

# 15

## GRÁFICOS

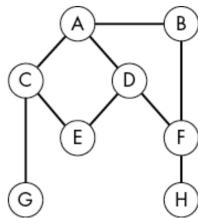


Los grafos son una de las estructuras de datos fundamentales en informática. Surgen en numerosos problemas y tareas de programación. A diferencia de otras estructuras de datos de este libro, diseñadas para optimizar ciertos cálculos, la estructura de los grafos surge naturalmente de los propios datos. En otras palabras, los grafos reflejan los datos que representan. Examinar los algoritmos de grafos nos permite comprender cómo podemos definir algoritmos que aprovechen la estructura inherente de los datos.

Los capítulos anteriores se centraron en el problema de estructurar los datos para facilitar los algoritmos. Problemas de alto nivel, como la búsqueda de un valor, motivaron e impulsaron el diseño de las estructuras de datos facilitadoras. Este capítulo aborda el problema opuesto: los grafos nos muestran cómo la estructura de los datos puede impulsar el desarrollo de nuevos algoritmos. En otras palabras, dados los datos en forma de grafo, examinamos cómo crear algoritmos que los utilicen. Este capítulo examina tres algoritmos de grafos que utilizan diferentes aspectos de la estructura del grafo: el algoritmo de Dijkstra para caminos más cortos, el algoritmo de Prim para árboles de expansión de mínimo coste y el algoritmo de Kahn para ordenamiento topológico.

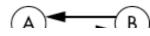
### Introducción a los gráficos

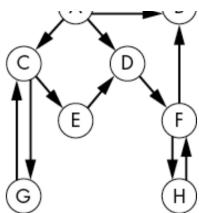
Los grafos se componen de un conjunto de *nodos* y un conjunto de *aristas*. Como se muestra en [la Figura 15-1](#), cada arista conecta un par de nodos. Esta estructura es similar a la de un gran número de sistemas del mundo real, incluyendo redes sociales (los nodos son personas y las aristas son sus conexiones), redes de transporte (los nodos son ciudades y las aristas representan caminos) y redes informáticas (los nodos son computadoras y las aristas representan las conexiones entre ellas). Esta variedad de analogías del mundo real hace que los algoritmos de grafos sean divertidos de visualizar, ya que las búsquedas simples se transforman en exploraciones minuciosas de castillos o carreras frenéticas por los concurridos callejones de una ciudad.



[Figura 15-1](#) : Un gráfico con aristas no dirigidas

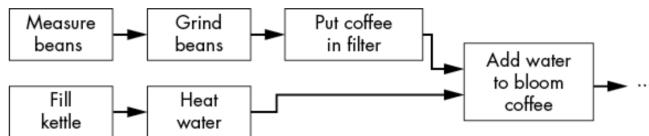
Las aristas de un grafo pueden tener propiedades adicionales para capturar las complejidades del mundo real de los datos, como si las aristas son direccionales o no. Las *aristas no dirigidas*, como las del grafo de [la Figura 15-1](#), representan relaciones bidireccionales, como la mayoría de las carreteras y las amistades felices. Las *aristas dirigidas*, como se ilustra en [la Figura 15-2](#), son como calles de un solo sentido e indican un flujo en una sola dirección. Para representar el acceso no dirigido, utilizamos un par de aristas dirigidas (una en cada dirección) entre los nodos. En un contexto social, las aristas dirigidas podrían representar un interés romántico en un drama adolescente de televisión: una arista de Alice hacia Bob indica que a Alice le gusta Bob, mientras que la ausencia de una arista de Bob hacia Alice ilustra la devastadora falta de reciprocidad.





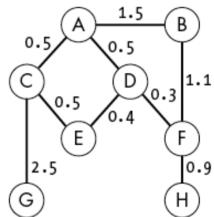
[Figura 15-2](#) : Un gráfico con aristas dirigidas

Además de permitirnos modelar calles de un solo sentido o amores no correspondidos, las aristas dirigidas nos permiten modelar problemas más abstractos, como la dependencia de tareas. Podemos especificar un conjunto de tareas como nodos y usar aristas dirigidas para indicar la dependencia del orden entre tareas. De esta manera, podríamos crear un gráfico para representar las tareas requeridas para preparar la taza de café perfecta, como se muestra en [la Figura 15-3](#). Los nodos incluyen pasos como calentar el agua, medir los granos, moler los granos y agregar agua a los posos. Las aristas representan dependencias entre estos pasos. Necesitamos agregar una arista dirigida desde el nodo para "moler granos" al nodo para "poner los posos en el filtro" para indicar que debemos moler los granos primero. El orden de estos dos pasos es crítico, como cualquiera que haya intentado preparar granos sin moler puede atestiguar. Sin embargo, no necesitaríamos una arista entre "calentar el agua" y "moler los granos" en ninguna dirección. Podemos realizar esas tareas en paralelo.



[Figura 15-3](#) : Uso de un gráfico para representar el orden de operaciones para una tarea

Las ponderaciones de los bordes aumentan aún más la capacidad de modelado de los grafos. *Los bordes ponderados* capturan no solo el vínculo entre nodos, sino también su coste. Por ejemplo, podríamos ponderar los bordes en un grafo de transporte según la distancia entre ubicaciones. Podríamos ampliar nuestra red social con una medida de cercanía, como el número de veces que dos nodos han hablado en el último mes. [La Figura 15-4](#) muestra nuestro grafo de ejemplo con bordes ponderados.



[Figura 15-4](#) : Un gráfico con bordes ponderados

El uso de una combinación de aristas ponderadas y dirigidas nos permite capturar interrelaciones complejas entre los nodos. Dramas sociales completos pueden representarse y desarrollarse a través de los nodos y aristas de un grafo bien construido.

