# Algoritmi elementari per grafi 22

Questo capitolo descrive i metodi per rappresentare un grafo e per effettuare delle ricerche in un grafo. Effettuare delle ricerche in un grafo significa seguire sistematicamente gli archi del grafo in modo da visitare i vertici del grafo. Un algoritmo di ricerca in un grafo può scoprire molte cose sulla struttura di un grafo. Molti algoritmi iniziano ricercando nel loro grafo di input per ottenere queste informazioni strutturali. Altri algoritmi per grafi sono organizzati come semplici elaborazioni di algoritmi di ricerca di base. Le tecniche di ricerca in un grafo sono il cuore degli algoritmi che operano con i grafi.

Il Paragrafo 22.1 descrive le due più comuni rappresentazioni computazionali dei grafi: le liste di adiacenza e le matrici di adiacenza. Il Paragrafo 22.2 presenta un semplice algoritmo di ricerca per grafi, detto visita in ampiezza (breadth-first search), e spiega come creare l'albero risultante da una visita in ampiezza. Il Paragrafo 22.3 descrive il processo di ricerca in profondità e prova alcuni risultati standard sull'ordine in cui la ricerca in profondità visita i vertici. Il Paragrafo 22.4 descrive la prima applicazione reale della visita in profondità: l'ordinamento topologico di un grafo orientato aciclico. Una seconda applicazione della visita in profondità – trovare le componenti fortemente connesse di un grafo orientato – è descritta nel Paragrafo 22.5.

## 22.1 Rappresentazione dei grafi

Ci sono due metodi standard per rappresentare un grafo $G = (V, E)$: come una collezione di liste di adiacenza o come una matrice di adiacenza. Entrambi i metodi possono essere applicati ai grafi orientati e non orientati. Di solito, si preferisce la rappresentazione con liste di adiacenza, perché permette di rappresentare in modo compatto i grafi ***sparsi*** – per i quali $|E|$ è molto più piccolo di $|V|^2$. Per la maggior parte degli algoritmi per grafi descritti in questo libro si suppone che un grafo di input sia rappresentato tramite liste di adiacenza. Tuttavia, potrebbe essere preferibile una rappresentazione con matrice di adiacenza, quando il grafo è ***denso*** – $|E|$ è prossimo a $|V|^2$ – o quando dobbiamo essere in grado di dire rapidamente se c'è un arco che collega due vertici particolari. Per esempio, due degli algoritmi per cammini minimi fra tutte le coppie di vertici, descritti nel Capitolo 25, suppongono che i loro grafi di input siano rappresentati da matrici di adiacenza.

La ***rappresentazione con liste di adiacenza*** di un grafo $G = (V, E)$ consiste in un array $Adj$ di $|V|$ liste, una per ogni vertice in $V$. Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici $v$ tali che esista un arco $(u, v) \in E$. Ovvero $Adj[u]$ include tutti i vertici adiacenti a $u$ in $G$ (in alternativa, potrebbe contenere i puntatori a questi vertici). I vertici in ogni lista di adiacenza, tipicamente, sono

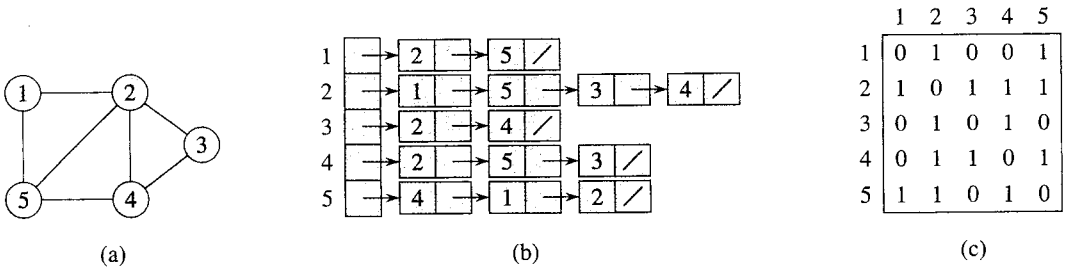


**Figura 22.1** Due rappresentazioni di un grafo non orientato. **(a)** Un grafo non orientato $G$ con cinque vertici e sette archi. **(b)** Una rappresentazione con liste di adiacenza di $G$. **(c)** La rappresentazione con matrice di adiacenza di $G$.

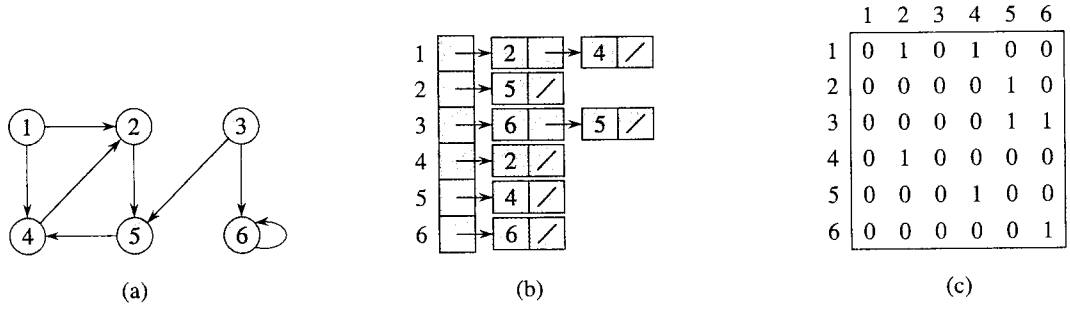


**Figura 22.2** Due rappresentazioni di un grafo orientato. **(a)** Un grafo orientato $G$ con sei vertici e otto archi. **(b)** Una rappresentazione con liste di adiacenza di $G$. **(c)** La rappresentazione con matrice di adiacenza di $G$.

memorizzati in un ordine arbitrario. La Figura 22.1(b) è una rappresentazione con liste di adiacenza del grafo non orientato della Figura 22.1(a). Analogamente, la Figura 22.2(b) è una rappresentazione con liste di adiacenza del grafo orientato della Figura 22.2(a).

Se $G$ è un grafo orientato, la somma delle lunghezze di tutte le liste di adiacenza è $|E|$, perché un arco della forma $(u, v)$ è rappresentato inserendo $v$ in $Adj[u]$. Se $G$ è un grafo non orientato, la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, perché se $(u, v)$ è un arco non orientato, allora $u$ appare nella lista di adiacenza di $v$ e viceversa. Per i grafi orientati e non orientati, la rappresentazione con liste di adiacenza ha l'interessante proprietà che la quantità di memoria richiesta è $\Theta(V + E)$.

Le liste di adiacenza possono essere facilmente adattate per rappresentare i ***grafi pesati***, cioè, i grafi per i quali ogni arco ha un ***peso*** associato, tipicamente dato da una ***funzione peso*** $w : E \to \mathbf{R}$. Per esempio, sia $G = (V, E)$ un grafo pesato con la funzione peso $w$. Il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene memorizzato semplicemente assieme al vertice $v$ nella lista di adiacenza di $u$. La rappresentazione con le liste di adiacenza è molto robusta, nel senso che può essere modificata per supportare molte altre varianti di grafi.

Uno svantaggio potenziale della rappresentazione con liste di adiacenza è che non c'è un modo più veloce per determinare se un particolare arco $(u, v)$ è presente nel grafo che cercare $v$ nella lista di adiacenza $Adj[u]$. Per porre un rimedio a questo svantaggio, si può rappresentare il grafo con una matrice di adiacenza, al costo di usare asintoticamente una maggiore quantità di memoria (l'Esercizio 22.1-8 include alcuni suggerimenti su come modificare le liste di adiacenza per accelerare le ricerche degli archi).

Per la ***rappresentazione con matrice di adiacenza*** di un grafo $G = (V, E)$ si suppone che i vertici siano numerati $1, 2, \ldots, |V|$ in modo arbitrario. La rappresentazione con matrice di adiacenza di un grafo $G$ consiste in una matrice $A = (a_{ij})$ di dimensioni $|V| \times |V|$ tale che

$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{negli altri casi} \end{cases}$$

Le Figure 22.1(c) e 22.2(c) sono le matrici di adiacenza del grafo non orientato e del grafo orientato illustrati, rispettivamente, nelle Figure 22.1(a) e 22.2(a). La matrice di adiacenza di un grafo richiede una memoria $\Theta(V^2)$, indipendentemente dal numero di archi nel grafo.

Osservate la simmetria rispetto alla diagonale principale della matrice di adiacenza nella Figura 22.1(c). Definiamo ***trasposta*** di una matrice $A = (a_{ij})$ la matrice $A^{\mathrm{T}} = (a_{ij}^{\mathrm{T}})$ con $a_{ij}^{\mathrm{T}} = a_{ji}$. Poiché $(u, v)$ e $(v, u)$ rappresentano lo stesso arco in un grafo non orientato, la matrice di adiacenza $A$ di un grafo non orientato è uguale alla sua trasposta: $A = A^{\mathrm{T}}$. In alcune applicazioni conviene memorizzare soltanto gli elementi che si trovano sopra e lungo la diagonale della matrice di adiacenza, riducendo così la memoria richiesta per memorizzare il grafo quasi della metà.

Come la rappresentazione con liste di adiacenza di un grafo, anche la rappresentazione con matrice di adiacenza può essere utilizzata per i grafi pesati. Per esempio, se $G = (V, E)$ è un grafo pesato con la funzione peso $w$, il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene semplicemente memorizzato come l'elemento nella riga $u$ e nella colonna $v$ della matrice di adiacenza. Se un arco non esiste, si può memorizzare un valore NIL come il suo corrispondente elemento nella matrice, sebbene per molti problemi sia preferibile utilizzare un valore come 0 o $\infty$.

Sebbene la rappresentazione con liste di adiacenza sia asintoticamente efficiente almeno quanto la rappresentazione con matrice di adiacenza, tuttavia quando i grafi sono abbastanza piccoli potrebbe essere preferita la matrice di adiacenza per la sua semplicità. Inoltre, se il grafo non è pesato, c'è un ulteriore vantaggio per la rappresentazione con matrice di adiacenza che riguarda la memoria richiesta. Anziché utilizzare una parola della memoria del calcolatore per ogni elemento della matrice di adiacenza, basta utilizzare un solo bit per ogni elemento.

