capital
3 permiso-de-obras┐obtener-licencia-empresa┐proyecto-aprobado
4 aportar-capital┐ahorrar
5 perforar┐permiso-de-obras┐alquilar-perforadora┐contratar-perforadores
6 explotar-pozo┐contratar-plantilla-explotacion┐perforar

```

La tarea `perforar` requiere haber completado las tareas `obtener-licencia-empresa` y `proyecto-aprobado`.

Diseña un programa que lea el fichero de dependencias y muestre por pantalla un orden de ejecución de tareas que permita que toda tarea se ejecute sólo cuando se han ejecutado aquellas de las que depende.

► **159** Demuestra que todo grafo acíclico tiene un sumidero (vértice con grado de salida nulo) y una fuente (vértice con grado de entrada nulo).

► **160** Un algoritmo alternativo de ordenación topológica consiste en lo siguiente: en un bucle, se busca un vértice con grado de entrada nulo, se extrae y se ubica en la siguiente posición del vector de vértices ordenados; el vértice y todas las aristas que parten de él se eliminan de  $G$  antes de efectuar una nueva iteración del bucle. El bucle finaliza cuando el grafo queda sin vértice alguno.

Implementa el algoritmo y analiza su complejidad computacional.

► **161** Deseamos implementar una versión simplificada del programa `make` a la que llamaremos `minimake`. El programa `minimake` lee un fichero en el que cada línea tiene varios campos separados por el carácter «:»: el nombre de un fichero  $f_0$ , el nombre de uno o más ficheros  $f_1, f_2, \dots$  y un comando que permite generar el fichero  $f_0$  a partir de  $f_1, f_2, \dots$ .

Por ejemplo, una línea como ésta

```
prog.exe:prog.c:prog.h:libreria.o:gcc prog.c libreria.o -o prog.exe
```

da una receta para generar un fichero `prog.exe` que depende de otros tres: `prog.c`, `prog.h` (éste se incluye en `programa.c` a través de una directiva `#include`) y `libreria.o`: compilar el fichero `prog.c` con el compilador `gcc` y la opción `-o prog.exe` y enlazar el fichero resultante con `libreria.o`. Nótese que `libreria.o` puede tener sus propias dependencias y acción que permite generarlo.

Las reglas no se aplican siempre: sólo se debe ejecutar la regla asociada a un fichero  $f_0$  si no existe o si la fecha de modificación de cualquiera de los ficheros de los que depende es posterior a la de  $f_0$ .

Se pide una implementación de `minimake`. El programa leerá una especificación que sigue el formato descrito y construirá un grafo que represente todas las dependencias. A continuación, comprobará si se trata de un grafo acíclico. En caso de que no lo sea, avisará al usuario del problema y se detendrá. Si se superó la comprobación, el programa obtendrá la fecha de modificación de todos los ficheros y decidirá cuáles debe generar. Finalmente, ejecutará una sola vez cada una de las órdenes que resulta estrictamente necesario ejecutar.

► **162** Un árbol dirigido es un grafo acíclico. Si numeramos sus vértices con el orden con el que son procesados siguiendo estrategias de exploración

- 1) en preorden,
- 2) en postorden,
- 3) por primero en anchura,

obtenemos

- a) un orden topológico,
- b) un orden topológico inverso,
- c) ninguno de los anteriores.

### 9.3. El problema de las componentes conexas

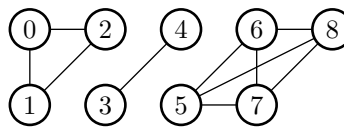
Un problema de interés y con muchas aplicaciones es la determinación de las componentes conexas de un grafo no dirigido. La estructura de datos MFSet, que ya hemos estudiado, ayuda a resolver eficientemente este problema. Nos remitimos a la sección 4.9.3 para un estudio detallado de ésta. Nos limitamos a mostrar aquí su aplicación a grafos:

```

connected_components.py  connected_components.py
1 from mfset import MFset
2 from sets import Set
3
4 def connected_components(G):
5     sets = MFset()
6     for u in G.V:
7         sets.new(u)
8     for u in G.V:
9         for v in G.succs(u):
10            sets.merge(u, v)
11
12     conncomps = {}
13     for u in G.V:
14         v = sets.find(u)
15         if v in conncomps:
16             conncomps[v].add(u)
17         else:
18             conncomps[v] = Set([u])
19     return conncomps.values()

```

Probemos el algoritmo con el grafo de la figura 9.12.



**Figura 9.12:** Un grafo no dirigido con varias componentes conexas.

```

test_connected_components.py  test_connected_components.py
1 from graph import Graph
2 from connected_components import connected_components
3
4 G = Graph(V=range(9), E=[(0,1), (0,2), (1,2), (3,4), (5,6),
5                          (5,7), (5,8), (6,7), (6,8), (7,8)], directed=False)
6
7 print connected_components(G)

```

```
[Set([0, 1, 2]), Set([3, 4]), Set([8, 5, 6, 7])]
```

Ciertos algoritmos sólo funcionan correctamente si el grafo es conexo, así que el test de conectividad puede constituir un paso previo necesario.

- .....PROBLEMAS.....
- **163** ¿Qué complejidad computacional presenta el algoritmo *connected\_components*? (Exprésalo en función de  $|V|$  y/o  $|E|$ .)
  - **164** Diseña un programa que determine si un grafo no dirigido es o no conexo.
  - **165** Diseña una función que reciba un laberinto como los descritos en el ejercicio 151 y nos diga si la entrada y la salida están conectadas. Utiliza al algoritmo de cálculo de las componente conexas.
- .....

## 9.4. Alcanzabilidad y clausura transitiva de un grafo: el algoritmo de Warshall

En ciertos problemas es necesario conocer qué vértices son **alcanzables** desde un vértice dado, es decir, están conectados por un camino. La **clausura transitiva** de un grafo dirigido es un nuevo grafo dirigido en el que hay una arista de  $u$  a  $v$  si y solo si  $v$  es alcanzable desde  $u$  en el grafo original.

¿De dónde viene el término clausura transitiva? Recordemos que un grafo dirigido  $G = (V, E)$  describe una relación binaria transitiva  $\prec$  que sobre su conjunto de vértices. Diseñemos a partir de él una nueva relación transitiva. Para toda arista del grafo  $(u, v) \in E$  estableceremos la relación  $u \prec v$ . La transitividad de la relación hace que  $u \prec v$  y  $v \prec w$  implique  $u \prec w$ . Si existe un camino de  $u$  a  $w$  en el grafo  $G$ , digamos  $(v_1, v_2, \dots, v_n)$  donde  $v_1 = u$  y  $v_n = w$ , la relación transitiva entre vértices adyacentes en el camino,  $v_i \prec v_{i+1}$ , hace que  $u \prec w$ . La relación binaria  $\prec$  define un nuevo grafo con el mismo conjunto de vértices que se obtiene mediante un cierre o clausura transitiva de la relación del grafo original.

Conviene pensar en términos de la representación de los grafos dirigidos mediante matrices de adyacencia, pues la clausura transitiva puede obtenerse como resultado de un producto matricial reiterado en el que se sustituyen sumas por «o-lógicas» y productos por «y-lógicas». Esta función, por ejemplo, recibe una matriz booleana cuadrada  $M$  (una lista de listas de booleanos) y devuelve el valor de  $M^2$ :

**Alcanzabilidad:** *Reachability.*

**Alcanzable:** *Reachable.*

**Clausura transitiva:**  
*Transitive closure.*

```
boolean_matrix_square.py
boolean_matrix_square.py
1 def boolean_matrix_square(M):
2     R = []
3     for i in range(len(M)):
4         R.append([False] * len(M))
5
6     for i in range(len(M)):
7         for j in range(len(M)):
8             for k in range(len(M)):
9                 R[i][j] = R[i][j] or M[i][k] and M[k][j]
10    return R
```

Si suministramos la matriz de adyacencia  $E$  que representa un grafo dirigido, la función devolverá la matriz de un nuevo grafo en el que dos aristas  $u$  y  $w$  estarán conectadas si lo estaban en el grafo original o si  $u$  estaba conectada a otra arista  $v$  que, a su vez, estaba conectada a  $w$ . Podemos calcular  $E^2$  para obtener, mediante una nueva llamada a la función,  $E^4$  y usar esta nueva matriz para calcular  $E^8$ ... La matriz de adyacencia que corresponde a la clausura transitiva es  $A^n$  para  $n \geq |V|$ , pues ningún camino en el grafo puede tener más de  $|V| - 1$  vértices sin repetir ninguno. Podemos calcular  $E^n$  para  $n \geq |V|$  en  $\lceil \lg |V| \rceil$  productos matriciales como el descrito, cada uno de los cuales se ejecuta en tiempo  $\Theta(|V|^3)$ . El coste temporal de esta aproximación es, pues,  $O(|V|^3 \lg |V|)$ .

Hay un modo más eficiente de efectuar el mismo cálculo:

```
boolean_matrix_transitive_closure.py
boolean_matrix_transitive_closure.py
1 def boolean_matrix_transitive_closure(M):
2     R = []
3     for i in range(len(M)):
4         R.append([False] * len(M))
5
6     for k in range(len(M)):
7         for i in range(len(M)):
8             for j in range(len(M)):
9                 R[i][j] = R[i][j] or M[i][k] and M[k][j]
10    return R
```

Lo único que diferencia a esta función de la anterior es el orden de los bucles.

**Teorema 9.2** La función `boolean_matrix_transitive_closure` calcula  $M^n$  para  $n$  mayor o igual al número de filas de la matriz booleana cuadrada  $M$ , es decir, su clausura transitiva.

*Demostración.* Podemos demostrarlo por inducción sobre  $k$ , el índice del bucle exterior e interpretando la matriz en términos de la matriz de adyacencia para un grafo cuyos vértices forman un rango de enteros consecutivos y empezado en cero. La hipótesis de inducción es «la iteración  $k$ -ésima pone a *True* la celda  $M[i][j]$  si y sólo si hay un camino de  $i$  a  $j$  que no incluye vértices mayores que  $k$ ».

**Base de inducción** Tras la primera iteración  $M[i][j]$  vale *True* si valía *True* originalmente o si tanto  $M[i][0]$  como  $M[0][j]$  valían *True*.

**Paso de inducción** Si es cierto para la iteración  $k$ , ¿lo es también para la  $k+1$ ? Tras la iteración  $k+1$ , la celda  $M[i][j]$  vale *True* si valía *True* tras la iteración  $k$  o si  $M[i][k+1]$  y  $M[k+1][j]$  valían *True*. En el primero de los casos, sigue siendo cierta la hipótesis de inducción. En el segundo, hemos encontrado un camino del vértice  $i$  a  $k+1$  y un camino de  $k+1$  a  $j$  que, por hipótesis de inducción, sólo atraviesan vértices de valor inferior o igual a  $k$ . Este nuevo camino no atraviesa vértices de valor superior a  $k+1$ , como queríamos demostrar.  $\square$

He aquí una versión del algoritmo formulada en términos de grafos que devuelve una matriz de adyacencia indexada por vértices a partir de la cual es inmediato construir el grafo inducido por la clausura transitiva de  $G$ :

```
warshall.py
1 def warshall(G):
2     D = {}
3     for u in G.V:
4         for v in G.V:
5             D[u,v] = u == v or v in G.succs(u)
6
7     for v in G.V:
8         for u in G.V:
9             for w in G.V:
10                D[u,w] = D[u,w] or (D[u,v] and D[v,w])
11
12     return D
```

El coste temporal del algoritmo es, evidentemente,  $\Theta(|V|^3)$ .

En la práctica, es posible reducir el tiempo de ejecución si observamos que sólo tiene sentido ejecutar el bucle interior cuando  $D[u,v]$  vale *True*, pues en caso contrario deja intacta la matriz de adyacencia.

```
warshall.py
1 def warshall(G):
2     D = {}
3     for u in G.V:
4         for v in G.V:
5             D[u,v] = u == v or v in G.succs(u)
6
7     for v in G.V:
8         for u in G.V:
9             if D[u,v]:
10                for w in G.V:
11                    D[u,w] = D[u,w] or (D[u,v] and D[v,w])
12
13     return D
```

El coste de esta nueva versión es  $O(|V|^3)$ . El algoritmo de Warshall es el procedimiento elegido cuando se desea calcular la clausura transitiva en grafos densos.

#### PROBLEMAS

► **166** Utiliza el algoritmo de Warshall para detectar las componentes conexas de un grafo no dirigido.

► **167** Es posible calcular la clausura transitiva con una aproximación completamente diferente. Podemos determinar los vértices alcanzables desde un vértice dado con una exploración por primero en profundidad: los vértices visitados al iniciar la exploración en un vértice  $v$  son alcanzables desde  $v$ . ¿Con qué coste espacial y temporal podemos calcular la clausura transitiva si ejecutamos tantas exploraciones por primero en profundidad como resulte necesario?

► **168** ¿Cuál es la clausura transitiva de estos grafos?

a) Un grafo en el que un vértice está unido por sendas aristas a todos los demás vértices.

b) Un grafo en el que las aristas forman un ciclo euleriano.

c) Un grafo completo.

► **169** Es posible calcular la clausura positiva de un grafo dirigido acíclico más eficientemente a partir de su ordenación topológica. ¿Cómo? ¿Con qué coste?

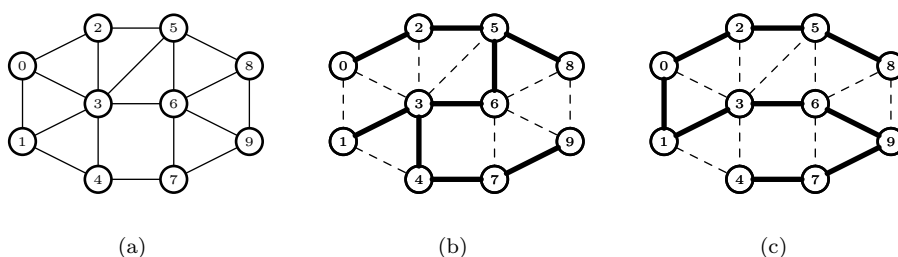
► **170** La operación «inversa» recibe el nombre de **reducción transitiva** (*transitive reduction*) y es un grafo tal que  $u$  y  $v$  están unidos por un camino si y sólo si  $(u, v)$  es una arista del grafo original. Tiene interés calcular la reducción transitiva de menor número de aristas (también conocida por «digrafo equivalente mínimo»). ¿Cómo puede calcularse?

## 9.5. El problema del árbol de recubrimiento de coste mínimo

Un **árbol de recubrimiento** en un grafo no dirigido  $G = (V, E)$  es un conjunto de aristas  $T \subseteq E$  que inducen un grafo con estructura de árbol.

**Árbol de recubrimiento:**  
*Spanning tree.*

La figura 9.13 (a) muestra un grafo no dirigido y las figuras 9.13 (b) y 9.13 (c), sendos árboles de recubrimiento en trazo grueso.



**Figura 9.13:** (a) Grafo no dirigido. (b) Árbol de recubrimiento sobre el grafo (aristas en trazo grueso). (c) Otro árbol de recubrimiento.

Nótese que es posible transitar de cualquier vértice a cualquier otro visitando únicamente aristas del árbol de recubrimiento.

Un **árbol de recubrimiento mínimo** o MST de un grafo no dirigido y ponderado  $G = (V, E, d)$  es aquel árbol de recubrimiento  $T \subseteq E$  cuya suma de pesos

**Árbol de recubrimiento mínimo:** *Minimum spanning tree (MST).*

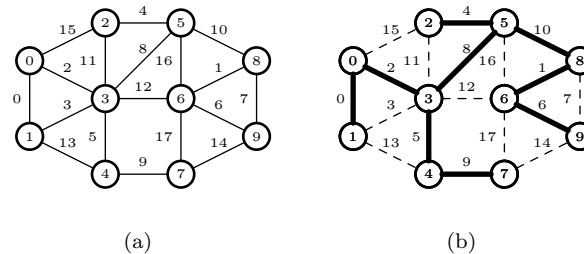
$$D(T) = \sum_{(u,v) \in T} d(u,v)$$

es menor o igual que la de cualquier otro árbol de recubrimiento.

Nótese, pues, que sería más apropiado referirnos al árbol de recubrimiento *con coste mínimo*, pero en la literatura se le conoce como «árbol de recubrimiento mínimo».

Encontrar el MST presenta interés en el diseño de redes de transporte o comunicaciones: permite interconectar todos los vértices de un grafo con el menor coste posible (menor número de kilómetros asfaltados en el caso de carreteras o menor número de kilómetros de cable en el caso de redes para la transmisión de datos o voz).

La figura 9.14 (a) muestra un grafo no dirigido y ponderado. La figura 9.14 (b) muestra un MST en trazo grueso. El coste del MST es de 45.



**Figura 9.14:** (a) Grafo no dirigido y ponderado. (b) Árbol de recubrimiento mínimo (MST) sobre el grafo (aristas en trazo continuo).

### PROBLEMAS

- **171** Dado un grafo  $G$  no dirigido y conexo, modifica el algoritmo de recorrido por primero en profundidad para que devuelva las aristas que formarían parte de un árbol de recubrimiento para el grafo  $G$ .
- **172** Dado un grafo no dirigido, ponderado y conexo, ¿devuelve un MST el algoritmo que has diseñado como solución al ejercicio 171? Razona tu respuesta.

## 9.5.1. El algoritmo de Kruskal

Estudiaremos ahora un algoritmo para el cálculo de esta estructura: el **algoritmo de Kruskal**. Este algoritmo inicializa el conjunto de aristas que forman parte del árbol de recubrimiento mínimo,  $T$ , al conjunto vacío. A continuación, considera todas y cada una de las aristas del grafo en orden de menor a mayor peso. Si la arista no forma un ciclo en el grafo  $G' = (V, T)$ , se añade a  $T$ ; en caso contrario, se descarta. Este procedimiento va formando un bosque (un conjunto de árboles) inicializado con un árbol por vértice. Los árboles se unen cuando se considera una arista  $(u, v)$  tal que  $u$  está en un árbol diferente del árbol en el que se encuentra  $v$ . La forma de detectar si una arista  $(u, v)$  forma ciclo o no es sencilla: si  $u$  y  $v$  forman parte del mismo árbol, la arista  $(u, v)$  forma un ciclo. Los MFset son una estructura idónea para efectuar las operaciones sobre conjuntos de vértices que nos permiten detectar la formación de bucles: determinar a qué conjunto pertenecen  $|V|$  vértices y unir  $|V| - 1$  conjuntos de vértices son operaciones con un coste global que consideramos  $O(|V|)$ .

La figura 9.15 muestra una traza del algoritmo descrito sobre un grafo.

Con lo dicho resulta fácil ofrecer una implementación en Python si recurrimos a los MFsets que ya implementamos en un módulo.