### Representación de gráficos

Si bien la estructura abstracta de un grafo es relativamente simple, existen múltiples maneras de representar nodos y aristas en la memoria de la computadora. Dos de las representaciones más comunes son *las matrices de adyacencia* y *las listas de adyacencia*. Ambas representaciones pueden manejar nodos dirigidos, no dirigidos, ponderados y... Aristas no ponderadas. Al igual que con todas las demás estructuras de datos de este libro, la diferencia entre estas estructuras radica en cómo se almacenan los datos en memoria y, por lo tanto, cómo los diferentes algoritmos pueden acceder a ellos.

para cada nodo. Podríamos usar una matriz o una lista enlazada de vecinos dentro de la estructura de datos compuesta del nodo:

```
Node {
    String: name
    Array of Nodes: neighbors
}
```

O incluso podríamos crear una estructura de datos de aristas independiente para almacenar información auxiliar sobre las aristas, como su direccionalidad o pesos. Para los ejemplos siguientes, también proporcionamos un único ID numérico para cada nodo, correspondiente al índice del nodo en la estructura de datos del grafo principal:

```
Edge {
    Integer: to_node
    Integer: from_node
    Float: weight
}

Node {
    String: name
    Integer: id
    Array of Edges: edges
}
```

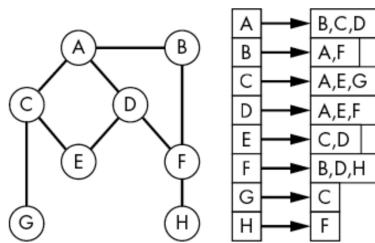
En cualquier caso, el gráfico en sí contendría una matriz de nodos:

```
Graph {
    Integer: num_nodes
    Array of Nodes: nodes
}
```

Independientemente de la implementación exacta, podemos acceder a los vecinos de cualquier nodo mediante una lista enlazada desde el propio nodo. [La Figura 15-5](#) muestra un ejemplo de esta estructura.

En el caso de aristas dirigidas, la lista de aristas o nodos vecinos de un nodo contiene solo aquellas a las que se puede acceder al *salir* del nodo. Por ejemplo, el nodo A puede tener una arista hacia el nodo B, mientras que el nodo B no la tiene hacia el nodo A.

Las listas de adyacencia proporcionan una visión localizada de las relaciones vecinales que refleja casos reales como las redes sociales. Cada nodo rastrea únicamente el nodo con el que tiene conexiones. De forma similar, en una red social, cada persona determina quiénes son sus amigos, manteniendo así una lista de sus propias conexiones. No necesitamos un repositorio central independiente que nos indique quiénes son nuestros amigos, y es posible que no tengamos una visión completa de los amigos de los demás. Podría decirse que ni siquiera sabemos cuáles de nuestros amigos (el lado saliente) realmente nos consideran amigos (el lado entrante). Solo conocemos nuestras propias conexiones salientes.



[Figura 15-5](#) : Un grafo (izquierda) y su representación en lista de adyacencia (derecha). Cada nodo almacena una lista de nodos vecinos.

En cambio, una matriz de adyacencia representa un grafo como una matriz, como se muestra en [la Figura 15-6](#), con una fila y una columna para cada nodo. El valor en la fila  $i$ , columna  $j$  representa el peso de la arista del nodo  $i$  al nodo  $j$ . Un valor de cero indica que no existe dicha arista. Esta representación permite consultar

directamente si existe una arista entre dos nodos cualesquiera desde una única fuente de datos central.

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	0	0	0	1	0	0
C	1	0	0	0	1	0	1	0
D	1	0	0	0	1	1	0	0
E	0	0	1	1	0	0	0	0
F	0	1	0	1	0	0	0	1
G	0	0	1	0	0	0	0	0
H	0	0	0	0	1	0	0	0

*Figura 15-6 : Representación de la matriz de adyacencia de un gráfico*

Esta visión global del gráfico surge en situaciones reales donde un solo planificador visualiza toda la red. Por ejemplo, una aerolínea puede usar una visión global de las rutas de vuelo, donde los nodos son aeropuertos y los bordes indican los vuelos entre ellos, para planificar un nuevo servicio.

Si bien la representación del grafo de adyacencia es útil en algunos casos, en el resto de este capítulo nos centraremos en la representación de la lista de adyacencia. Esta representación se adapta perfectamente al enfoque basado en punteros que hemos utilizado para otras estructuras de datos. Además, el uso de estructuras de datos de nodos individuales ofrece mayor flexibilidad para el almacenamiento de datos auxiliares.

### Búsqueda de gráficos

Si volvemos a nuestro ejemplo de rastreo web del Capítulo 4 , donde exploramos nuestra enciclopedia en línea favorita para obtener información relacionada con el café.Molinillos, podemos ver inmediatamente cómo los enlaces en nuestra enciclopedia en línea favorita forman un grafo de temas, donde cada página representa un nodo y cada hipervínculo representa una arista dirigida. Podemos explorar temas progresivamente, profundizando cada vez más en el mundo de los molinillos de café, explorando iterativamente cada nodo y añadiendo nuevos nodos a nuestra lista de temas para explorar en el futuro. Este tipo de exploración constituye la base de una búsqueda gráfica.

