della pagina web `//www.w3.org/Addressing/Addressing.html` descrive la specifica tecnica delle URL e la loro sintassi: la parte racchiusa tra `//` e il primo `/` successivo (`www.w3.org`) è un riferimento simbolico a un indirizzo di un calcolatore ospite (*host*) connesso a Internet. Tale indirizzo è una sequenza di 32 bit se viene utilizzato il protocollo IPv4 (oppure di 128 bit nel caso di IPv6). Infine, la parte rimanente dell'URL, ovvero `/Addressing/Addressing.html`, indica un percorso interno al *file system* dell'host, che identifica il file corrispondente alla pagina web. Un collegamento da una pagina a un'altra è specificato, all'interno della prima, utilizzando la seguente sintassi definita nel linguaggio HTML (*HyperText Markup Language*), che permette di associare, tra l'altro, un testo a ogni link con la seguente sintassi `<a href="http:URL">`*testo*`</a>`.

Queste informazioni sono utilizzate da specifici programmi dei motori di ricerca, detti *crawler* o *spider*, per attraversare il grafo del web in maniera sistematica ed efficiente raccogliendo informazioni sulle pagine visitate. Infatti, vista la dimensione del grafo del Web, è improponibile generare tutti gli indirizzi a 32 o 128 bit dei possibili host di siti web, per accedere alle loro pagine. I crawler effettuano invece la **visita** del grafo del Web partendo da un insieme `S` di pagine selezionate: in pratica, `S` viene formato con gli indirizzi disponibili in alcune collezioni, come *Open Directory Project*, che contengono un insieme di pagine web raccolte e classificate da editori "umani". Quando un crawler scopre una nuova pagina, la lista dei link in uscita da essa permette di estendere la visita a ulteriori pagine (in realtà la situazione è più complessa per la presenza di pagine dinamiche e di formati diversi e, inoltre, per altre problematiche come la ripartizione del carico di lavoro tra i vari crawler).

Possiamo quindi modellare il comportamento dei crawler come una visita di tutti i nodi e gli archi di un grafo raggiungibili da un nodo di partenza, ipotizzando che i vertici siano identificati con numeri compresi tra `0` e `n - 1` (quindi `V = {0, 1, ..., n - 1}`) e il numero di archi sia indicato con `m = |E|`. Osserviamo che gli algoritmi di visita discussi nel seguito utilizzano le pile e le code discusse nel Capitolo 2 e funzionano sia per grafi orientati che per grafi non orientati.

7.2.1 Visita in ampiezza di un grafo

La caratteristica della visita in ampiezza o BFS (*Breadth-First Search*) è che essa esplora i nodi in ordine crescente di distanza da un nodo iniziale tenendo presente l'esigenza di evitare che la presenza di cicli possa portare a esaminare ripetutamente gli stessi cammini. Nell'esporre la visita in ampiezza su un grafo, faremo inizialmente l'ipotesi che l'insieme dei nodi sia conosciuto: successivamente considereremo il caso più generale in cui tale insieme non sia preventivamente noto.

Al fine di esaminare ogni arco un numero limitato di volte nel corso della visita, usiamo un array booleano di appoggio `raggiunto`, tale che `raggiunto[u]` vale `TRUE` se e solo se il nodo `u` è stato scoperto nel corso della visita effettuata fino a ora. Ogni volta che viene raggiunto un nuovo nodo, tutti i suoi vicini vengono