```

kruskal.py
1 from mfset import MFset
2
3 def Kruskal(G, d):
4     # Empezamos por construir una lista de aristas con sus respectivos pesos.
5     E = [ (d(u,v), (u,v)) for u in G.V for v in G.succs(u) ]
6     # Y ordenamos la lista por valor creciente del peso.
7     E.sort()
8
9     # Inicializamos el conjunto de aristas, que podemos representar con una lista.
10    T = [None] * (len(G.V)-1)
11

```

El algoritmo de Kruskal se dio a conocer en el artículo «On the shortest spanning subtree of a graph and the travelling salesman problem», *American Mathematical Society* 7, pp. 48–50, 1956.

Estudiaremos otro algoritmo para calcular el MST cuando estudiemos la estrategia algorítmica voraz: el algoritmo de Prim. Como veremos entonces, Kruskal también es un algoritmo que podemos calificar de voraz.



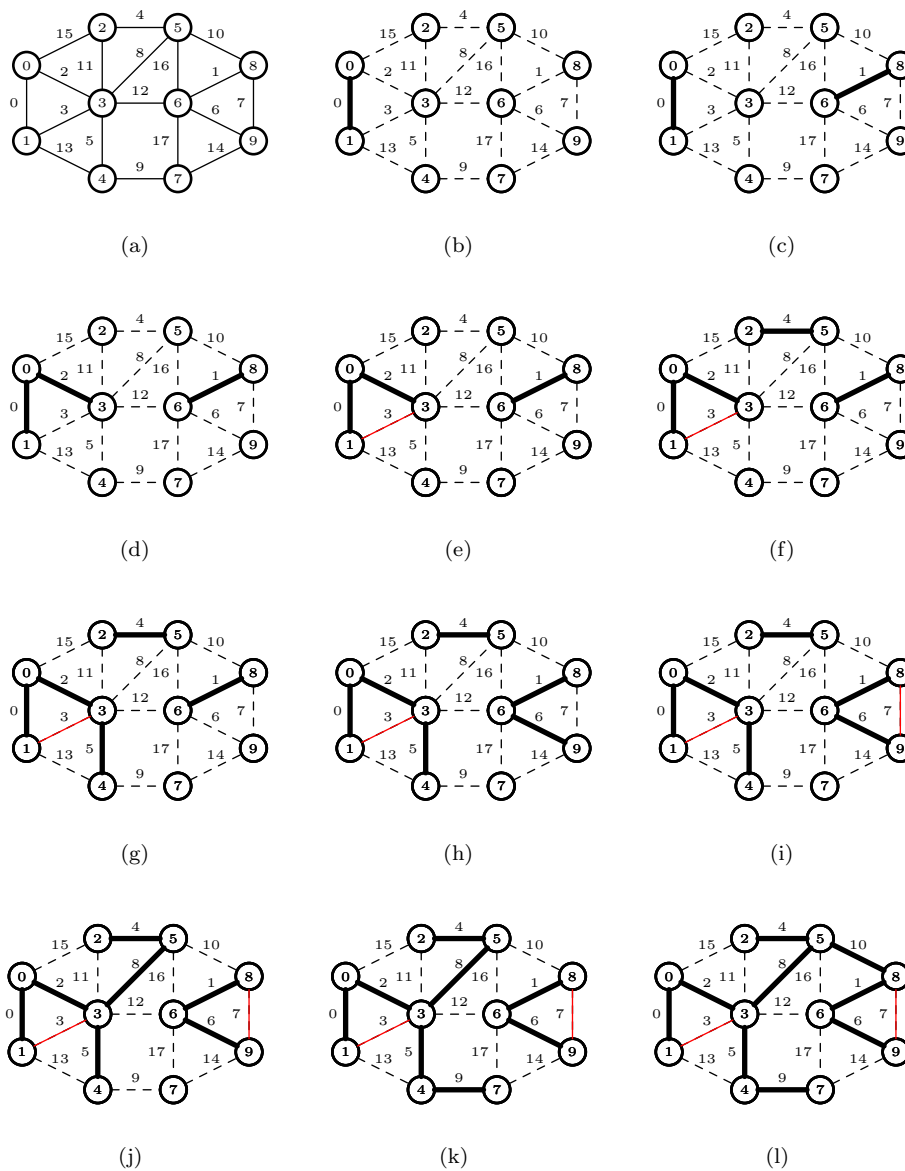



Figura 9.15: Trazo del algoritmo de Kruskal.

```

12 forest = MFset()
13 for u in G.V: forest.new(u)
14
15 i = 0
16 for (weight, (u,v)) in E: # Recorremos las aristas.
17     if forest.find(u) != forest.find(v): # si u y v están en conjuntos diferentes...
18         T[i] = (u,v)
19         i += 1
20     forest.merge(u, v) # ... unimos sus respectivos conjuntos en uno solo.
21     if i == len(G.V)-1: break
22 return T

```

 test\_kruskal.py

test\_kruskal.py

```

1 from graph import Graph
2 from kruskal import Kruskal
3
4 G = Graph(V=range(10), E=[(0,1), (0,2), (0,3), (1,3), (1,4), (2,3), (2,5),
5                             (3,4), (3,5), (3,6), (4,7), (5,6), (5,8), (6,7),
6                             (6,8), (6,9), (7,9), (8,9)], directed=False)

```

```

7
8  dist = {(0,1): 0, (0,2): 15, (0,3): 2, (1,3): 3, (1,4): 13, (2,3): 11, (2,5): 4,
9        (3,4): 5, (3,5): 8, (3,6): 12, (4,7): 9, (5,6): 16, (5,8): 10, (6,7): 17,
10       (6,8): 1, (6,9): 6, (7,9): 14, (8,9): 7}
11 def d(u,v):
12     if (u,v) in dist: return dist[u,v]
13     return dist[v,u]
14
15 E = Kruskal(G, d)
16 print 'MST:', E
17 print 'Peso del MST:', sum([d(u,v) for (u,v) in E])

```

```

MST: [(0, 1), (6, 8), (0, 3), (2, 5), (3, 4), (6, 9), (3, 5), (4, 7), (5, 8)]
Peso del MST: 45

```

La figura 9.16 muestra gráficamente el resultado de aplicar el algoritmo de Kruskal al mapa de la península ibérica.



**Figura 9.16:** Árbol de recubrimiento mínimo para las ciudades de la península ibérica.

### Corrección del algoritmo

Es evidente que el algoritmo acaba encontrando un árbol de recubrimiento si el grafo es conexo, pues considera todas las aristas y va uniendo los árboles del bosque (que son disjuntos) dos a dos hasta formar un sólo árbol. Que el árbol encontrado sea de coste mínimo no resulta evidente. Supondremos de momento que sólo hay un MST.

**Lema 9.1** Sea  $G = (V, E)$  un grafo no dirigido y ponderado, sea  $X$  un subconjunto de  $V$  y sea  $e = (u, v)$  la arista de menor peso que conecta  $X$  con  $V - X$ . La arista  $e$  es parte del MST.

*Demostración.* Supongamos que  $T$  es un árbol de recubrimiento que no contiene a  $e$ . Vamos a demostrar que  $T$  no es el MST. En consecuencia,  $e$  deberá formar parte del

MST. La arista  $e$  es un par  $(u, v)$  donde  $u \in X$  y  $v \in V - X$ . Como  $T$  es un árbol de recubrimiento, ya conectaba  $X$  con  $V - X$ . Sea  $e' = (u', v')$  la arista que efectuaba esta conexión. Si añadimos a  $T$  la arista  $e$ , se producirá un ciclo. El árbol  $T' = (T \cup e) - e'$  elimina el ciclo, conecta a todos los vértices y es, por tanto, un árbol de recubrimiento. El peso  $D(T')$  es menor que el de  $D(T)$ , pues  $d(u, v) < d(u', v')$ . Así pues,  $T$  no es el MST.  $\square$

Si hubiésemos admitido que pudiera haber más de un MST, hubiésemos tenido que considerar la posibilidad de que  $D(T') = D(T)$ , lo que implicaría  $d(u, v) = d(u', v')$ . En tal caso, tanto  $T$  como  $T'$  serían MST y hubiera dado igual escoger  $e$  o  $e'$ , pues ambos forman parte de un MST.

**Corolario 9.1** *El algoritmo de Kruskal calcula el MST.*

*Demostración.* Con cada iteración, el algoritmo escoge la arista  $e$  que conecta dos regiones disjuntas del grafo y lo hace seleccionando la arista de menor peso de cuantas no ha considerado todavía.  $\square$

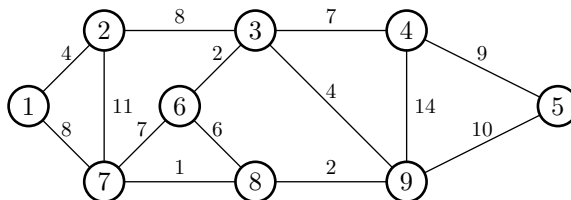
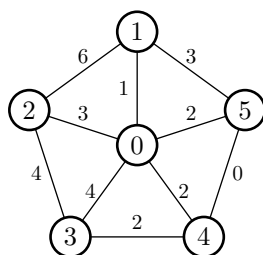
### Análisis de complejidad

La complejidad temporal es  $O(|V| + |E| \lg |E|)$ . Ordenar el vector de aristas por orden creciente de peso es  $O(|E| \lg |E|)$ . El algoritmo efectúa un máximo de  $|E|$  iteraciones y un máximo de  $|E|$  consultas de pertenencia y de  $|V|$  fusiones de conjuntos sobre el MFSet. El coste temporal de estas operaciones se considera globalmente  $O(|E|)$ .

El algoritmo de Prim, que estudiaremos más adelante, resuelve este problema en tiempo  $O(|V|^2)$ , así que es preferible al de Kruskal si el grafo es denso.

### PROBLEMAS

► **173** Realiza una traza del algoritmo de Kruskal para los grafos de las siguientes figuras:

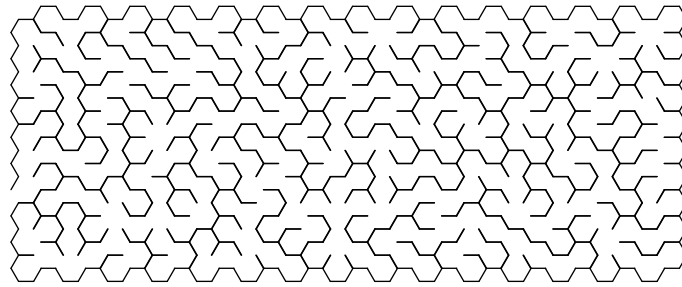


► **174** Utilizando el algoritmo de Kruskal, ¿cómo se podría determinar si un grafo es conexo? ¿cómo se podrían obtener los vértices que forman cada una de las componentes conexas del grafo?

► **175** Supongamos que hemos obtenido un MST para un grafo  $G$  no dirigido, conexo y ponderado. Si a continuación se añadiera un nuevo vértice y la arista o aristas que lo conectan al resto de los vértices, ¿de qué forma se podría actualizar rápidamente el MST, sin necesidad de ejecutar nuevamente el algoritmo?

► **176** El algoritmo de Kruskal puede utilizarse para generar laberintos como los descritos en el ejercicio 151. El procedimiento a seguir es éste: Dispón muros sólo en el perímetro exterior del laberinto. Pondera cada enlace entre dos celdas vecinas con peso nulo y calcula el MST (todos los árboles de recubrimiento serán MSTs). Recorre los arcos en cualquier orden (todos pesan lo mismo) y obtén una arborescencia de recubrimiento. Pon muros de separación entre todo par de vértices contiguos que no estén unidos en el MST. Finalmente, elimina dos muros exteriores para crear la entrada y la salida del laberinto. Implementa este método de generación y representa gráficamente el laberinto resultante.

► **177** El laberinto puede construirse sobre una malla hexagonal. He aquí un ejemplo:



Diseña un algoritmo que construya laberintos hexagonales. (Pista: El «soporte» del laberinto sigue siendo una matriz convencional. La principal diferencia estriba en las relaciones de vecindad entre casillas.)

► **178** Deseamos montar un sistema de comunicaciones por cable que comunique entre sí a todas las ciudades de la península ibérica y queremos minimizar la cantidad de cable instalado.

- Diseña un programa que indique qué pares de ciudades hemos de conectar entre sí suponiendo que es posible comunicar dos ciudades con una cantidad de cable igual a la distancia que las separa en línea recta, haya o no carretera que las una.
- Diseña un programa que indique qué pares de ciudades hemos de conectar entre sí suponiendo que es posible comunicar dos ciudades con una cantidad de cable igual a la distancia que las separa en línea recta, pero sólo si hay carretera entre ellas.

En ambos casos, representa gráficamente el resultado.

► **179** El algoritmo de Kruskal es muy costoso sobre grafos euclídeos en el plano en los que cada vértice está conectado con todos los demás. Existe un preproceso, llamado triangulación de Delauney, que permite reducir sensiblemente el coste del algoritmo. Averigua qué es la triangulación de Delauney y por qué puede mejorar la eficiencia asintótica del algoritmo de Kruskal.

## 9.6. El problema del camino más corto

El **problema del camino más corto entre dos vértices** de un grafo se enuncia así:

«Dado un grafo ponderado  $G = (V, E, d)$  y dos vértices  $s$  y  $t$ , a los que denominamos vértice inicial o fuente y vértice final u objetivo, respectivamente, encuéntrase un camino que parta de  $s$  y finalice en  $t$  tal que ningún otro presenta menor distancia.»

Formalmente, buscamos un camino  $(v_1, v_2, \dots, v_n)$  tal que  $v_1 = s$  y  $v_n = t$  con distancia mínima.

$$(\hat{v}_1, \hat{v}_2, \dots, \hat{v}_n) = \arg \min_{(v_1, v_2, \dots, v_n) \in P(s, t)} D(v_1, v_2, \dots, v_n).$$

Este otro problema, denominado **problema de la distancia mínima entre dos vértices** está directamente relacionado con el anterior:

«Dado un grafo ponderado  $G = (V, E, d)$  y dos vértices  $s$  y  $t$ , encuéntrase la distancia del camino más corto entre  $s$  y  $t$ .»

Formalmente, deseamos calcular el valor

$$\hat{D} = \min_{(v_1, v_2, \dots, v_n) \in P(s, t)} D(v_1, v_2, \dots, v_n).$$

Obsérvese que la solución de este segundo problema no es un camino, sino su distancia.

**Vértice inicial:** *Source vertex.*

No debe confundirse el término «source vertex» en este contexto y su significado general de «vértice sin predecesores».

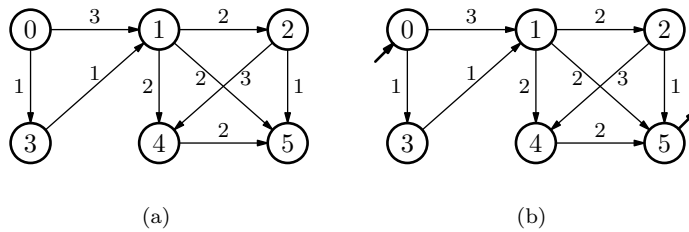
**Vértice final:** *Target vertex.*

En los desarrollos que presentamos asumiremos que la distancia de un camino es la suma de los pesos de sus aristas:

$$D(v_1, v_2, \dots, v_n) = \sum_{1 \leq i < n} d(v_i, v_{i+1}).$$

Muchos de los problemas que estudiaremos se formulan finalmente como una instancia del problema del camino más corto, del problema de la distancia mínima o una variante de estos. Será frecuente en temas posteriores, pues, plantear la búsqueda de un camino óptimo entre dos vértices.

En las representaciones gráficas de los problemas de búsqueda del camino más corto en un grafo, destacaremos los vértices inicial y final con, respectivamente, una flecha de entrada y una de salida. Si deseamos calcular el camino más corto entre los vértices 0 y 5 del grafo de la figura 9.17 (a), representaremos el grafo como se muestra en la figura 9.17 (b).



**Figura 9.17:** (a) Grafo dirigido y ponderado. (b) El mismo grafo, pero con los vértices inicial ( $s = 0$ ) y final ( $t = 5$ ) destacados.

No hay problema en considerar más de un vértice final. El camino más corto entre  $s$  y uno cualquiera de dos vértices finales,  $t_1$  y  $t_2$ , por ejemplo, es el camino más corto de entre el camino más corto de  $s$  a  $t_1$  y el camino más corto de  $s$  a  $t_2$ .

Vamos a proponer algoritmos diferentes en función de ciertas propiedades que deben cumplir los grafos. La única restricción que imponemos a todos los grafos es natural: el grafo no podrá tener ciclos cuya suma de pesos sea negativa. Si los pesos de un ciclo son negativos, no hay «camino más corto»: siempre es posible mejorar el peso de un camino añadiendo el ciclo de peso negativo. El problema estaría, pues, mal definido.

### 9.6.1. Un algoritmo basado en la fuerza bruta

Un método directo para resolver el problema consiste en enumerar todos los caminos del grafo entre  $s$  y  $t$  y seleccionar el de menor distancia. Recordemos que el conjunto de los caminos entre  $s$  y  $t$  es  $P(s, t)$ , que definimos como:

$$P(s, t) = \{(v_1, v_2, \dots, v_n) \mid v_1 = s; v_n = t; (v_i, v_{i+1}) \in E, 1 \leq i < n\}.$$

Podemos expresar  $P(s, t)$  en función de  $P(s, u)$  para todo  $u$  predecesor de  $t$ :

$$\begin{aligned} P(s, t) &= \{(v_1, v_2, \dots, v_n) \mid v_1 = s; v_n = t; (v_i, v_{i+1}) \in E, 1 \leq i < n\} \\ &= \{(v_1, v_2, \dots, v_n) \mid v_1 = s; v_n = t; v_{n-1} \in \text{preds}(t); (v_i, v_{i+1}) \in E, 1 \leq i < n-1\} \\ &= \{(v_1, v_2, \dots, v_n) \mid v_1 = s; v_n = t; (v_1, v_2, \dots, v_{n-1}) \in P(s, u); (u, t) \in E\}. \end{aligned}$$

Hemos llegado a una definición recursiva del conjunto de caminos entre dos vértices, pues expresa  $P(s, t)$  en función de  $P(s, u)$ , para todo  $u$  predecesor de  $t$ . Ello conduce a un algoritmo, también recursivo, que genera todos los caminos entre  $s$  y  $t$ :

