1. Grafi

I grafi sono utilizzati per rappresentare insiemi di oggetti e le relazioni che intercorrono fra essi. Molti problemi di calcolo sono equivalenti a problemi su grafi, e possono essere risolti con tecniche sperimentate.

1.1 Definizioni

Per poter ragionare su una struttura di dati così fondamentale, è importante definirne precisamente la terminologia.

1.1.1 Grafi orientati

Un *grafo orientato* (o *grafo diretto*, *directed graph* in inglese) è una coppia $G = (V, E)$, con V insieme finito di elementi, detti *nodi* (o vertici, *vertex* in inglese), ed E insieme finito di coppie ordinate di nodi, detti *archi* (o spigoli, *edge* in inglese). I grafi orientati possono essere rappresentati graficamente disegnando ogni nodo con un cerchietto e ogni arco (u, v) con una freccia che esce dal nodo u ed entra nel nodo v. Il numero di nodi e il numero di archi sono di solito indicati, rispettivamente, con n ed m, oppure con $|V|$ ed $|E|$. Poiché possono esserci al più $n-1$ archi uscenti da ciascun nodo, risulta $0 \leq m \leq n(n-1)$.

Esempio (Grafo orientato) $G = (V, E)$ con $V = \{1, 2, 3, 4\}$ ed $E = \{(1,2), (1,3), (2,4), (3,2), (4,2), (4,3)\}$ è un grafo orientato, ove $n = |V| = 4$ ed $m = |E| = 6$. G può essere rappresentato come mostrato nella Figura 1.1. ■

In un grafo orientato, un nodo v si dice *adiacente* ad un nodo u se esiste un arco $(u, v) \in E$. Tale arco si dice *incidente* da u in v. Il *grado entrante* (*uscente*) di un nodo u è il numero di archi incidenti in (da) u.

In un grafo orientato G, un *cammino* è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tali che $(u_i, u_{i+1}) \in E$, per $i = 0, \ldots, k-1$. Il cammino parte dal nodo u_0, attraversa i nodi interni $u_1, \ldots, u_{k-1}$, arriva al nodo u_k, ed ha *lunghezza* uguale al numero k di archi. Se non ci sono nodi ripetuti, cioè $u_i \neq u_j$ per

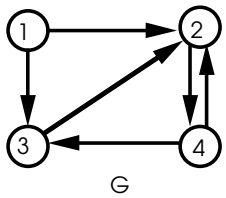

Figura 1.1: Un grafo orientato.

$0 \leq i < j \leq k$, allora il cammino è semplice, mentre se $u_0 = u_k$, allora è chiuso. Un cammino sia semplice che chiuso, dove l'unica ripetizione è $u_0 = u_k$, è detto *ciclo*.

Esempio (Cammini) Nel grafo G di Figura 1.1, la sequenza 1-2-4-3-2 è un cammino, 1-2-4 è un cammino semplice, e 2-4-3-2 è un ciclo. ■

1.1.2 Grafi non orientati

Talvolta, gli archi di un grafo G sono formati da coppie non ordinate $[u,v]$. In tal caso, G è un *grafo non orientato*. Mentre in un grafo orientato (u,v) e (v,u) indicano due archi distinti, in uno non orientato $[u,v]$ e $[v,u]$ indicano lo stesso arco. Grafi non orientati sono usati per rappresentare relazioni simmetriche tra oggetti.

Esempio (Grafo non orientato) Il grafo non orientato $G = (V,E)$ con $V = \{1,2,3,4\}$ e $E = \{[1,2],[1,3],[2,4],[3,2],[4,3]\}$ è mostrato nella Figura 1.2. I nodi 1 e 3 sono adiacenti, ma 1 e 4 non lo sono. ■

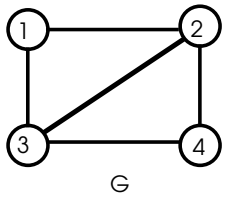

Figura 1.2: Un grafo non orientato.

Esempio (Cabala) Il biblico "Legno (o Albero) della Vita", con i suoi 10 Sefiroth e i 22 Sentieri, è rappresentato nella tradizione cabalistica con il grafo non orientato mostrato all'inizio di questo testo. ■

Anche nei grafi non orientati ci sono nozioni analoghe a quelle di cammino e ciclo. In un grafo non orientato G, una *catena* è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tali che $[u_i, u_j] \in E$, per $i = 0, \ldots, k-1$. La catena collega u_0 con u_k, o viceversa, ed ha lunghezza uguale a k. Se non ci sono nodi ripetuti, allora la catena è semplice, mentre se $u_0 = u_k$, allora è chiusa. Una catena sia semplice che chiusa i cui archi sono tutti distinti è un *circuito*. Si noti che nei grafi orientati la mancanza di ripetizioni di nodi implica la mancanza di ripetizione degli archi, ma questo non è vero per i grafi non orientati.

Esempio (Catene e circuiti) Nel grafo G di Figura 1.2, la sequenza 1-2-4-2-3-1 è una catena chiusa e 2-4-3-2 è un circuito. La sequenza 1-2-1 è una catena chiusa ma non un circuito, benché non ci siano nodi ripetuti, perché è ripetuto l'arco $[1,2]$. ■

1.1.3 Connessione e connessione forte

Sia $G = (V,E)$ un grafo orientato. G è *fortemente connesso* se per ogni coppia di nodi u e v esiste almeno un cammino da u a v ed almeno un cammino da v ad u. Un grafo $G' = (V',E')$ è una *componente fortemente connessa* di G se e solo se è un sottografo di G fortemente connesso e massimale:

- G' è un *sottografo* di G ($G' \subseteq G$) se e solo se $V' \subseteq V$ e $E' \subseteq E$;
- G' è massimale se non è possibile trovare un sottografo G'' di G che sia fortemente connesso e "più grande" di G', ovvero tale per cui $G' \subseteq G'' \subseteq G$.

Esempio (Connessione) Il grafo G di Figura 1.1 non è fortemente connesso. Infatti esiste un cammino da 1 a 4, ma non da 4 a 1. Il grafo è infatti composto da due componenti fortemente connesse: $\{1\}$ e $\{2,3,4\}$. ■

Sia $G = (V,E)$ un grafo non orientato. G è *connesso* se e solo se per ogni coppia di nodi distinti u e v esiste una catena tra u e v. Un grafo $G' = (V',E')$ è una *componente connessa* di G se e solo se è un sottografo di G connesso e massimale.

Esempio Il grafo G di Figura 1.2 è connesso. Il grafo di Figura 1.3 è formato da tre componenti connesse.

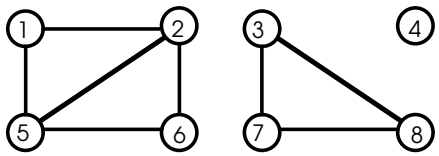

Figura 1.3: Un grafo non orientato con tre componenti connesse.

1.1.4 Relazioni fra alberi e grafi

Un grafo non orientato è un *albero libero* se è connesso e per ogni coppia di nodi esiste una e una sola catena semplice. È facile verificare che una definizione equivalente di albero libero richiede, oltre alla connessione, che il numero di archi sia minimo (cioè uguale al numero di nodi meno uno) oppure che non esista alcun circuito. Un *albero radicato* è ottenuto da uno libero stabilendo un ordinamento "verticale" tra i nodi: un nodo r è designato arbitrariamente come radice e i nodi rimanenti sono ordinati per livelli, cioè in base alla loro distanza dalla radice. Dato un albero libero di n nodi, da questo si possono ricavare pertanto n alberi radicati tra loro distinti. Un albero ordinato è ottenuto da uno radicato stabilendo un ordinamento "orizzontale" tra i nodi che stanno allo stesso livello [cfr. Capitolo 5].