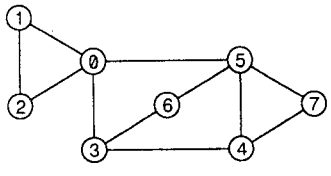
inseriti in una coda Q dalla quale viene prelevato il prossimo nodo da visitare. Come vedremo in seguito, l'utilizzo della coda ci garantirà l'ordine di visita in ampiezza. La lista di adiacenza del vertice corrente fornisce, come al solito, i riferimenti ai suoi vicini. Il Codice 7.1 riporta lo schema di visita a partire da un vertice prescelto s, dove `listaAdiacenza[u].inizio` indica il riferimento all'inizio della lista di adiacenza per il vertice u. Dopo aver inizializzato `raggiunto` e la coda Q (righe 2-4), inizia il ciclo di visita. Il vertice u in testa alla coda viene estratto (riga 6) e, se non è stato ancora raggiunto, viene marcato come tale e la sua lista di adiacenza viene scandita a partire dal primo elemento (righe 7-13). Poiché per ogni vertice u i suoi vicini vengono inseriti nella coda soltanto se u non è ancora marcato, ciascuna delle liste di adiacenza esaminate viene anch'essa considerata una sola volta: in conseguenza di ciò, il costo totale della visita è dato dalla somma delle lunghezze delle liste di adiacenza esaminate, ovvero al più dalla somma dei gradi di tutti i vertici del grafo, ottenendo un tempo totale $O(n + m)$ e $O(n + m)$ celle di memoria aggiuntive.

ALVIE Codice 7.1 Visita in ampiezza di un grafo con n vertici a partire dal vertice s, utilizzando una coda Q inizialmente vuota e un array `raggiunto` per marcare i vertici visitati.

```
1   BreadthFirstSearch( s ):
2     FOR (u = 0 ; u < n; u = u+1)
3       raggiunto[u] = FALSE;
4     Q.Enqueue( s );
5     WHILE (!Q.Empty( )) {
6       u = Q.Dequeue( );
7       IF (!raggiunto[u]) {
8         raggiunto[u] = TRUE;
9         FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
10          v = x.dato;
11          Q.Enqueue( v );
12        }
13      }
14    }
```

ESEMPIO 7.1

Prendiamo come riferimento il grafo mostrato nella figura che segue.



L'ordine dei vertici in ciascuna lista di adiacenza determina l'ordine di visita dei vertici stessi nel grafo. Supponendo che i vertici in ciascuna lista di adiacenza siano mantenuti in ordine crescente e che il nodo di partenza sia s = 0, la coda Q e l'array `raggiunto` evolveranno nel modo indicato di seguito.

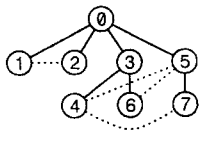
		0	1	2	3	4	5	6	7
Q = 0	raggiunto =	F	F	F	F	F	F	F	F
Q = 1, 2, 3, 5	raggiunto =	T	F	F	F	F	F	F	F
Q = 2, 3, 5, 0, 2	raggiunto =	T	T	F	F	F	F	F	F
Q = 3, 5, 0, 2, 0, 1	raggiunto =	T	T	T	F	F	F	F	F
Q = 5, 0, 2, 0, 1, 0, 4, 6	raggiunto =	T	T	T	T	F	F	F	F
Q = 0, 2, 0, 1, 0, 4, 6, 0, 4, 6, 7	raggiunto =	T	T	T	T	F	T	F	F

A ogni passo viene estratto l'elemento in testa alla coda (quello più a sinistra) e, non essendo visitato, viene marcato e tutti i suoi vicini vengono messi in coda. I nodi che seguono in coda sono stati già marcati quindi vengono solo tolti dalla coda.

		0	1	2	3	4	5	6	7
Q = 4, 6, 0, 4, 6, 7	raggiunto =	T	T	T	T	F	T	F	F
Q = 6, 0, 4, 6, 7, 3, 5, 7	raggiunto =	T	T	T	T	T	T	F	F
Q = 0, 4, 6, 7, 3, 5, 7, 3, 5	raggiunto =	T	T	T	T	T	T	T	F
Q = 7, 3, 5, 7, 3, 5	raggiunto =	T	T	T	T	T	T	T	F
Q = 3, 5, 7, 3, 5, 4, 5	raggiunto =	T	T	T	T	T	T	T	T
Q = null	raggiunto =	T	T	T	T	T	T	T	T

Con la coda vuota la procedura ha termine.

L'ordine con cui i vertici vengono raggiunti dalla visita del grafo, illustrato nella figura che segue, è dato da 0, 1, 2, 3, 5, 4, 6, 7 (il loro ordine di estrazione dalla coda e la conseguente marcatura mediante `raggiunto`).