```
brute_force_shortest_path.py brute_force_shortest_path.py
1 def brute_force_paths(G, s, t):
2     if s == t:
3         return [[s]]
4     paths = []
5     for u in G.preds(t):
```

```

6     for prefix in brute_force_paths(G, s, u):
7         paths.append( prefix + [t] )
8     return paths

```

Y si somos capaces de enumerar el conjunto de caminos, podemos calcular la distancia con la que cada uno enlaza los vértices  $s$  y  $t$  para quedarnos con la menor y obtener así el camino de distancia mínima:

```

brute_force_shortest_path.py  brute_force_shortest_path.py
10 def path_distance(G, d, path):
11     dist = 0
12     for i in range(len(path)-1):
13         dist += d(path[i], path[i+1])
14     return dist
15
16 def brute_force_shortest_path(G, d, s, t):
17     mindist = None
18     for path in brute_force_paths(G, s, t):
19         dist = path_distance(G, d, path)
20         if mindist == None or dist < mindist:
21             mindist = dist
22             shortest_path = path
23     return shortest_path

```

Tomemos por caso el grafo de la figura 9.18 y la búsqueda del camino más corto entre sus vértices (0,0) y (2,2).

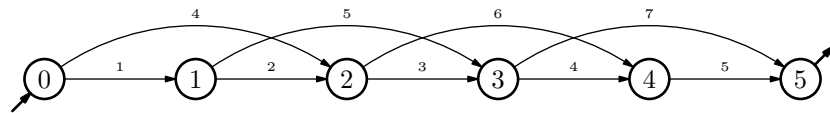


Figura 9.18: Un ejemplo de grafo ponderado. Cada vértice se une a los dos siguientes

```

test_brute_force_shortest_path.py  test_brute_force_shortest_path.py
1 from graph import Graph
2 from brute_force_shortest_path import brute_force_paths, brute_force_shortest_path, \
3     path_distance
4
5 n = 6
6 G = Graph(V=range(n),
7           E=[(u,u+1) for u in range(n-1)]+[(u,u+2) for u in range(n-2)])
8
9 def d(u,v):
10     if v-u == 1: return v
11     return v+2
12
13 all_paths = brute_force_paths(G, 0, n-1)
14 print 'Caminos con sus respectivas distancias:'
15 for path in all_paths:
16     print path, path_distance(G, d, path)
17
18 print 'Camino más corto:', brute_force_shortest_path(G, d, 0, n-1)

```

```

Caminos con sus respectivas distancias:
[0, 1, 3, 5] 13
[0, 2, 3, 5] 14
[0, 1, 2, 3, 5] 13
[0, 2, 4, 5] 15
[0, 1, 2, 4, 5] 14
[0, 1, 3, 4, 5] 15

```

[0, 2, 3, 4, 5] 16  
 [0, 1, 2, 3, 4, 5] 15  
 Camino más corto: [0, 1, 3, 5]

Nuestro algoritmo presenta un problema: sólo funciona si el grafo es acíclico. ¿Por qué? Porque si el grafo contiene ciclos, el número de caminos es infinito. Podríamos diseñar una versión que detectara los ciclos y evitara generar caminos en los que el ciclo se repite una y otra vez ya que no presentan interés si estamos interesados en el camino más corto. Pero hay una razón de peso para no invertir más esfuerzo es esta aproximación: requiere tiempo exponencial con el número de vértices. Incluso con grafos acíclicos con la estructura del presentado en el ejemplo, el número de caminos crece con el número de vértices de modo tal que resulta inviable utilizar la enumeración de todos los caminos para calcular el más corto. En un grafo con  $n$  vértices y la estructura del utilizado en el ejemplo, el número de caminos,  $C(n)$ , se calcula con esta expresión recursiva:

$$C(n) = \begin{cases} 1, & \text{si } n = 1 \text{ o } n = 2; \\ C(n-1) + C(n-2), & \text{si } n > 2. \end{cases}$$

O sea, el número de caminos,  $C(n)$ , es  $n$ -ésimo el número de Fibonacci. Y ya vimos que la secuencia de los números de Fibonacci crecía exponencialmente. Podemos concluir que visitar explícitamente todos los caminos resulta prohibitivo. Afortunadamente podemos evitarlo. Veamos primero cómo hacerlo en el caso de grafos acíclicos.

### 9.6.2. En grafos acíclicos

En lugar de resolver directamente el problema del camino más corto abordaremos primero, por ser más sencillo, el problema del cálculo de la distancia del camino más corto.

#### Cálculo de la distancia del camino más corto

Dado un grafo  $G = (V, E)$  y dos vértices  $s$  y  $t$ , busquemos

$$\hat{D} = \min_{(v_1, v_2, \dots, v_n) \in P(s, t)} D(v_1, v_2, \dots, v_n),$$

donde, recordemos,  $P(s, t)$  puede definirse recursivamente, es decir, en función de  $P(s, u)$  para todo  $u$  predecesor de  $t$ :

$$P(s, t) = \{(v_1, v_2, \dots, v_n) \mid v_1 = s; v_n = t; (v_1, v_2, \dots, v_{n-1}) \in P(s, u); (u, t) \in E\}.$$

Introduzcamos un nuevo elemento de notación. La distancia del camino más corto entre  $s$  y  $t$  se denotará con  $\mathcal{D}(s, t)$ :

$$\begin{aligned} \mathcal{D}(s, t) &= \min_{(v_1, v_2, \dots, v_n) \in P(s, t)} D(v_1, v_2, \dots, v_n) \\ &= \min_{(v_1, v_2, \dots, v_n) \in P(s, t)} \left( \sum_{1 \leq i < n} d(v_i, v_{i+1}) \right). \end{aligned}$$

Podemos sustituir  $P(s, t)$  por su expresión recursiva:

$$\mathcal{D}(s, t) = \min_{(u, t) \in E} \left( \min_{(v_1, v_2, \dots, v_{n-1}) \in P(s, u)} \left( \sum_{1 \leq i < n-1} d(v_i, v_{i+1}) \right) + d(u, t) \right).$$

Y, ahora, extraer de la minimización interior el último término del sumatorio:

$$\mathcal{D}(s, t) = \min_{(u, t) \in E} \left( \left( \min_{(v_1, v_2, \dots, v_{n-1}) \in P(s, u)} \left( \sum_{1 \leq i < n-1} d(v_i, v_{i+1}) \right) \right) + d(u, t) \right).$$

Podemos reescribir el sumatorio interior, que no es más que la distancia de un camino:

$$\mathcal{D}(s, t) = \min_{(u, t) \in E} \left( \left( \min_{(v_1, v_2, \dots, v_{n-1}) \in P(s, u)} D(v_1, v_2, \dots, v_{n-1}) \right) + d(u, t) \right).$$

Es evidente que hemos llegado a una relación recursiva:

$$\mathcal{D}(s, t) = \min_{(u, t) \in E} (\mathcal{D}(s, u) + d(u, t)).$$

No hay nada especial en  $t$ , así que podríamos haber deducido la recursión para un vértice  $v$  cualquiera:

$$\mathcal{D}(s, v) = \min_{(u, v) \in E} (\mathcal{D}(s, u) + d(u, v)).$$

Hay un caso que podemos considerar base de la recursión: cuando  $v = s$  se propone la búsqueda de la distancia óptima de  $s$  a  $s$ . Dicha distancia es, por definición, nula:

$$\mathcal{D}(s, s) = 0.$$

Otro caso especial es el de los vértices sin predecesores. La minimización se propone entonces sobre un conjunto vacío de elementos. Definimos  $\min(\emptyset)$  como  $+\infty$ .

$$\mathcal{D}(s, v) = \begin{cases} 0 & \text{si } v = s; \\ +\infty & \text{si } \nexists (u, v) \in E; \\ \min_{(u, v) \in E} (\mathcal{D}(s, u) + d(u, v)) & \text{en otro caso.} \end{cases} \quad (9.1)$$

El valor que deseamos calcular es

$$\hat{D} = \mathcal{D}(s, t).$$

#### PROBLEMAS

► **180** Hemos dicho antes que podemos ponderar un camino mediante el producto de los pesos de sus aristas, y no sólo mediante la suma. ¿Es posible efectuar una derivación como la que acabamos de presentar si sustituimos sumatorios por productorios?

### Un algoritmo recursivo

Podemos «traducir» la ecuación recursiva a un programa Python y obtener así la implementación de un algoritmo recursivo.

```
shortest_path.py shortest_path.py
1 def recursive_dag_shortest_distance(G, d, s, v, infinity= 3.4e+38):
2     if v == s:
3         return 0
4     elif len(G.preds(v)) == 0:
5         return infinity
6     else:
7         return min([recursive_dag_shortest_distance(G, d, s, u, infinity) + d(u, v) \
8                     for u in G.preds(v)])
```

### Un algoritmo iterativo

Si al calcular  $\mathcal{D}(s, v)$  se produce una «llamada» recursiva a  $\mathcal{D}(s, u)$ , es porque  $(u, v)$  es una arista. O sea,  $u$  es un vértice anterior a  $v$  en cualquier orden topológico de los vértices del grafo (recordemos que es un grafo acíclico). Podemos diseñar una versión iterativa basada en un recorrido de los vértices del grafo en orden topológico:

```
shortest_path.py shortest_path.py
10 from topsort import topsort
11
12 def dag_shortest_distance(G, d, s, t, infinity=3.4e+38):
```



```

13  D = {}
14  for v in topsort(G):
15      if v == s:
16          D[s,v] = 0
17      elif len(G.preds(v)) == 0:
18          D[s,v] = infinity
19      else:
20          D[s,v] = min( [ D[s,u] + d(u,v) for u in G.preds(v) ] )
21      if v == t: break
22
23  return D[s, t]

```

Probemos el programa. Diseñemos un programa que calcule el camino más corto entre los vértices 0 y 5 del grafo acíclico y ponderado de la figura 9.19.

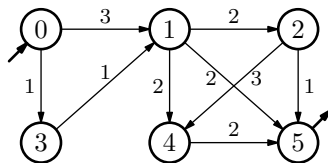


Figura 9.19: Grafo acíclico y ponderado.