### Esercizi

***22.1-1***

Data una rappresentazione con liste di adiacenza di un grafo orientato, quanto tempo occorre per calcolare il grado uscente di ogni vertice? Quanto tempo occorre per calcolare il grado entrante di ogni vertice?

***22.1-2***

Utilizzate le liste di adiacenza per rappresentare un albero binario completo con 7 vertici. Create l'equivalente rappresentazione con la matrice di adiacenza. Supponete che i vertici siano numerati da 1 a 7 come in un heap binario.

***22.1-3***

Il ***trasposto*** di un grafo orientato $G = (V, E)$ è il grafo $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, dove $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$. Quindi, $G^{\mathrm{T}}$ è $G$ con tutti i suoi archi invertiti. Descrivete degli algoritmi efficienti per calcolare $G^{\mathrm{T}}$ da $G$, rappresentando $G$ sia con le liste di adiacenza sia con la matrice di adiacenza. Analizzate i tempi di esecuzione dei vostri algoritmi.

***22.1-4***

Data una rappresentazione con liste di adiacenza di un multigrafo $G = (V, E)$, descrivete un algoritmo con tempo $O(V + E)$ per calcolare la rappresentazione con liste di adiacenza del grafo non orientato "equivalente" $G' = (V, E')$, dove $E'$ è formato dagli archi in $E$ con tutti gli archi multipli fra due vertici sostituiti da un singolo arco e con tutti i cappi rimossi.

***22.1-5***

Il ***quadrato*** di un grafo orientato $G = (V, E)$ è il grafo $G^2 = (V, E^2)$ tale che $(u, w) \in E^2$ se e soltanto se, per qualche $v \in V$, si abbia $(u, v) \in E$ e $(v, w) \in E$. Ovvero $G^2$ contiene un arco fra $u$ e $w$, se $G$ contiene un cammino con due soli archi fra $u$ e $w$. Descrivete degli algoritmi efficienti per calcolare $G^2$ da $G$, rappresentando $G$ sia con le liste di adiacenza sia con la matrice di adiacenza. Analizzate i tempi di esecuzione dei vostri algoritmi.

***22.1-6***

Quando viene utilizzata una rappresentazione con matrice di adiacenza, la maggior parte degli algoritmi per grafi richiede un tempo $\Omega(V^2)$, ma ci sono alcune eccezioni. Dimostrate che per determinare se un grafo orientato $G$ contiene un ***pozzo universale*** – un vertice con grado entrante $|V| - 1$ e grado uscente 0 – basta un tempo $O(V)$, data una matrice di adiacenza per $G$.

***22.1-7***

La ***matrice di incidenza*** di un grafo orientato $G = (V, E)$ è una matrice $B = (b_{ij})$ di dimensioni $|V| \times |E|$ tale che

$$b_{ij} = \begin{cases} -1 & \text{se l'arco } j \text{ esce dal vertice } i \\ 1 & \text{se l'arco } j \text{ entra nel vertice } i \\ 0 & \text{negli altri casi} \end{cases}$$

Descrivete che cosa rappresentano gli elementi del prodotto matriciale $BB^{\mathrm{T}}$, dove $B^{\mathrm{T}}$ è la matrice trasposta di $B$.

***22.1-8***

Supponete che, anziché una lista concatenata, ogni elemento dell'array $Adj[u]$ sia una tavola hash che contiene i vertici $v$ per i quali $(u, v) \in E$. Se tutte le ricerche degli archi sono equamente probabili, qual è il tempo atteso per determinare se un arco si trova nel grafo? Quali svantaggi ha questo schema? Suggerite una struttura dati alternativa per le liste di archi che risolve questi problemi. La vostra struttura dati alternativa ha degli svantaggi rispetto alla tavola hash?

## 22.2 Visita in ampiezza

La ***visita in ampiezza*** (breadth-first search) è uno dei più semplici algoritmi di ricerca nei grafi e l'archetipo per molti algoritmi importanti che operano con i grafi. L'algoritmo di Prim per l'albero di connessione minimo (Paragrafo 23.2) e l'algoritmo di Dijkstra per i cammini minimi da sorgente unica (Paragrafo 24.4) si basano su concetti simili a quelli della visita in ampiezza.

Dato un grafo $G = (V, E)$ e un vertice distinto $s$, detto ***sorgente***, la visita in ampiezza ispeziona sistematicamente gli archi di $G$ per "scoprire" tutti i vertici che sono raggiungibili da $s$. Calcola la distanza (il minimo numero di archi) da $s$ a ciascun vertice raggiungibile. Genera anche un albero BF (breadth-first tree) con radice $s$ che contiene tutti i vertici raggiungibili. Per ogni vertice $v$ raggiungibile

da $s$, il cammino nell'albero BF che va da $s$ a $v$ corrisponde a un "cammino minimo" da $s$ a $v$ in $G$, cioè un percorso che contiene il minor numero di archi. L'algoritmo opera sui grafi orientati e non orientati.

La visita in ampiezza è chiamata così perché espande la frontiera fra i vertici scoperti e quelli da scoprire in maniera uniforme lungo l'ampiezza della frontiera. Ovvero l'algoritmo scopre tutti i vertici che si trovano a distanza $k$ da $s$, prima di scoprire i vertici a distanza $k + 1$.

Per tenere traccia del lavoro svolto, la visita in ampiezza colora i vertici di bianco, di grigio o di nero. Inizialmente tutti i vertici sono bianchi; dopo possono diventare grigi e poi neri. Un vertice viene ***scoperto*** quando viene incontrato per la prima volta durante la visita; in quel momento cessa di essere un vertice bianco. I vertici grigi e neri, quindi, sono vertici che sono stati scoperti, ma l'algoritmo li mantiene distinti per fare in modo che la visita proceda in ampiezza. Se $(u, v) \in E$ e il vertice $u$ è nero, allora il vertice $v$ è grigio oppure nero; ovvero tutti i vertici adiacenti ai vertici neri sono stati scoperti. I vertici grigi possono avere qualche vertice bianco adiacente; essi rappresentano la frontiera fra i vertici scoperti e quelli da scoprire.

La visita in ampiezza costruisce l'albero BF che, inizialmente contiene soltanto la sua radice, che è il vertice sorgente $s$. Quando un vertice bianco $v$ viene scoperto durante l'ispezione della lista di adiacenza di un vertice $u$ già scoperto, il vertice $v$ e l'arco $(u, v)$ vengono aggiunti all'albero. Il vertice $u$ è detto ***predecessore*** o ***padre*** di $v$ nell'albero BF. Poiché un vertice viene scoperto una sola volta, esso ha al più un padre. Le relazioni di antenato e discendente nell'albero BF sono definite rispetto alla radice $s$ come di consueto: se $u$ è lungo il cammino che va dalla radice $s$ al vertice $v$, allora $u$ è un antenato di $v$ e $v$ è un discendente di $u$.