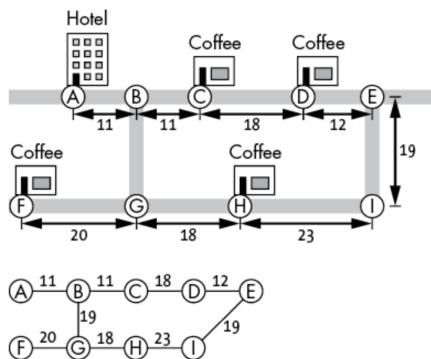
Imaginemos que nos interesa encontrar un nodo específico en el grafo. Quizás estamos realizando una investigación en línea y buscando una marca de café cuyo nombre hemos olvidado hace mucho tiempo. Exploramos las páginas web relevantes (nodos del grafo) una a la vez, leyendo la información de una página antes de pasar a otra. Como vimos en el Capítulo 4 , el orden en que exploramos los nodos influye enormemente en nuestro patrón de búsqueda. Al usar una estructura de datos de pila para rastrear nuestras futuras opciones de exploración, realizamos una búsqueda en profundidad sobre el grafo. Seguimos rutas individuales cada vez más profundamente hasta llegar a un callejón sin salida. Luego retrocedemos y probamos otras opciones que omitimos en el camino. Si, en cambio, usamos una cola para rastrear nuestros futuros estados de búsqueda, realizamos una búsqueda en amplitud sobre los nodos. Comprobamos los nodos más cercanos a nuestra ubicación inicial antes de aventurarnos más y más en el grafo. Por supuesto, hay otras formas de ordenar nuestra búsqueda. Por ejemplo, la búsqueda "mejor primero" ordena los futuros nodos según una función de clasificación, centrándose en explorar primero los nodos con mayor puntuación. En nuestra búsqueda de cafeterías cercanas en una nueva ciudad, esta priorización de nodos puede evitar que perdamos horas deambulando por barrios residenciales en lugar de centrarnos en zonas comerciales.

Independientemente del orden, el concepto de buscar en un grafo explorando un nodo a la vez ilustra el impacto de la estructura de los datos en el algoritmo. Utilizamos los enlaces entre nodos (aristas) para restringir y guiar la exploración. En las siguientes secciones, analizaremos algoritmos comunes y útiles que hacen precisamente esto.

### Encontrar caminos más cortos con el algoritmo de Dijkstra

Probablemente la tarea más común al trabajar con grafos del mundo real es encontrar la distancia más corta entre dos nodos. Imaginemos que visitamos una ciudad nueva por primera vez. Al amanecer, salimos a trompicones de la habitación del hotel, con jet lag, en busca de algo para refrescarnos. Como buenos viajeros, hemos investigado a fondo la oferta cafetalera de la ciudad y hemos creado una lista de cuatro cafeterías para probar durante nuestra estancia. Al llegar el ascensor al vestíbulo, sacamos un plano de la ciudad, cuidadosamente marcado con la ubicación del hotel y las cafeterías. Es hora de decidir cómo llegar a las tiendas de nuestra lista.

*El algoritmo de Dijkstra*, inventado por el informático Edsger W. Dijkstra, encuentra la ruta más corta desde cualquier nodo inicial hasta todos los demás nodos del grafo. Puede funcionar en grafos dirigidos, no dirigidos, ponderados o no ponderados. La única restricción es que todos los pesos de las aristas deben ser positivos. Nunca se puede reducir la longitud total de la ruta añadiendo una arista. En nuestro ejemplo de turismo con temática de café, buscamos la ruta más corta desde el hotel hasta cada una de las cafeterías. Como se muestra en [la Figura 15-7](#), los nodos representan intersecciones de calles o tiendas a lo largo de la calle. Las aristas ponderadas y no dirigidas representan la distancia a lo largo de las carreteras entre estos puntos.



*Figura 15-7*: Los puntos a lo largo de un mapa con sus distancias correspondientes (arriba) se pueden representar como un gráfico ponderado (abajo).

Nuestro objetivo es encontrar el camino más corto desde el nodo de inicio hasta cada uno de los nodos de cafetería. Los nodos de intersección no son objetivos en sí mismos, pero permiten que nuestro camino se bifurque en diferentes calles.

El algoritmo de Dijkstra funciona manteniendo un conjunto de nodos no visitados y actualizando continuamente la distancia *tentativa* a cada uno. En cada iteración, visitamos el nodo no visitado más cercano. Al hacerlo, eliminamos este nuevo nodo del conjunto y actualizamos las distancias a cada uno de sus vecinos no visitados. Específicamente, examinamos los vecinos del nuevo nodo y nos preguntamos si hemos encontrado una ruta mejor para cada vecino. Calculamos la longitud de la nueva ruta propuesta tomando la distancia al nodo actual y sumando la distancia (peso de la arista) al vecino. Si esta nueva distancia es menor que la mejor distancia observada hasta el momento, la actualizamos.

```

Dijkstras(Graph: G, Integer: from_node_index):
    ① Array: distance = inf for each node id in G
    Array: last = -1 for each node in G
    Set: unvisited = set of all node indices in G
    distance[from_node_index] = 0.0

    ② WHILE unvisited is NOT empty:
        ③ Integer: next_index = the node index in unvisited
                    with the minimal distance
        Node: current = G.nodes[next_index]
        Remove next_index from unvisited

        ④ FOR EACH edge IN current.edges:
            ⑤ Float: new_dist = distance[edge.from_node] +
                        edge.weight
            ⑥ IF new_dist < distance[edge.to_node]:
                distance[edge.to_node] = new_dist
                last[edge.to_node] = edge.from_node

```

El código comienza creando una serie de estructuras de datos auxiliares ❶, incluyendo una matriz de distancias a cada nodo (`distance`), una matriz que indica el último nodo visitado antes de un nodo dado (`last`), y un conjunto de nodos no visitados (`unvisited`). Luego, el código procesa los nodos no visitados uno por uno. Un `WHILE` bucle itera hasta que el conjunto de nodos no visitados está vacío ❷. En cada iteración, el código elige el nodo con la distancia mínima y lo elimina del conjunto de nodos no visitados ❸. Un `FOR` bucle itera sobre cada uno de los vecinos del nodo ❹, calculando la distancia a ese vecino a través del nodo actual ❺ y actualizando las matrices `distance` y `last` si el código ha encontrado una mejor ruta ❻.

**NOTA**

Aunque tanto el pseudocódigo como las ilustraciones utilizan una matriz de distancias explícita y un conjunto de no visitados a modo de ilustración, estos pueden combinarse en un min-heap para mayor eficiencia. El min-heap funciona como una cola de prioridad que siempre devuelve el elemento de menor prioridad. En este caso, almacena la lista de nodos no visitados, clasificada según su distancia actual. Encontrar el nodo más cercano consiste en devolver la clave mínima de la cola, y actualizar la mejor distancia hasta el momento corresponde a actualizar la prioridad de un nodo.