```

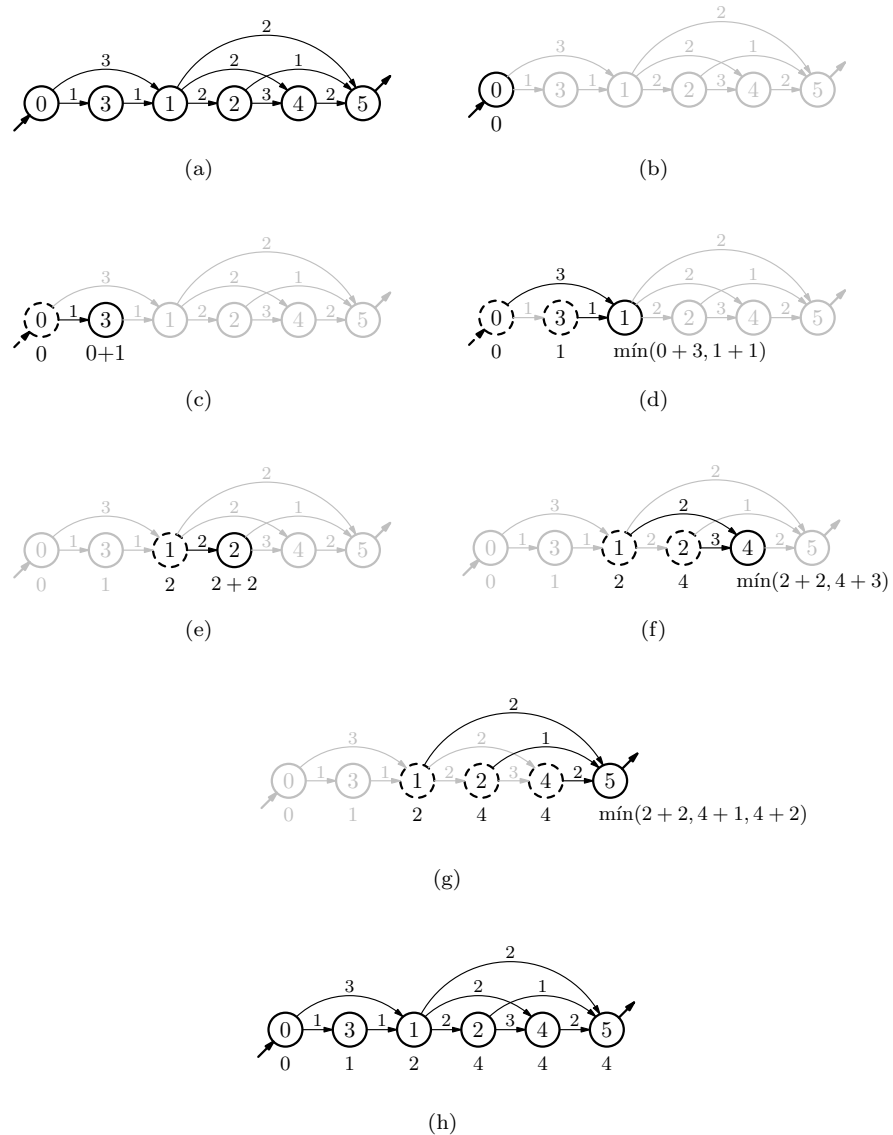
test_shortest_path.a.py  test_shortest_path.a.py
1  from graph import Graph
2  from shortest_path import dag_shortest_distance
3
4  G = Graph(V=range(6), E=[(0,1), (0,3), (1,2), (1,4), (1,5), (2,4), (2,5),
5                          (3,1), (4,5)])
6
7  dist = {(0,1):3, (0,3):1, (1,2):2, (1,4):2, (1,5):2, (2,4): 3,
8          (2,5):1, (3,1):1, (4,5):2}
9
10 def d(u,v):
11     return dist[u,v]
12
13 print 'Distancia entre los vértices 0 y 5:', dag_shortest_distance(G, d, 0, 5)

```

El resultado de ejecutar el programa es éste:

Distancia entre los vértices 0 y 5: 4

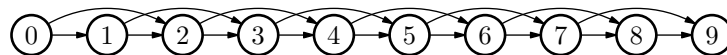
Hagamos una traza paso a paso. La función empieza ordenando topológicamente los vértices (figura 9.20 (a)). Se procede entonces a visitar los vértices en ese orden. Se empieza por visitar el vértice 0. Como es el vértice inicial, el valor de  $D[0,0]$  es 0. Mostramos el valor de cada celda de  $D$  bajo el correspondiente vértice (figura 9.20 (b)). Pasamos ahora al vértice 3. Sólo hay un predecesor (el vértice 0). El valor de  $D[0,0]$  es 0 y el peso de la arista  $(0,1)$  es 1. El valor de  $D[0,3]$  es el mínimo de un conjunto formado por un solo elemento de valor 1, o sea, es 1 (figura 9.20 (c)). Pasamos al vértice 1. Se puede llegar a este vértice desde dos vértices: el vértice 0 y el vértice 3. Si venimos del primero, hemos de considerar el valor  $D[0,0]$  sumado al valor  $d(0,1)$ . Si venimos del segundo, hemos de considerar el valor  $D[0,3]$  sumado al valor  $d(3,1)$ . La primera suma da 3 y la segunda, 2. El menor de ambos valores es 2, así que lo asignamos a  $D[0,1]$  (figura 9.20 (d)). Pasamos al vértice 2. Sólo se puede llegar desde el vértice 1. Asignamos a  $D[0,2]$  el valor que resulta de sumar  $D[0,1]$  a  $d(1,2)$  (figura 9.20 (e)). Pasamos al vértice 4. Calculamos el valor de  $D[0,4]$  como el menor entre  $D[0,1]$  más  $d(1,4)$  y  $D[0,2]$  más  $d(2,4)$  (figura 9.20 (f)). Finalmente, calculamos  $D[0,5]$ . En este caso tenemos tres predecesores (figura 9.20 (g)). El resultado se muestra en la figura 9.20 (h).



**Figura 9.20:** Trazo del algoritmo de cálculo del camino más corto en el grafo acíclico ponderado de la figura 9.19.

### PROBLEMAS

► **181** Queremos calcular la distancia del camino más corto en grafos ponderados estructurados como éste:



Nótese que cada vértice tiene únicamente dos (o menos) aristas: una a cada uno de los dos vértices siguientes. El algoritmo presentado requiere espacio  $\Theta(|V|)$  para resolver el problema. ¿Puedes diseñar un algoritmo inspirado en éste que reduzca el espacio necesario a  $\Theta(1)$ ?

### Cálculo del camino más corto: la técnica «seguir la pista hacia atrás»

Hemos aprendido a calcular el peso del camino más corto, pero no sabemos cómo calcular el camino. En realidad es sencillo si aplicamos una técnica que utilizaremos muchas veces de ahora en adelante: la técnica que llamamos «seguir la pista hacia atrás». Consiste en recordar, para cada vértice  $v$ , de qué vértice  $u$  «viene» la solución óptima hasta dicho punto. Recordar significa escribir en un vector indexado por  $v$  el valor  $u$ . Esta primera versión implementa esta idea de un modo un tanto farragoso:

**Seguir la pista hacia atrás:**  
*Backtracing.*

```

1 def dag_shortest_path(G, d, s, t, infinity=3.4e+38):
2     # Devuelve la distancia del camino mínimo y la lista de vértices que lo forman.
3     D = {}
4     backp = {}
5     for v in topsort(G):
6         if v == s:
7             D[s, v] = 0
8             backp[v] = None
9         elif G.preds(v):
10            D[s, v] = infinity
11            for u in G.preds(v):
12                if D[s, u] + d(u, v) < D[s, v]:
13                    D[s, v] = D[s, u] + d(u, v)
14                    backp[v] = u
15            else:
16                D[s, v] = infinity
17                backp[v] = None
18            if v == t: break
19
20    path = [t]
21    while backp[path[-1]] != None:
22        path.append( backp[path[-1]] )
23    path.reverse()
24
25    return D[s, t], path

```

El vector *backp*, cuyo nombre es prefijo del término inglés «backpointer» (por «puntero hacia atrás»), almacena la información necesaria para recuperar el camino. Es posible recuperar el camino si se sigue la sucesión de punteros hacia atrás desde el nodo *t*.

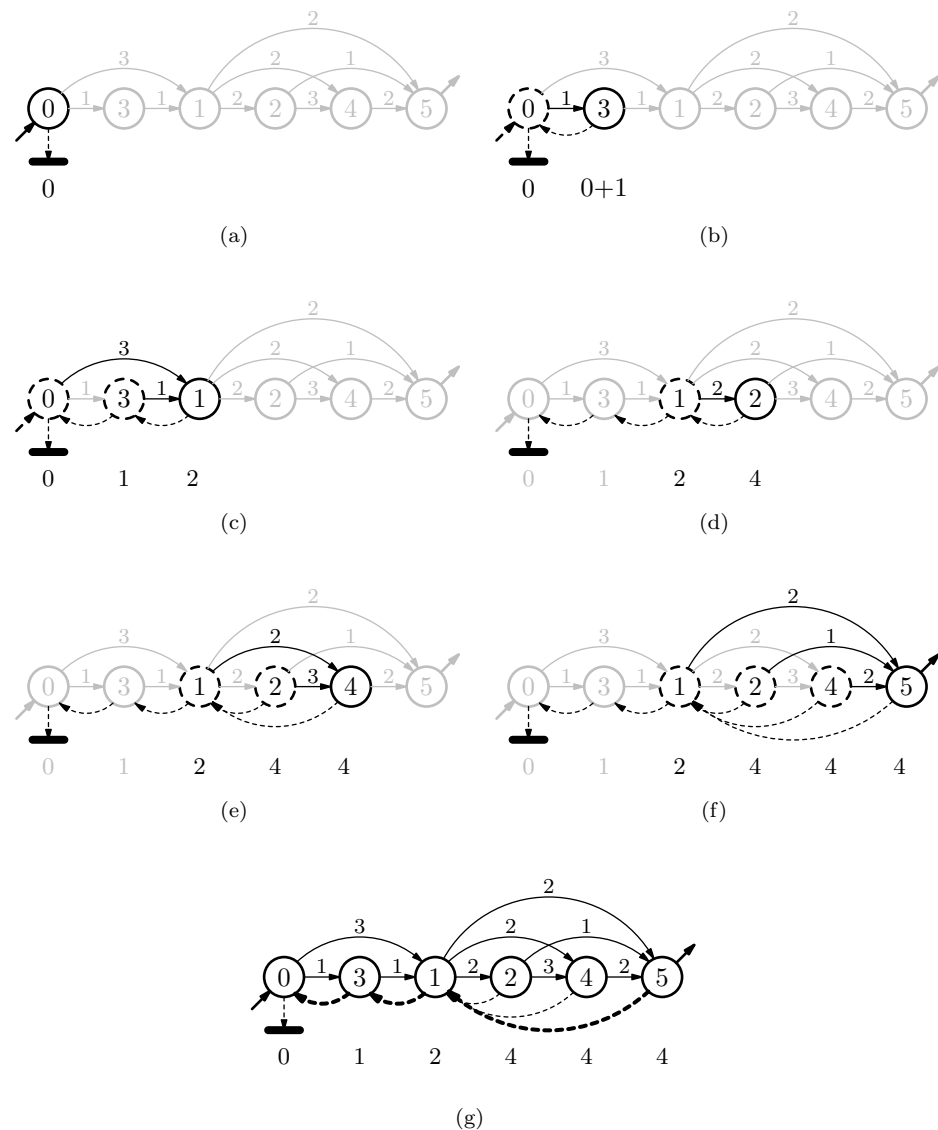
Hagamos una traza paso a paso centrándonos en *backp*, como ilustra la figura 9.21. Tras ordenar topológicamente los vértices del grafo, empezamos con el vértice 0. Como es el vértice inicial, su puntero atrás, *backp[s]* se inicializa a *None* (figura 9.21 (a)). Sólo hay un predecesor del vértice 3, el vértice 0, así que *backp[3]* apunta a él (figura 9.21 (b)). Pasamos al vértice 1. Se puede llegar a este vértice desde dos vértices. El camino óptimo viene del vértice 3, así que *backp[1]* apunta al vértice 3 (figura 9.21 (c)). Pasamos al vértice 2. Sólo se puede llegar desde el vértice 1 (figura 9.21 (d)). Pasamos al vértice 4. El camino óptimo viene del vértice 1 (figura 9.21 (e)). Ahora calculamos *D[0,5]* y *backp[5]*. Este último apunta al vértice 1 (figura 9.21 (f)). El camino óptimo del vértice 0 al vértice 5 se destaca ahora en trazo más grueso (figura 9.21 (h)). Recorriendo hacia atrás los punteros de *backp* desde el vértice 5 y hasta llegar al valor *None* vamos formando el camino de distancia mínima. El recorrido obtiene el camino en orden inverso: [5, 1, 3, 0]. Es necesario, pues, invertir el resultado para obtener el camino óptimo: [0, 3, 1, 5].

Podemos codificar el procedimiento anterior así:

```

shortest_path.py shortest_path.py
25 def dag_shortest_path(G, d, s, t, infinity=3.4e+38):
26     # Devuelve la distancia del camino mínimo y la lista de vértices que lo forman.
27     sorted = topsort(G)
28     D = {}
29     backp = {}
30
31     for v in sorted:
32         if v == s:
33             D[v], backp[v] = 0, None
34         elif G.preds(v):
35             D[v], backp[v] = min( [ (D[u] + d(u, v), u) for u in G.preds(v) ] )
36         else:
37             D[v], backp[v] = infinity, None
38     if v == t: break

```



**Figura 9.21:** Trazas del cálculo del camino más corto en el grafo acíclico de la figura 9.19. En la zona inferior del grafo se muestran los punteros hacia atrás (*backp*) a partir de los cuales se recupera el camino de distancia mínima.

```

39
40 path = [t]
41 while backp[path[-1]] != None:
42     path.append( backp[path[-1]] )
43 path.reverse()
44
45 return D[t], path

```

Hemos introducido dos mejoras:

- Dado que el vértice inicial es siempre  $s$ , lo hemos suprimido en expresiones de la forma  $D[s, v]$ , que pasan a ser de la forma  $D[v]$ .
- En una misma línea calculamos el valor de  $D[v]$  y el de  $backp[v]$ . El caso más curioso es la expresión

$$D[v], backp[v] = \min( [ (D[u] + d(u, v), u) \text{ for } u \text{ in } G.preds(v) ] ).$$

Estúdiala con detenimiento para entender cómo funciona.

Pongamos a prueba nuestro programa:

```

test_shortest_path_b.py
1 from graph import Graph
2 from shortest_path import dag_shortest_path
3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,2), (1,4), (1,5), (2,4), (2,5),
5                          (3,1), (4,5)])
6
7 dist = {(0,1):3, (0,3):1, (1,2):2, (1,4):2, (1,5):2, (2,4): 3, (2,5):1,
8         (3,1):1, (4,5):2}
9 def d(u,v):
10     return dist[u,v]
11
12 distance, path = dag_shortest_path(G, d, 0, 5)
13 print 'Distancia entre los vértices 0 y 5:', distance
14 print 'Camino de distancia mínima entre los vértices 0 y 5:', path

```

Distancia entre los vértices 0 y 5: 4

Camino de distancia mínima entre los vértices 0 y 5: [0, 3, 1, 5]

### Corrección del algoritmo

La corrección del algoritmo *recursive\_dag\_shortest\_distance* se deduce de la derivación de la ecuación recursiva (9.1) (véase la página 228), que hemos obtenido a partir de la formulación del problema del cálculo de la distancia del camino más corto entre  $s$  y  $t$ . El algoritmo *dag\_shortest\_distance* es el resultado de efectuar una transformación recursivo-iterativa del procedimiento anterior.

La técnica de los punteros hacia atrás no hace más que memorizar en cada vértice  $v$  qué predecesor  $u$  proporcionó el valor  $\mathcal{D}(s, v)$ . Si  $\mathcal{D}(s, v)$ , el coste del camino más corto de  $s$  a  $v$ , puede expresarse como  $\mathcal{D}(s, u) + d(u, v)$  es porque se forma concatenando el camino más corto de  $s$  a  $u$  con el arco  $(u, v)$ . Un puntero hacia atrás asociado a  $v$  apunta a  $u$ , y el de  $u$  apuntará al vértice del que viene el camino más corto de  $s$  a  $u$ . Recorrer los punteros hacia atrás, que es lo que hace finalmente el algoritmo *dag\_shortest\_path*, recorre los vértices del camino más corto.

### Análisis de complejidad computacional

El coste temporal de cualquiera de las versiones del método presentado (recursivas o iterativas) es  $O(|V| + |E|)$  si utilizamos cualquier implementación de los grafos excepto la matriz de adyacencia. En tal caso, el coste temporal se elevaría a  $\Theta(|V|^2)$ .

En las versiones recursivas está claro que cada arco del grafo se considera una sola vez. En la versión iterativa, la ordenación topológica de los vértices requiere tiempo  $O(|V| + |E|)$ . Una vez ordenados los vértices, se recorren uno a uno y para cada uno se consulta una vez el valor de cada uno de los arcos que inciden en él. El coste total es, pues,  $O(|V| + |E|)$ . La recuperación del camino no afecta, en principio, al coste temporal: recorrer los punteros hacia atrás, formar la lista con los vértices atravesados e invertirla al final requiere tiempo que podemos acotar superiormente con una función  $O(|V|)$ .

#### PROBLEMAS

► **182** Aunque la recuperación del camino se puede ejecutar en tiempo  $O(|V|)$ , nuestra versión es  $O(|V|^2)$ , pues forma el camino con sucesivas llamadas al método *append* de una lista, que es un método que se ejecuta en tiempo proporcional al tamaño de la lista. ¿Puedes diseñar una versión que garantice un tiempo de ejecución  $O(|V|)$  al construir el camino?

### El principio de optimalidad

Hemos dicho que si  $\mathcal{D}(s, v)$ , el coste del camino más corto de  $s$  a  $v$ , puede expresarse como  $\mathcal{D}(s, u) + d(u, v)$  es porque se forma concatenando el camino más corto de  $s$  a  $u$

con el arco  $(u, v)$ . Podemos decir, pues, que el camino más corto entre  $s$  y  $v$  se forma añadiendo un arco al camino más corto entre  $s$  y otro vértice  $u$ .

**Principio de optimalidad:**  
*Optimality principle.*

Esta es una propiedad interesante y se la conoce como «**Principio de optimalidad**». Podríamos reformularla así: «los prefijos de una solución óptima son, a su vez, óptimos». Haremos uso frecuente del principio de optimalidad cuando estudiemos Programación Dinámica.

### 9.6.3. En grafos sin ciclos negativos: el algoritmo de Bellman-Ford

Si el grafo presenta ciclos, no es posible encontrar un orden topológico con el que resolver la ecuación recursiva del anterior apartado, por lo que la aproximación seguida no es válida. ¿Qué hacer entonces? El algoritmo de Bellman-Ford ofrece una solución a este problema (siempre que el grafo no contenga ciclos de peso negativo).

Para resolver el problema del camino más corto en un grafo con ciclos (no negativos) empezaremos por abordar un problema diferente: el cálculo del camino más cortos entre dos vértices  $s$  y  $t$  formado con, exactamente,  $k$  aristas.

#### La distancia del camino más corto con $k$ aristas

Definamos ahora  $\mathcal{D}(s, t, k)$  como peso del camino con  $k$  aristas más corto entre  $s$  y  $t$ . Tenemos  $\mathcal{D}(s, t, k)$ :

$$\begin{aligned}\mathcal{D}(s, t, k) &= \min_{(v_1, v_2, \dots, v_{k+1}) \in P(s, t, k)} D(v_1, v_2, \dots, v_{k+1}) \\ &= \min_{(v_1, v_2, \dots, v_{k+1}) \in P(s, t, k)} \left( \sum_{1 \leq i < k+1} d(v_i, v_{i+1}) \right).\end{aligned}$$

donde  $P(s, t, k)$  es el conjunto de todos los caminos que unen  $s$  y  $t$  con  $k$ . Podemos definir este conjunto recursivamente:

$$P(s, t, k) = \{(v_1, v_2, \dots, v_{k+1}) \mid v_1 = s; v_{k+1} = t; (v_1, v_2, \dots, v_k) \in P(s, u, k-1); (u, t) \in E\}.$$

El caso base es  $P(s, s, 0) = \{s\}$  y  $P(s, v, 0) = \emptyset$  si  $v \neq s$ .

Sustituimos ahora  $P(s, t, k)$  en la definición de  $\mathcal{D}(s, t, k)$  por su expresión recursiva:

$$\mathcal{D}(s, t, k) = \min_{(u, t) \in E} \left( \min_{(v_1, v_2, \dots, v_k) \in P(s, u, k-1)} \left( \sum_{1 \leq i < k-1} d(v_i, v_{i+1}) \right) + d(u, t) \right).$$

Extraemos de la minimización interior el último término del sumatorio:

$$\mathcal{D}(s, t, k) = \min_{(u, t) \in E} \left( \left( \min_{(v_1, v_2, \dots, v_k) \in P(s, u, k-1)} \left( \sum_{1 \leq i < k-1} d(v_i, v_{i+1}) \right) \right) + d(u, t) \right).$$

Podemos reescribir el sumatorio interior, que no es más que la distancia de un camino:

$$\mathcal{D}(s, t, k) = \min_{(u, t) \in E} \left( \left( \min_{(v_1, v_2, \dots, v_k) \in P(s, u, k-1)} D(v_1, v_2, \dots, v_k) \right) + d(u, t) \right).$$

Hemos llegado a una relación recursiva:

$$\mathcal{D}(s, t, k) = \min_{(u, t) \in E} (\mathcal{D}(s, u, k-1) + d(u, t)).$$

Y como no hay nada especial en  $t$ , podríamos haber deducido la recursión para un vértice  $v$  cualquiera:

$$\mathcal{D}(s, v, k) = \min_{(u, v) \in E} (\mathcal{D}(s, u, k) + d(u, v)).$$

Hay un caso que podemos considerar base de la recursión: cuando  $v = s$  y  $k = 0$  se propone la búsqueda de la distancia óptima de  $s$  a  $s$  sin arista alguna. Dicha distancia es, por definición, nula:

$$\mathcal{D}(s, s, 0) = 0.$$

Otro caso base tiene lugar si  $v \neq s$  y  $k = 0$ :

$$\mathcal{D}(s, v, 0) = +\infty.$$

Un caso especial es el de los vértices sin predecesores. La minimización se propone entonces sobre un conjunto vacío de elementos.

$$\mathcal{D}(s, v, k) = \begin{cases} 0 & \text{si } v = s \text{ y } k = 0; \\ +\infty & \text{si } v \neq s \text{ y } k = 0; \\ +\infty & \text{si } \nexists (u, v) \in E \text{ y } k > 0; \\ \min_{(u,v) \in E} (\mathcal{D}(s, u) + d(u, v)) & \text{en otro caso.} \end{cases} \quad (9.2)$$

En principio, el valor que deseamos calcular es

$$\hat{D} = \mathcal{D}(s, t, k).$$

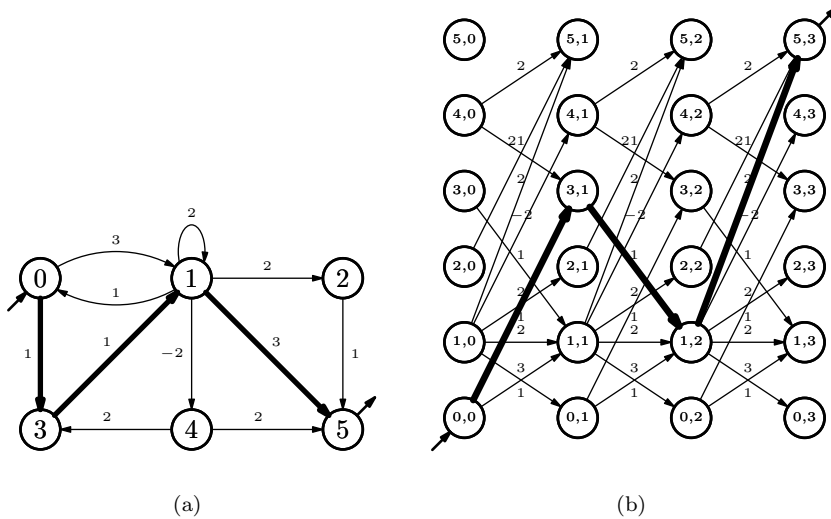
Hay una interpretación interesante de esta ecuación recursiva que se ilustra en la figura 9.22. Si buscamos la distancia del camino con 3 aristas más corto entre los vértices 0 y 5 del grafo de la figura 9.22 (a), por ejemplo, buscamos la distancia del camino más corto entre los vértices (0,0) y (5,3) del grafo de la figura 9.22 (b). Se trata de un grafo multietapa que se puede formar a partir del primero. Dado un grafo  $G = (V, E)$ , un valor positivo  $k$  y una función de ponderación  $d$ , su grafo asociado,  $G' = (V', E')$ , es

$$\begin{aligned} V' &= \{(v, i) \mid v \in V, 0 \leq i \leq k\}, \\ E' &= \{((u, i-1), (v, i)) \mid (u, v) \in E, 0 < i \leq k\}; \end{aligned}$$

y su función de ponderación asociada  $d'$  se define así a partir de  $d$ :

$$d'((u, i-1), (v, i)) = d(u, v).$$

Nótese que todo camino  $(v_1, v_2, \dots, v_{k+1})$  en  $G$  con  $k$  aristas tiene un camino equivalente (con el mismo peso) en  $G'$ :  $((v_1, 0), (v_2, 1), \dots, (v_{k+1}, k))$ . Y viceversa: todo camino en  $G'$  tiene un camino equivalente en  $G$ . Así pues, resulta evidente que calcular la distancia del camino más corto en  $G'$  entre  $(s, 0)$  y  $(t, k)$  es un problema equivalente a calcular la distancia del camino entre  $s$  y  $t$  de  $k$  aristas más corto en  $G$ .



**Figura 9.22:** (a) Grafo ponderado con ciclos. En trazo grueso se muestran las aristas del camino (0, 3, 1, 5). (b) Grafo multietapa asociado al anterior sobre el que se efectúa la búsqueda del camino más corto entre 0 y 5 con 3 aristas. En trazo grueso se muestra el camino equivalente en este grafo al que se destacó en el grafo de la izquierda.

No es necesario construir explícitamente  $G'$  para calcular su camino más corto entre  $(s, 0)$  y  $(t, k)$ . He aquí una función iterativa implementada en Python:

```

bellman_ford_1.py
1 def shortest_distance_k(G, d, s, t, k, infinity=3.4e+38):
2     D = {}
3     for v in G.V: D[s, v, 0] = infinity
4     D[s, s, 0] = 0
5
6     for i in range(1, k+1):
7         for v in G.V:
8             if len(G.preds(v)) > 0:
9                 D[s, v, i] = min( [ D[s, u, i-1] + d(u,v) for u in G.preds(v) ] )
10            else:
11                D[s, v, i] = infinity
12
13     return D[s, t, k]

```

Probemos el algoritmo con el grafo de la figura 9.22 (a).

```

test_bellman_ford_1.py
1 from graph import Graph
2 from bellman_ford_1 import shortest_distance_k
3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,0), (1,1), (1,2), (1,4), (1,5),
5                          (2,5), (3,1), (4,3), (4,5)])
6
7 dist = {(0,1):3, (0,3):1, (1,0): 1, (1,1): 2, (1,2):2, (1,4):-2, (1,5):3,
8         (2,5): 1, (3,1):1, (4,3):2, (4,5):2}
9
10 def d(u,v):
11     return dist[u,v]
12
13 for i in range(6):
14     print 'Distancia de 0 a 5 con %d aristas:' %i, shortest_distance_k(G, d, 0, 5, i)

```

El resultado de ejecutar el programa es éste:

```

Distancia de 0 a 5 con 0 aristas: 3.4e+38
Distancia de 0 a 5 con 1 aristas: 3.4e+38
Distancia de 0 a 5 con 2 aristas: 6
Distancia de 0 a 5 con 3 aristas: 3
Distancia de 0 a 5 con 4 aristas: 2
Distancia de 0 a 5 con 5 aristas: 4

```

No es necesario usar tres índices para acceder a  $D$ : en toda expresión de la forma  $D[s, v, i]$  podemos eliminar el índice  $s$ , pues siempre aparece:

```

bellman_ford.py
1 def shortest_distance_k(G, d, s, t, k, infinity=3.4e+38):
2     D = {}
3     for v in G.V: D[v, 0] = infinity
4     D[s, 0] = 0
5
6     for i in range(1, k+1):
7         for v in G.V:
8             if len(G.preds(v)) > 0:
9                 D[v, i] = min( [ D[u, i-1] + d(u,v) for u in G.preds(v) ] )
10            else:
11                D[v, i] = infinity
12
13     return D[t, k]

```



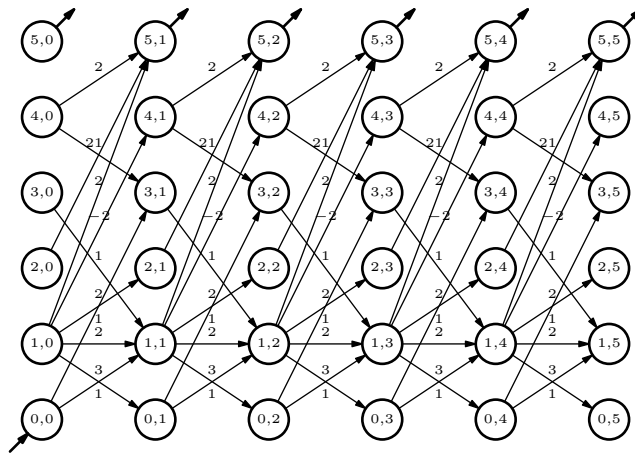
### La distancia del camino más corto entre dos vértices (con cualquier número de aristas)

El camino más corto entre  $s$  y  $t$  no puede tener más de  $|V| - 1$  aristas. Si tuviera más, contendría un ciclo. Como no admitimos ciclos negativos, siempre sería posible mejorar el peso del camino eliminado ese hipotético ciclo. Así pues,

$$\mathcal{D}(s, t) = \min_{0 \leq k < |V|} \mathcal{D}(s, t, k).$$

Es decir, el peso camino más corto entre  $s$  y  $t$  es el menor de entre los pesos de los caminos más cortos de  $s$  a  $t$  con  $k$  aristas, para  $k$  inferior a  $|V|$ .

No es necesario efectuar  $|V|$  llamadas a la función `shortest_distance_k`. Podemos calcular el camino más corto con  $|V| - 1$  aristas y obtendremos, como subproducto el valor de  $\mathcal{D}(s, t, k)$  para todo  $k < |V| - 1$ . En la figura 9.23 se muestra el grafo que hubiésemos construido implícitamente al calcular el camino más corto entre 0 y 5 con 5 aristas en el grafo de la figura 9.22 (a). En ese grafo, hemos considerado que todos los vértices de la forma  $(t, i)$ , para  $0 \leq i < 5$ . La distancia mínima de entre las calculadas para cada uno de esos vértices es  $\mathcal{D}(0, 6)$ . Se trata de un grafo acíclico, así que el algoritmo desarrollado en el apartado anterior encuentra aplicación en este nuevo problema.



**Figura 9.23:** Grafo multietapa asociado al grafo de la figura 9.22 (a) sobre el que se efectúa implícitamente la búsqueda del camino más corto entre 0 y 5 con cualquier número de aristas.

```

bellman_ford.py
def shortest_distance(G, d, s, t, infinity=3.4e+38):
    D = {}
    for v in G.V: D[v, 0] = infinity
    D[s, 0] = 0

    for i in range(1, len(G.V)):
        for v in G.V:
            if len(G.preds(v)) > 0:
                D[v, i] = min( [ D[u, i-1] + d(u, v) for u in G.preds(v) ] )
            else:
                D[v, i] = infinity

    return min([D[t, i] for i in range(len(G.V))])

```

Ponemos a prueba el programa con el grafo 9.23

```

test_bellman_ford.b.py
1 from graph import Graph
2 from bellman_ford import shortest_distance

```

```

3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,0), (1,1), (1,2), (1,4), (1,5),
5                        (2,5), (3,1), (4,3), (4,5)])
6
7 dist = {(0,1):3, (0,3):1, (1,0): 1, (1,1): 2, (1,2):2, (1,4):-2, (1,5):3,
8        (2,5): 1, (3,1):1, (4,3):2, (4,5):2}
9
10 def d(u,v):
11     return dist[u,v]
12
13 print 'Distancia de 0 a 5:', shortest_distance(G, d, 0, 5)

```

Distancia de 0 a 5: 2

### El camino más corto

A la vista de que hemos reducido el cálculo del camino más corto en un grafo con ciclos (no negativos) al del camino más corto en un grafo acíclico obtenido a partir del problema, resulta inmediato aplicar la técnica de los punteros hacia atrás para recuperar el camino óptimo.

bellman\_ford.py

bellman\_ford.py

```

29 def shortest_path(G, d, s, t, infinity=3.4e+38):
30     D = {}
31     backp = {}
32
33     for v in G.V:
34         D[v, 0], backp[s, 0] = infinity, None
35     D[s, 0], backp[s, 0] = 0, None
36
37     for i in range(1, len(G.V)):
38         for v in G.V:
39             if len(G.preds(v)) > 0:
40                 D[v, i], backp[v, i] = min([(D[u, i-1] + d(u,v), u) for u in G.preds(v)])
41             else:
42                 D[v, i], backp[v, i] = infinity, None
43
44     mindist = min([D[t, i] for i in range(len(G.V))])
45     k = [D[t, i] for i in range(len(G.V))].index(mindist)
46     path = [t]
47     while backp[path[-1], k] != None:
48         path.append( backp[path[-1], k] )
49         k -= 1
50     path.reverse()
51
52     return mindist, path

```

test\_bellman\_ford.c.py

test\_bellman\_ford.c.py

```

1 from graph import Graph
2 from bellman_ford import shortest_path
3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,0), (1,1), (1,2), (1,4), (1,5),
5                        (2,5), (3,1), (4,3), (4,5)])
6
7 dist = {(0,1):3, (0,3):1, (1,0): 1, (1,1): 2, (1,2):2, (1,4):-2, (1,5):3,
8        (2,5): 1, (3,1):1, (4,3):2, (4,5):2}
9
10 def d(u,v):
11     return dist[u,v]

```