La seguente procedura BFS per la visita in ampiezza suppone che il grafo di input $G = (V, E)$ sia rappresentato con le liste di adiacenza; mantiene diverse altre strutture dati addizionali per ogni vertice nel grafo.

```
BFS(G, s)
 1  for ogni vertice u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for ogni v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

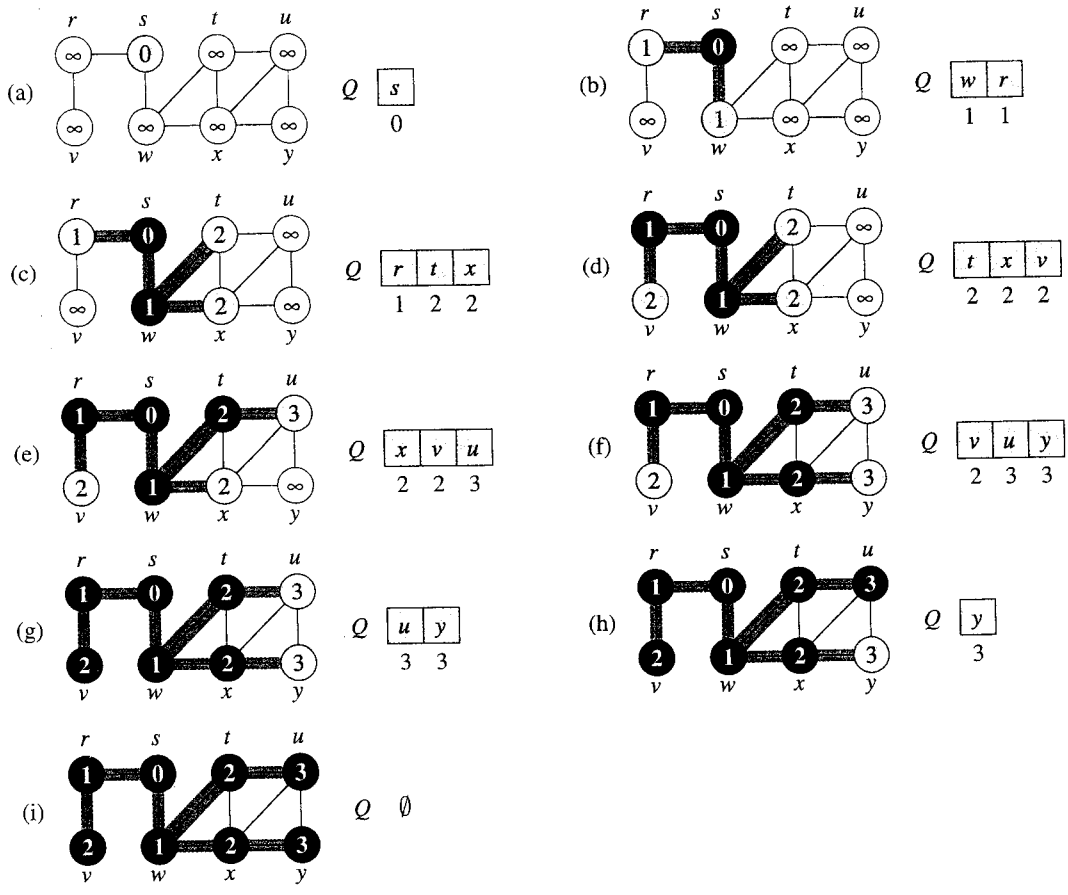


**Figura 22.3** Il funzionamento della procedura BFS con un grafo non orientato. Gli archi sono rappresentati in grigio quando vengono prodotti da BFS. All'interno di ciascun vertice $u$ è indicato il valore di $d[u]$. La coda $Q$ è rappresentata all'inizio di ogni iterazione del ciclo **while** (righe 10–18). In corrispondenza dei vertici nella coda sono annotate le distanze dei vertici.

Il colore di ogni vertice $u \in V$ è memorizzato nella variabile $color[u]$ e il predecessore di $u$ è memorizzato nella variabile $\pi[u]$. Se $u$ non ha un predecessore (per esempio, se $u = s$ o $u$ non è stato ancora scoperto), allora $\pi[u] = \text{NIL}$. La distanza dalla sorgente $s$ al vertice $u$ calcolata dall'algoritmo è memorizzata nella variabile $d[u]$. L'algoritmo usa anche una coda $Q$ con schema FIFO (vedere il Paragrafo 10.1) per gestire l'insieme dei vertici grigi. La Figura 22.3 illustra varie fasi della procedura BFS con un grafo campione.

La procedura BFS opera nel seguente modo. Con l'eccezione del vertice sorgente $s$, le righe 1–4 colorano di bianco tutti i vertici, assegnano alla variabile $d[u]$ il valore infinito per ogni vertice $u$ e assegnano al padre di ogni vertice il valore NIL. La riga 5 colora $s$ di grigio, perché questo vertice è considerato scoperto quando inizia la procedura. La riga 6 inizializza $d[s]$ a 0; la riga 7 assegna al predecessore della sorgente il valore NIL. Le righe 8–9 inizializzano $Q$ con la coda che contiene il solo vertice $s$.

Il ciclo **while** (righe 10–18) si ripete finché restano dei vertici grigi, che sono vertici scoperti le cui liste di adiacenza non sono state ancora completamente esaminate. Questo ciclo **while** conserva la seguente invariante:

> Quando viene eseguito il test della riga 10, la coda $Q$ è formata dall'insieme dei vertici grigi.

Sebbene non vogliamo utilizzare questa invariante di ciclo per provare la correttezza, tuttavia è facile capire che l'invariante è vera prima della prima iterazione e che ogni iterazione del ciclo conserva l'invariante. Prima della prima iterazione, l'unico vertice grigio, e l'unico vertice in $Q$, è il vertice sorgente $s$. La riga 11 determina il vertice grigio $u$ che si trova in testa alla coda $Q$ e lo elimina da $Q$.

Il ciclo **for** (righe 12–17) esamina ciascun vertice $v$ nella lista di adiacenza di $u$. Se $v$ è bianco, allora non è stato ancora scoperto; l'algoritmo lo scopre eseguendo le righe 14–17. Innanzi tutto esso viene colorato di grigio e la sua distanza $d[v]$ viene impostata a $d[u] + 1$. Poi, $u$ viene memorizzato come padre di $v$. Infine, $v$ viene posto in fondo alla coda $Q$. Dopo che tutti i vertici nella lista di adiacenza di $u$ sono stati esaminati, $u$ viene colorato di nero nella riga 18. L'invariante di ciclo si conserva perché, quando un vertice viene colorato di grigio (riga 14), viene anche accodato (riga 17) e, quando un vertice viene eliminato dalla coda (riga 11), viene anche colorato di nero (riga 18).

I risultati della visita in ampiezza possono dipendere dall'ordine in cui i vicini di un dato vertice vengono visitati nella riga 12: l'albero BF può variare, ma le distanze $d$ calcolate dall'algoritmo sono le stesse (Esercizio 22.2-4).

### Analisi

Prima di dimostrare le varie proprietà della visita in ampiezza, vogliamo analizzare il suo tempo di esecuzione con un grafo di input $G = (V, E)$. Utilizziamo il metodo dell'aggregazione, che abbiamo descritto nel Paragrafo 17.1. Dopo l'inizializzazione, nessun vertice sarà colorato di bianco, quindi il test nella riga 13 garantisce che ciascun vertice venga accodato al più una volta e, di conseguenza, venga eliminato dalla coda al più una volta.

Le operazioni di inserimento e cancellazione dalla coda richiedono un tempo $O(1)$, quindi il tempo totale dedicato alle operazioni con la coda è $O(V)$. Poiché la lista di adiacenza di ciascun vertice viene ispezionata soltanto quando il vertice viene rimosso dalla coda, ogni lista di adiacenza viene ispezionata al più una volta. Poiché la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(E)$, il tempo totale impiegato per ispezionare le liste di adiacenza è $O(E)$. Il costo generale di inizializzazione è $O(V)$, quindi il tempo di esecuzione totale di BFS è $O(V+E)$. In conclusione, la visita in ampiezza viene eseguita in un tempo che è lineare nella dimensione della rappresentazione con liste di adiacenza del grafo $G$.

### Cammini minimi

Come detto in precedenza, la visita in ampiezza trova la distanza di ciascun vertice raggiungibile in un grafo $G = (V, E)$ da un dato vertice sorgente $s \in V$. Definiamo la ***distanza di cammino minimo*** $\delta(s, v)$ da $s$ a $v$ come il numero minimo di archi in un cammino qualsiasi che va dal vertice $s$ al vertice $v$; se non c'è un cammino che va da $s$ a $v$, allora $\delta(s, v) = \infty$. Un cammino di lunghezza $\delta(s, v)$ da $s$ a $v$ è detto ***cammino minimo***[1] da $s$ a $v$. Prima di dimostrare che la visita in

[1]Nei Capitoli 24 e 25 generalizzeremo il nostro studio dei cammini minimi ai grafi pesati, in cui ogni arco ha un peso effettivo e il peso di un cammino è la somma dei pesi degli archi che lo compongono. I grafi considerati nel presente capitolo non sono pesati, ovvero tutti gli archi hanno peso unitario.

ampiezza calcola effettivamente le distanze di cammino minimo, analizziamo una importante proprietà di queste distanze.

***Lemma 22.1***
Se $G = (V, E)$ è un grafo orientato o non orientato e $s \in V$ è un vertice arbitrario, allora per qualsiasi arco $(u, v) \in E$ si ha

$$\delta(s, v) \leq \delta(s, u) + 1$$

***Dimostrazione*** Se $u$ è un vertice raggiungibile da $s$, allora lo è anche $v$. In questo caso, il cammino minimo da $s$ a $v$ non può essere più lungo del cammino minimo da $s$ a $u$ seguito dall'arco $(u, v)$, quindi la disuguaglianza è valida. Se $u$ non è raggiungibile da $s$, allora $\delta(s, u) = \infty$ e la disuguaglianza è valida. ■

Vogliamo dimostrare che BFS calcola correttamente $d[v] = \delta(s, v)$ per ogni vertice $v \in V$. Prima dimostriamo che $d[v]$ è un limite superiore per $\delta(s, v)$.

***Lemma 22.2***
Sia $G = (V, E)$ un grafo orientato o non orientato e supponiamo che BFS venga eseguita sul grafo $G$ da un dato vertice sorgente $s \in V$. Allora, al termine della procedura, per ogni vertice $v \in V$, il valore $d[v]$ calcolato da BFS soddisfa la relazione $d[v] \geq \delta(s, v)$.

***Dimostrazione*** Applichiamo l'induzione sul numero di operazioni ENQUEUE. La nostra ipotesi induttiva è che $d[v] \geq \delta(s, v)$ per ogni $v \in V$.

Il caso base dell'induzione è la situazione che si ha subito dopo che il vertice $s$ è stato inserito nella coda (riga 9 di BFS). L'ipotesi induttiva è vera qui, perché $d[s] = 0 = \delta(s, s)$ e $d[v] = \infty \geq \delta(s, v)$ per ogni $v \in V - \{s\}$.

Per il passaggio induttivo, consideriamo un vertice bianco $v$ che viene scoperto durante la visita a partire da un vertice $u$. L'ipotesi induttiva implica che $d[u] \geq \delta(s, u)$. Per l'assegnazione eseguita nella riga 15 e per il Lemma 22.1, si ha

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) \end{aligned}$$