Esempio (Alberi liberi e radicati) Il grafo G di Figura 1.2 non è un albero libero perché ha 4 nodi e più di 3 archi. Invece, la Figura 1.4 mostra quattro alberi $T_1,\ldots,T_4$. Visti come alberi liberi, essi sono tutti identici tra loro. T_1 non è radicato, mentre gli altri lo sono. Visti come

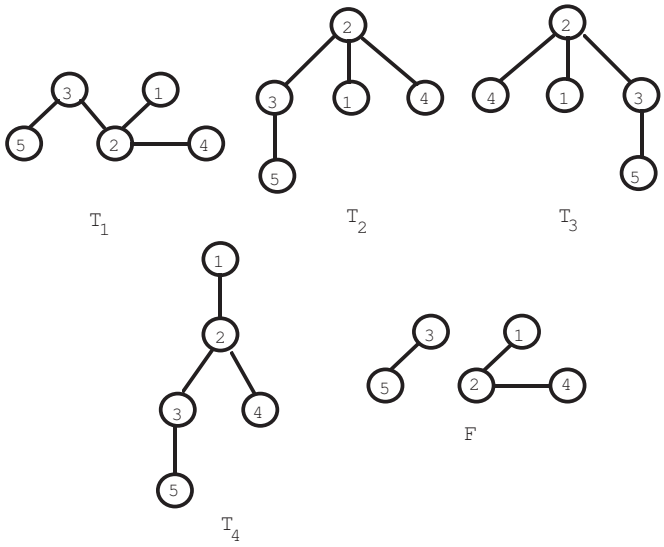

Figura 1.4: Alberi liberi, radicati e foreste.

alberi radicati, T_2 e T_3, avendo entrambi il nodo 2 come radice, sono identici tra loro, ma sono diversi da T_4 che è radicato in 1. La Figura 1.4 mostra anche un grafo F che non è un albero perché ha 5 nodi, ma meno di 4 archi e quindi non è connesso. Essendo formato da un insieme di alberi, F è una *foresta*. ■

Un *albero di copertura* T di un grafo connesso $G = (V,E)$ è un albero libero $T = (V,E')$ composto da tutti i nodi di V e da un sottoinsieme degli archi ($E' \subseteq E$), tale per cui tutte le coppie di nodi del grafo sono connesse da una sola catena nell'albero. Il nome deriva dal fatto che tutti i nodi devono essere "coperti" dall'albero.

Esempio (Chi ha ucciso il duca McPollock?) In molti problemi pratici, ci si imbatte in grafi che sono caratterizzati da proprietà particolari, che risultano utili per risolvere il problema stesso. Ad esempio, si consideri il problema seguente: Sherlock Holmes è stato incaricato di indagare sulla misteriosa morte del vecchio duca McPollock, ucciso da un'esplosione nel suo maniero scozzese. Il Dr. Watson si presenta a rapporto da Holmes in Baker Street:

Watson: "Ci sono novità, Holmes: pare che il testamento, andato distrutto nell'esplosione, fosse stato favorevole ad una delle sette 'amiche' del duca."

Holmes: "Ciò che è più strano, è che la bomba sia stata fabbricata appositamente per essere nascosta nell'armatura della camera da letto, il che fa supporre che *l'assassino abbia necessariamente fatto più di una visita al castello*."

Watson: "Ho interrogato personalmente le sette donne, ma ciascuna ha giurato di essere stata nel castello *una sola volta nella sua vita*. Dagli interrogatori risulta che:

(1) Ann ha incontrato Betty, Charlotte, Felicia e Georgia;
(2) Betty ha incontrato Ann, Charlotte, Edith, Felicia e Helen;
(3) Charlotte ha incontrato Ann, Betty e Edith;
(4) Edith ha incontrato Betty, Charlotte, Felicia;
(5) Felicia ha incontrato Ann, Betty, Edith, Helen;
(6) Georgia ha incontrato Ann e Helen;
(7) Helen ha incontrato Betty, Felicia e Georgia.

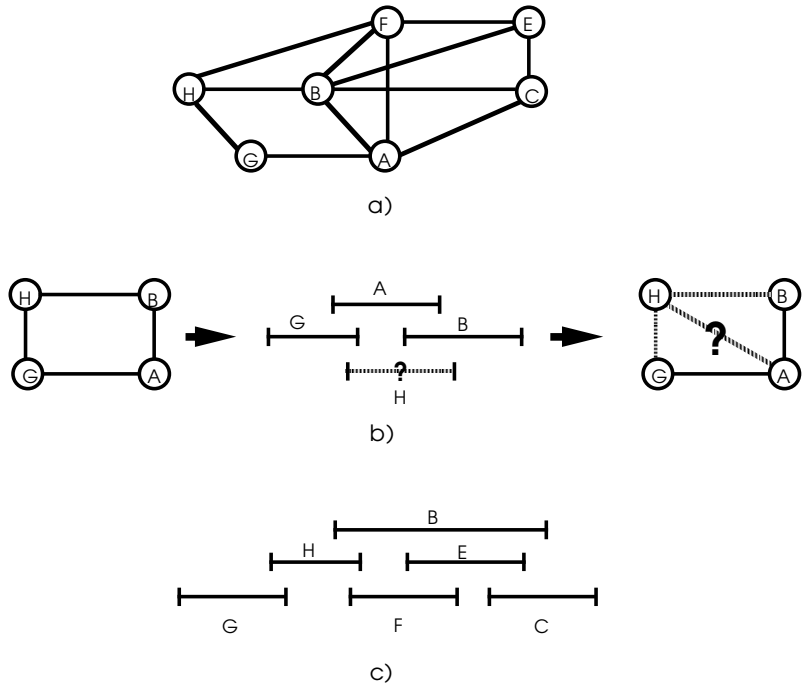

Figura 1.5: a) Il grafo degli incontri. b) Rappresentazione di circuiti di lunghezza 4 come intersezione di intervalli della retta. c) Possibili periodi di soggiorno al castello cancellando il nodo A.

Vedete, Holmes, che le testimonianze concordano. Ma chi sarà l'assassino?"
Holmes: "Elementare, mio caro Watson: ciò che mi avete detto individua inequivocabilmente l'assassino!"

Soluzione di Holmes: Il grafo degli incontri tra le donne, indicate con le loro iniziali, è disegnato nella Figura 1.5a). Se ogni donna avesse detto la verità, questo grafo dovrebbe rappresentare l'intersezione di 7 intervalli della retta reale (un grafo siffatto si dice *grafo di intervalli*). In tal caso, però, ogni circuito di lunghezza maggiore o uguale a 4 dovrebbe possedere un arco che collegasse due nodi non consecutivi nel circuito, come esemplificato nella Figura 1.5b). Il grafo degli incontri ha ben 3 circuiti che violano questa proprietà: (1) A, G, H, F, A, (2) A, G, A, B, A e (3) A, F, E, C, A. L'unico nodo che compare in tutti e tre i circuiti è il nodo A: l'assassino è dunque Ann! Infatti, cancellando A dal grafo, i periodi di soggiorno nel castello delle altre 6 donne potrebbero essere quelli illustrati nella Figura 1.5c). ■

1.2 Specifica

Poiché ogni grafo non orientato G può essere visto come un grafo orientato G', ottenuto da G sostituendo ogni arco $[u,v]$ con i due archi (u,v) e (v,u), sarà fornita una specifica solo per grafi orientati, che saranno chiamati semplicemente grafi. Le operazioni definite sui grafi prevedono la possibilità di inserire o cancellare nodi o archi, di ottenere l'insieme di tutti i nodi o dei nodi adiacenti ad un particolare nodo.

Di tutte queste procedure, discuteremo la realizzazione delle sole procedure adj() e V(); il motivo principale è che, in questo testo, i grafi sono usati per rappresentare relazioni tra oggetti che non mutano nel tempo. In altri termini, non si effettuano mai né cancellazioni né inserzioni, né di

GRAPH	
Graph( )	% Crea un grafo vuoto
insertNode(NODE u)	% Aggiunge il nodo u al grafo
insertEdge(NODE u, NODE v)	% Aggiunge l'arco (u,v) al grafo
deleteNode(NODE u)	% Rimuove il nodo u dal grafo
deleteEdge(NODE u, NODE v)	% Rimuove l'arco (u,v) nel grafo
SET adj(NODE u)	% Restituisce l'insieme dei nodi adiacenti ad u
SET V()	% Restituisce l'insieme di tutti i nodi

nodi né di archi, ma si esplorano sistematicamente grafi forniti in ingresso, al fine di determinarne sottoinsiemi di nodi e/o di archi che soddisfino certe proprietà.

Qualora il grafo sia dinamico, la struttura di dati più adatta per rappresentarlo dipende dalla particolare utilizzazione del grafo stesso. Realizzazioni efficienti di insiemi e dizionari sono state trattate ampiamente nei capitoli precedenti, ed il lettore può cercare di combinarle per esercizio in modo da fornire realizzazioni efficienti per le operazioni di inserzione e cancellazione di nodi e/o archi.