In particolare, dal vertice 0 raggiungiamo i vertici 1, 2, 3 e 5, dal vertice 3 raggiungiamo 4 e 6 e dal vertice 5 raggiungiamo 7 (mentre i rimanenti vertici non permettono di raggiungerne altri).

Esercizio svolto 7.1 Modificare la visita BFS in modo che metta in coda soltanto i vicini non ancora visitati del vertice corrente. Valutare la dimensione massima della coda in tal caso.

Soluzione Prima di inserire un vertice in coda, verifichiamo attraverso un array `inCoda` che questo non sia già presente nella coda `Q`. L'invariante è che il vertice `u` estratto da `Q` sicuramente soddisfa la condizione che `raggiunto[u]` è falso. Per cui possiamo tranquillamente porlo a vero e mettere in coda i suoi vicini per cui `inCoda` è falso, ponendolo poi a vero, come illustrato nel codice riportato di seguito. La dimensione massima della coda diventa `n - 1` in quanto contiene soltanto vertici ancora da raggiungere.

```
1  BreadthFirstSearchSemplificata( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      raggiunto[u] = FALSE;
4      inCoda[u] = FALSE;
5    }
6    Q.Enqueue( s );
7    inCoda[s] = TRUE;
8    WHILE (!Q.Empty( )) {
9      u = Q.Dequeue( );
10     raggiunto[u] = TRUE;
11     FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
12       v = x.dato;
13       IF (!inCoda[v]) {
14         Q.Enqueue( v );
15         inCoda[v] = TRUE;
16       }
17     }
18   }
```

Elenchiamo alcune proprietà interessanti della visita. Gli archi che conducono a vertici ancora non visitati, permettendone la scoperta, formano un albero detto **albero BFS**, la cui struttura dipende dall'ordine di visita (si veda l'ultima figura dell'Esempio 7.1). Per poter costruire tale albero, modifichiamo lo schema di visita illustrato nel Codice 7.1: invece di usare una coda `Q` in cui sono inseriti i *vertici*, usiamo `Q` come coda in cui gli *archi* sono inseriti ed estratti una sola volta. Il Codice 7.2 riporta tale modifica della visita in ampiezza: dopo aver estratto l'arco (v, u) dalla coda (riga 6), scandiamo la lista di adiacenza di `u` solo se quest'ultimo non è stato scoperto (riga 7). La visita richiede $O(n+m)$ tempo, in quanto ogni arco è inserito ed estratto una sola volta e le liste di adiacenza sono scandite solo quando i corrispondenti vertici sono visitati la prima volta.

ALVIE Codice 7.2 Visita in ampiezza di un grafo in cui la coda Q contiene archi anziché vertici.

```
1  BreadthFirstSearch( s ):
2  FOR (u = 0; u < n; u = u + 1)
3    raggiunto[u] = FALSE;
4  Q.Enqueue( (null, s) );
5  WHILE (!Q.Empty( )) {
6    (u', u) = Q.Dequeue( );
7    IF (!raggiunto[u]) {
8      raggiunto[u] = TRUE;
9      FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
10       v = x.dato;
11       Q.Enqueue( (u, v) );
12     }
13   }
14 }
```

Utilizzando il Codice 7.2, non è difficile individuare gli archi dell'albero BFS: basta memorizzare l'arco (u', u) quando il nodo u viene marcato come raggiunto nella riga 8 e, inoltre, u' diventa il padre di u nell'albero BFS. In generale, gli archi individuati in tal modo formano un sottografo aciclico e, quando gli archi di tale sottografo sono incidenti a tutti i vertici, l'albero BFS ottenuto è un **albero di ricoprimento** (*spanning tree*) del grafo, vale a dire un albero i cui nodi coincidono con quelli del grafo. Cambiando l'ordine relativo dei vertici all'interno delle liste di adiacenza, possiamo ottenere alberi diversi. Notiamo infine che tali alberi, avendo grado variabile, possono essere rappresentati come alberi ordinali (Paragrafo 1.4.2).