Il vertice $v$ viene inserito nella coda e non sarà mai più reinserito perché viene anche colorato di grigio e la clausola **then** (righe 14–17) viene eseguita soltanto per i vertici bianchi. Quindi, il valore di $d[v]$ non cambierà più e l'ipotesi induttiva resta valida. ■

Per dimostrare che $d[v] = \delta(s, v)$, dobbiamo prima vedere più dettagliatamente come opera la coda $Q$ durante l'esecuzione di BFS. Il prossimo lemma dimostra che in qualsiasi istante ci sono al più due distinti valori $d$ nella coda.

***Lemma 22.3***
Supponiamo che durante l'esecuzione di BFS su un grafo $G = (V, E)$, la coda $Q$ contenga i vertici $\langle v_1, v_2, \ldots, v_r \rangle$, dove $v_1$ è l'inizio della coda $Q$ e $v_r$ è la fine. Allora, $d[v_r] \leq d[v_1] + 1$ e $d[v_i] \leq d[v_{i+1}]$ per $i = 1, 2, \ldots, r - 1$.

***Dimostrazione*** La dimostrazione è per induzione sul numero di operazioni eseguite con la coda. Inizialmente, quando la coda contiene soltanto $s$, il lemma è certamente valido.

Per il passaggio induttivo, dobbiamo provare che il lemma è valido dopo l'inserimento e la cancellazione di un vertice nella coda. Se il vertice $v_1$ viene eliminato dalla coda, $v_2$ diventa il nuovo inizio della coda (se la coda si svuota, allora il lemma è banalmente valido). Per l'ipotesi induttiva, $d[v_1] \leq d[v_2]$; ma allora abbiamo $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$ e le restanti disuguaglianze restano inalterate. Dunque, il lemma è dimostrato se $v_2$ è l'inizio della coda.

L'inserimento di un vertice nella coda richiede un esame più attento del codice. Quando inseriamo nella coda un vertice $v$ (riga 17 di BFS), esso diventa $v_{r+1}$. In quel momento, abbiamo già eliminato dalla coda $Q$ il vertice $u$, la cui lista di adiacenza è quella correntemente ispezionata. Se $u$ era l'unico elemento della coda, la sua rimozione svuota la coda, dopodiché ogni vertice $v$ inserito alla riga 17 ha lo stesso valore $d[v] = d[u] + 1$ e quindi le disuguaglianze restano vere. Altrimenti, per l'ipotesi induttiva, il nuovo inizio della coda $v_1$ ha $d[v_1] \geq d[u]$. Pertanto, $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. Per l'ipotesi induttiva, abbiamo anche $d[v_r] \leq d[u] + 1$, quindi $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$ e le restanti disuguaglianze restano inalterate. Dunque, il lemma è dimostrato quando $v$ viene inserito nella coda. ■

Il seguente corollario dimostra che i valori $d$ dei vertici quando vengono inseriti nella coda sono monotonicamente crescenti nel tempo.

***Corollario 22.4***
Supponiamo che i vertici $v_i$ e $v_j$ siano inseriti nella coda durante l'esecuzione di BFS e che $v_i$ sia inserito nella coda prima di $v_j$. Allora $d[v_i] \leq d[v_j]$ nell'istante in cui $v_j$ viene inserito nella coda.

***Dimostrazione*** È una conseguenza immediata del Lemma 22.3 e della proprietà che ogni vertice riceve un valore finito $d$ al più una volta durante l'esecuzione di BFS. ■

Adesso possiamo dimostrare che la visita in ampiezza calcola correttamente le distanze dei cammini minimi.

***Teorema 22.5 (Correttezza della visita in ampiezza)***
Sia $G = (V, E)$ un grafo orientato o non orientato e supponiamo che la procedura BFS venga eseguita sul grafo $G$ da un dato vertice sorgente $s \in V$. Allora, durante la sua esecuzione, BFS scopre tutti i vertici $v \in V$ che sono raggiungibili dalla sorgente $s$ e, alla fine dell'esecuzione, $d[v] = \delta(s, v)$ per ogni $v \in V$. Inoltre, per qualsiasi vertice $v \neq s$ che è raggiungibile da $s$, uno dei cammini minimi da $s$ a $v$ è un cammino minimo da $s$ a $\pi[v]$ seguito dall'arco $(\pi[v], v)$.

***Dimostrazione*** Supponiamo, per assurdo, che qualche vertice riceva un valore $d$ che non è uguale alla distanza del suo cammino minimo. Sia $v$ il vertice con il minimo $\delta(s, v)$ che riceve questo valore errato $d$; chiaramente $v \neq s$. Per il Lemma 22.2, $d[v] \geq \delta(s, v)$, e quindi $d[v] > \delta(s, v)$. Il vertice $v$ deve essere raggiungibile da $s$, perché se non lo fosse, allora $\delta(s, v) = \infty \geq d[v]$. Sia $u$ il vertice che precede immediatamente $v$ in un cammino minimo da $s$ a $v$, cosicché $\delta(s, v) = \delta(s, u) + 1$. Poiché $\delta(s, u) < \delta(s, v)$ e per come abbiamo scelto $v$, deve essere $d[u] = \delta(s, u)$. Ponendo insieme queste proprietà, si ha

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \tag{22.1}$$

Adesso consideriamo il momento in cui BFS sceglie di eliminare il vertice $u$ dalla coda $Q$ (riga 11). In quel momento, il vertice $v$ può essere bianco, grigio o nero. Dimostreremo che in ciascuno di questi casi, si arriva a una contraddizione della disuguaglianza (22.1). Se il vertice $v$ è bianco, allora la riga 15 imposta $d[v] = d[u] + 1$, che contraddice la disuguaglianza (22.1). Se il vertice $v$ è nero, allora era stato già rimosso dalla coda e, per il Corollario 22.4, si ha $d[v] \leq d[u]$, che contraddice ancora la disuguaglianza (22.1). Se il vertice $v$ è grigio, allora era stato colorato di grigio dopo la cancellazione dalla coda di qualche vertice $w$, che era stato cancellato da $Q$ prima di $u$ e per il quale $d[v] = d[w] + 1$. Per il Corollario 22.4, però, $d[w] \leq d[u]$, e quindi si ha $d[v] \leq d[u] + 1$, che contraddice ancora la disuguaglianza (22.1). Dunque, possiamo concludere che $d[v] = \delta(s, v)$ per ogni $v \in V$. Tutti i vertici raggiungibili da $s$ devono essere scoperti, perché se non lo fossero, avrebbero il valore $d$ infinito. Per concludere la dimostrazione del teorema, osserviamo che, se $\pi[v] = u$, allora $d[v] = d[u] + 1$. Quindi, possiamo ottenere un cammino minimo da $s$ a $v$ prendendo un cammino minimo da $s$ a $\pi[v]$ e poi attraversando l'arco $(\pi[v], v)$. ■

### Alberi di visita in ampiezza

La procedura BFS costruisce un albero BF mentre visita il grafo, come illustra la Figura 22.3. L'albero è rappresentato dal campo $\pi$ in ciascun vertice. Più formalmente, per un grafo $G = (V, E)$ con sorgente $s$, definiamo il ***sottografo dei predecessori*** di $G$ come $G_\pi = (V_\pi, E_\pi)$, dove

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

e

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$$

Il sottografo dei predecessori $G_\pi$ è un ***albero BF*** (o albero di visita in ampiezza) se $V_\pi$ è formato dai vertici raggiungibili da $s$ e, per ogni $v \in V_\pi$, c'è un solo cammino semplice da $s$ a $v$ in $G_\pi$ che è anche un cammino minimo da $s$ a $v$ in $G$. Un albero BF è effettivamente un albero, perché è connesso e $|E_\pi| = |V_\pi| - 1$ (vedere il Teorema B.2). Gli archi in $E_\pi$ sono detti ***archi d'albero***.

Dopo che la procedura BFS è stata eseguita da una sorgente $s$ su un grafo $G$, il seguente lemma dimostra che il sottografo dei predecessori è un albero BF.

***Lemma 22.6***

Quando viene applicata a un grafo $G = (V, E)$, orientato o non orientato, la procedura BFS costruisce $\pi$ in modo tale the il sottografo dei predecessori $G_\pi = (V_\pi, E_\pi)$ sia un albero BF.

***Dimostrazione*** La riga 16 della procedura BFS imposta $\pi[v] = u$ se e soltanto se $(u, v) \in E$ e $\delta(s, v) < \infty$ – ovvero, se $v$ è raggiungibile da $s$ – e quindi $V_\pi$ è formato dai vertici in $V$ che sono raggiungibili da $s$. Poiché $G_\pi$ forma un albero, per il Teorema B.2, esso contiene un unico cammino da $s$ a ciascun vertice in $V_\pi$. Applicando il Teorema 22.5 in modo induttivo, concludiamo che ciascuno di questi cammini è un cammino minimo. ■

La seguente procedura stampa i vertici di un cammino minimo da $s$ a $v$, supponendo che la procedura BFS sia già stata eseguita per calcolare l'albero dei cammini minimi.