Inoltre, nella vita di un programmatore capita di lavorare con i grafi senza menzionarli esplicitamente. Ad esempio, un file system può essere visto come un albero di directory e file, ma la presenza di *hard link* e *symbolic link* fanno sì che l'albero sia in realtà un grafo. Le operazioni per la creazione di nodi e archi sono quindi condizionate dalla particolare "semantica" associata al grafo, e parleremo di creazione di file al posto di creazione di nodi.

Infine, la situazione può essere ulteriormente complicata dal fatto che in molte applicazioni ulteriori informazioni (dette *etichette*, o *pesi*) sono associate ai nodi e/o agli archi. In tal caso, si parla di grafi *etichettati* (o *pesati*), i quali richiedono ulteriori operatori, per reperire e/o modificare le informazioni associate a nodi e/o archi. Non è quindi un caso che nelle librerie dei più diffusi linguaggi di programmazione siano disponibili numerose realizzazioni per le strutture di dati insieme, dizionario, etc., ma che non vi sia una realizzazione standard per i grafi. Non staremo quindi noi ad inventarne una fittizia.

Nelle prossime sezioni discuteremo dei principali approcci per realizzare la struttura di dati grafo. Partiremo dall'ipotesi semplificativa che l'insieme statico degli n nodi possa essere rappresentato dai numeri da 1 a n. Per eseguire un'operazione su tutti i nodi e su tutti gli archi, utilizzeremo il costrutto **foreach** come segue:

```
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        fai un'operazione sull'arco (u, v)
```

Il primo **foreach** scorre tutti i nodi del grafo, mentre il secondo scorre tutti i nodi adiacenti al nodo u preso in considerazione. Per semplificare il testo inoltre, useremo il termine $G.n$ come abbreviazione di $G.\mathsf{V}().size()$.

1.3 Realizzazioni con matrici

Uno dei modi più semplici per rappresentare un grafo è basato su matrici. Dato un grafo $G = (V,E)$, la *matrice di adiacenza* $M = [m_{uv}]$ (detta anche *matrice di adiacenza nodi-nodi*) è una matrice di

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Figura 1.6: Matrice di adiacenza per il grafo non orientato di Figura 1.2.

dimensione $n \times n$, tale che:

$$m_{uv} = \begin{cases} 1, & \text{se } (u,v) \in E, \\ 0, & \text{se } (u,v) \notin E. \end{cases}$$

Se il grafo non è orientato, la matrice è simmetrica, ovvero $m_{uv} = m_{vu}$ per ogni u, v. Un esempio di matrice di adiacenza è mostrata nella Figura 1.6.

Con questa realizzazione è banale verificare in tempo $O(1)$ se un dato arco è presente nel grafo oppure no. Purtroppo, il tempo per ricavare l'insieme di adiacenza $G.\mathsf{adj}(u)$ di un qualsiasi nodo u è sempre $\Theta(n)$, indipendentemente dalla cardinalità di $G.\mathsf{adj}(u)$, perché occorre scandire un'intera riga della matrice alla ricerca degli elementi $m_{uv} = 1$. Inoltre, lo spazio di memoria occupato è $\Theta(n^2)$ e il tempo per esaminare tutti gli archi è $\Theta(n^2)$, anche se il grafo è sparso, ovvero contiene un numero m di archi che è $O(n)$. Se il grafo è pesato, allora nella matrice M si utilizzano i pesi degli archi al posto degli elementi binari. Se p_{uv} è il peso dell'arco (u,v), allora la matrice M diventa:

$$m_{uv} = \begin{cases} p_{uv}, & \text{se } (u,v) \in E, \\ +\infty \text{ (oppure } -\infty) & \text{se } (u,v) \notin E. \end{cases}$$

La *matrice di incidenza nodi-archi* è una rappresentazione su matrice alternativa, in cui si utilizza una matrice rettangolare B di dimensione $n \times m$, dove ciascuna riga rappresenta un nodo e ciascuna colonna rappresenta un arco, tale che:

$$b_{uk} = \begin{cases} -1, & \text{se l'arco } k\text{-esimo esce dal nodo } u, \\ +1, & \text{se l'arco } k\text{-esimo entra nel nodo } u, \\ 0, & \text{altrimenti.} \end{cases}$$

Se il grafo è non orientato, invece, si ha:

$$b_{uk} = \begin{cases} 1, & \text{se l'arco } k\text{-esimo è incidente nel nodo } u, \\ 0, & \text{altrimenti.} \end{cases}$$

Esempio (Matrice di incidenza nodi-archi) La matrice di incidenza del grafo orientato di Figura 1.1 è mostrata nella Figura 1.7(a) (si noti che ogni colonna ha esattamente un $+1$ e un -1), mentre quella del grafo non orientato di Figura 1.2 è mostrata nella Figura 1.7(b) (si noti che ogni colonna ha esattamente due 1). ■

La matrice di incidenza è molto utile per rappresentare un grafo in certi problemi di PROGRAMMAZIONE LINEARE, nei quali si cerca un vettore x di m incognite (reali o intere) associate agli archi che soddisfi un sistema di equazioni del tipo $Bx = c$, dove c è un vettore di termini noti interi.

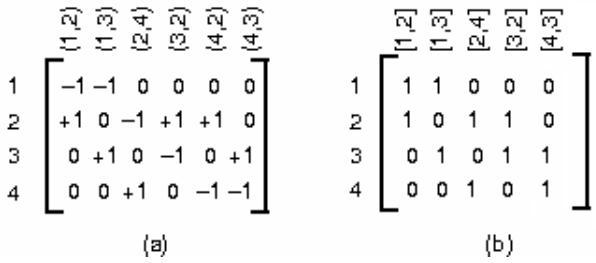

Figura 1.7: Matrici di incidenza nodi-archi: a) per il grafo orientato di Figura 1.1; b) per il grafo non orientato di Figura 1.2.

Purtroppo, con questa rappresentazione, lo spazio di memoria occupato è sempre $\Theta(nm)$. Inoltre, dato un nodo, non è agevole ricavare l'insieme di adiacenza. Infatti per calcolare un qualsiasi $G.\mathsf{adj}(u)$ occorre scandire la riga u di B alla ricerca delle colonne k con elementi $b_{uk} = -1$ e per ciascuna colonna k scandirla per individuare l'indice di riga v tale che $b_{vk} = +1$, per un totale di $\Theta(nm)$ tempo.

1.4 Realizzazioni con liste di adiacenza

È spesso conveniente mantenere esplicitamente gli insiemi di adiacenza per ogni nodo del grafo. Una possibilità consiste nell'utilizzare le cosiddette *liste di adiacenza*. Questa realizzazione è analoga a quella con liste di trabocco viste per i dizionari. Si usa un vettore di dimensione n dove l'u-esima posizione del vettore contiene l'identificativo di una lista, di solito realizzata con puntatori, che memorizza gli elementi di $G.\mathsf{adj}(u)$. Lo spazio di memoria è $\Theta(n)$ per il vettore e $\Theta(|G.\mathsf{adj}(u)|)$ per ciascuna lista, per un totale di $\Theta(n+\sum_{u\in V}|G.\mathsf{adj}(u)|) = \Theta(n+m)$, che è ottimo. Se il grafo è sparso, cioè m è $O(n)$, lo spazio risulta quindi essere $\Theta(n)$ anziché $\Theta(n^2)$. Inoltre, la scansione di ciascun insieme di adiacenza $G.\mathsf{adj}(u)$ richiede tempo ottimo $\Theta(|G.\mathsf{adj}(u)|)$ mentre la scansione di tutti gli archi del grafo richiede tempo ottimo $\Theta(n+m)$. Purtroppo, anche per verificare se $(u,v) \in E$ occorre una scansione di $G.\mathsf{adj}(u)$ che richiede $O(|G.\mathsf{adj}(u)|)$ tempo.

Nell'ipotesi semplificativa che non vengano effettuate né inserzioni né cancellazioni di nodi e archi, è possibile rappresentare gli insiemi di adiacenza senza utilizzare liste, ma utilizzando invece un *vettore di adiacenza* (di dimensione fissa) per ognuno dei nodi u del grafo, contenente esattamente i nodi adiacenti ad u.