ESEMPIO 7.2

La sequenza degli archi visitati dal Codice 7.2 sul grafo dell'Esempio 7.1 è (null, 0), (0, 1), (0, 2), (0, 3), (0, 5), (3, 4), (3, 6) e (5, 7). Infatti, per esempio, i nodi 1, 2, 3 e 5 sono scoperti la prima volta dopo la visita del nodo 0 che sarà il loro padre. Analogamente il nodo 6 viene scoperto la prima volta dopo la visita del nodo 3 e quindi quest'ultimo sarà il suo padre.

L'albero BFS è utile per rappresentare i cammini minimi dal vertice di partenza s verso tutti gli altri vertici in un grafo non pesato: tale proprietà è vera in quanto gli archi *non* sono pesati (altrimenti non è detto che valga, e vedremo successivamente come gestire il caso in cui gli archi sono pesati). Per verificare tale proprietà, basta osservare che l'algoritmo visita prima i vertici a distanza 1 (ovvero i vertici adiacenti a s), poi quelli a distanza 2 e così via, come un semplice ragionamento per induzione può stabilire. In altre parole vale il seguente risultato.

Teorema 7.1 *La distanza minima di un vertice* v *da* s *nel grafo equivale alla profondità di* v *nell'albero BFS.*

Dimostrazione Per verificare quanto affermato sopra ragioniamo in modo induttivo rispetto alla distanza dei nodi da s: il caso base dell'induzione è banalmente verificato in quanto l'unico nodo a profondità 0 è s che evidentemente ha distanza 0 da se stesso.
Per mostrare il passo induttivo, supponiamo che per ogni nodo u a distanza $p' < p$ da s la profondità di u sia pari a p' e ipotizziamo che esista, per assurdo, un nodo v a distanza δ da s la cui profondità nell'albero BFS sia $p \neq \delta$, e quindi tale che il cammino minimo da s a v sia di lunghezza δ. Consideriamo allora sia il vertice v' (a distanza $\delta - 1$ da s) che precede v in tale cammino, che il padre u di v a profondità $p - 1$ nell'albero BFS. Evidentemente, non può essere $p < \delta$ in quanto, in tal caso, il cammino da s a v che attraversa u avrebbe lunghezza minore di δ, il che contraddice l'ipotesi che δ sia la distanza tra s e v. Al tempo stesso, se $\delta < p$ l'algoritmo di visita avrebbe raggiunto v da v' e quindi v avrebbe profondità δ (infatti v' sarebbe stato visitato prima di u). Deve quindi essere $p = \delta$. ❑

La proprietà appena discussa ha due conseguenze rilevanti.

1. Gli archi del grafo che non sono nell'albero BFS sono chiamati **all'indietro** (*back*): possono collegare solo due vertici alla stessa profondità nell'albero BFS oppure a profondità consecutive p e p+1.
2. Il *diametro* del grafo può essere calcolato come la massima tra le altezze degli alberi BFS radicati nei diversi vertici del grafo (in generale, ne possono esistere n diversi). Il tempo richiesto per calcolare il diametro è $O(n(n+m))$.

Osserviamo che i nodi irraggiungibili da s sono a distanza infinita e non vengono inclusi nell'albero BFS.

Un'altra applicazione interessante è che la visita BFS ci permette di stabilire se un grafo non orientato G è connesso. Infatti, G è connesso se e solo se `raggiunto[u]` vale TRUE per ogni $0 \leq u \leq n-1$, ovvero tutti i vertici sono stati raggiunti e quindi abbiamo che l'albero BFS è un albero di ricoprimento. Il costo computazionale è lo stesso della visita BFS, quindi $O(n+m)$ tempo e $O(n+m)$ celle di memoria.