```
PRINT-PATH(G, s, v)
1  if v = s
2     then stampa s
3     else if π[v] = NIL
4             then stampa "non ci sono cammini da" s "a" v
5             else PRINT-PATH(G, s, π[v])
6                  stampa v
```

Questa procedura viene eseguita in un tempo lineare nel numero di vertici del cammino stampato, perché ogni chiamata ricorsiva riguarda un cammino che ha un vertice in meno.

**Esercizi**

***22.2-1***

Calcolate i valori $d$ e $\pi$ che si ottengono effettuando una visita in ampiezza del grafo orientato della Figura 22.2(a), utilizzando il vertice 3 come sorgente.

***22.2-2***

Calcolate i valori $d$ e $\pi$ che si ottengono effettuando una visita in ampiezza del grafo non orientato della Figura 22.3, utilizzando il vertice $u$ come sorgente.

***22.2-3***

Qual è il tempo di esecuzione della procedura BFS se il suo grafo di input è rappresentato da una matrice di adiacenza e l'algoritmo viene modificato per gestire questa forma di input?

***22.2-4***

Dimostrate che in una visita in ampiezza il valore $d[u]$ assegnato a un vertice $u$ è indipendente dall'ordine in cui si trovano i vertici in ogni lista di adiacenza. Utilizzando la Figura 22.3 come esempio, dimostrate che l'albero BF calcolato dalla procedura BFS può dipendere dall'ordinamento delle liste di adiacenza.

***22.2-5***

Indicate un esempio di un grafo orientato $G = (V, E)$, un vertice sorgente $s \in V$ e un insieme di archi $E_\pi \subseteq E$ tali che, per ogni vertice $v \in V$, l'unico cammino da $s$ a $v$ nel grafo $(V, E_\pi)$ sia un cammino minimo in $G$ eppure l'insieme degli archi $E_\pi$ non può essere prodotto eseguendo la procedura BFS sul grafo $G$, indipendentemente da come siano ordinati i vertici all'interno di ciascuna lista di adiacenza.

***22.2-6***

Ci sono due tipi di lottatori professionisti: "buoni" e "cattivi". Fra una coppia qualsiasi di lottatori professionisti ci può essere o no rivalità. Supponete di avere $n$ lottatori e una lista di $r$ coppie di lottatori fra i quali c'è rivalità. Create un algoritmo con tempo $O(n + r)$ che determina se sia possibile designare alcuni lottatori come buoni e tutti gli altri come cattivi, in modo che la rivalità sia sempre fra un lottatore buono e uno cattivo. Se tale designazione è possibile, il vostro algoritmo dovrebbe effettuarla.

***22.2-7*** $\star$

Il ***diametro*** di un albero $T = (V, E)$ è dato da

$$\max_{u,v \in V} \delta(u, v) ;$$

In altre parole, il diametro è la più grande fra tutte le distanze di cammino minimo nell'albero. Create un algoritmo efficiente per calcolare il diametro di un albero e analizzate il tempo di esecuzione del vostro algoritmo.

*22.2-8*

Sia $G = (V, E)$ un grafo non orientato e connesso. Create un algoritmo con tempo $O(V + E)$ per calcolare un cammino in $G$ che attraversa ciascun arco in $E$ esattamente un volta in ogni direzione. Spiegate come potete trovare l'uscita in un labirinto, avendo a disposizione una grande quantità di monetine da un centesimo.

## 22.3 Visita in profondità

La strategia adottata dalla visita in profondità (depth-first search) consiste, come è implicito nel suo nome, nel visitare il grafo sempre più "in profondità" se possibile. Nella visita in profondità, gli archi vengono ispezionati a partire dall'ultimo vertice scoperto $v$ che ha ancora archi non ispezionati che escono da esso. Quando tutti gli archi di $v$ sono stati ispezionati, la visita "fa marcia indietro" per ispezionare gli archi che escono dal vertice dal quale $v$ era stato scoperto. Questo processo continua finché non saranno stati scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originale. Se restano dei vertici non scoperti, allora uno di essi viene selezionato come nuovo vertice sorgente e la visita riparte da questa sorgente. L'intero processo viene ripetuto finché non saranno scoperti tutti i vertici del grafo.

Come nella visita in ampiezza, quando un vertice $v$ viene scoperto durante un'ispezione della lista di adiacenza di un vertice $u$ già scoperto, la visita in profondità registra questo evento assegnando $u$ al campo $\pi[v]$ del predecessore di $v$. Diversamente dalla visita in ampiezza, il cui sottografo dei predecessori forma un albero, il sottografo dei predecessori prodotto da una visita in profondità può essere formato da più alberi, perché la visita può essere ripetuta da più sorgenti.[2] Il ***sottografo dei predecessori*** di una visita in profondità è quindi definito in modo leggermente diverso da quello di una visita in ampiezza: poniamo $G_\pi = (V, E_\pi)$, dove

$$E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$$

Il sottografo dei predecessori di una visita in profondità forma una ***foresta DF*** (depth-first forest) composta da vari ***alberi DF*** (depth-first tree). Gli archi in $E_\pi$ sono detti ***archi d'albero***.

Come nella visita in ampiezza, i vertici vengono colorati durante la visita in profondità per indicare il loro stato. Inizialmente, tutti i vertici sono bianchi. Un vertice diventa grigio quando viene ***scoperto*** durante la visita; diventa nero quando viene ***completato***, ovvero quando la sua lista di adiacenza è stata completamente

[2] Potrebbe sembrare arbitrario che una visita in ampiezza sia limitata a una sola sorgente, mentre una visita in profondità può effettuare la visita da più sorgenti. Sebbene, in teoria, una visita in ampiezza possa procedere da più sorgenti e una visita in profondità possa essere limitata a una sola sorgente, tuttavia il nostro approccio riflette il modo in cui vengono tipicamente utilizzati i risultati di queste visite. La visita in ampiezza, di solito, è utilizzata per trovare le distanze dei cammini minimi (e il corrispondente sottografo dei predecessori) da una data sorgente. La visita in profondità, di solito, è una subroutine in un altro algoritmo, come vedremo più avanti in questo capitolo.

ispezionata. Questa tecnica garantisce che ogni vertice vada a finire in un solo albero DF, in modo che questi alberi siano disgiunti.

Oltre a creare una foresta DF, la visita in profondità associa anche a ciascun vertice delle informazioni temporali. Ogni vertice $v$ ha due informazioni temporali: la prima $d[v]$ registra il momento in cui il vertice $v$ viene scoperto (e colorato di grigio); la seconda $f[v]$ registra il momento in cui la visita completa l'ispezione della lista di adiacenza del vertice $v$ (che diventa nero). Queste informazioni temporali sono utilizzate in molti algoritmi che operano con i grafi e, in generale, agevolano l'analisi del comportamento della visita in profondità.

La seguente procedura DFS registra nella variabile $d[u]$ il momento in cui scopre il vertice $u$ e nella variabile $f[u]$ il momento in cui completa la visita del vertice $u$. Queste informazioni temporali sono numeri interi compresi fra 1 e $2|V|$, perché ciascuno dei $|V|$ vertici può essere scoperto una sola volta e la sua visita può essere completata una sola volta. Per ogni vertice $u$ si ha

$$d[u] < f[u] \tag{22.2}$$

Il vertice $u$ è WHITE prima del tempo $d[u]$, GRAY fra il tempo $d[u]$ e il tempo $f[u]$, e BLACK successivamente.

Il seguente pseudocodice è l'algoritmo di base che effettua una visita in profondità. Il grafo di input $G$ può essere orientato o non orientato. La variabile *time* è una variabile globale che utilizziamo per registrare le informazioni temporali.