> **Esempio (Vettori di adiacenza)** I vettori di adiacenza che rappresentano il grafo di Figura 1.2 sono mostrati nella Figura 1.8. Si noti che, essendo il grafo non orientato, ogni arco $[u,v]$ è rappresentato due volte, poiché $u \in G.\mathsf{adj}(v)$ e $v \in G.\mathsf{adj}(u)$. ■

Lo spazio di memoria occupato dai vettori di adiacenza è $\Theta(n+m)$, che è ottimo in ordine di grandezza come nel caso delle liste di adiacenza. Rispetto alle liste di adiacenza, però, c'è un ulteriore risparmio dovuto all'assenza dei campi per i puntatori nelle celle delle liste. Se il grafo è pesato, i pesi sui nodi e/o sugli archi possono essere memorizzati ad esempio aggiungendo vettori simili (un vettore di dimensione n per i pesi sui nodi, un vettore di vettori per i pesi sugli archi).

1.5 Esplorazione di un grafo

Una *visita* di un grafo è una procedura sistematica per esplorare un grafo, visitando almeno una volta ogni suo nodo ed arco.

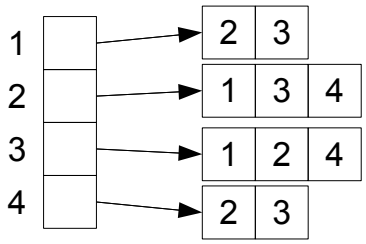

Figura 1.8: Vettori di adiacenza per il grafo non orientato di Figura 1.2.

Esempio I motori di ricerca utilizzano i cosiddetti "web spider" per raccogliere i dati necessari a rispondere alle richieste dell'utente. Un web spider visita il grafo del Web, in cui le pagine web sono nodi e i link in esse contenuti sono archi. ■

Genericamente, la visita di un grafo può essere descritta dalla procedura visit().

```
visit(GRAPH G, NODE r)
SET S = Set()
S.insert(r)
{ marca il nodo r come "scoperto" }
while S.size() > 0 do
    NODE u = S.remove()
    { esamina il nodo u }
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u,v) }
        if v non è già stato scoperto then
            { marca il nodo v come "scoperto" }
            S.insert(v)
```

L'insieme S contiene i nodi che sono stati scoperti, ma non ancora visitati. I nodi vengono estratti da S uno dopo l'altro per essere visitati, e nel contempo si esaminano tutti gli archi uscenti dal nodo visitato per scoprire nuovi nodi. Grazie al controllo effettuato nel costrutto **if**, ogni nodo del grafo è inserito nell'insieme S una sola volta; poiché per ogni nodo v, l'insieme $G.\mathsf{adj}(v)$ viene scandito una sola volta, questo algoritmo è ottimo e ha complessità $\Theta(n+m)$. Nel caso di grafi orientati e fortemente connessi, o non orientati e connessi, la visita parte da un generico nodo r e raggiunge tutti i nodi in una singola "passata". Nei grafi non connessi o non fortemente connessi, è necessario invece fare più passate, ovvero far ripartire l'algoritmo da un nodo non scoperto in precedenza.

L'ordine in cui vengono visitati i nodi dipende dalla politica di estrazione della procedura remove(). Esistono due approcci principali:

- *Breadth-First-Search* (BFS), ovvero visita in ampiezza (detta anche a ventaglio);
- *Depth-First-Search* (DFS), ovvero visita in profondità (detta anche a scandaglio).

1.5.1 Visita BFS

Nella visita BFS, i nodi sono visitati in ordine di distanza crescente dal nodo di partenza r, dove la distanza da r ad un generico nodo u è il minimo numero di archi in un cammino da r a u. Tale

metodo di visita è un'estensione di una visita per livelli di un albero radicato, in cui i figli di un nodo sono visitati dopo aver visitato tutti gli altri nodi che stanno allo stesso livello del padre.

Poiché la BFS visita i nodi più vicini ad r prima di quelli lontani, l'insieme S è una coda (con politica di rimozione FIFO). Il processo di marcatura è realizzato con un vettore di booleani; la posizione corrispondente al nodo u viene inizializzata a **false**, per diventare **true** quando il nodo viene "scoperto" durante la visita.

```
bfs(GRAPH G, NODE r)
    QUEUE S = Queue()
    S.enqueue(r)
    boolean[] visited = new boolean[1...G.n]
    foreach u ∈ G.V() − {r} do visited[u] = false
    visited[r] = true
    while not S.isEmpty() do
        NODE u = S.dequeue()
        { esamina il nodo u }
        foreach v ∈ G.adj(u) do
            { esamina l'arco (u,v) }
            if not visited[v] then
                visited[v] = true
                S.enqueue(v)
```

Esempio (BFS) L'ordine di visita BFS dei nodi e degli archi del grafo mostrato nella Figura 1.9, assumendo di partire dal nodo $r = 1$ e che gli insiemi $\mathsf{adj}(v)$ siano memorizzati in ordine crescente, è il seguente, dove per chiarezza i nodi sono evidenziati in grassetto:
1, (1,2), (1,3), (1,4), (1,6), (1,8), **2**, (2,1), (2,4), (2,5), (2,7), (2,9), **3**, (3,1), (3,6), **4**, (4,1), (4,2), (4,5), **6**, (6,1), (6,3), **8**, (8,1), (8,7), **5**, (5,2), (5,4), **7**, (7,2), (7,8), (7,9), **9**, (9,2), (9,7). ■

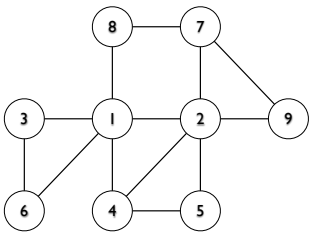

Figura 1.9: Un grafo non orientato

1.5.2 Applicazione BFS: il numero di Erdős

La storia della matematica è piena di personaggi eccentrici; Paul Erdős è uno di questi. Fu un matematico estremamente prolifico, che arrivò a pubblicare circa 1500 articoli, con più di 500 collaboratori diversi. Gran parte della sua carriera fu spesa con una valigia in mano, passando da un dipartimento di matematica all'altro, collaborando con chi fosse disposto ad ospitarlo.

Come tributo scherzoso alla sua estrema prolificità, è nato il concetto di "numero di Erdős", un coefficiente assegnato ad ogni matematico (in generale, ad ogni autore scientifico) per misurare

la sua "distanza" da Erdős. Paul Erdős ha *distance* $= 0$. I co-autori dei suoi articoli hanno *distance* $= 1$. Chi ha scritto articoli in collaborazione con autori con *distance* $= 1$ (ma non con Erdős) ha *distance* $= 2$. Ricorsivamente, chi ha collaborato con autori con *distance* $= k$ (ma non con autori con *distance* $< k$), ha *distance* $= k+1$. Chi non è raggiunto da questa definizione, ha *distance* $= \infty$. Il massimo numero di Erdős finito è 15; è scherzosamente considerato un punto d'onore avere un numero di Erdős basso. Gli autori di questo libro hanno *distance* 3 e 4, rispettivamente.

Dato il grafo delle collaborazioni scientifiche (dove ogni autore è un nodo e due autori sono uniti da un arco se e solo se hanno scritto un articolo scientifico insieme), è possibile calcolare il numero di Erdős di ogni autore utilizzando una visita in ampiezza a partire da Erdős.

```
distance(GRAPH G, NODE r, int[] distance, NODE[] p)
    QUEUE S = Queue()
    S.enqueue(r)
    foreach u ∈ G.V() − {r} do distance[u] = ∞
    distance[r] = 0
    p[r] = nil
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do                          % Esamina l'arco (u, v)
            if distance[v] = ∞ then                      % Il nodo u non è già stato scoperto
                distance[v] = distance[u] + 1
                p[v] = u
                S.enqueue(v)
```

In questa versione della BFS, invece di utilizzare un vettore di booleani per la marcatura, utilizziamo il vettore *distance* di interi, in cui i nodi non ancora scoperti hanno numero di Erdős pari a ∞. Il nodo radice (Erdős) viene inizializzato a zero; ogni volta che viene analizzato un arco (u,v) e il nodo v non è ancora stato scoperto, *distance*$[v]$ viene posto uguale a *distance*$[u]+1$, come da definizione. Al termine, tutti i nodi non raggiunti dalla singola passata conservano (correttamente) il valore ∞.