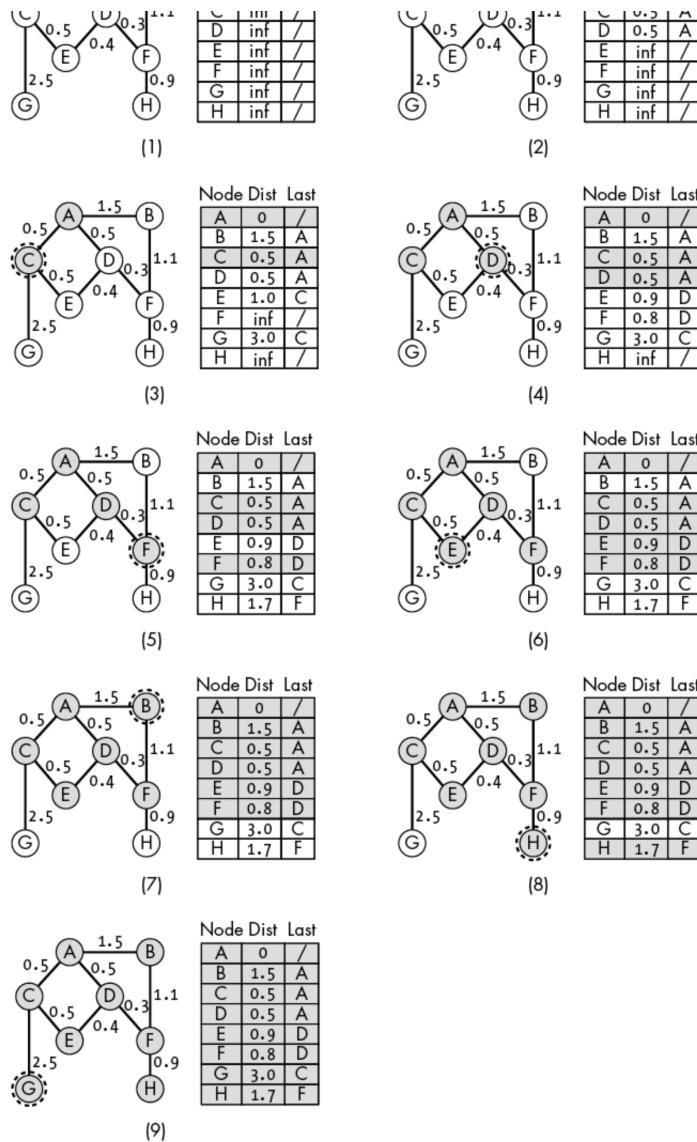
[La Figura 15-8](#) muestra un ejemplo de búsqueda de la ruta más corta desde el nodo A en el gráfico ponderado de [la Figura 15-4](#). El nodo encerrado en un círculo es el que se está examinando actualmente. Los nodos y las entradas de la lista en gris representan nodos que se han eliminado de la lista de no visitados y, por lo tanto, ya no están disponibles para su consideración.

Para la búsqueda en [la Figura 15-8](#), iniciamos el algoritmo de Dijkstra con todas las distancias en el infinito excepto para el nodo A, que se establece en cero ([Figura 15-8](#) (1)). Esta configuración inicial corresponde a nuestro conocimiento inicial sobre los mejores caminos. Ya estamos en el nodo A, por lo que el mejor camino allí es trivial. Dado que no hemos encontrado caminos a ninguno de los otros nodos, podrían estar a cualquier distancia. También mantenemos información para cada nodo de qué nodo lo precede en nuestra búsqueda. La `Last` columna indica el nodo precedente. Esta información nos permite rastrear caminos hacia atrás. Si bien no todos los usos necesitarán reconstruir el camino, nuestra búsqueda de café ciertamente lo hace. No tiene sentido encontrar la distancia más corta al café si no encontramos también el camino real. Para construir el camino al nodo F, seguimos los últimos punteros hacia atrás hasta que llegamos al nodo A.

Nuestra búsqueda comienza, como se muestra en [la Figura 15-8](#) (2), seleccionando el nodo con la menor distancia (nodo A), eliminándolo de la lista de no visitados y examinando sus vecinos. Para cada nodo vecino de A, comprobamos si el trayecto hasta él a través de A es más corto que cualquier ruta observada hasta el momento. Dado que la distancia al nodo A es cero, la distancia a través de A hasta cada uno de sus vecinos será igual a los pesos de arista correspondientes. Cada vez que actualizamos la distancia a un nodo no visitado, también actualizamos el puntero hacia atrás para reflejar la mejor ruta hasta el momento. Tres nodos apuntan ahora a A ([Figura 15-8](#) (2)).

La búsqueda avanza, seleccionando el siguiente nodo más cercano y no visitado. En este caso, podría ser C o D. Desempatamos según el orden de los nodos en nuestra lista: ¡el nodo C gana! Nuevamente, consideramos los vecinos de C y actualizamos sus mejores distancias ([Figura 15-8](#) (3)). Recuerde que las distancias representan la mejor distancia total desde nuestro nodo inicial. Las nuevas distancias son la suma de la distancia a C y la distancia de C a cada vecino.





*Figura 15-8* : Un ejemplo del algoritmo de Dijkstra en un gráfico ponderado

La búsqueda avanza al nodo D, el nuevo nodo no visitado con la distancia mínima (*Figura 15-8* (4)). Al examinar los vecinos del nodo D, encontramos nuevas distancias más cortas a ambos nodos E y F. El nodo E es particularmente interesante, ya que ya teníamos una ruta candidata a E a través de C. Podemos viajar de A a C a E con una distancia de 1.0. Sin embargo, esta no es la mejor ruta posible. Nuestra búsqueda reveló una nueva ruta, a través de D, que es ligeramente más corta con una distancia total de 0.9. Actualizamos tanto la distancia potencial como el puntero hacia atrás. Nuestra mejor ruta a E ahora pasa por D. ¡Continuemos hacia el siguiente nodo más cercano en nuestro conjunto no visitado, el nodo F!