```
DFS(G)
1  for ogni vertice u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for ogni vertice u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ Il vertice bianco u è stato appena scoperto.
2  time ← time +1
3  d[u] ← time
4  for ogni v ∈ Adj[u]    ▷ Ispeziona l'arco (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Colora di nero u; visita completata.
9  f[u] ← time ← time +1
```

La Figura 22.4 illustra le varie fasi della procedura DFS applicata al grafo illustrato nella Figura 22.2.

La procedura DFS opera nel seguente modo. Le righe 1–3 colorano di bianco tutti i vertici e inizializzano i loro campi $\pi$ a NIL. La riga 4 azzera il contatore globale del tempo. Le righe 5–7 controllano, uno alla volta, tutti i vertici in $V$ e, quando trovano un vertice bianco, lo visitano utilizzando la procedura DFS-VISIT. Ogni volta che viene chiamata la procedura DFS-VISIT$(u)$ nella riga 7, il

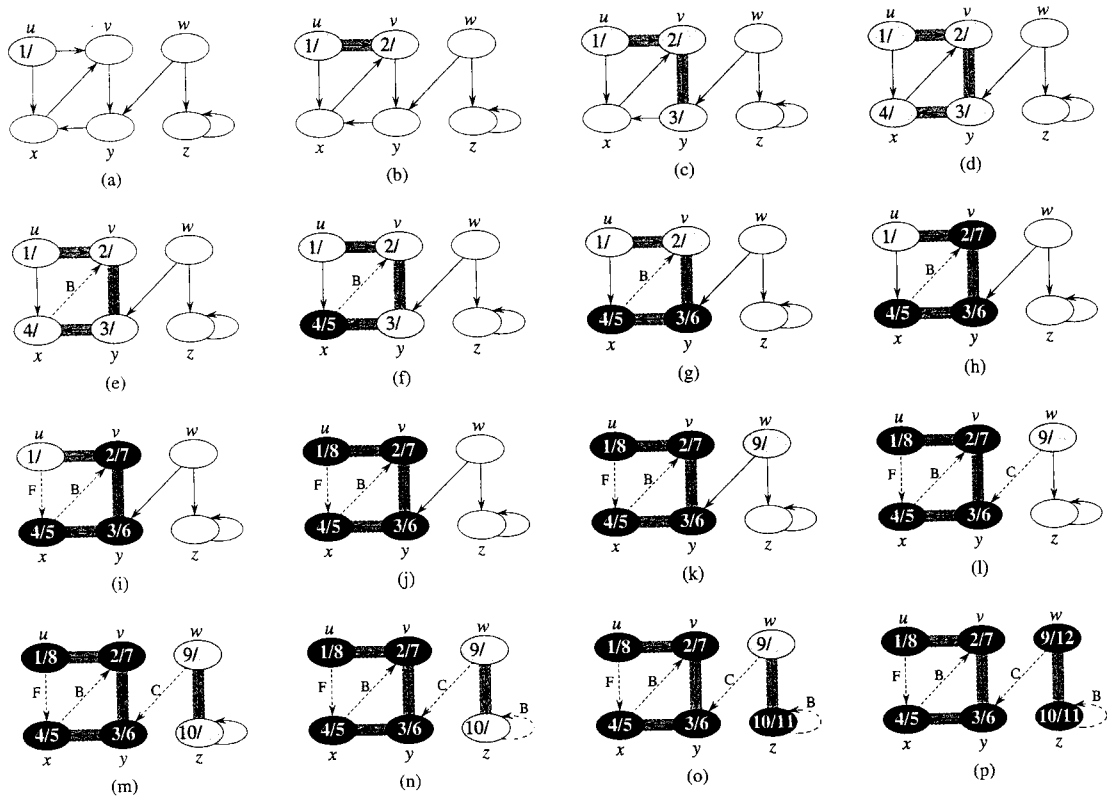


**Figura 22.4** Le fasi dell'algoritmo DFS di visita in profondità con un grafo orientato. Dopo che gli archi sono stati ispezionati dall'algoritmo, vengono rappresentati su uno sfondo grigio (se sono archi d'albero) o tratteggiati (negli altri casi). Quelli che non sono archi d'albero sono etichettati con le lettere B, C o F a seconda che siano archi all'indietro, trasversali o in avanti. All'interno dei vertici sono riportate le informazioni temporali (tempo di scoperta/tempo di completamento).

vertice $u$ diventa la radice di un nuovo albero della foresta DF. Quando la procedura DFS termina, a ogni vertice $u$ è stato assegnato un ***tempo di scoperta*** $d[u]$ e un ***tempo di completamento*** $f[u]$.

In ogni chiamata di DFS-VISIT$(u)$, il vertice $u$ è inizialmente bianco. La riga 1 colora di grigio $u$, la riga 2 incrementa la variabile globale *time* e la riga 3 registra il nuovo valore di *time* come il tempo di scoperta $d[u]$. Le righe 4–7 ispezionano ogni vertice $v$ adiacente a $u$ e visitano in modo ricorsivo il vertice $v$, se è bianco. Quando un vertice $v \in Adj[u]$ viene esaminato nella riga 4, diciamo che l'arco $(u, v)$ è stato ***ispezionato*** dalla visita in profondità. Infine, dopo che tutti i nodi che escono da $u$ sono stati ispezionati, le righe 8–9 colorano di nero $u$ e registrano in $f[u]$ il tempo di completamento della visita.

Notate che i risultati della visita in profondità potrebbero dipendere dall'ordine in cui i vertici sono ispezionati nella riga 5 della procedura DFS e dall'ordine in cui i vicini di un vertice vengono visitati nella riga 4 della procedura DFS-VISIT. In pratica, queste differenze nell'ordine in cui vengono effettuate le visite non causano problemi, in quanto *qualsiasi* risultato della visita in profondità di solito può essere efficacemente utilizzato, perché i risultati ottenuti sono essenzialmente equivalenti.

Qual è il tempo di esecuzione della procedura DFS? I cicli nelle righe 1–3 e nelle righe 5–7 di DFS impiegano un tempo $\Theta(V)$, escluso il tempo per ese-

guire le chiamate di DFS-VISIT. Come abbiamo fatto per la visita in ampiezza, applichiamo il metodo dell'aggregazione. La procedura DFS-VISIT è chiamata esattamente una volta per ogni vertice $v \in V$, perché DFS-VISIT viene invocata soltanto se un vertice è bianco e la prima cosa che fa è colorare di grigio il vertice. Durante un'esecuzione di DFS-VISIT($v$), il ciclo nelle righe 4–7 viene eseguito $|Adj[v]|$ volte. Poiché

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

il costo totale per eseguire le righe 4–7 di DFS-VISIT è $\Theta(E)$. Il tempo di esecuzione di DFS è dunque $\Theta(V + E)$.

### Proprietà della visita in profondità

La visita in profondità fornisce informazioni preziose sulla struttura di un grafo. Forse la più importante proprietà della visita in profondità è che il sottografo dei predecessori $G_\pi$ forma effettivamente una foresta di alberi, in quanto la struttura degli alberi DF rispecchia esattamente la struttura delle chiamate ricorsive di DFS-VISIT. Ovvero, $u = \pi[v]$ se e soltanto se DFS-VISIT($v$) è stata chiamata durante una visita della lista di adiacenza di $u$. In aggiunta, il vertice $v$ è un discendente del vertice $u$ nella foresta DF se e soltanto se $v$ viene scoperto durante il periodo in cui $u$ è grigio.

Un'altra importante proprietà della visita in profondità è che i tempi di scoperta e di completamento hanno una ***struttura di parentesi***. Se rappresentiamo la scoperta del vertice $u$ con una parentesi aperta "$(u$" e il suo completamento con una parentesi chiusa "$u)$", allora la storia delle scoperte e dei completamenti produce un'espressione ben formata, nel senso che le parentesi sono opportunamente annidate. Per esempio, la visita in profondità della Figura 22.5(a) corrisponde alla parentesizzazione illustrata nella Figura 22.5(b). Il seguente teorema consente di definire la proprietà della struttura di parentesi in un altro modo.

***Teorema 22.7 (Teorema delle parentesi)***
In qualsiasi visita in profondità di un grafo $G = (V, E)$ (orientato o non orientato), per ogni coppia di vertici $u$ e $v$, è soddisfatta una sola delle seguenti tre condizioni:

- Gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono completamente disgiunti; inoltre $u$ e $v$ non sono discendenti l'uno dell'altro nella foresta DF.
- L'intervallo $[d[u], f[u]]$ è interamente contenuto nell'intervallo $[d[v], f[v]]$; inoltre $u$ è un discendente di $v$ in un albero DF.
- L'intervallo $[d[v], f[v]]$ è interamente contenuto nell'intervallo $[d[u], f[u]]$; inoltre $v$ è un discendente di $u$ in un albero DF.

***Dimostrazione*** Iniziamo con il caso in cui $d[u] < d[v]$. Ci sono due sottocasi da considerare, a seconda che $d[v] < f[u]$ oppure no. Il primo sottocaso si verifica quando $d[v] < f[u]$, quindi $v$ è stato scoperto mentre $u$ era ancora grigio. Questo implica che $v$ è un discendente di $u$. Inoltre, poiché $v$ è stato scoperto più recentemente di $u$, vengono ispezionati tutti i suoi archi uscenti e l'ispezione di $v$ viene completata prima che la visita riprenda da $u$ e venga completata l'ispezione di $u$. In questo caso, quindi, l'intervallo $[d[v], f[v]]$ è interamente contenuto nell'intervallo $[d[u], f[u]]$. Nel secondo sottocaso, $f[u] < d[v]$, e la disuguaglianza (22.2)

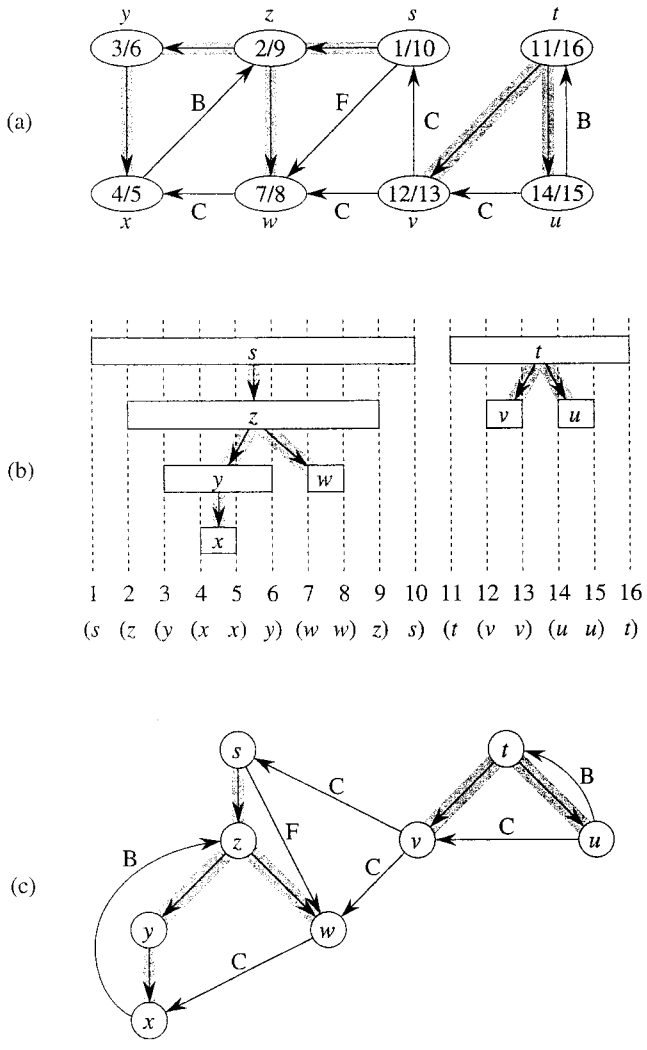


**Figura 22.5** Proprietà della visita in profondità. **(a)** Il risultato di una visita in profondità di un grafo orientato. I vertici includono le informazioni temporali e i tipi di archi sono indicati come nella Figura 22.4. **(b)** Gli intervalli fra il tempo di scoperta e il tempo di completamento di ciascun vertice corrispondono alla parentesizzazione illustrata. Ogni rettangolo si estende nell'intervallo definito dai tempi di scoperta e di completamento del corrispondente vertice. Sono illustrati gli archi d'albero. Se due intervalli si sovrappongono, allora uno viene annidato all'interno dell'altro e il vertice che corrisponde all'intervallo più piccolo è un discendente del vertice che corrisponde all'intervallo più grande. **(c)** Il grafo della parte (a) ridisegnato con tutti gli archi d'albero e in avanti che scendono in un albero DF e con tutti gli archi all'indietro che salgono da un discendente a un antenato.

implica che gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ siano disgiunti. Poiché gli intervalli sono disgiunti, nessuno dei due vertici è stato scoperto mentre l'altro era grigio, quindi nessuno dei due vertici è un discendente dell'altro.

Il caso in cui $d[v] < d[u]$ è simile, perché basta invertire i ruoli di $u$ e $v$ nella precedente discussione. ■

***Corollario 22.8 (Annidamento degli intervalli dei discendenti)***
Il vertice $v$ è un discendente proprio del vertice $u$ nella foresta DF per un grafo $G$ (orientato o non orientato) se e soltanto se $d[u] < d[v] < f[v] < f[u]$.

***Dimostrazione*** È una conseguenza immediata del Teorema 22.7. ■

Il prossimo teorema fornisce un'altra caratterizzazione importante di quando un vertice è un discendente di un altro vertice nella foresta DF.