```

12
13 mindist, path = shortest_path(G, d, 0, 5)
14 print 'Distancia de 0 a 5:', mindist
15 print 'Camino de distancia mínima de 0 a 5:', path

```

Distancia de 0 a 5: 2

Camino de distancia mínima de 0 a 5: [0, 3, 1, 4, 5]

#### PROBLEMAS

► **183** Aplica el algoritmo de Bellman-Ford al mapa de la península ibérica diseñando una función que devuelva una lista con las ciudades que deben visitarse al ir por el camino más corto de una ciudad a otra.

### Complejidad computacional

Más allá de un análisis directo de la complejidad, interesa reflexionar brevemente acerca de esta técnica de resolución. El problema de calcular el camino más corto en un grafo con ciclos se ha reducido a calcular el camino más corto en un grafo acíclico y multietapa. Hay una correspondencia biunívoca entre caminos con menos de  $|V|$  vértices en el grafo original y caminos en el nuevo grafo. Hemos podido, de este modo, aplicar una técnica ya conocida para resolver el problema: encontramos el camino más corto en el nuevo grafo y devolvemos el camino asociado en el grafo original. El coste tiene, por tanto, tres componentes diferentes:

- la construcción del grafo multietapa,
- el cálculo del camino más corto en él,
- y la obtención del camino asociado en el grafo original.

El primero paso no tiene coste alguno porque no se efectúa explícitamente. Si  $G = (V, E)$  es el grafo original, el grafo acíclico sobre el que efectuamos el cálculo del camino más corto presenta  $|V|(|V| + 1)$  vértices y  $(|V| - 1) \cdot |E|$  aristas. Como el cálculo del camino más corto en un grafo acíclico presenta una complejidad temporal  $O(|E|)$ , esa es la complejidad temporal de este paso. La obtención del camino asociado sólo requiere eliminar la información relativa a las etapas del nuevo grafo y puede realizarse en tiempo proporcional a la longitud del camino, es decir, en tiempo  $O(|V|)$ .

El coste temporal del algoritmo es, pues,  $O(|V||E|)$ . El coste espacial es  $O(|V|^2)$ , pues ése es el número de vértices del grafo multietapa.

#### PROBLEMAS

► **184** Aplica el algoritmo de Bellman-Ford al mapa de la península ibérica diseñando una función que devuelva la distancia más camino más corta entre dos ciudades (por carretera, se entiende).

### 9.6.4. En grafos ponderados positivos: el algoritmo de Dijkstra

Si los grafos presentan ciclos pero todas sus aristas tienen pesos positivos, es posible resolver el problema de calcular el camino más corto entre  $s$  y  $t$  con una menor complejidad computacional. El algoritmo de Dijkstra resuelve el problema en tiempo  $O(|E| \lg |V|)$ . Empezaremos, no obstante, por presentar una versión ineficiente (cuadrática con  $|V|$ ) cuya idea fundamental se ilustra en el siguiente ejemplo.

Edsger W. Dijkstra (1930–2002) es uno de los pioneros de la informática. El conocido como «algoritmo de Dijkstra» se publicó en el artículo «A note on two problems in connection with graphs», *Numerical Mathematics*, núm. 1, pp. 269–271, 1959.

### Un ejemplo

El algoritmo de Dijkstra se inspira en el recorrido por primero en anchura de un grafo. Presentaremos sus ideas fundamentales desarrollando un ejemplo: el cálculo de la distancia del camino más corto entre las ciudades de Andratx ( $s$ ) y Manacor ( $t$ ) en la isla de Mallorca. (El mapa de carreteras de la isla de Mallorca se mostró en la figura 7.5, en la página 155.)

Usaremos una tabla  $D$  indexada por vértices (ciudades) y un conjunto de vértices  $S$ . La tabla  $D$  asociará a cada ciudad la distancia del camino más corto desde  $s$  visto hasta el momento. La segunda,  $S$ , contendrá los vértices para los que conocemos la distancia del camino más corto desde el vértice inicial.

Empezaremos con este estado (mostramos los elementos que han ingresado en  $S$  con una marca en la casilla correspondiente):

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$S$														

La tabla refleja lo que sabemos inicialmente: el camino más corto de Andratx a Andratx tiene 0 kilómetros y la distancia más corta de Andratx a cualquiera de las restantes ciudades se supone, de momento, infinita.

El algoritmo selecciona ahora la ciudad de  $V - S$  con menor valor de  $D$ . Seleccionamos la única posibilidad que tenemos: la ciudad de Andratx. Esta ciudad ingresa ahora en el conjunto  $S$ :

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$S$		■												

Procedemos ahora como haríamos con la exploración en anchura: consideramos los vértices adyacentes a Andratx, que son Sóller, Calvià y Palma de Mallorca. En lugar de limitarnos a marcar dichos vértices como visitados, anotamos en su entrada de la tabla  $D$  la distancia con la que es posible llegar hasta ellos desde Andratx con lo que sabemos de momento. Podemos ir a Sóller siguiendo una carretera de 56 kilómetros, a Calvià por una carretera de 14 y a Palma de Mallorca por una de 30:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	14	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	30	$\infty$	$\infty$	56
$S$		■												

No estamos seguros de saber cuál es el camino más corto desde Andratx hasta cada una de esas ciudades, así que no la añadimos a  $S$ .

Consideramos ahora la ciudad con menor valor de  $D$  tal que no pertenece a  $S$ : Calvià. Según la tabla  $D$ , su distancia a Andratx es de 14 kilómetros. Ahora estamos completamente seguros de que esa es la menor distancia con la que podemos ir de una Andratx a Calvià. Cualquier otra forma de llegar a Calvià se formará sumando una cantidad positiva a una distancia igual o mayor, pues todos los valores almacenados en  $D$  para vértices de  $V - S$  son mayores. Éste es el estado actual:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	14	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	30	$\infty$	$\infty$	56
$S$		■		■										

A continuación vemos si es posible mejorar el valor de  $D$  asociado a las ciudades directamente conectadas con Calvià.

- De Calvià a Andratx podemos ir recorriendo 14 kilómetros. Si hiciésemos eso, habríamos encontrado una forma de ir de Andratx a Andratx recorriendo un total de 28 kilómetros (14 de Andratx a Calvià más 14 de Calvià a Andratx). ¡Pero ya sabemos ir recorriendo 0 kilómetros! No hace falta, pues, modificar el valor de  $D[\text{'Andratx'}]$ . De hecho, en adelante no consideraremos las conexiones con ciudades que están en  $S$ .
- De Calvià podemos ir a Palma de Mallorca siguiendo una carretera de 14 kilómetros, que sumados a los 14 con los que podemos ir de Andratx a Calvià hacen un total de 28 kilómetros. Hasta el momento sabíamos ir de Andratx a Palma de Mallorca por un camino de 30 kilómetros. Como el nuevo es menor, actualizamos  $D[\text{'Palma de Mallorca'}]$ .

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	14	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	28	$\infty$	$\infty$	56
$S$		■		■										

Seleccionamos ahora a Palma de Mallorca: de entre las ciudades que no están en  $S$ , es la de menor valor de  $D$ . La añadimos a  $S$  y actualizamos, si procede, el valor de  $D$  de sus ciudades adyacentes (Calvià, Andratx, Marratxí y Llucmajor):

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	14	$\infty$	$\infty$	$\infty$	48	$\infty$	42	28	$\infty$	$\infty$	56
$S$		■		■							■			

Le toca el turno a Marratxí, que proporciona un camino que llega a Inca.

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	14	$\infty$	$\infty$	54	48	$\infty$	42	28	$\infty$	$\infty$	56
$S$		■		■						■	■			

Y ahora a Llucmajor, que permite alcanzar Campos del Port a través de un camino de 62 kilómetros:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
$D$	$\infty$	0	$\infty$	14	62	$\infty$	54	48	$\infty$	42	28	$\infty$	$\infty$	56
$S$		■		■				■		■	■			

Es el turno de Inca. Podemos ir de Andratx a Manacor o a Alcúdia pasando por Inca siguiendo un camino de 79 kilómetros.

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	$\infty$	14	62	$\infty$	54	48	79	42	28	$\infty$	$\infty$	56
<i>S</i>		■		■			■	■		■	■			

Ya sabemos llegar de algún modo a Manacor: a través de Inca; pero aún no estamos seguros de que éste sea el camino más corto. Seleccionamos ahora a la ciudad de Sóller, que nos da acceso a Pollença:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	$\infty$	14	62	$\infty$	54	48	79	42	28	110	$\infty$	56
<i>S</i>		■		■			■	■		■	■			■

Toca considerar a Campos del Port:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	$\infty$	14	62	$\infty$	54	48	79	42	28	110	75	56
<i>S</i>		■		■	■		■	■		■	■			■

Pasamos a estudiar Santanyí. Desde Andratx, pasando por Santanyí, es posible ir a Manacor por un camino con 102 kilómetros. No hemos mejorado, pues, el que ya conocíamos. Pero seguimos sin estar seguros de haber encontrado el mejor camino hasta Manacor (¿y si hubiera uno mejor que viniera, por ejemplo, de Artà?).

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	$\infty$	14	62	$\infty$	54	48	79	42	28	110	75	56
<i>S</i>		■		■	■		■	■		■	■		■	■

Ahora consideramos Alcúdia, que da acceso por ve primera a Artà, pero que también permite mejorar nuestra estimación del camino más corto entre Andratx y Pollença:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	115	14	62	$\infty$	54	48	79	42	28	89	75	56
<i>S</i>	■	■		■	■		■	■		■	■		■	■

Ahora extraemos Manacor y, por fin, estamos seguros de que no hay camino de Andratx a ella con menos de 79 kilómetros: cualquier otra forma de llegar a Manacor obligará a sumar una cantidad positiva (nula, en el mejor de los casos) al este valor. Saber que Manacor está accesible desde Andratx con un camino de 79 kilómetros nos permite actualizar nuestra estimación de la distancia del camino más corto de Andratx a Artà:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	96	14	62	$\infty$	54	48	79	42	28	89	75	56
<i>S</i>	■	■		■	■		■	■	■	■	■		■	■

Podemos seguir, aunque ya conocemos el resultado final: el algoritmo devolverá el valor 79 como distancia del camino más corto entre Andratx y Manacor. Completamos la traza. Seleccionamos Pollença, sin mejorar la estimación de la distancia del camino más corto de Andratx a las ciudades adyacentes a Pollença:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	96	14	62	$\infty$	54	48	79	42	28	89	75	56
<i>S</i>	■	■		■	■		■	■	■	■	■	■	■	■

Seleccionamos Artà y encontramos así un camino que nos lleva a Capdepera:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	96	14	62	104	54	48	79	42	28	89	75	56
<i>S</i>	■	■	■	■	■		■	■	■	■	■	■	■	■

Finalmente, seleccionamos Capdepera:

	Alcúdia	Andratx	Artà	Calvià	Campos del Port	Capdepera	Inca	Llucmajor	Manacor	Marratxí	Palma de Mallorca	Pollença	Santanyí	Sóller
<i>D</i>	79	0	96	14	62	104	54	48	79	42	28	89	75	56
<i>S</i>	■	■	■	■	■	■	■	■	■	■	■	■	■	■

Y al haber incluido en *S* a todos los vértices, el algoritmo se detiene.

Una observación: aunque sólo nos interesaba la distancia el camino más corto de Andratx a Manacor, hemos obtenido como subproducto la distancia del camino más corto de Andratx a cualquiera de las ciudades de la isla.

### Una implementación directa

Podemos implementar el algoritmo del ejemplo siguiendo una aproximación directa (y que supone, además, que el grafo es conexo):

```

dijkstra_1.py
1 from sets import Set
2
3 def dijkstra_shortest_path_distance(G, d, s, t, infinity=3.4e+38):
4     S = Set()
5     D = {}
6     for v in G.V: D[v] = infinity
7     D[s] = 0
8
9     while len(S) != len(G.V):
10         dmin = infinity
11         for v in G.V:
12             if v not in S and D[v] < dmin:
13                 u = v
14                 dmin = D[u]
15         S.add(u)
16
17         for v in G.succs(u):
18             if v not in S:
19                 D[v] = min(D[v], D[u] + d(u,v))
20
21     return D[t]
```

Pongamos a prueba nuestra implementación con el mapa de carreteras de Mallorca. Usaremos el fichero `mallorca.py`, que definimos en la página 8.4.

```

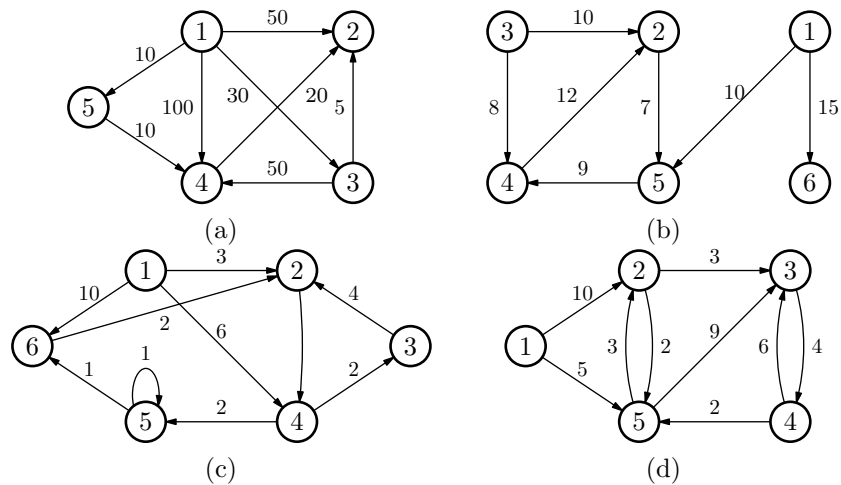
test_dijkstra_1.py
1 from graph import Graph
2 from dijkstra_1 import dijkstra_shortest_path_distance
3 from mallorca import Mallorca, d
4
5 distance = dijkstra_shortest_path_distance(Mallorca, d, 'Andratx', 'Manacor')
6 print 'Distancia de Andratx a Manacor:', distance

```

Distancia de Andratx a Manacor: 79

### PROBLEMAS

► **185** Haz una traza del algoritmo de Dijkstra para los siguientes grafos, tomando como vértice origen el 1.



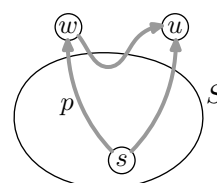
### Corrección del algoritmo

**Teorema 9.3** El algoritmo de Dijkstra calcula el camino más corto entre  $s$  y  $t$ .

*Demostración.* Demostraremos por inducción sobre la talla de  $S$  que, para todo  $v \in S$ ,  $D[v]$  es la distancia mínima entre  $s$  y  $v$  y que  $D[v]$ , para todo  $v \in V - S$ , es la distancia mínima entre  $s$  y  $t$  que únicamente visita vértices de  $V - S$  (excepto  $v$ , que no está en el conjunto  $S$ ).

**Base de inducción.** Cuando  $|S| = 1$ , su único vértice es  $s$  y  $D[s]$  vale 0, que es la distancia del camino más corto de  $s$  a  $s$ .

**Paso de inducción.** Supongamos que  $D[v]$  es la distancia del camino más corto de  $s$  a  $v$  para todo elemento de  $S$  en un instante dado, es decir, para una talla determinada de  $S$ . Sea  $u$  el vértice de  $V - S$  seleccionado en las líneas 10–14 del algoritmo. Nuestro objetivo es demostrar que  $D[u]$  es la distancia del camino más corto de  $s$  a  $u$ . Supongamos que no lo es, es decir, que existe un camino  $p$  más corto. Dicho camino debe tener al menos un vértice diferente de  $u$  que no esté en  $S$ . Sea  $w$  el primer vértice de  $p$  que no está en  $S$ .





Supongamos que no hay aristas de peso nulo. Por fuerza, la distancia de  $s$  a  $w$  ha de ser menor que  $D[u]$ , el tramo de  $p$  que va de  $v$  a  $w$  añade una distancia que no puede ser negativa (no hay aristas de peso negativo). Como el prefijo de  $p$  que llega a  $w$  está formado por nodos de  $S$ , esto equivale a suponer que  $D[w] < D[u]$ . Pero eso es una contradicción: de ser así, en la iteración actual no habríamos seleccionado el vértice  $u$ , sino el vértice  $w$ . La conclusión es que el camino  $p$  no existe.

Si hubiera aristas de peso nulo, la única posibilidad de que  $p$  existiera es que la porción que conecta  $w$  con  $u$  se formara con una secuencia de aristas de peso nulo. En tal caso,  $D[w]$  sería igual a  $D[u]$  y, por tanto,  $D[u]$  correspondería a la distancia del camino más corto entre  $s$  y  $u$ , como queríamos demostrar.

Nótese que una vez ingresa en  $S$  un vértice  $u$ , su valor  $D[u]$  no se modifica nunca más, así que mantiene la distancia mínima entre  $s$  y  $u$ .

El algoritmo finaliza cuando ingresan en  $S$  todos los vértices y devuelve el valor  $D[t]$ , así que calcula correctamente la distancia mínima de  $s$  a  $t$ .  $\square$

#### .....PROBLEMAS.....

► **186** Muestra un ejemplo sencillo de un grafo dirigido con pesos negativos en las aristas, para el que el algoritmo de Dijkstra produciría una solución errónea. ¿Podría darse algún caso en el que, aunque el grafo contuviera pesos negativos, la solución obtenida por Dijkstra fuera la correcta? Si es así da algún ejemplo.

► **187** Deseamos encontrar un camino de coste mínimo entre la esquina inferior izquierda y la esquina superior derecha de una matriz en la el valor de la celda de índices  $(i, j)$  representa la altura del terreno en esas coordenadas. Desde una casilla podemos desplazarnos únicamente hacia el norte y hacia el este. Buscamos el camino de altura media mínima.

► **188** Si en lugar de la suma de pesos de aristas consideramos el producto de sus pesos para calcular el peso de un camino, ¿es suficiente con exigir que el peso de las aristas sea positivo para poder aplicar el algoritmo de Dijkstra?

### Complejidad computacional de la implementación directa

La complejidad espacial es, evidentemente,  $\Theta(|V|)$ . La complejidad temporal es  $\Theta(|V|^2)$ . Con cada iteración del bucle **while** se selecciona un vértice recorriendo los  $O(|V|)$  vértices aún no seleccionados.

### Una versión más eficiente

Nótese que la operación «crítica» es la selección del elemento de menor valor de  $D[v]$  en  $V - S$ . Podemos seleccionar el menor de una serie de elementos con una cola de prioridad. El problema estriba en que los valores de la cola de prioridad se modifican durante la ejecución del algoritmo: el valor de  $D[v]$  puede ir decreciendo conforme evoluciona la ejecución. El min-heap indexado ofrece una solución atractiva, pues permiten, decrementar el valor asociado a un elemento en tiempo logarítmico con la talla de la cola de prioridad.

 dijkstra\_2.py

dijkstra\_2.py