La búsqueda continúa a través de los nodos restantes, pero no ocurre nada más interesante. Los nodos restantes se encuentran al final de los caminos más cortos y no ofrecen oportunidades para caminos más cortos. Por ejemplo, al considerar los vecinos del nodo E (*Figura 15-8* (6)), examinamos los nodos C y D. La distancia a cualquiera de los nodos al viajar a través de E sería 1.4, más larga que los caminos que ya hemos descubierto. De hecho, tanto C como D ya han sido visitados, por lo que ni siquiera los consideraríamos. Se aplica una lógica similar al considerar los nodos B, H y G, como se muestra en *las Figuras 15-8* (7), 15-8(8) y 15-8(9). Dado que todos los vecinos de esos nodos han sido visitados, no los consideramos.

Al examinar cómo el algoritmo de Dijkstra recorre un grafo mientras encuentra la ruta más corta, podemos observar la clara interrelación entre la estructura de los datos y el propio algoritmo. Los algoritmos de ruta más corta, como el de Dijkstra, solo son necesarios debido a la estructura del problema. Si pudiéramos saltar sin esfuerzo de un nodo a otro, no habría necesidad de encontrar una ruta a lo largo de los bordes. Esto es el equivalente en el mundo real a teletransportarse desde el

vestíbulo del hotel a la cafetería de destino: conveniente, pero no permitido por la estructura del mundo físico. Por lo tanto, al buscar estas rutas más cortas, debemos obedecer la estructura del propio grafo.

## Encontrar árboles de expansión mínimos con el algoritmo de Prim

El problema de encontrar el *árbol de expansión mínimo* de un grafo proporciona otro ejemplo de cómo la estructura de los datos del grafo nos permite plantear nuevas preguntas y, por lo tanto, crear nuevos algoritmos adecuados para responderlas. El árbol de expansión mínimo de un grafo no dirigido es el conjunto más pequeño de aristas tal que todos los nodos estén conectados (si es posible). Podemos pensar en estos árboles como en un urbanista con presupuesto limitado que intenta determinar qué carreteras pavimentar. ¿Cuál es el conjunto mínimo absoluto de carreteras necesario para garantizar que cualquier persona pueda ir de un lugar (nodo) a cualquier otro lugar (nodo) en una carretera pavimentada? Si las aristas están ponderadas, como por la distancia o el coste de pavimentar una carretera, extendemos el concepto para encontrar el conjunto que minimiza el peso total: el *árbol de expansión de coste mínimo* es el conjunto de aristas con el peso total mínimo que conecta todos los nodos.

Un método para encontrar el árbol de expansión mínimo es el *algoritmo de Prim*, propuesto independientemente por varias personas, entre ellas el informático RC Prim y el matemático Vojtěch Jarník. El algoritmo funciona de forma muy similar al algoritmo de Dijkstra de la sección anterior, trabajando con un conjunto no visitado y construyendo un árbol de expansión mínimo nodo a nodo. Empezamos con un conjunto de nodos no visitados y elegimos uno al azar. Este nodo visitado constituye el inicio de nuestro árbol de expansión mínimo. Luego, en cada iteración, encontramos el nodo no visitado con el menor peso de arista en comparación con *cualquiera* de los nodos visitados previamente. Nos preguntamos: "¿Qué nodo está más cerca de la periferia de nuestro conjunto y, por lo tanto, se puede añadir con el menor coste?". Eliminamos este nuevo nodo del conjunto no visitado y añadimos la arista correspondiente a nuestro árbol de expansión de coste mínimo. Seguimos añadiendo nodos y aristas, uno por iteración, hasta que se hayan visitado todos los nodos.

Podemos imaginar el algoritmo de Prim como una empresa constructora contratada para construir puentes entre las islas de un archipiélago. Los constructores empiezan en una sola isla y van conectando más y más islas. En cada paso, eligen la isla más cercana a las del conjunto conectado. Un extremo del puente se asienta sobre una isla del conjunto conectado y el otro sobre una isla fuera del conjunto conectado (incorporando así la nueva isla al conjunto conectado). Al empezar siempre nuevos puentes desde una isla del conjunto conectado, los constructores pueden mover su equipo a la isla inicial utilizando los puentes existentes. Y al terminar siempre los puentes en islas fuera del conjunto conectado, los constructores aumentan la cobertura del conjunto conectado en cada etapa.

Podemos simplificar el código del algoritmo rastreando información adicional. En cada paso, mantenemos una lista de las mejores aristas (incluido el peso) que hemos encontrado para cada nodo. Cada vez que eliminamos un nuevo nodo del conjunto de nodos no visitados, examinamos sus vecinos no visitados y comprobamos si existen aristas mejores (es decir, de menor coste) para alguno de ellos. De ser así, actualizamos la entrada del vecino en la lista con la nueva arista y el peso.

```
Prims(Graph G):
    ① Array: distance = inf for each node in G
    Array: last = -1 for each node in G
    Set: unvisited = set of all node indices in G
    Set: mst_edges = empty set

    ② WHILE unvisited is NOT empty:
        ③ Integer: next_id = the node index in unvisited with
            the minimal distance
        ④ IF last[next_id] != -1:
            Add the edge between last[next_id] and
            next id to mst edges
```

```

Remove next_id from unvisited

Node: current = G.nodes[next_id]
⑤ FOR EACH edge IN current.edges:
    IF edge.to_node is in unvisited:
        IF edge.weight < distance[edge.to_node]:
            distance[edge.to_node] = edge.weight
            last[edge.to_node] = current.id
return mst_edges

```

El código comienza creando una serie de estructuras de datos auxiliares ①, que incluyen un array de distancias a cada nodo (`distance`), un array que indica el último nodo visitado antes de un nodo dado (`last`), un conjunto de nodos no visitados (`unvisited`) y el conjunto final de aristas para el árbol de expansión mínimo (`mst_edges`). Al igual que con el algoritmo de Dijkstra, el pseudocódigo (y las figuras que analizaremos en breve) utiliza una combinación de listas y conjuntos a modo de ilustración. Podemos implementar el algoritmo de forma más eficiente almacenando los nodos no visitados en un min-heap codificado por la distancia. Por ahora, listaremos todos los valores para ilustrar explícitamente lo que sucede.