Nel caso in cui l'insieme dei nodi del grafo non sia preventivamente noto, come succede nell'esplorazione della struttura a grafo del Web, non è possibile utilizzare un array per distinguere i nodi raggiunti da quelli non ancora trovati. Per ottenere tale funzionalità è necessario fare uso di una struttura di dati che consenta di rappresentare un insieme, nello specifico l'insieme dei vertici raggiunti, dando la possibilità di aggiungere nuovi elementi all'insieme e di verificare l'appartenenza di un elemento all'insieme stesso. Inoltre, l'array `listaAdiacenza[u]` viene sostituito da una funzione `listaAdiacenza(u)` che estrae dal contenuto di u le connessioni ai suoi vertici adiacenti (per esempio, u è una pagina web che al suo interno contiene delle URL ai vicini).

Tali funzionalità sono offerte da un dizionario (Capitolo 4), che quindi impieghiamo nel Codice 7.3, che è una semplice riscrittura del Codice 7.1. Nel codice in questione, il dizionario `D`, inizialmente vuoto, viene utilizzato in sostituzione dell'array `raggiunto` per rappresentare, a ogni istante, l'insieme dei nodi già raggiunti dalla visita.

ALVIE Codice 7.3 Esplorazione mediante visita in ampiezza di un grafo, utilizzando una coda `Q` e un dizionario `D` per memorizzare i vertici visitati.

```
1  BreadthFirstSearchExplore( s ):
2    Q.Enqueue( s );
3    WHILE (!Q.Empty( )) {
4      u = Q.Dequeue( );
5      IF (!D.Appartiene(u)) {
6        D.Inserisci(u);
7        FOR (x = listaAdiacenza(u).inizio; x != null; x = x.succ) {
8          v = x.dato;
9          Q.Enqueue( v );
10       }
11     }
12   }
```

Dal punto di vista del costo computazionale, la sostituzione dell'array con un dizionario fa sì che tale costo dipenda dal costo delle operazioni definite sul dizionario, dipendente a sua volta dall'implementazione adottata per tale struttura di dati.

In particolare, esaminando il Codice 7.3 risulta che l'operazione `Inserisci` è eseguita $O(n)$ volte, mentre l'operazione `Appartiene` è invocata $O(m)$ volte: per esempio, utilizzando una tabella hash, per la quale le due operazioni richiedono tempo medio $O(1)$, il costo complessivo della visita è $O(n + m)$ nel caso medio. Utilizzando invece un albero di ricerca bilanciato, i costi delle due operazioni sono $O(\log n)$ nel caso peggiore, e quindi il costo conseguente dell'algoritmo è $O((n + m) \log n)$.

7.2.2 Visita in profondità di un grafo

Se nello schema di visita illustrato nei Codici 7.1 e 7.2 sostituiamo la coda `Q` con una pila `P`, otteniamo un altro algoritmo di visita di grafi, noto come algoritmo di visita in profondità, o DFS (*Depth-First Search*), realizzato dai Codici 7.4 e 7.5. Dato che in una pila un insieme di elementi viene estratto in ordine opposto a quello di inserimento, volendo estrarre dalla pila gli archi incidenti a un nodo `u` nello stesso ordine con cui li incontriamo scandendo la lista di adiacenza di `u`, dobbiamo inserire nella pila tali archi in ordine inverso rispetto a quello della lista. Analogamente alla visita in ampiezza, anche nella visita in profondità viene costruito un albero, detto **albero DFS**, i cui archi vengono individuati in corrispondenza alla scoperta di nuovi vertici.

Codice 7.4 Visita in profondità di un grafo utilizzando una pila P di archi, inizialmente vuota. Ciascuna lista di adiacenza viene scandita all'indietro.

```
1  DepthFirstSearch( s ):
2  FOR (u = 0; u < n; u = u + 1)
3    raggiunto[u] = FALSE;
4  P.Push( s );
5  WHILE (!P.Empty( )) {
6    u = P.Pop( );
7    IF (!raggiunto[u]) {
8      raggiunto[u] = TRUE;
9      FOR (x = listaAdiacenza[u].fine; x != null; x = x.pred) {
10       v = x.dato;
11       P.Push( v );
12     }
13   }
14 }
```