1.5.3 Alberi di copertura BFS

Oltre al valore *distance*$[s]$ di un autore s, può essere interessante ottenere anche il cammino di lunghezza *distance*$[s]$ che porta da r a s (o meglio, uno dei cammini, visto che è possibile che ve ne sia più di uno). Per ottenere questa informazione, è possibile utilizzare l'*albero dei cammini* BFS, che è un albero di copertura del grafo G radicato in r. In questo albero, i figli di un nodo u sono tutti i nodi v scoperti durante la visita di archi (u,v). Il cammino nell'albero che unisce il nodo r ad un nodo s è ovviamente un cammino anche nel grafo, ed ha lunghezza *distance*$[s]$.

La procedura distance() memorizza l'albero dei cammini BFS in un vettore di padri p, in cui $p[v]$ contiene u se v è stato scoperto durante la visita del nodo u e in particolare dell'arco (u,v). Il vettore p è passato in input alla procedura distance(), con ogni elemento del vettore inizializzato a **nil**.

Una volta ottenuto il vettore p, è possibile ottenere il cammino dal nodo radice r ad un generico nodo s tramite la procedura printPath(). Si noti l'interessante uso della ricorsione, che permette di stampare gli autori nell'ordine atteso da r ad s.

```
printPath(NODE r, NODE s, NODE[] p)
if r = s then
    print s
else if p[s] = nil then
    print "nessun cammino da r a s"
else
    printPath(r,p[s],p)
    print s
```

1.5.4 Visita DFS

Il metodo di visita DFS è una diretta estensione della visita in ordine anticipato di un albero ordinato. Quando viene visitato un nuovo nodo u, ci si allontana da esso il più possibile lungo un cammino (o catena), visitando nodi (e archi) nel cammino fino a giungere in un "vicolo cieco", ovvero in un nodo v i cui nodi adiacenti sono già stati tutti visitati. Quando si è raggiunto v si torna indietro lungo l'ultimo arco visitato e si riprende la visita allontanandosi lungo un altro cammino di nodi (e archi) non ancora visitati. Questa tecnica di visita è un'applicazione della tecnica backtrack [cfr. Capitolo 16].

Nella visita DFS, l'insieme S potrebbe essere realizzato tramite una pila (politica di rimozione LIFO). È più naturale invece descrivere l'algoritmo in maniera ricorsiva, in quanto più coinciso ed elegante. L'algoritmo viene richiamato su un grafo G a partire da un nodo u, mentre *visited* è il vettore di booleani che rappresenta l'insieme di nodi già scoperti (inizializzato a tutti **false**). Come negli alberi, è possibile distinguere fra una previsita (riga (1)) e una postvisita (riga (2)).

```
dfs(GRAPH G, NODE u, boolean[] visited)
    visited[u] = true
(1) { esamina il nodo u (caso previsita) }
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u,v)
        if not visited[v] then
            dfs(G,v,visited)
(2) { esamina il nodo u (caso postvisita) }
```

Esempio (DFS) L'ordine di visita DFS dei nodi e degli archi del grafo di Figura 1.9, assumendo di partire dal nodo $r = 1$ e che gli insiemi adj(v) siano memorizzati in ordine crescente, è il seguente, dove per chiarezza i nodi sono evidenziati in grassetto:
1, $(1,2)$, **2**, $(2,1)$, $(2,4)$, **4**, $(4,1)$, $(4,2)$, $(4,5)$, **5**, $(5,2)$, $(5,4)$, $(2,5)$, $(2,7)$, **7**, $(7,2)$, $(7,8)$, **8**, $(8,1)$, $(8,7)$, $(7,9)$, **9**, $(9,2)$, $(9,7)$, $(2,9)$, $(1,3)$, **3**, $(3,1)$, $(3,6)$, **6**, $(6,1)$, $(6,3)$, $(1,4)$, $(1,6)$, $(1,8)$. ■

1.5.5 Applicazioni DFS: componenti connesse

Modificando opportunamente la DFS appena introdotta, è possibile risolvere in tempo $\Theta(n+m)$ molti problemi fondamentali relativi alla connessione dei grafi, quali ad esempio (i) verificare se un grafo non orientato G è connesso oppure no, oppure (ii) trovare le componenti connesse di cui G è composto.

La soluzione del primo problema è banale: un grafo è connesso se, al termine della visita DFS, tutti i nodi sono stati marcati. Se questo non accade, il grafo è composto da più componenti connesse e si passa al secondo problema. In questo caso, una singola passata non è sufficiente ed è

quindi necessario far ripartire la visita da un nodo non marcato, scoprendo una nuova porzione del grafo. Vediamo quindi l'algoritmo per identificare le componenti connesse.

```
int[] cc(GRAPH G, STACK S)
    int[] id = new int[1...G.n]
    foreach u ∈ G.V() do id[u] = 0
    int conta = 0
    while not S.isEmpty() do
        u = S.pop()
        if id[u] = 0 then
            conta = conta + 1
            ccdfs(G, conta, u, id)
    return id

 ccdfs(GRAPH G, int conta, NODE u, int[] id)
    id[u] = conta
    foreach v ∈ G.adj(u) do
        if id[v] = 0 then
            ccdfs(G, conta, v, id)
```

Il vettore *id* contiene, per ogni elemento u, un intero che identifica la componente connessa a cui u appartiene. Il contatore *count* conta il numero di componenti connesse scoperte finora; ogni volta che viene trovato un nodo u non ancora scoperto ($id[u] = 0$), si incrementa di uno tale contatore e si inizia una nuova visita DFS a partire da u, assegnando a tutti i nodi scoperti finora il valore di *count*.

Questa versione di cc() prende in input anche una pila contenente una permutazione dei nodi di G; i nodi da cui far partire la visita vengono quindi scelti nell'ordine di estrazione dallo pila. Un simile comportamento non è necessario per il buon funzionamento dell'algoritmo, ma verrà utile in seguito per identificare le componenti fortemente connesse di un grafo orientato.

Esempio (Componenti connesse) Il grafo di Figura 1.3 è formato da tre componenti connesse. Eseguendo la procedura cc() l'ordine di visita dei nodi è 1, 2, 5, 6, 3, 7, 8, 4 ed alla fine il vettore *id* contiene: $id[1] = 1$, $id[2] = 1$, $id[3] = 2$, $id[4] = 3$, $id[5] = 1$, $id[6] = 1$, $id[7] = 2$, $id[8] = 2$. ■

1.5.6 Alberi di copertura DFS

Così come nelle visite BFS, è possibile definire l'*albero dei cammini* T generato dalla visita DFS. Tutte le volte che viene incontrato un arco che connette un nodo marcato ad uno non marcato, esso viene inserito nell'albero T. Qualora non tutti i nodi siano raggiungibili dalla radice (ovvero dal nodo di partenza r), perché il grafo non è (fortemente) connesso, allora una visita completa del grafo si ottiene richiamando iterativamente la DFS, come mostrato nella procedura cc. In tal caso, la visita forma una *foresta* di copertura DFS. Se il grafo non è orientato, la visita provoca un orientamento arbitrario (u,v) oppure (v,u) di ciascun arco $[u,v]$, dovuto alla direzione in cui è esaminato, mentre se il grafo è orientato viene rispettato l'orientamento.

Gli archi non inclusi in T possono essere divisi in tre categorie durante la visita:

– se l'arco è esaminato passando da un nodo di T ad un altro nodo che è suo antenato in T, è detto *arco all'indietro*;

– se l'arco è esaminato passando da un nodo di T ad un suo discendente (che non sia figlio) in T è detto *arco in avanti*;
– altrimenti, è detto *arco di attraversamento*.

Se il grafo è non orientato, ciascun arco sarà o un arco di T o un arco all'indietro. Se il grafo è orientato, tutte le quattro classi sono possibili.