El código entonces procede como el algoritmo de Dijkstra, procesando los nodos no visitados uno a la vez. Un `WHILE` bucle itera hasta que el conjunto de nodos no visitados está vacío ②. Durante cada iteración, el nodo con la distancia mínima a cualquiera de los nodos visitados es elegido y eliminado del conjunto no visitado ③. El código comprueba si existe una arista entrante al nodo, lo cual es necesario porque el primer nodo visitado no tendrá una arista entrante ④, y agrega las aristas correspondientes al árbol de expansión mínimo. Después de agregar el nuevo nodo, un `FOR` bucle itera sobre cada uno de los vecinos del nodo ⑤, comprobando si el vecino no ha sido visitado y, de ser así, comprobando su distancia al nodo actual. En este caso, la distancia es simplemente el peso de la arista. El código termina devolviendo el conjunto de aristas que conforman el árbol de expansión mínimo.

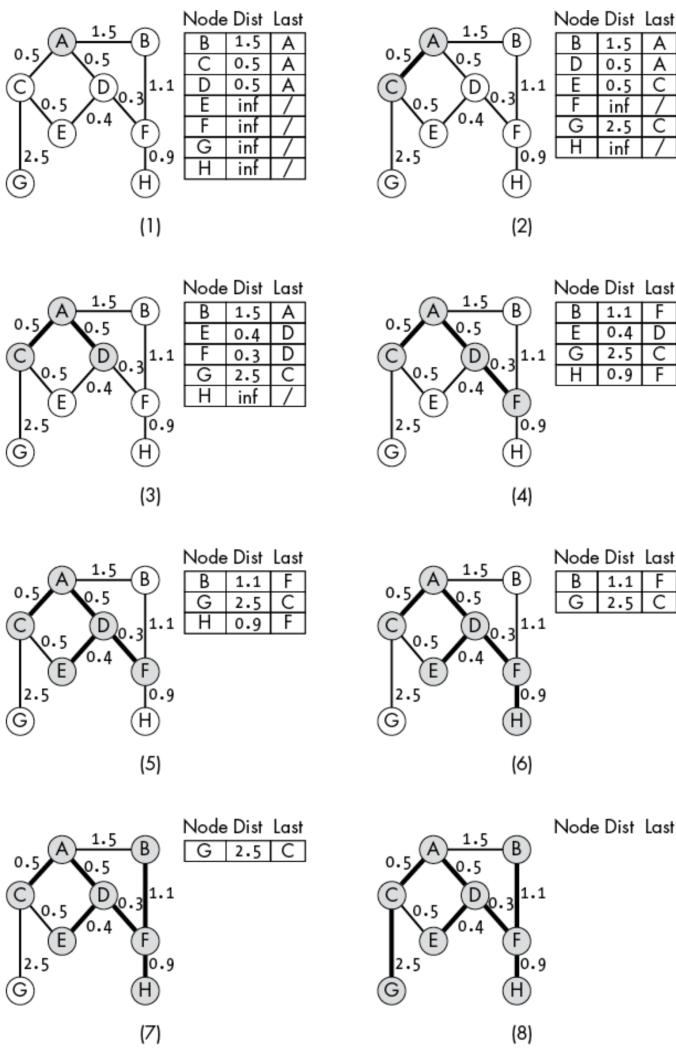
Consideré lo que sucede al ejecutar el algoritmo de Prim en el grafo ponderado de [la Figura 15-4](#), como se ilustra en [la Figura 15-9](#). Comenzamos con todas las últimas aristas establecidas como nulas (aún no hemos encontrado ninguna) y todas las "mejores" distancias hasta el infinito. Para simplificar, desempataremos los nodos en orden alfabético.

Para comenzar, eliminamos el primer nodo A de nuestro conjunto no visitado. Luego, consideramos todos los vecinos de A y comprobamos si existe una arista de menor coste entre A y ese vecino. Dado que todas nuestras mejores distancias actuales son infinitas, esto no es difícil. Encontramos aristas de menor coste para todos los vecinos de A: (A, B), (A, C) y (A, D). [La Figura 15-9](#) (1) muestra este nuevo estado.

Durante la segunda iteración, encontramos dos nodos potenciales en nuestro conjunto no visitado: C y D. Utilizando el orden alfabético para desempatar, seleccionamos C. Eliminamos C del conjunto no visitado y añadimos la arista (A, C) a nuestro árbol de expansión de mínimo coste. Al examinar los vecinos no visitados de C, encontramos mejores aristas candidatas para los nodos E y G ([Figura 15-9](#) (2)).

El siguiente nodo más cercano es D. Lo eliminamos de nuestro conjunto no visitado y añadimos la arista (A, D) al árbol de expansión de mínimo coste. Al examinar los vecinos no visitados de D, encontramos nuevas aristas de menor coste para ambos nodos E y F ([Figura 15-9](#) (3)). Nuestra mejor arista candidata para el nodo E ahora proviene del nodo D en lugar del nodo C.