Codice 7.5 Visita in profondità di un grafo in cui la pila P contiene archi anziché vertici.

```
1  DepthFirstSearch( s ):
2  FOR (u = 0; u < n; u = u + 1)
3    raggiunto[u] = false;
4  P.Push( (null, s) );
5  WHILE (!P.Empty( )) {
6    (u', u) = P.Pop( );
7    IF (!raggiunto[u]) {
8      raggiunto[u] = TRUE;
9      FOR (x = listaAdiacenza[u].fine; x != null; x = x.pred) {
10       v = x.dato;
11       P.Push( (u, v) );
12     }
13   }
14 }
```

La visita in profondità, per la natura stessa della politica LIFO che adotta la pila, si presta in modo naturale a un'implementazione ricorsiva, riportata nel Codice 7.6. Tale implementazione è ampiamente usata in varie applicazioni discusse in seguito, in alternativa a quella iterativa; notate che in questo caso il fatto che l'utilizzo della ricorsione non richieda una gestione esplicita di una pila fa sì che non sia più necessario effettuare una scansione al contrario delle liste di adiacenza.

ALVIE Codice 7.6 Visita in profondità di un grafo utilizzando la ricorsione.

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

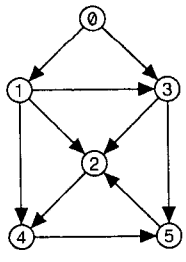
```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
```

Unitamente alla visita ricorsiva, il codice mostra la funzione `Scansione`, che esamina tutti i vertici del grafo alla ricerca di quelli non ancora scoperti, invocando la ricorsione su ciascun nodo s di questo tipo, utilizzandolo come vertice di partenza di una nuova visita. Osserviamo che l'esame di tutti i vertici del grafo in effetti non richiede necessariamente l'adozione di una visita in profondità per ogni nodo non ancora raggiunto: in linea di principio, anzi, potremmo utilizzare tipi di visita diversi.

Il costo computazionale delle visite in profondità discusse sopra è analogo a quello della visita in ampiezza, ovvero $O(n + m)$ tempo sia per grafi orientati che per grafi non orientati. Il numero di celle di memoria richieste per la visita iterativa è $O(m)$ mentre per quella ricorsiva è $O(n)$.

ESEMPIO 7.3

Riportiamo un esempio di visita del grafo orientato mostrato nella figura, a partire dal vertice s = 0.



Ecco come evolvono la pila P e l'array `raggiunto` utilizzando il Codice 7.5.

`P = (null, 0)`

`raggiunto =`

0	1	2	3	4	5
F	F	F	F	F	F

Viene estratto dalla pila l'arco in testa (`null`, `0`), il nodo `0` viene marcato e tutti gli archi uscenti da `0` vengono aggiunti in pila in ordine inverso rispetto a quello della lista di adiacenza di `0`.

P = (0, 1), (0, 3)	raggiunto =	0 1 2 3 4 5 T F F F F F
P = (1, 2), (1, 3), (1, 4), (0, 3)	raggiunto =	0 1 2 3 4 5 T T F F F F
P = (2, 4), (1, 3), (1, 4), (0, 3)	raggiunto =	0 1 2 3 4 5 T T T F F F
P = (4, 5), (1, 3), (1, 4), (0, 3)	raggiunto =	0 1 2 3 4 5 T T T F T F
P = (5, 2), (1, 3), (1, 4), (0, 3)	raggiunto =	0 1 2 3 4 5 T T T F T T

Poiché il nodo 2 dell'arco in testa alla pila è già stato visitato, l'arco viene rimosso dalla pila senza dare seguito ad altro.

P = (1, 3), (1, 4), (0, 3)	raggiunto =	0 1 2 3 4 5 T T T T T T
P = null	raggiunto =	0 1 2 3 4 5 T T T T T T

Ora che la pila è vuota l'algoritmo ha termine.

Nella Figura 7.12 sono mostrati sia l'albero BFS che albero DFS per il grafo orientato dell'Esempio 7.3. Una caratteristica importante della visita descritta nel codice di `DepthFirstSearchRicorsiva` è che mantiene implicitamente, e in ordine inverso, il cammino π nell'albero DFS dal nodo di partenza s al vertice u attualmente considerato nella visita: i vertici lungo π sono quelli in cui la visita ricorsiva è iniziata ma non ancora terminata. In altre parole, eseguendo tale codice, otteniamo implicitamente la visita anticipata dell'albero DFS. Per esempio, quando la chiamata di `DepthFirstSearchRicorsiva` esamina il vertice `u = 3` nella Figura 7.12, i vertici corrispondenti al cammino π nell'albero DFS sono 3, 1, 0.