Esempio (Albero di copertura DFS) La Figura 1.10 mostra un grafo orientato G e la corrispondente partizione degli archi ottenuta eseguendo una DFS a partire dal nodo 1. L'ordine di visita dei nodi e degli archi è: **1**, $(1,2)$, **2**, $(2,3)$, **3**, $(3,1)$, $(1,3)$, $(1,4)$, **4**, $(4,2)$, $(4,3)$. Gli archi $(1,2)$, $(2,3)$ e $(1,4)$ sono in T, l'arco $(1,3)$ è in avanti, l'arco $(3,1)$ è all'indietro, e gli archi $(4,2)$ e $(4,3)$ sono di attraversamento. ■

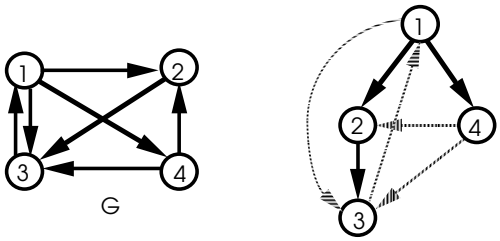

Figura 1.10: Un grafo orientato e la partizione degli archi indotta da una DFS.

La distinzione tra le quattro classi di archi per i grafi orientati (due per quelli non orientati) è importante perché permette di definire uno schema generale di visita dfs-schema(), che può essere poi particolarizzato per risolvere problemi specifici.

Nello schema si fa uso di due vettori *dt* e *ft* (rispettivamente *discovery time* and *finish time*) di dimensione n e di un intero *time* con le funzioni di orologio discreto; per semplicità assumiamo che queste tre variabili siano globali. La variabile *time* è inizializzata a 0 ed è incrementata di uno sia all'inizio che alla fine della procedura DFS. Tutte le posizioni dei vettori *dt* e *ft* sono inizializzate a 0 (a significare che la visita non è ancora iniziata); durante la visita, nelle posizioni $dt[u]$ e $ft[u]$ sono registrati i valori dell'orologio, rispettivamente, all'inizio e alla fine della procedura di visita sul nodo u.

Confrontando i valori di $dt[u]$, $dt[v]$ e $ft[v]$ è facile stabilire il tipo dell'arco (u,v). Se $dt[v] = 0$, allora la visita del nodo v non è ancora iniziata, e quindi (u,v) è in T; altrimenti, se $dt[v] \neq 0$, allora (u,v) non è in T. In quest'ultimo caso, se $dt[u] > dt[v]$ e $ft[v] = 0$, la visita di u è iniziata dopo quella di v, la quale però non è ancora terminata, e pertanto (u,v) è all'indietro; quando invece $dt[u] < dt[v]$ e $ft[v] \neq 0$, la visita di v è sia iniziata che finita dopo quella di u, che però non è ancora terminata, e quindi (u,v) è in avanti; nei rimanenti casi (u,v) è di attraversamento.

Per concludere, nelle prossime sezioni accenniamo alle risoluzioni di alcuni semplici problemi, ottenibili modificando opportunamente lo schema proposto.

1.5.7 Applicazione schema DFS: grafo aciclico

Si consideri il problema di verificare se un grafo $G = (V,E)$ è aciclico. Un grafo è aciclico se e solo se non esistono archi all'indietro. Infatti, se esiste un arco all'indietro (u,v), dove v è un antenato di u, allora esiste un cammino da v a u e un arco da u a v, ovvero un ciclo. Se esiste un ciclo, sia v il primo nodo del ciclo che viene visitato e sia (u,v) un arco del ciclo. Il cammino che connette v ad u verrà prima o poi visitato, e da u verrà scoperto l'arco all'indietro (u,v). Adattando opportunamente lo schema di cui sopra, si ottiene l'algoritmo hasCycle() che restituisce **true** se

```
dfs-schema(GRAPH G, NODE u)
    esamina il nodo u prima (caso pre-visita)
    time = time + 1;  dt[u] = time
    foreach v ∈ G.adj(u) do
        esamina l'arco (u,v) di qualsiasi tipo
        if dt[v] = 0 then
            esamina l'arco (u,v) in T
            dfs-schema(G,v)
        else if dt[u] > dt[v] and ft[v] = 0 then
            esamina l'arco (u,v) all'indietro
        else if dt[u] < dt[v] and ft[v] ≠ 0 then
            esamina l'arco (u,v) in avanti
        else
            esamina l'arco (u,v) di attraversamento
    esamina il nodo u dopo (caso post-visita)
    time = time + 1;  ft[u] = time
```

il grafo contiene un ciclo. Si noti che non appena viene trovato un arco all'indietro, la procedura termina restituendo **true** e chiudendo tutte le chiamate ricorsive.

```
boolean hasCycle(GRAPH G, NODE u)
    time = time + 1;  dt[u] = time
    foreach v ∈ G.adj(u) do
        if dt[v] = 0 then
            if hasCycle(G,v) then return true
        else if dt[u] > dt[v] and ft[v] = 0 then
            return true
    time = time + 1;  ft[u] = time
    return false;
```

1.5.8 Applicazione schema DFS: Ordinamento topologico

In alcuni casi, è possibile dimostrare la correttezza di un algoritmo in base ai tempi di inizio e di fine, senza doverli effettivamente calcolare! Come esempio, si consideri il problema dell'ordine *ordine topologico* (o linearizzazione) di un grafo orientato (non necessariamente connesso), che consiste nell'identificare un ordinamento dei nodi tale che se (u,v) è un arco del grafo, allora u precede v nell'ordinamento.

Esempio (Ristrutturazione) Ristrutturare casa è un compito improbo. Bisogna coordinare diverse figure professionali (muratori, idraulici, elettricisti, piastrellisti, imbianchini, etc.), ognuna delle quali è impegnata in varie attività (costruire un nuovo muro – creare le tracce per l'idraulico e l'elettricista – riempire le tracce con tubature e canalette – inserire i fili nelle canalette – ricoprire le tracce – imbiancare – applicare le piastrelle – applicare rubinetteria, ecc.). Il coordinamento di queste attività è un problema di ordine topologico: dato il grafo delle dipendenze tra attività, in quale ordine queste attività vanno eseguite? ■

Innanzitutto, dobbiamo chiederci se tutti i grafi hanno un ordine topologico. La risposta è no: solo i grafi aciclici possono essere ordinati topologicamente. Consideriamo infatti un qualsiasi ordinamento, e supponiamo per assurdo che esista un ciclo. Sia v_j il primo nodo del ciclo a comparire nell'ordinamento; per definizione, esiste un nodo v_i che precede v_j nel ciclo; esiste quindi un arco $(v_i, v_j) \in E$. Per definizione di ordine topologico, v_i deve comparire prima di v_j nell'ordine topologico, ma questo contraddice la definizione di v_j. L'implicazione inversa (dato un grafo acicliclo G, esiste sempre un ordinamento topologico di esso) verrà dimostrata in maniera costruttiva, proponendo un algoritmo che costruisce tale ordinamento.

La procedura topsort(), coadiuvata da ts-dfs(), costruisce un ordine topologico eseguendo una visita DFS che calcola l'istante $ft[u]$ in cui finisce la visita di ciascun nodo u e ordinando i nodi per istanti di fine decrescenti. È interessante notare che questo algoritmo sfrutta il concetto di tempo di scoperta introdotto nel Capitolo 1.5.6, ma in realtà non richiede che questo valore debba essere calcolato esplicitamente. È facile provare che in un grafo aciclico non esistono archi all'indietro, ed è pertanto agevole dimostrare che se (u,v) è un arco, allora $ft[v] < ft[u]$.

Anzichè calcolare il tempo di fine, quindi, è sufficiente memorizzare il nodo u in una pila S, presa in input, in modo che al momento dell'estrazione i nodi vengano analizzati in ordine inverso di tempo di fine.

```
topsort(GRAPH G, STACK S)
    boolean[] visited = boolean[1...G.n]
    foreach u ∈ G.V() do visited[u] = false
    foreach u ∈ G.V() do
        if not visited[u] then
            ts-dfs(G, u, visited, S)

ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK S)
    visited[u] = true
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            ts-dfs(G, v, visited, S)
    S.push(u)
```

1.5.9 Applicazione schema DFS: componenti fortemente connesse

In alcuni casi, è possibile dimostrare la correttezza di un algoritmo in base ai tempi di inizio e di fine, senza doverli effettivamente calcolare! Come esempio, si consideri il problema di identificare le componenti fortemente connesse di un grafo orientato. Vediamo ora un altro algoritmo che non calcola esplicitamente i tempi di inizio e di fine; tali concetti, tuttavia, vengono sfruttati pesantemente per dimostrare la correttezza dell'algoritmo. Il problema considerato è quello dell'identificazione delle componenti fortemente connesse di un grafo orientato.