El algoritmo avanza a través de los nodos restantes del conjunto no visitado. A continuación, visitamos el nodo F, añadiendo la arista (D, F), como se muestra en [la Figura 15-9](#) (4). Despues, como se muestra en [la Figura 15-9](#) (5), añadimos el nodo E y la arista (D, E). El algoritmo finaliza añadiendo los nodos H, B y G en ese orden. En cada paso, añadimos la mejor arista observada hasta el momento: (F, H), (F, B) y (C, G). Los tres pasos finales se muestran en [las Figuras 15-9](#) (6), [15-9](#) (7) en 15-9 (8) reseñativamente



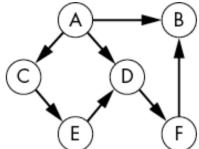
[Figura 15-9](#) : Un ejemplo del algoritmo de Prim en un gráfico ponderado

El algoritmo de Prim no considera la longitud total de las rutas desde nuestro nodo inicial. Solo nos interesa el coste de añadir el nuevo nodo a nuestro conjunto conectado: el peso de la arista que lo vinculará con cualquier otro nodo del conjunto visitado. No optimizamos los tiempos de recorrido final entre nodos, sino que buscamos minimizar el coste de pavimentar carreteras o construir nuevos puentes.

¿Qué pasaría si hubiéramos resuelto los empates aleatoriamente en lugar de por orden alfabetico? Al decidir entre elegir los nodos D o E de nuestro conjunto no visitado después de [la Figura 15-9](#) (2), podríamos haber usado cualquiera de los dos. Si hubiéramos elegido E en lugar de D, habríamos encontrado un peso de arista de menor costo que enlazara D con nuestro grafo. El algoritmo enlazaría el nodo D a través de E en lugar de A. Esto significa que podemos encontrar diferentes árboles de expansión de costo mínimo para el mismo grafo. Varios árboles diferentes pueden tener el mismo costo. El algoritmo de Prim solo garantiza que encontraremos uno de los árboles con el costo mínimo.

### Ordenamiento topológico con el algoritmo de Kahn

Nuestro último ejemplo de algoritmo de grafos utiliza las aristas de un *grafo acíclico dirigido (DAG)* para ordenar los nodos. Un grafo acíclico dirigido es un grafo con aristas dirigidas, dispuestas de tal manera que no contiene *ciclos* ni rutas que regresen al mismo nodo, como se muestra en [la Figura 15-10](#). Los ciclos son cruciales en las redes de carreteras del mundo real. Sería terrible que las carreteras se construyeran de tal manera que pudieramos ir de nuestro apartamento a nuestra cafetería favorita, pero nunca pudieramos regresar. Sin embargo, esto es exactamente lo que ocurre en un grafo acíclico: la ruta de salida de cualquier nodo nunca regresará a ese mismo nodo.



*Figura 15-10:* Un gráfico acíclico dirigido

Podemos usar aristas dirigidas para indicar el orden de los nodos. Si el grafo tiene una arista de A a B, el nodo A debe ir antes del nodo B. Ordenamos los nodos de esta manera en nuestro ejemplo de preparación de café al principio del capítulo: cada nodo representaba un paso del proceso y cada arista indicaba la dependencia de un paso con el siguiente. La persona que prepara el café debe realizar un paso determinado antes de poder realizar cualquiera de los pasos siguientes. Este tipo de dependencias surgen tanto en la informática como en el resto de la vida. Un algoritmo que ordena los nodos según sus aristas se denomina *ordenamiento topológico*.

El informático Arthur B. Kahn desarrolló un enfoque, ahora llamado *algoritmo de Kahn*, para realizar una ordenación topológica en un grafo acíclico dirigido que representa eventos. Este algoritmo funciona encontrando los nodos sin aristas entrantes, eliminándolos de nuestra lista de nodos pendientes, añadiéndolos a nuestra lista ordenada y, finalmente, eliminando las aristas salientes de ese nodo. El algoritmo se repite hasta que hayamos añadido todos los nodos a nuestra lista ordenada. Intuitivamente, esta ordenación refleja cómo podríamos realizar una tarea compleja en el mundo real. Comenzamos con una subtarea que podemos realizar (una sin dependencias). Realizamos esa subtarea y luego elegimos otra. Cualquier subtarea que requiera que hayamos realizado una tarea aún no completada debe esperar en nuestra lista hasta que hayamos completado todas sus dependencias.

Al implementar el algoritmo de Kahn, no es necesario eliminar aristas de nuestro grafo. Basta con mantener un array auxiliar que cuente el número de aristas entrantes a cada nodo y modifique dicho conteo.

```

Kahns(Graph G):
    ❶ Array: sorted = empty array to store result
    Array: count = 0 for each node in G
    Stack: next = empty stack for the next nodes to add

    # Count the incoming edges.
    ❷ FOR EACH node IN G.nodes:
        FOR EACH edge IN node.edges:
            count[edge.to_node] = count[edge.to_node] + 1

    # Find the initial nodes without incoming edges.
    ❸ FOR EACH node IN G.nodes:
        IF count[node.id] == 0:
            next.Push(node)

    # Iteratively process the remaining nodes without
    # incoming connections.
    ❹ WHILE NOT next.IsEmpty():
        Node: current = next.Pop()
        Append current to the end of sorted
        ❺ FOR EACH edge IN current.edges:
            count[edge.to_node] = count[edge.to_node] - 1
            ❻ IF count[edge.to_node] == 0:
                next.Push(G.nodes[edge.to_node])

    return sorted
  