***Teorema 22.9 (Teorema del cammino bianco)***
In una foresta DF di un grafo $G = (V, E)$ (orientato o non orientato), il vertice $v$ è un discendente del vertice $u$ se e soltanto se, al tempo $d[u]$ in cui viene sco-

perto $u$, il vertice $v$ può essere raggiunto da $u$ lungo un cammino che è formato esclusivamente da vertici bianchi.

***Dimostrazione*** $\Rightarrow$: Supponiamo che $v$ sia un discendente di $u$. Sia $w$ un vertice qualsiasi lungo il cammino fra $u$ e $v$ nell'albero DF; quindi $w$ è un discendente di $u$. Per il Corollario 22.8, $d[u] < d[w]$, e quindi $w$ è bianco nel tempo $d[u]$.

$\Leftarrow$: Supponiamo che il vertice $v$ sia raggiungibile da $u$ lungo un cammino di vertici bianchi al tempo $d[u]$ e che $v$ non diventi un discendente di $u$ nell'albero DF. Senza perdere in generalità, supponiamo che tutti gli altri vertici lungo il cammino diventino discendenti di $u$ (altrimenti, indichiamo con $v$ il vertice più vicino a $u$ lungo il cammino che non diventa un discendente di $u$). Sia $w$ il predecessore di $v$ lungo il cammino, cosicché $w$ è un discendente di $u$ (in effetti, $w$ e $u$ potrebbero essere lo stesso vertice) e, per il Corollario 22.8, si ha $f[w] \leq f[u]$.

Notate che $v$ deve essere scoperto dopo di $u$, ma prima che sia completata la visita di $w$. Di conseguenza, si ha $d[u] < d[v] < f[w] \leq f[u]$. Il Teorema 22.7 allora implica che l'intervallo $[d[v], f[v]]$ sia interamente contenuto nell'intervallo $[d[u], f[u]]$. Per il Corollario 22.8, $v$ deve essere un discendente di $u$.

■

### Classificazione degli archi

Un'altra interessante proprietà della visita in profondità è che la visita può essere utilizzata per classificare gli archi del grafo di input $G = (V, E)$. Questa classificazione degli archi può essere utilizzata per raccogliere informazioni importanti su un grafo. Per esempio, nel prossimo paragrafo, vedremo che un grafo orientato è aciclico se e soltanto se una visita in profondità non genera archi "all'indietro" (Lemma 22.11).

Possiamo definire quattro tipi di archi in base alla foresta DF $G_\pi$ prodotta da una visita in profondità del grafo $G$.

1. ***Archi d'albero***: sono gli archi nella foresta DF $G_\pi$. L'arco $(u, v)$ è un arco d'albero se $v$ viene scoperto la prima volta durante l'esplorazione di $(u, v)$.
2. ***Archi all'indietro***: sono quegli archi $(u, v)$ che collegano un vertice $u$ a un antenato $v$ in un albero DF. I cappi, che possono presentarsi nei grafi orientati, sono considerati archi all'indietro.
3. ***Archi in avanti***: sono gli archi $(u, v)$ (diversi dagli archi d'albero) che collegano un vertice $u$ a un discendente $v$ in un albero DF.
4. ***Archi trasversali***: tutti gli altri archi. Possono connettere i vertici nello stesso albero DF, purché un vertice non sia un antenato dell'altro, oppure possono connettere vertici di alberi DF differenti.

Nelle Figure 22.4 e 22.5 gli archi sono etichettati per indicare il loro tipo. La Figura 22.5(c) mostra anche come il grafo della Figura 22.5(a) può essere ridisegnato in modo che tutti gli archi d'albero e in avanti puntino in basso in un albero DF e tutti gli archi all'indietro puntino in alto. Qualsiasi grafo può essere ridisegnato in questo modo.

L'algoritmo DFS può essere modificato per classificare gli archi che incontra. L'idea chiave è che ogni arco $(u, v)$ può essere classificato in base al colore del vertice $v$ che viene raggiunto quando l'arco viene ispezionato per la prima volta (tranne gli archi in avanti e trasversali che non sono distinti):

1. WHITE indica un arco d'albero.
2. GRAY indica un arco all'indietro.
3. BLACK indica un arco in avanti o trasversali.

Il primo caso è immediato dalla specifica dell'algoritmo. Per il secondo caso, notate che i vertici grigi formano sempre una catena lineare di discendenti che corrisponde allo stack delle chiamate attive di DFS-VISIT; il numero di vertici grigi è uno più della profondità dell'ultimo vertice scoperto nella foresta DF.

L'ispezione procede sempre a partire dal vertice grigio più profondo, quindi un arco che raggiunge un altro vertice grigio raggiunge un antenato. Il terzo caso gestisce l'ultima possibilità; si può dimostrare che tale arco $(u, v)$ è un arco in avanti se $d[u] < d[v]$ e un arco trasversale se $d[u] > d[v]$ (Esercizio 22.3-4).

In un grafo non orientato potrebbe esserci qualche ambiguità nella classificazione degli archi, perché $(u, v)$ e $(v, u)$ in effetti sono lo stesso arco. In tale caso, l'arco viene classificato con il *primo* tipo della lista di classificazione che può essere applicato. In modo equivalente (vedere l'Esercizio 22.3-5), l'arco viene classificato a seconda se si incontra prima $(u, v)$ o $(v, u)$ durante l'esecuzione dell'algoritmo.

Dimostriamo adesso che in una visita in profondità di un grafo non orientato non si presentano mai archi in avanti e trasversali.

***Teorema 22.10***
In una visita in profondità di un grafo non orientato $G$, gli archi di $G$ possono essere archi d'albero o archi all'indietro.

***Dimostrazione*** Sia $(u, v)$ un arco arbitrario di $G$ e, senza perdere in generalità, supponiamo che $d[u] < d[v]$. Allora, il vertice $v$ deve essere scoperto e completato prima che sia completato $u$ (mentre $u$ è grigio), perché $v$ si trova nella lista di adiacenza di $u$. Se l'arco $(u, v)$ viene ispezionato prima nella direzione da $u$ a $v$, allora $v$ è un vertice non scoperto (bianco) fino a quel momento, perché altrimenti avremmo già ispezionato questo arco nella direzione da $v$ a $u$. Quindi, $(u, v)$ diventa un arco d'albero. Se $(u, v)$ viene ispezionato prima nella direzione da $v$ a $u$, allora $(u, v)$ è un arco all'indietro, perché $u$ è ancora grigio quando l'arco viene esplorato per la prima volta. ■

Nei prossimi paragrafi vedremo varie applicazioni di questi teoremi.

## Esercizi

***22.3-1***
Create una tabella $3 \times 3$ con le etichette WHITE, GRAY e BLACK per le righe e le colonne. In ogni cella $(i, j)$ indicate se, in qualsiasi punto durante una visita in profondità di un grafo orientato, ci può essere un arco che va da un vertice di colore $i$ a un vertice di colore $j$. Per ciascuno degli archi possibili indicate il tipo di arco. Create una tabella analoga per la visita in profondità di un grafo non orientato.

***22.3-2***
Illustrate come funziona una visita in profondità con il grafo della Figura 22.6. Supponete che il ciclo **for** (righe 5–7) della procedura DFS consideri i vertici in ordine alfabetico e che ogni lista di adiacenza sia ordinata alfabeticamente. Indi-

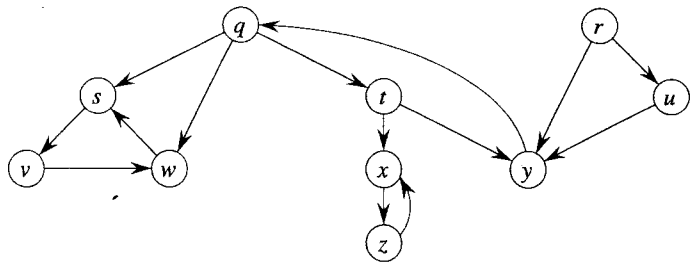


**Figura 22.6** Un grafo orientato per gli Esercizi 22.3-2 e 22.5-2.

cate i tempi di scoperta e di completamento di ciascun vertice e la classificazione di ciascun arco.

***22.3-3***
Illustrate la struttura delle parentesi della visita in profondità del grafo illustrato nella Figura 22.4.

***22.3-4***
Dimostrate che l'arco $(u, v)$ è

*a.* Un arco d'albero o un arco in avanti se e soltanto se $d[u] < d[v] < f[v] < f[u]$.

*b.* Un arco all'indietro se e soltanto se $d[v] \leq d[u] < f[u] \leq f[v]$.

*c.* Un arco trasversale se e soltanto se $d[v] < f[v] < d[u] < f[u]$.

***22.3-5***
Dimostrate che in un grafo non orientato classificare un arco $(u, v)$ come arco d'albero o arco all'indietro, a seconda se viene incontrato prima $(u, v)$ o $(v, u)$ durante la visita in profondità, equivale a classificare l'arco secondo la priorità dei tipi nello schema di classificazione.

***22.3-6***
Riscrivete la procedura DFS utilizzando uno stack per eliminare la ricorsione.

***22.3-7***
Trovate un controesempio alla congettura che, se esiste un cammino da $u$ a $v$ in un grafo orientato $G$ e se $d[u] < d[v]$ in una visita in profondità di $G$, allora $v$ è un discendente di $u$ nella foresta DF risultante.

***22.3-8***
Trovate un controesempio alla congettura che, se esiste un cammino da $u$ a $v$ in un grafo orientato $G$, allora per qualsiasi visita in profondità deve risultare $d[v] \leq f[u]$.

***22.3-9***
Modificate lo pseudocodice per la visita in profondità in modo che stampi tutti gli archi del grafo orientato $G$ insieme ai loro tipi. Descrivete quali eventuali modifiche devono essere apportate se $G$ non è un grafo orientato.