L'algoritmo cc() visto nel Capitolo 1.5.5 non risolve questo specifico problema, perché il risultato dipende dall'ordine in cui i nodi vengono visitati. Ad esempio, se la visita del grafo in Figura 1.1 inizia nel nodo 2, cc() restituirà correttamente due componenti connesse $\{2,3,4\}$ e $\{1\}$; partendo dal nodo 1, restituirà invece un'unica componente $\{1,2,3,4\}$. Questo perché dalla componente $\{1\}$ è possibile raggiungere $\{2,3,4\}$ ma non viceversa.

Un algoritmo del 1978 attribuito a Kosaraju funziona nel modo seguente: (i) esegue una visita in profondità del grafo G utilizzando l'algoritmo topsort(); (ii) calcola il grafo trasposto G^T di G; (iii) esegue l'algoritmo cc() per ottenere le componenti del grafo G^T, ma nel ciclo principale

seleziona i nodi in ordine inverso di tempo di fine rispetto alla prima visita. Ciascuno dei tre passi richiede tempo $O(m+n)$.

Dato un grafo $G=(V,E)$, il *grafo trasposto* $G^T=(V,E^T)$ è formato dagli stessi nodi, mentre gli archi hanno direzioni invertite: i.e, $E^T=\{(u,v)|(v,u)\in E\}$.

L'algoritmo è realizzato dalla procedura scc(). È interessante notare che i tempi di inizio e di fine non vengono calcolati esplicitamente; al loro posto, i nodi vengono inseriti in una pila al termine della prima chiamata. La pila verrà passata alla procedura cc(), che la utilizzerà come ordine di visita.

```
int[] scc(GRAPH G)
    STACK S = topsort(G)                          % Prima visita
    GRAPH G^T = Graph()                           % Calcolo grafo trasposto
    foreach u ∈ G.V() do G^T.insertNode(u)
    foreach u ∈ G.V() do
        foreach v ∈ G.adj(u) do
            G^T.insertEdge(v,u)
    return cc(G^T,S)                              % Seconda visita
```

Questo algoritmo si basa sull'osservazione che un grafo G e il suo trasposto G^T hanno le stesse componenti fortemente connesse; questo deriva dalla simmetria della definizione, che prevede, per ogni coppia di nodi u,v, che esista almeno un cammino da u a v e almeno un cammino da v ad u.

Per capire il funzionamento, è utile considerare il *grafo delle componenti* $G^c=(V^c,E^c)$ del grafo G, definito come segue:

- $V^c=\{C_1,C_2,\ldots,C_k\}$, dove C_i è l'i-esima componente connessa di G;
- $E^c=\{(C_i,C_j):\exists(u_i,u_j)\in E:u_i\in C_i,u_j\in C_j\}$

In altre parole, le componenti sono "contratte" in un singolo vertice, e così gli archi che vanno da componente a componente. È facile notare che il grafo delle componenti è aciclico: se così non fosse, i nodi delle componenti appartenenti al ciclo potrebbero tutti raggiungersi l'uno con l'altro; farebbero quindi parte di un'unica componente, una contraddizione.

Estendiamo il concetto di tempo di inizio e di fine per le componenti. Sia C un sottoinsieme di V; definiamo $dt(C)$ come il primo tempo di scoperta e $ft(C)$ come l'ultimo tempo di completamento:

$$dt(C)=\min\{dt[u]:u\in C\}$$
$$ft(C)=\max\{ft[u]:u\in C\}$$

In base a queste definizioni, possiamo enunciare il seguente teorema.

Teorema 1.1 Siano C e C' due componenti distinte nel grafo orientato $G=(V,E)$. Supponiamo che esista un arco $(C,C')\in E^c$. Allora $ft(C)>ft(C')$.

Dimostrazione

Sono dati due casi: viene scoperto prima un nodo di C oppure di C'.

- $dt(C)<dt(C')$: sia x il primo nodo scoperto in C, all'istante $dt(C)$. Poiché tutti i nodi in $C\cup C'$ sono raggiungibili da x e non ancora scoperti, la loro visita inizierà e terminerà prima della terminazione della visita di x. Quindi $ft(C)=ft(x)>ft(y)$, per ogni $y\in C\cup C'-\{x\}$. Ne segue che $ft(C)>ft(C')$.
- $dt(C')<dt(C)$: sia y il primo nodo scoperto in C'; tutti i nodi in C' sono raggiungibili da y e non ancora scoperti. Al termine della visita di tutti i nodi di C', tutti i nodi di C non sono ancora stati scoperti. La loro visita inizierà e terminerà dopo $ft(C')$; ne segue che $ft(C)>ft(C')$. ■

Questo teorema può essere letto nel modo seguente: l'ultimo nodo della componente C ad essere stato visitato verrà inserito nella pila prima dell'ultimo nodo della componente C'. Poiché la pila viene svuotata in ordine inverso, questo significa che almeno un nodo di C verrà visitato prima dei nodi di C' nella seconda visita. A questo punto entra in gioco il grafo trasposto: se in G^c esiste un arco che connette C con C', in G^{Tc} esso è sostituito da un arco che connette C' con C: nella seconda visita quindi i nodi di C' non sono raggiungibili dai nodi di C. C verrà quindi individuata come una componente a sé stante.

1.6 Reality check

Le applicazioni dei grafi sono innumerevoli. All'ordinamento topologico su grafi orientati aciclici (in inglese *directed acyclic graph*, o DAG), ad esempio, è addirittura dedicato un programma di utilità specifico: `tsort`, presente in molti sistemi Unix e nelle GNU Coreutils. `tsort` prende in input la descrizione di un grafo come questo:

```
3 8
3 10
5 11
7 8
7 11
8 9
11 2
11 9
11 10
```

in cui ogni coppia di numeri identifica un arco, e va letta come "il primo numero precede il secondo", e restituisce il seguente ordinamento topologico: `3 5 7 11 8 10 2 9`. Questo algoritmo è usato poi come punto di partenza per programmi di utilità quali GNU `make` e Apache `ant`, che permettono la compilazione automatica di progetti software di grandi dimensioni. `make` e `ant` costruiscono un ordine topologico di tutte le "regole" (rules) per la compilazione del progetto; se esiste un ciclo, essi si bloccano. L'ordinamento topologico viene anche utilizzato per risolvere le dipendenze fra librerie nei gestori di pacchetti software dei sistemi Linux, nei compilatori per determinare l'ordine ottimale delle istruzioni e negli spreadsheet per aggiornare i valori calcolati tramite formule.

1.7 Esercizi

Esercizio 1.1 (Occupazione di memoria)**.** Sia $G = (V, E)$ un grafo orientato con n vertici ed m archi. Si supponga che puntatori e interi richiedano d bit. Indicare per quali valori di n ed m la rappresentazione a liste (monodirezionali) di adiacenza occupa meno memoria della matrice di adiacenza.

Esercizio 1.2 (DFS iterativa)**.** Scrivere una versione iterativa della DFS basata su pila dove l'ordine di visita dei nodi e degli archi sia lo stesso di quello della versione ricorsiva descritta nel Capitolo 1.5.4 ed eseguirla sul grafo dell'Esempio (2).

Esercizio 1.3 (Grafo bipartito)**.** Un grafo non orientato G è *bipartito* se l'insieme dei nodi può essere partizionato in due sottoinsiemi tali che nessun arco connette due nodi appartenenti alla stessa parte. G è 2-colorabile se ogni nodo può essere colorato di bianco o di rosso in modo che nodi connessi da archi siano colorati con colori distinti. Si dimostri che G è bipartito: (1) se e solo se è 2-colorabile; (2) se e solo se non contiene circuiti di lunghezza dispari.

Esercizio 1.4 (Grafo bipartito)**.** Scrivere una procedura basata su una visita BFS per verificare se un grafo non orientato è bipartito oppure no.

Esercizio 1.5 (Larghezza albero radicato)**.** Si modifichi una visita BFS per calcolare la larghezza di un albero radicato [cfr. Esercizio 5.3].

Esercizio 1.6 (Albero libero, non orientato)**.** Si scriva una procedura per verificare se un grafo non orientato è un albero libero.