```

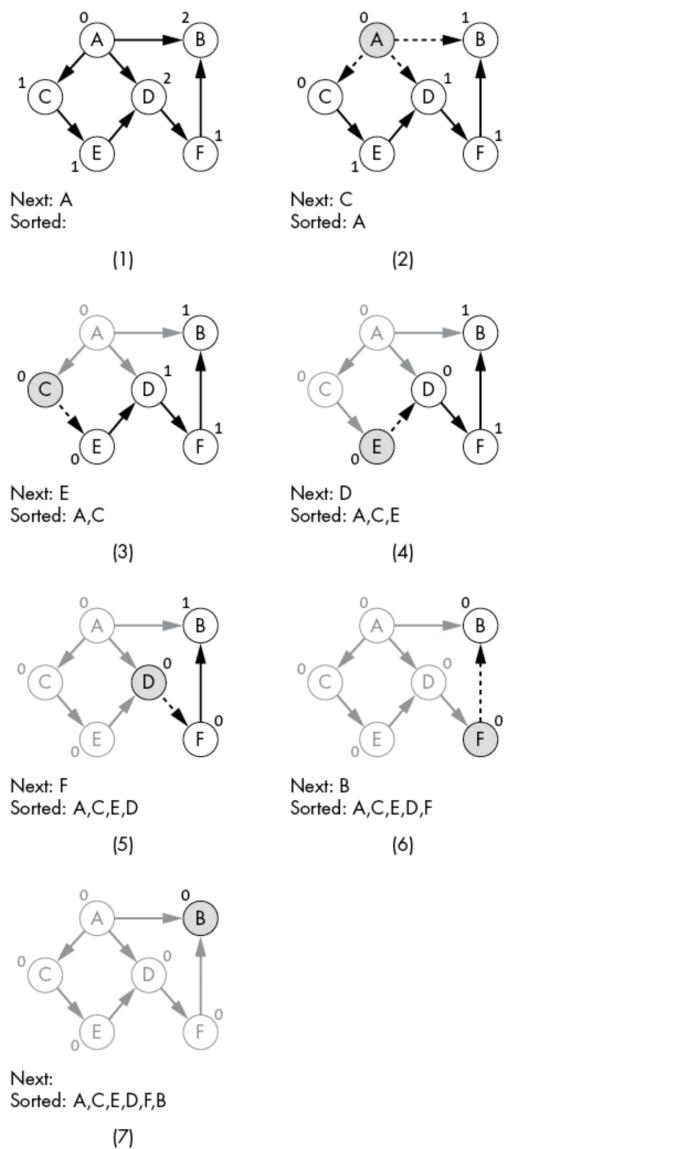
El código comienza creando varias estructuras de datos auxiliares ❶, incluyendo un array para almacenar la lista ordenada de nodos (`sorted`), un array que almacena el recuento de aristas entrantes de cada nodo (`count`) y una pila del siguiente nodo al que se añadirá `sorted` (`next`). El código utiliza un par de `FOR` bucles anidados sobre los nodos (bucle externo) y las aristas de cada nodo (bucle interno) para contar el número de aristas entrantes de cada nodo ❷. A continuación, un `FOR` bucle sobre el `count` array encuentra los nodos sin aristas en-

continuación, un bucle recorre la lista `next` y elimina los nodos que tienen entrantes y los inserta en `next`.

El código utiliza un `WHILE` bucle para procesar la `next` pila hasta que se vacía ④. Durante cada iteración, el código extrae un nodo de la pila y lo añade al final del `sorted` array. Un `FOR` bucle itera sobre las aristas del nodo y reduce el recuento (eliminando así la arista entrante) para cada vecino ⑤. Cualquier vecino con un recuento entrante de cero se añade a `next` ⑥. Finalmente, el código devuelve el array de nodos ordenados.

Si nuestro grafo contiene ciclos, la lista ordenada estará incompleta. Podemos añadir una comprobación adicional al final de la función para comprobar que el número de elementos en la lista ordenada es igual al número de nodos del grafo.

Consideré ejecutar este algoritmo en el gráfico de [la Figura 15-10](#), como se ilustra en [la Figura 15-11](#). Comenzamos contando el número de aristas entrantes (mos trado como el número adyacente a cada nodo) y determinandoEse nodo A es el único nodo sin aristas entrantes ([Figura 15-11](#) (1)). El algoritmo de Kahn luego agrega A a la lista ordenada y elimina sus aristas salientes (al disminuir los conteos correspondientes), como se muestra en [la Figura 15-11](#) (2).



[Figura 15-11](#): Una ordenación topológica en un gráfico acíclico dirigido

Continuamos el algoritmo en el nodo C ([Figura 15-11](#) (3)), que ya no presenta aristas entrantes. Eliminamos la única arista al procesar el nodo A. Eliminamos C de la lista de nodos considerados (nuestra pila `next`), eliminamos sus aristas del grafo y lo añadimos al final de nuestra lista ordenada. En el proceso, dejamos el nodo E sin vecinos entrantes. E pasa a nuestra pila.

La ordenación avanza por el resto de la lista. Mientras procesamos el nodo E, eliminamos las últimas aristas entrantes del nodo D, convirtiéndolo en el siguiente en el algoritmo ([Figura 15-11](#) (4)). La ordenación añade D, luego F y luego B a nuestra lista ordenada, como se muestra en [las Figuras 15-11](#) (5), [15-11](#) (6) y [15-11](#) (7), respectivamente.

El algoritmo de Kahn presenta un ejemplo de la utilidad de las aristas dirigidas en un grafo y de cómo podemos diseñar un algoritmo para operar con ellas. La direccionalidad de las aristas restringe aún más la exploración de los nodos.

### Por qué esto importa

Los grafos son omnipresentes en la informática. Su estructura les permite reflejar una gran variedad de fenómenos del mundo real, desde calles y redes sociales o informáticas hasta conjuntos de tareas complejas. Los grafos son útiles para tareas como la planificación de rutas y la determinación del orden de compilación del código fuente de un programa. Existe una gran cantidad de algoritmos diseñados para operar sobre estas estructuras de datos, realizando tareas como la búsqueda en el grafo, la determinación del árbol de expansión mínimo o el flujo máximo a través de un grafo. Podríamos dedicar un libro entero a esta singular estructura de datos de gran impacto.

Sin embargo, para los fines de este capítulo, nos centraremos en la estrecha relación entre la estructura de los datos y los algoritmos que operan con ellos. La estructura gráfica de los datos genera nuevos problemas, como la búsqueda del árbol de expansión mínimo, y, por consiguiente, nuevos algoritmos. A su vez, los algoritmos utilizan la estructura gráfica de los datos, recorriendo las aristas y explotando de nodo en nodo. Esta interacción demuestra la importancia de comprender la estructura de los datos al definir tanto problemas como nuevas soluciones.

Capítulo anterior

< [Chapter 14: Skip Lists](#)

Siguiente capítulo

[Conclusion](#) >