***22.3-10***
Spiegate come un vertice $u$ di un grafo orientato possa finire in un albero DF che contiene soltanto $u$, anche se $u$ ha degli archi entranti e uscenti nel grafo $G$.

***22.3-11***
Dimostrate che una visita in profondità di un grafo non orientato $G$ può essere utilizzata per identificare le componenti connesse di $G$ e che la foresta DF contiene

tanti alberi quante sono le componenti connesse di $G$. Più precisamente, spiegate come modificare la visita in profondità in modo che a ogni vertice $v$ sia assegnata un'etichetta $cc[v]$ di numeri interi compresi fra 1 e $k$, dove $k$ è il numero di componenti connesse di $G$, tale che $cc[u] = cc[v]$ se e soltanto se $u$ e $v$ si trovano nella stessa componente connessa.

***22.3-12*** $\star$

Un grafo orientato $G = (V, E)$ è ***singolarmente connesso*** se $u \rightsquigarrow v$ implica che ci sia al più un cammino semplice da $u$ a $v$ per tutti i vertici $u, v \in V$. Create un algoritmo efficiente per determinare se un grafo orientato è singolarmente connesso.

## 22.4 Ordinamento topologico

Questo paragrafo spiega come utilizzare la visita in profondità per eseguire l'ordinamento topologico di un grafo orientato aciclico o semplicemente "dag" (dall'inglese *directed acyclic graph*). Un ***ordinamento topologico*** di un dag $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici tale che, se $G$ contiene un arco $(u, v)$, allora $u$ appare prima di $v$ nell'ordinamento (se il grafo non è aciclico, allora non è possibile effettuare alcun ordinamento lineare). Un ordinamento topologico di un grafo può essere visto come un ordinamento dei suoi vertici lungo una linea orizzontale in modo che tutti gli archi orientati siano diretti da sinistra a destra. L'ordinamento topologico è quindi diverso dal tipo usuale di "ordinamento" studiato nella Parte II.

I grafi orientati aciclici sono utilizzati in molte applicazioni per indicare le precedenze fra gli eventi. La Figura 22.7 illustra un caso che si verifica tutte le mattine quando il professor Bumstead si veste. Il professore deve indossare determinati indumenti prima di altri (per esempio, le calze prima delle scarpe). Altri indumenti possono essere indossati in qualsiasi ordine (per esempio, le calze e i pantaloni). Un arco orientato $(u, v)$ nel dag della Figura 22.7(a) indica che l'indumento $u$ deve essere indossato prima dell'indumento $v$. Un ordinamento topologico di questo dag, quindi, determina la sequenza ordinata in cui indossare gli indumenti per vestirsi. La Figura 22.7(b) illustra il dag ordinato topologicamente come un sequenza ordinata di vertici lungo una linea orizzontale tale che tutti gli archi orientati siano diretti da sinistra a destra.

Il seguente semplice algoritmo ordina topologicamente un dag.

TOPOLOGICAL-SORT$(G)$

1 Chiama DFS$(G)$ per calcolare i tempi di completamento $f[v]$
  per ciascun vertice $v$
2 Una volta completata l'ispezione di un vertice,
  inserisce il vertice in testa alla lista concatenata
3 **return** la lista concatenata dei vertici

La Figura 22.7(b) illustra come si presentano i vertici ordinati topologicamente in senso inverso rispetto ai loro tempi di completamento.

È possibile eseguire un ordinamento topologico nel tempo $\Theta(V + E)$, perché la visita in profondità impiega un tempo $\Theta(V + E)$ e occorre un tempo $O(1)$ per inserire ciascuno dei $|V|$ vertici in testa alla lista concatenata.

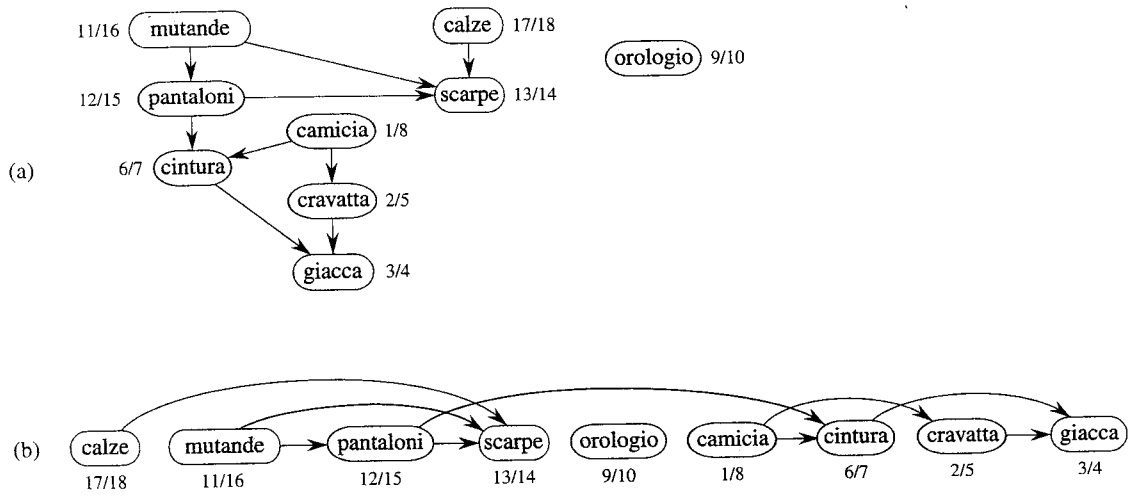


**Figura 22.7** **(a)** Il professor Bumstead ordina topologicamente i sui indumenti quando si veste. Ogni arco orientato $(u, v)$ significa che l'indumento $u$ deve essere indossato prima dell'indumento $v$. I tempi di scoperta e di completamento di una visita in profondità sono illustrati accanto a ciascun vertice. **(b)** Lo stesso grafo rappresentato secondo l'ordinamento topologico. I suoi vertici sono ordinati da sinistra a destra in senso decrescente rispetto ai tempi di completamento. Notate che tutti gli archi orientati sono diretti da sinistra a destra.

Dimostriamo la correttezza di questo algoritmo utilizzando il seguente lemma chiave che caratterizza i grafi orientati aciclici.

***Lemma 22.11***
Un grafo orientato $G$ è aciclico se e soltanto se una visita in profondità di $G$ non genera archi all'indietro.

***Dimostrazione*** $\Rightarrow$: Supponiamo che ci sia un arco all'indietro $(u, v)$. Allora, il vertice $v$ è un antenato del vertice $u$ nella foresta DF. Quindi, esiste un cammino che va da $v$ a $u$ nel grafo $G$ e l'arco all'indietro $(u, v)$ completa un ciclo.

$\Leftarrow$: Supponiamo che il grafo $G$ contenga un ciclo $c$. Dimostriamo che una visita in profondità di $G$ genera un arco all'indietro. Sia $v$ il primo vertice che viene scoperto in $c$ e sia $(u, v)$ l'arco precedente in $c$. Al tempo $d[v]$, i vertici di $c$ formano un cammino di vertici bianchi da $v$ a $u$. Per il teorema del cammino bianco, il vertice $u$ diventa un discendente di $v$ nella foresta DF. Dunque, $(u, v)$ è un arco all'indietro. ■

***Teorema 22.12***
TOPOLOGICAL-SORT$(G)$ produce un ordinamento topologico di un grafo orientato aciclico $G$.

***Dimostrazione*** Supponiamo che la procedura DFS venga eseguita su un dato dag $G = (V, E)$ per determinare i tempi di completamento dei suoi vertici. È sufficiente dimostrare che per una coppia qualsiasi di vertici distinti $u, v \in V$, se esiste un arco in $G$ che va da $u$ a $v$, allora $f[v] < f[u]$. Consideriamo un arco qualsiasi $(u, v)$ ispezionato da DFS$(G)$. Quando questo arco viene ispezionato, il vertice $v$ non può essere grigio, perché altrimenti $v$ sarebbe un antenato di $u$ e $(u, v)$ sarebbe un arco all'indietro, contraddicendo il Lemma 22.11. Quindi, il vertice $v$ deve essere bianco o nero. Se $v$ è bianco, diventa un discendente di $u$ e

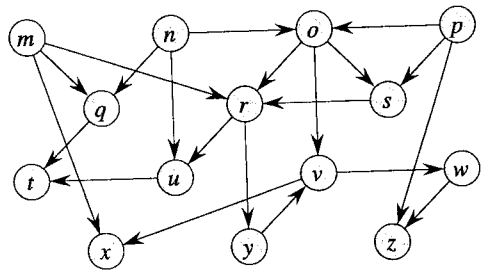


**Figura 22.8** Un grafo orientato aciclico per l'ordinamento topologico.

quindi $f[v] < f[u]$. Se $v$ è nero, la sua ispezione è stata già completata, quindi il valore di $f[v]$ è già stato impostato. Poiché stiamo ancora ispezionando dal vertice $u$, dobbiamo ancora assegnare un'informazione temporale a $f[u]$ e, quando lo faremo, avremo ancora $f[v] < f[u]$. Quindi, per qualsiasi arco $(u, v)$ nel dag, si ha $f[v] < f[u]$, e questo dimostra il teorema. ■

### Esercizi

***22.4-1***
Illustrate come vengono ordinati i vertici quando la procedura TOPOLOGICAL-SORT viene eseguita sul grafo orientato aciclico della Figura 22.8, con le ipotesi fatte nell'Esercizio 22.3-2.

***22.4-2***
Descrivete un algoritmo con tempo lineare che riceve come input un grafo orientato aciclico $G = (V, E)$ e due vertici $s$ e $t$, e restituisce il numero di cammini da $s$ a $t$ nel grafo $G$. Per esempio, nel grafo orientato aciclico della Figura 22.8 ci sono esattamente quattro cammini dal vertice $p$ al vertice $v$: $pov$, $poryv$, $posryv$ e $psryv$ (l'algoritmo dovrà soltanto contare i cammini, senza elencarli).

***22.4-3***
Descrivete un algoritmo che determina se un grafo non orientato $G = (V, E)$ contiene oppure no un ciclo. Il vostro algoritmo dovrebbe essere eseguito nel tempo $O(V)$, indipendentemente da $|E|$.

***22.4-4***
Dimostrare o confutare: se un grafo orientato $G$ contiene dei cicli, allora la procedura TOPOLOGICAL-SORT$(G)$ produce un ordinamento di vertici che minimizza il numero di archi "cattivi" che non sono coerenti con l'ordinamento ottenuto.

***22.4-5***
Un altro modo di eseguire un ordinamento topologico su un grafo orientato aciclico $G = (V, E)$ consiste nel trovare ripetutamente un vertice di grado entrante pari a 0, nel generare un output di questo vertice e nel rimuovere il vertice dal grafo insieme a tutti i suoi archi uscenti. Spiegate come implementare questa idea in modo che l'algoritmo di ordinamento possa essere eseguito nel tempo $O(V + E)$. Che cosa accade a questo algoritmo se il grafo $G$ ha dei cicli?

## 22.5 Componenti fortemente connesse

Consideriamo adesso una classica applicazione della visita in profondità: scomporre un grafo orientato nelle sue componenti fortemente connesse. Questo para-