Esercizio 1.7 (Scheduling)**.** Siano dati n programmi $P_1, P_2, \ldots, P_n$ che richiedono rispettivamente tempo $t_1, t_2, \ldots, t_n$ per essere eseguiti. Tra i programmi è definita una relazione $<$ di precedenza aciclica tale che $P_i < P_j$ se l'esecuzione di P_i deve essere completata prima dell'inizio dell'esecuzione di P_j. Si fornisca una procedura per determinare in tempo polinomiale l'istante in cui iniziare l'esecuzione di ciascun programma in modo da rispettare i vincoli di precedenza e terminare l'esecuzione di tutti i programmi il prima possibile.

Esercizio 1.8 (Massima sovrapposizione di intervalli)**.** Siano dati n intervalli distinti $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ della retta reale. Si progetti un algoritmo di complessità $O(n \log n)$ per trovare il più grande sottoinsieme di intervalli che si intersecano tutti tra di loro (nel corrispondente grafo di intervalli, il problema consiste nel trovare un sottoinsieme di cardinalità massima di nodi tale che per ogni coppia di nodi esista un arco che li connetta).

Esercizio 1.9 (Distanza massima)**.** Dato in input un grafo orientato $G = (V, E)$ rappresentato tramite liste di adiacenza e un nodo $r \in V$, restituire il numero dei nodi in G raggiungibili da r che si trovano alla massima distanza da r. Analizzare la complessità.

Esercizio 1.10 (Grafo quadrato)**.** 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 solo se $\exists u : (v, u) \in E \wedge (u, w) \in E$. In altre parole, se esiste un percorso di due archi fra i nodi u e w. Scrivere un algoritmo che, dato un grafo G rappresentato con matrice d'adiacenza, restituisce il grafo G^2.

Esercizio 1.11 (Diametro)**.** Il *diametro* di un grafo è la lunghezza del "più lungo cammino breve", ovvero la più grande lunghezza del cammino minimo (in termini di numero di archi) fra tutte le coppie di nodi. Progettare un algoritmo che misuri il diametro di un grafo e valutare la sua complessità.

1.8 Soluzioni

Soluzione Esercizio 1.1

Per rappresentare la matrice, abbiamo bisogno di n^2 bit; per rappresentare le liste di adiacenza, abbiamo bisogno di n puntatori di inizio lista (d bit) più m strutture dati con $2d$ bit (d per il puntatore al successore della lista e d per il puntatore al nodo della lista). Le liste di adiacenza occupano meno memoria della matrice quando $n^2 > 2dm + dn$, ovvero quando $m < \frac{n^2 - dn}{2d}$

Soluzione Esercizio 1.4

Come visto nell'Esercizio 1.3, un grafo è bipartito se e solo se è 2-colorabile. Per verificarlo, la procedura bipartire() utilizza una visita BFS colorando la radice di rosso, i nodi raggiungibili dalla radice in un passo di bianco, i nodi raggiungibili da questi di rosso, e così via. Se nel processo si scopre un nodo che deve essere colorato sia di rosso che di bianco, il grafo non è 2-colorabile e quindi nemmeno bipartito. La colorazione viene mantenuta nel vettore di interi *color*. L'ultimo colore utilizzato viene mantenuto in *current*, che può assumere i valori red $= 0$, white $= 1$ e uncolored $= 2$. Grazie alla scelta delle costanti di colore, l'espressione $1 - current$ trasforma il rosso in bianco e viceversa.

```
boolean bipartire(GRAPH G, NODE r)
    QUEUE S = Queue()
    S.enqueue(r)
    int[] color = new int[1...G.n]
    foreach u ∈ G.V() − {r} do color[u] = uncolored
    color[r] = red
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do
            if color[v] = uncolored then
                color[v] = 1 − color[u]
                S.enqueue(v)
            else
                if color[v] ≠ color[u] then return false
    return true
```

Soluzione Esercizio 1.7

Si possono rappresentare i programmi come nodi di un grafo orientato aciclico dove gli archi rappresentano le relazioni di precedenza, cioè se $P_i < P_j$ allora (i, j) è un arco del grafo. Gli istanti di partenza si calcolano scandendo la sequenza dei nodi ordinati topologicamente [cfr. Capitolo 1.6], e assegnando all'istante di partenza di ogni nodo un valore che è determinato dalla somma dei tempi di esecuzione dei nodi precedenti. Oltre al vettore *ordine*, si utilizzano i vettori *length* e *start*, che contengono, rispettivamente, i tempi di esecuzione di ogni programma e gli istanti di partenza di ciascun nodo. La complessità dell'algoritmo è $\Theta(n+m)$.

```
scheduling(GRAPH G, int[] length)
    ordine = new int[1...G.n]
    topsort(G, ordine)
    start = new int[1...G.n]
    start[ordine[1]] = 0
    for i = 2 to G.n do
        int j = ordine[i]
        start[j] = start[j] + length[j]
```

Soluzione Esercizio 1.8

Senza perdere in generalità assumiamo che tutti gli estremi degli intervalli siano distinti (se così non fosse, potremmo sempre perturbare due estremi coincidenti in modo da renderli distinti mantenendo la sovrapposizione tra i rispettivi intervalli). Osserviamo che se c intervalli hanno tutti intersezione non vuota, allora esiste un punto x che appartiene a tutti i c intervalli. Se il punto x non è un estremo di un intervallo, si può sempre spostarlo fino ad incontrare il più vicino estremo (destro o sinistro) di un intervallo. Pertanto possiamo cercare il punto x di massima sovrapposizione tra i $2n$ estremi degli intervalli.

Ordiniamo tali estremi $\{a_1, \ldots, a_n, b_1, \ldots, b_n\}$ in modo crescente, ponendoli in un vettore $S[1 \ldots 2n]$. Scandiamo S da sinistra verso destra e durante la scansione impostiamo opportuni valori in un altro vettore $Q[1 \ldots 2n]$. Per $k = 1, \ldots, 2n$, se l'elemento scandito $S[k]$ è un estremo iniziale, cioè un a_i, allora assegnamo $+1$ a $Q[k]$, mentre se $S[k]$ è un estremo finale, cioè un b_i, allora assegnamo -1 a $Q[k]$. Terminata la scansione, calcoliamo le somme prefisse di Q, ovvero

per $k = 2, \ldots, 2n$ eseguiamo $Q[k] = Q[k] + Q[k-1]$. Osserviamo che $Q[k]$ fornisce il numero di sovrapposizioni nel punto $S[k]$, se $S[k]$ è un estremo iniziale, ed il numero di sovrapposizioni meno uno, se $S[k]$ è un estremo finale. Adesso, basta trovare $c = \max_{1 \le k \le 2n}\{Q[k]\}$, che fornisce la massima sovrapposizione richiesta. Per ricavare anche il sottoinsieme di intervalli, si memorizza l'indice c che dà il massimo e poi si controlla, per ciascun intervallo I_i, se $a_i \le S[c] \le b_i$ oppure no. In caso affermativo, I_i è aggiunto alla soluzione. Il tempo complessivo è $O(n \log n)$, dovuto all'ordinamento. L'algoritmo si estende facilmente al caso più generale in cui ad ogni intervallo I_i sia associato un peso w_i e si voglia trovare un sottoinsieme di intervalli che si sovrappongono tra loro e per i quali la somma dei pesi sia massima. Basta infatti assegnare a $Q[k]$ il valore $+w_i$, se $S[k]$ è un estremo iniziale a_i, oppure $-w_i$, se $S[k]$ è un estremo finale b_i, e poi procedere come prima.

Soluzione Esercizio 1.9

È possibile adattare una visita BFS che calcoli le distanze in modo simile a quanto fatto per i numeri di Erdős. Tutte le volte che viene trovato un nodo a distanza maggiore della maggiore distanza trovata finora, il contatore *count* viene riportato a 1 e la variabile *max* viene aggiornata; se il nodo trovato è a distanza uguale alla distanza massima, il contatore viene incrementato.

```
int maxDistance(GRAPH G, NODE r)
    int[] d = new int[1...G.n]                          % Vettore distanze
    int max = 0                                         % Distanza massima trovata finora
    int count = 0                                       % Numero di nodi alla distanza massima
    QUEUE S = Queue()
    S.enqueue(r)
    foreach u ∈ G.V() − {r} do d[u] = ∞
    d[r] = 0
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do                         % Esamina l'arco (u, v)
            if d[v] = ∞ then                            % Il nodo u non è già stato scoperto
                d[v] = d[u] + 1
                if d[v] > max then
                    max = d[v]
                    count = 1
                else if d[v] = max then
                    count = count + 1
                S.enqueue(v)
    return count
```