```
1 from heap import IndexedMinHeap
2 from sets import Set
3
4 def dijkstra_shortest_path_distance(G, d, s, t, infinity=3.4e+38):
5     S = Set()
6     D = {}
7     for v in G.V: D[v] = infinity
8     D[s] = 0
9
10    Q = IndexedMinHeap(len(G.V), [(v, D[v]) for v in G.V])
```

```

11
12 while not Q.is_empty():
13     u, aux = Q.extract_min()
14     S.add(u)
15     for v in G.succs(u):
16         if v not in S:
17             if D[u] + d(u, v) < D[v]:
18                 D[v] = D[u] + d(u, v)
19                 Q.decrease(v, D[v])
20
21 return D[t]

```

test\_dijkstra.2.py

test\_dijkstra.2.py

```

1 from graph import Graph
2 from dijkstra_2 import dijkstra_shortest_path_distance
3 from mallorca import Mallorca, d
4
5 distance = dijkstra_shortest_path_distance(Mallorca, d, 'Andratx', 'Manacor')
6 print 'Distancia de Andratx a Manacor:', distance

```

Distancia de Andratx a Manacor: 79

### Análisis de complejidad

El coste espacial es idéntico al de la versión anterior, esto es,  $\Theta(|V|)$ . El coste temporal es ahora  $O(|E| \lg |V|)$ : extraemos  $|V|$  vértices de  $Q$  y cada extracción necesita ejecutar  $O(\lg |V|)$  pasos; por otra parte, cada arista del grafo puede generar una llamada a  $Q.decrease$ , operación que también tiene un coste  $O(\lg |V|)$ .

- .....PROBLEMAS.....
- **189** Supóngase que se modifica la condición del bucle «while» del algoritmo de Dijkstra: **while not**  $Q.is\_empty()$  por **while**  $len(Q) > 1$ . Esta modificación provocaría que el bucle se repitiera  $|V| - 1$  veces en lugar de  $|V|$  veces. ¿Puede afirmarse que el algoritmo modificado resolvería igualmente el problema? ¿Por qué?
  - **190** ¿Cómo podría comprobarse si un grafo dirigido es fuertemente conexo utilizando el algoritmo de Dijkstra? Comenta qué modificaciones serían necesarias realizar sobre el algoritmo.
  - **191** ¿Qué coste temporal tendría el algoritmo de Dijkstra si el grafo estuviera representado mediante una matriz de adyacencia? Razona la respuesta adecuadamente.
  - **192** Es posible diseñar una versión más eficiente del algoritmo de Dijkstra, capaz de resolver el problema del camino más corto en  $O(|V| \lg |V|)$  pasos. Averigua cómo.
- .....

### Un refinamiento

El algoritmo de Dijkstra, tal cual lo hemos presentado, espera a alcanzar todos los vértices para finalizar. Es posible, no obstante, finalizar antes: tan pronto ingresa  $t$  en el conjunto  $S$ :

dijkstra.py

dijkstra.py

```

1 from heap import IndexedMinHeap
2 from sets import Set
3
4 def dijkstra_shortest_path_distance(G, d, s, t, infinity=3.4e+38):
5     S = Set()
6     D = {}
7     for v in G.V: D[v] = infinity
8     D[s] = 0

```

```

9
10  $Q = \text{IndexedMinHeap}(\text{len}(G.V), [(v, D[v]) \text{ for } v \text{ in } G.V])$ 
11
12 while not  $Q.is\_empty()$ :
13      $u, aux = Q.extract\_min()$ 
14     if  $u == t$ : break
15      $S.add(u)$ 
16     for  $v$  in  $G.succs(u)$ :
17         if  $v$  not in  $S$ :
18             if  $D[u] + d(u, v) < D[v]$ :
19                  $D[v] = D[u] + d(u, v)$ 
20                  $Q.decrease(v, D[v])$ 
21
22 return  $D[t]$ 

```

```

test.dijkstra.a.py  test.dijkstra.a.py
1 from graph import Graph
2 from dijkstra import dijkstra_shortest_path_distance
3 from mallorca import Mallorca, d
4
5 distance = dijkstra_shortest_path_distance(Mallorca, d, 'Andratx', 'Manacor')
6 print 'Distancia de Andratx a Manacor:', distance

```

Distancia de Andratx a Manacor: 79

#### .....PROBLEMAS.....

- **193** Analiza la complejidad computacional de la última versión del algoritmo.

### El camino más corto

Recuperar el camino resulta sencillo: podemos usar punteros hacia atrás.

```

dijkstra.py  dijkstra.py
24 def  $dijkstra\_shortest\_path(G, d, s, t, infinity=3.4e+38)$ :
25      $S = Set()$ 
26      $D = \{\}$ 
27      $backp = \{\}$ 
28     for  $v$  in  $G.V$ :  $D[v], backp[v] = infinity, None$ 
29      $D[s] = 0$ 
30      $backp[s] = None$ 
31
32      $Q = \text{IndexedMinHeap}(\text{len}(G.V), [(v, D[v]) \text{ for } v \text{ in } G.V])$ 
33
34     while not  $Q.is\_empty()$ :
35          $u, aux = Q.extract\_min()$ 
36         if  $u == t$ : break
37          $S.add(u)$ 
38         for  $v$  in  $G.succs(u)$ :
39             if  $v$  not in  $S$ :
40                 if  $D[u] + d(u, v) < D[v]$ :
41                      $D[v] = D[u] + d(u, v)$ 
42                      $Q.decrease(v, D[v])$ 
43                      $backp[v] = u$ 
44
45      $v = t$ 
46      $path = [v]$ 
47     while  $backp[v] != None$ :

```

```

48     v = backp[v]
49     path.append(v)
50     path.reverse()
51     return D[t], path

```

test\_dijkstra.b.py

test\_dijkstra.b.py

```

1 from graph import Graph
2 from dijkstra import dijkstra_shortest_path
3 from mallorca import Mallorca, d
4
5 distance, path = dijkstra_shortest_path(Mallorca, d, 'Andratx', 'Manacor')
6 print 'Camino más corto de Andratx a Manacor:', path
7 print 'Distancia recorrida:', distance

```

```

Camino más corto de Andratx a Manacor: ['Andratx', 'Calvià', 'Palma de Mallorca',
'Marratxí', 'Inca', 'Manacor']
Distancia recorrida: 79

```

### Algunas observaciones

Hemos dicho que el algoritmo de Dijkstra guarda relación con la exploración por primero en anchura de un grafo. Si la función de ponderación asignara peso 1 a todas las aristas y la cola de prioridad respetara el orden de ingreso para las extracciones (primero en entrar, primero en salir), tendríamos un algoritmo equivalente (aunque más costoso) a la exploración por primero en anchura.

El algoritmo de Dijkstra también guarda relación con otro que ya hemos estudiado detenidamente: el algoritmo de Kruskal. Ambos siguen una estrategia de exploración por «primero según prioridad». En el algoritmo de Kruskal se seleccionan aristas con un criterio de prioridad que podemos calificar de estático: su peso. En el algoritmo de Dijkstra se seleccionan vértices con una priorización dinámica, que se va actualizando conforme la ejecución del algoritmo evoluciona.

El algoritmo de Dijkstra puede utilizarse como base para la resolución de otros problemas, como proponen los siguientes ejercicios.

.....PROBLEMAS.....

► **194** Dado un grafo dirigido  $G = (V, E)$  ponderado con pesos no negativos en las aristas, modifica el algoritmo de Dijkstra para que devuelva como resultado los costes de los caminos mínimos entre cualquier par de vértices del grafo. ¿Cuál sería el coste temporal del algoritmo?

► **195** Es posible que haya dos o más caminos entre dos vértices con el mismo peso. Explica cómo modificar el algoritmo de Dijkstra para que si hay más de un camino de coste mínimo entre el origen  $s$  y  $t$ , se escoja el camino con el menor número de aristas.

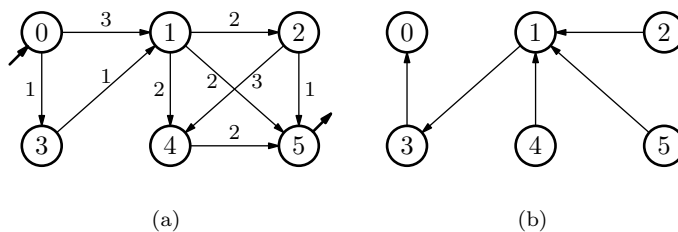
► **196** Modifica el algoritmo de Dijkstra para que contabilice, para cada vértice  $v \in V$ , el número de caminos de distancia mínima que existen desde el origen a  $v$ .

.....

## 9.7. El camino más corto de un vértice a todos los demás

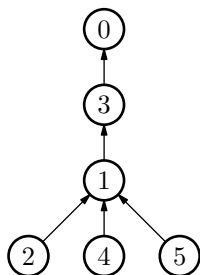
Cualquiera de los algoritmos considerados (recorrido en orden topológico para grafos acíclicos, Bellman-Ford para grafos con ciclos no negativos y Dijkstra para grafos ponderados positivos) construye una estructura de punteros hacia atrás cuando se propone recuperar el camino más corto, y no sólo obtener su distancia. La estructura *backp* almacena un «puntero» para cada vértice. Dicho puntero señala al vértice del que proviene, directamente, el camino más corto de  $s$  a  $t$ .

En la figura 9.24 se muestra a la izquierda un grafo acíclico ponderado y, a su derecha, una representación gráfica de la estructura *backp* cuando calculamos el camino más corto de 0 a 5.



**Figura 9.24:** (a) Grafo acíclico y ponderado. (b) Estructura de punteros hacia atrás.

La estructura *backp* codifica un árbol dirigido en el que vértice apunta a su padre. La figura 9.25 muestra una representación gráfica alternativa para la figura 9.24 (b) que pone de manifiesto la estructura de árbol definida por *backp*:



**Figura 9.25:** Árbol de caminos más cortos con destino en el vértice 5.

Este tipo de árbol recibe el nombre de **árbol de caminos más cortos**, pues con él resulta sencillo calcular el camino más corto de  $s$  a cualquier vértice  $v$  de  $V$ : basta con recorrer los punteros hacia atrás desde  $v$  hasta  $s$  e invertir la secuencia de vértices visitados.

Nótese que los árboles de caminos más cortos son árboles de recubrimiento del grafo.

**Árbol de caminos más cortos:** *Shortest paths tree.*

#### PROBLEMAS

► **197** El MST es el árbol de recubrimiento cuya suma de pesos de aristas es mínima. ¿Es, además, el árbol de caminos más cortos para algún vértice?

Presentamos ahora versiones de los tres algoritmos de cálculo del camino más corto modificados para devolver el árbol de caminos más cortos con origen en un vértice cualquiera  $s$ . La complejidad computacional de cada uno de ellos es la propia del algoritmo en el que se basa:  $O(|E|)$  para grafos acíclicos,  $O(|E||V|)$  para grafos con ciclos no negativos y  $O(|E| \lg |V|)$  para el algoritmo de Dijkstra.

### Árbol de caminos más cortos en grafos acíclicos

shortest\_path.py


shortest\_path.py

```

47 def dag_shortest_paths_tree(G, d, s, infinity=3.4e+38):
48     D = {}
49     backp = {}
50     for v in topsort(G):
51         if v == s:
52             D[v], backp[v] = 0, None
53         elif G.preds(v):
54             D[v], backp[v] = min([ (D[u] + d(u,v), u) for u in G.preds(v) ])
55         else:
56             D[v], backp[v] = infinity, None
57
58     return backp

```

La rutina *path\_from\_paths\_tree*, que presentamos a continuación, devuelve el camino más corto hasta *v* con la información del árbol de caminos más cortos:


 shortest\_path.py

shortest\_path.py

```

60 def path_from_paths_tree(backp, v):
61     path = [v]
62     while backp[path[-1]] != None:
63         path.append( backp[path[-1]] )
64     path.reverse()
65     return path

```

 test\_shortest\_path.c.py

test\_shortest\_path.c.py

```

1 from graph import Graph
2 from shortest_path import dag_shortest_paths_tree, path_from_paths_tree
3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,2), (1,4), (1,5), (2,4), (2,5),
5                           (3,1), (4,5)])
6
7 dist = {(0,1):3, (0,3):1, (1,2):2, (1,4):2, (1,5):2, (2,4): 3,
8         (2,5):1, (3,1):1, (4,5):2}
9
10 def d(u,v):
11     return dist[u,v]
12
13 tree = dag_shortest_paths_tree(G, d, 0)
14 for v in range(6):
15     print 'Camino más corto de 0 a %d:' % v, path_from_paths_tree(tree, v)

```

El resultado de ejecutar el programa es éste:

```


Camino más corto de 0 a 0: [0]
Camino más corto de 0 a 1: [0, 3, 1]
Camino más corto de 0 a 2: [0, 3, 1, 2]
Camino más corto de 0 a 3: [0, 3]
Camino más corto de 0 a 4: [0, 3, 1, 4]
Camino más corto de 0 a 5: [0, 3, 1, 5]

```

### Árbol de caminos más cortos en grafos ponderados sin ciclos negativos

Este es el único que merece algún comentario, pues la estructura *backp* contiene punteros hacia atrás calculados sobre el grafo multietapa asociado al grafo original.

Hemos de tener la precaución, pues, de devolver la etapa en la que acaba el camino más corto hasta cada vértice *v* y, al recuperar el camino, eliminar la información de etapas asociada a cada vértice.

 bellman\_ford.py

bellman\_ford.py

```

54 def shortest_paths_tree(G, d, s, infinity=3.4e+38):
55     D = {}
56     backp = {}
57
58     for v in G.V:
59         D[v, 0], backp[s, 0] = infinity, None
60     D[s, 0], backp[s, 0] = 0, None
61
62     for i in range(1, len(G.V)):
63         for v in G.V:
64             if len(G.preds(v)) > 0:
65                 D[v, i], backp[v, i] = min([(D[u, i-1] + d(u,v), u) for u in G.preds(v)])
66             else:


```

```

67      $D[v, i], \text{backp}[v, i] = \text{infinity}, \text{None}$ 
68
69      $\text{stage} = \{\}$ 
70     for  $v$  in  $G.V$ :
71          $\text{stage}[v] = 0$ 
72          $\text{mindist} = D[v, 0]$ 
73         for  $i$  in  $\text{range}(\text{len}(G.V))$ :
74             if  $[D[v, i] < \text{mindist}]$ :
75                  $\text{stage}[v] = i$ 
76                  $\text{mindist} = D[v, i]$ 
77
78     return  $\text{stage}, \text{backp}$ 
79
80 def  $\text{path\_from\_paths\_tree}(\text{stage}, \text{backp}, v)$ :
81      $k = \text{stage}[v]$ 
82      $\text{path} = [v]$ 
83     while  $\text{backp}[\text{path}[-1], k] \neq \text{None}$ :
84          $\text{path.append}(\text{backp}[\text{path}[-1], k])$ 
85          $k -= 1$ 
86      $\text{path.reverse}()$ 
87
88     return  $\text{path}$ 

```

### Árbol de caminos más cortos en grafos ponderados positivos

 `dijkstra.py` `dijkstra.py`

```

53 def  $\text{dijkstra\_shortest\_paths\_tree}(G, d, s, \text{infinity}=3.4\text{e}+38)$ :
54      $S = \text{Set}()$ 
55      $D = \{\}$ 
56      $\text{backp} = \{\}$ 
57     for  $v$  in  $G.V$ :  $D[v], \text{backp}[v] = \text{infinity}, \text{None}$ 
58      $D[s] = 0$ 
59      $\text{backp}[s] = \text{None}$ 
60
61      $Q = \text{IndexedMinHeap}(\text{len}(G.V), [(v, D[v]) \text{ for } v \text{ in } G.V])$ 
62
63     while not  $Q.\text{is\_empty}()$ :
64          $u, \text{aux} = Q.\text{extract\_min}()$ 
65         for  $v$  in  $G.\text{succs}(u)$ :
66             if  $v$  not in  $S$ :
67                 if  $D[u] + d(u, v) < D[v]$ :
68                      $D[v] = D[u] + d(u, v)$ 
69                      $Q.\text{decrease}(v, D[v])$ 
70                      $\text{backp}[v] = u$ 
71
72     return  $\text{backp}$ 

```

#### .....PROBLEMAS.....

► **198** Calcula el árbol de caminos más cortos que parten de Castellón de la Plana y llegan a cualquier de las ciudades de la península ibérica (ver ejercicio 178) si sólo permitimos desplazamientos hacia la izquierda. Representa gráficamente el árbol de caminos más cortos sobre el propio mapa, sin modificar la ubicación geográfica de las ciudades.

## 9.8. El problema de la distancia del camino más corto entre todo par de vértices: el algoritmo de Floyd

Podemos calcular la distancia del camino más corto entre todo par de vértices  $u$  y  $v$  aplicando  $|V|$  veces uno de los tres algoritmos anteriores, según el grafo sea acíclico, sin ciclos negativos o sin pesos negativos. El algoritmo de Floyd, que mostramos a continuación, efectúa el cálculo directamente.