Per quanto riguarda invece gli archi non appartenenti all'albero DFS, questi possono essere classificati ulteriormente nel caso di grafi orientati. In particolare, un arco (u, v) non appartenente all'albero DFS, può essere catalogato come segue:

- **all'indietro** (*back*): se v è antenato di u nell'albero DFS;
- **in avanti** (*forward*): se v è discendente di u nell'albero DFS (nipote, pronipote e così via, ma non figlio perché altrimenti l'arco apparterrebbe all'albero);
- **trasversale** (*cross*): se v e u non sono uno antenato dell'altro.

Nei grafi non orientati possono esserci solo archi in avanti o all'indietro, che sono gli unici a condurre a vertici già visitati durante la visita in profondità.

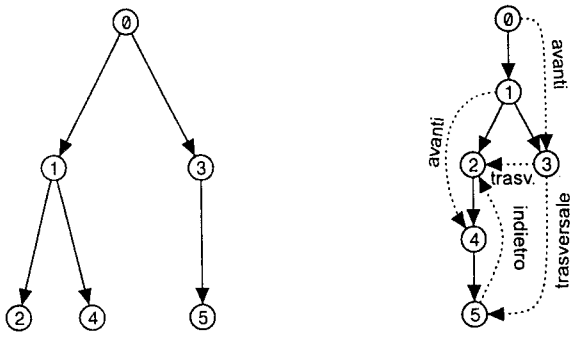


Figura 7.12 Per il grafo orientato dell'Esempio 7.3 viene mostrato il relativo albero BFS a partire dal vertice s, dove s = 0, e il relativo albero DFS a partire da s con la classificazione degli archi in avanti, all'indietro e trasversali.

Dall'esempio nella Figura 7.12 emerge anche la differenza tra le due visite. Nella visita in ampiezza, i vertici sono esaminati in ordine crescente di distanza dal nodo di partenza s, per cui la visita risulta adatta in problemi che richiedono la conoscenza della distanza e dei cammini minimi (non pesati): successivamente, vedremo come, in effetti, un limitato adattamento della visita in ampiezza consenta di individuare i cammini minimi anche in grafi con pesi sugli archi.

Nella visita in profondità, l'algoritmo raggiunge rapidamente vertici lontani dal vertice di partenza s, e quindi la visita è adatta per problemi collegati alla percorribilità, alla connessione e alla ciclicità dei cammini.

Anche per la visita in profondità, l'esplorazione di un grafo di cui non sono preventivamente noti i vertici richiede la sostituzione dell'array `raggiunto` con un dizionario: valgono rispetto a ciò le considerazioni effettuate per la visita in ampiezza nel Paragrafo 7.2.1. Nel caso specifico del grafo del Web, possiamo sostituire la coda o la pila con una coda che estrae gli elementi in base a un loro valore di rilevanza (il *rank* delle pagine web), garantendo in questo modo la priorità di caricamento, durante la visita, alle pagine classificate come più interessanti.

7.3 Applicazioni delle visite di grafi

7.3.1 Grafi diretti aciclici e ordinamento topologico

La visita in profondità trova applicazione, tra l'altro, nell'identificazione dei cicli in un grafo: vale infatti la seguente proprietà.

Teorema 7.2 *Un grafo* G *è ciclico se e solo se contiene almeno un arco all'indietro (definito per le visite BFS e DFS).*

Dimostrazione Esaminiamo prima il caso di un grafo non diretto. Consideriamo un grafo con un arco all'indietro (u, v) che, ricordiamo, non appartiene all'albero di ricoprimento (BFS o DFS) di G: esiste un cammino da v a u nell'albero in quanto, per definizione di arco all'indietro, v è antenato di u e da ciò deriva che,