```
floyd.py floyd.py
1 def floyd(G, d, infinity=3.4e+38):
2     D = {}
3     for u in G.V:
4         for v in G.V:
5             if u == v:
6                 D[u,v] = 0
7             elif v in G.succs(u):
8                 D[u,v] = d(u,v)
9             else:
10                D[u,v] = infinity
11
12    for v in G.V:
13        for u in G.V:
14            for w in G.V:
15                D[u,w] = min(D[u,w], D[u,v] + D[v,w])
16
17    return D
```

Como puede apreciarse, el algoritmo de Floyd está directamente inspirado en el algoritmo de Warshall para el cálculo de la clausura transitiva de un grafo (véase la sección 9.4). La corrección del cálculo del algoritmo de Floyd es análoga a la del algoritmo de Warshall.

Probemos el algoritmo sobre el grafo con el mapa de carreteras de la isla de Mallorca para obtener una tabla de distancias mínimas entre cualquier de pares de ciudades de la isla.

```
test.floyd.py test_floyd.py
1 from graph import Graph
2 from floyd import floyd
3 from mallorca import Mallorca, d
4
5 mindist = floyd(Mallorca, d)
6
7 print 'UUUU',
8 for u in Mallorca.V:
9     print '%4s' % u[:4],
10 print
11
12 for u in Mallorca.V:
13     print '%4s' % u[:4],
14     for v in Mallorca.V:
15         print '%4d' % mindist[u,v],
16     print
```

	Alcú	Andr	Artà	Calv	Camp	Capd	Inca	Lluc	Mana	Marr	Palm	Poll	Sant	Sóll
Alcú	0	79	36	65	85	44	25	71	50	37	51	10	77	64
Andr	79	0	96	14	62	104	54	48	79	42	28	89	75	56
Artà	36	96	0	82	57	8	42	71	17	54	68	46	44	100
Calv	65	14	82	0	48	90	40	34	65	28	14	75	61	70
Camp	85	62	57	48	0	65	60	14	40	48	34	95	13	118
Capd	44	104	8	90	65	0	50	79	25	62	76	54	52	108



Inca	25	54	42	40	60	50	0	46	25	12	26	35	52	89
Lluc	71	48	71	34	14	79	46	0	54	34	20	81	27	104
Mana	50	79	17	65	40	25	25	54	0	37	51	60	27	114
Marr	37	42	54	28	48	62	12	34	37	0	14	47	61	98
Palm	51	28	68	14	34	76	26	20	51	14	0	61	47	84
Poll	10	89	46	75	95	54	35	81	60	47	61	0	87	54
Sant	77	75	44	61	13	52	52	27	27	61	47	87	0	131
Sóll	64	56	100	70	118	108	89	104	114	98	84	54	131	0

El coste temporal del algoritmo de Floyd es  $\Theta(|V|^3)$ .

Los siguientes ejercicios pueden resolverse a partir de la información devuelta por el algoritmo de Floyd si ponderamos todas las aristas del grafo con peso unitario.

#### PROBLEMAS.....

► **199** La **excentricidad** de un vértice es la mayor longitud (número de aristas) del camino más corto a cualquiera de sus vértices. El *radio* de un grafo  $G$  es la excentricidad más pequeña de cualquiera de sus vértices. Diseña una función que calcule el radio de un grafo  $G$ .

**Excentricidad:** *Eccentricity*.

► **200** El *diámetro* de un grafo  $G$  es la excentricidad más grande de cualquiera de sus vértices. Diseña una función que calcule el radio de un grafo  $G$ .

► **201** El *centro* de un grafo  $G$  es el conjunto de vértices de menor excentricidad. Diseña una función que calcule el centro de un grafo  $G$ .

► **202** El *perímetro* de un grafo  $G$  es la longitud (número de aristas) de su ciclo más corto. Diseña una función que calcule el girth de un grafo ponderado positivo  $G$ .

**Perímetro de un grafo:**  
*Graph girth*.

## 9.9. Otros problemas sobre grafos

No pretendemos dar una relación exhaustiva de problemas sobre grafos, aunque sí reseñar algunos de los más interesantes y que encuentran aplicación práctica en numerosos campos. Algunos de ellos reaparecerán en capítulos posteriores.

### 9.9.1. Problemas de flujo de red

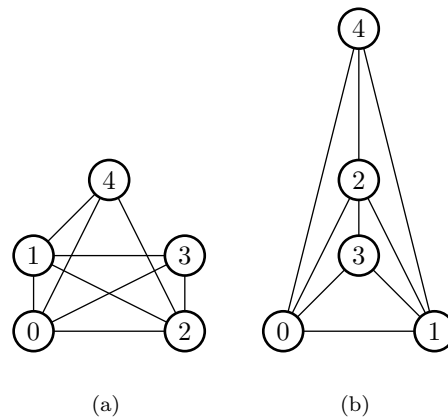
Una red de flujo es un grafo ponderado y dirigido en el que las aristas llevan asociado un valor de capacidad. Cada vértice es un punto de confluencia entre aristas entrantes y salientes. Si pensamos en que las aristas transportan un fluido (petróleo en un oleoducto o agua en un sistema de alcantarillado, aunque también encuentra aplicación en el tráfico de paquetes en una red de comunicaciones o en el suministro de componentes a fábricas), los vértices alcanzan una situación de equilibrio cuando entra en ellos tanto fluido como sale. El equilibrio representa una situación óptima en la que sale tanto fluido como puede entrar (naturalmente, por un vértice no puede pasar más fluido del que entra). Uno de los problemas consiste en determinar la cantidad máxima de fluido que puede transportar la red entre un par de vértices. Otro consiste en determinar el menor coste con el que es posible transportar el mayor flujo posible en un red cuyas aristas están ponderadas, además, por una función de coste.

Muchos problemas de programación lineal pueden reducirse a problemas de flujo de red y conducen así a algoritmos de resolución eficientes.

### 9.9.2. Determinación de la planaridad de un grafo

Un grafo es **planar** si es posible dibujarlo en el plano sin que dos aristas se crucen. El problema de determinar si un grafo es planar se conoce por **problema de la planaridad** puede resolverse en tiempo lineal. Nótese que una cosa es determinar si un grafo es o no es planar y otra es encontrar una representación sin aristas cruzadas. La figura 9.26 muestra un grafo planar y una disposición de sus vértices que evita cruces de aristas.

**Problema de la planaridad:**  
*Planarity problem*.

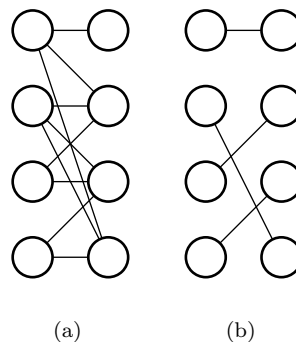


**Figura 9.26:** (a) Grafo planar. (b) Disposición de sus vértices que hace patente que es planar.

### 9.9.3. El problema de la correspondencia

**Correspondencia:** *Matching.*

El problema de la **correspondencia** propone el cálculo del subconjunto de aristas más grande tal que no hay dos aristas conectadas a un mismo vértice. La correspondencia se dice perfecta si comprende a todos los vértices. La figura 9.27 muestra (a) un grafo y (b) una posible correspondencia perfecta.



**Figura 9.27:** (a) Grafo no dirigido. (b) Correspondencia.

El problema puede resolverse en tiempo polinómico con  $|V|$  y  $|E|$ , pero el mejor algoritmo conocido resulta excesivamente costoso para grafos de talla grande.

**Correspondencia bipartita:**  
*Bipartite matching.*

Un caso particular de este problema es el problema de la **correspondencia bipartita**. En él, el grafo tiene dos tipos de vértices (por ejemplo, palabras y definiciones), las aristas conectan vértices de tipos distintos (definiendo, por ejemplo, relaciones de significado válido para cada palabra) y se desea poner en correspondencia el mayor número de vértices de un tipo con vértices del otro. El grafo de la figura 9.27 (a) es bipartito.

El problema de la **asignación** es una variante de este problema para grafos ponderados. Se propone encontrar la correspondencia cuya suma de pesos de aristas es mínima.

Este problema aparece cuando buscamos, por ejemplo, parejas idóneas en una agencia matrimonial, si queremos emparejar jugadores de nivel semejante al organizar un torneo, si queremos asignar empleados a puestos de trabajo...

### 9.9.4. Búsqueda del ciclo o camino hamiltoniano de coste mínimo

Dado un grafo  $G$  se desea calcular un camino que visite todos y cada uno de los vértices exactamente una vez y tal que la suma de pesos de sus aristas sea mínima. Este problema

se conoce en la literatura como **problema del viajante de comercio**.

Aparte de su evidente interés en la organización de rutas de reparto o comercio, este problema surge en la organización de rutas de montaje en sistemas de fabricación con robots u operarios que trabajan en serie o en la determinación de los movimientos del cabezal de un plotter que permite realizar un dibujo en el menor tiempo posible.

Es un problema para el que no se conoce algoritmo capaz de resolverlo en tiempo polinómico. Se recurre muchas veces a aproximaciones que producen caminos hamiltonianos de coste próximo al de menor coste.

Un problema relacionado es el que demanda el cálculo de un ciclo hamiltoniano.

**Problema del viajante de comercio:** *Traveling salesman problem.*

### 9.9.5. Búsqueda del ciclo o camino euleriano de coste mínimo

El problema de encontrar un ciclo euleriano, es decir, un camino que visite todas las aristas una sola vez. Un problema relacionado es el cálculo de un camino euleriano, igual que el anterior pero sin la exigencia de empezar y acabar en el mismo vértice.

Determinar si un grafo contiene o no un ciclo o un camino euleriano es sencillo. Hay algunas propiedades de los grafos que determinan la existencia de un ciclo o un camino euleriano:

- Un grafo no dirigido contiene un ciclo euleriano si y sólo si es conexo y todo vértice tiene grado 2.
- Un grafo no dirigido contiene un camino euleriano si y sólo si es conexo y todos los vértices, excepto dos, tienen grado 2.
- Un grafo dirigido contiene un ciclo euleriano si y sólo si es conexo y todos los vértices tienen el mismo grado de entrada que de salida.
- Un grafo dirigido contiene un camino euleriano si y sólo si es conexo y todos los vértices, excepto dos, tienen el mismo grado de entrada que de salida y los dos vértices especiales tienen grados de entrada y salida que difieren en una unidad.

Un problema relacionado demanda el cálculo de un ciclo o camino que pase por cada arista al menos una vez. El problema surge, por ejemplo, al asignar rutas a agentes (personas, camiones, etc.) de reparto de paquetería y correo o al determinar la ruta de los camiones de basura

El problema conocido como **problema del cartero chino** propone el cálculo del ciclo con menor número de aristas que visita todas y cada una de las aristas al menos una vez.

Hay pasatiempos infantiles que no son más que instancias de este problema: aquellos en que nos piden que tracemos una figura sin levantar el papel del lápiz, sin trazar dos veces la misma línea y finalizando en el punto de partida.

**Problema del cartero chino:** *Chinese postman problem.*

Es el problema del cartero *chino* porque fue planteado por un autor chino: M. Kwan.

### 9.9.6. Problemas de conectividad

Se desea conocer el menor número de aristas que, eliminadas, convierten un grafo conexo en dos componentes conexas disjuntas. Existe una formulación alternativa referida al número de vértices.

Este tipo de problemas tienen por objeto determinar la fiabilidad de una red de comunicaciones: ¿cuántos nodos o enlaces de la red han de caer para que sea imposible comunicarse entre dos puntos de la misma?

### 9.9.7. Problema del camino más largo

Dado un grafo ponderado  $G$ , encontrar el camino simple (sin ciclos) de mayor distancia recorrida entre dos vértices  $s$  y  $t$ . Aunque parezca un problema análogo al del camino más corto, es mucho más difícil.

### 9.9.8. Problema de la coloración

¿Cuál es el menor número de colores con el que es posible «pintar» los vértices de un grafo de modo tal que no haya dos vértices adyacentes del mismo color? Este problema

abstracto aparece en problemas de asignación de tareas interdependientes a procesadores o de valores a registros en la optimización de código de un compilador.

El menor número de colores con el que se pueden colorear los vértices un grafo es si **número cromático**. Calcular el color cromático es un problema difícil (requiere tiempo exponencial), así que también lo es el problema del coloreado.

Un problema relacionado propone el colorear de las aristas de modo que dos aristas con el mismo color no compartan un vértice. Si, por ejemplo, hemos de planificar una serie de entrevistas de idéntica duración, podemos tratar de colorear las aristas que representan actos de entrevistas o partidos de fútbol entre dos personas o equipos (dos vértices) con el menor número posible de colores. Todas las entrevistas o partidos con el mismo color pueden realizarse simultáneamente.

El menor número de colores necesario para colorear las aristas es el **número cromático de aristas** o **índice cromático**.

### 9.9.9. Cálculo de clique más grande

¿Cuál es el subconjunto de vértices de mayor talla tal que induce un grafo completo? Un subgrafo completo es un clique. El cálculo de cliques permite detectar grupos fuertemente relacionados, como grupos de investigadores que se citan entre sí en sus publicaciones científicas o grupos de alumnos que han copiado un examen (¡o estudiado conjuntamente!).

Se trata de un problema difícil para el que no hay algoritmos eficientes (es decir, con tiempo de ejecución polinómico).

### 9.9.10. Conjuntos independientes

¿Cuál es el conjunto de vértices más grande tal que no hay dos unidos entre sí por una arista? Este problema encuentra aplicación cuando queremos asignar una serie de recursos (farmacias, franquicias, etc.) a ciudades o calles sin que haya dos asignados a ciudades o calles vecinas.

Se trata, también, de un problema difícil.

### 9.9.11. Isomorfismo de grafos

Dados dos grafos, ¿son iguales si renombramos sus vértices? Este problema encuentra aplicación, por ejemplo, en problemas de visión artificial. Podemos representar una imagen con un grafo en el que las esquinas son vértices y las líneas que los unen son grafos. Mediante la detección de isomorfismos podemos decidir si dos imágenes representan la misma escena.

### 9.9.12. Dibujo de grafos

Encontrar una representación estéticamente aceptable de un grafo es un problema de gran interés. Existen infinidad de técnicas, muchas de las cuales se plantean en términos de la minimización del número de cruces de aristas, o del área ocupada, o de la longitud física de las aristas, etc.

Un caso particular de este problema es la representación de árboles.