Capitolo 9

Grafi

9.1 Introduzione

Un grafo non è altro che un insieme di entità collegate da un insieme di relazioni che possono essere interpretate in vari modi.

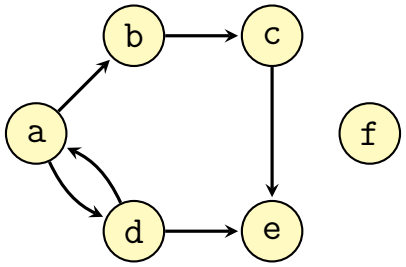
Il questa parte della lezione affronteremo i problemi che possono essere risolti tramite grafi non pesati, ossia:

- ricerca del cammino più breve, misurato in numero di archi (che differisce dal problema del cammino minimo definito su grafi pesati che cerca di stabilire quale sia il cammino meno costoso);
- componenti (fortemente) connesse;
- verifica ciclicità;
- ordinamento topologico.

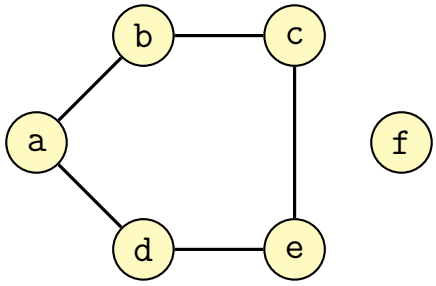
9.1.1 Definizioni

Definizione 9.1.1 (Grafo orientato, *directed*)**.** Un grafo orientato è una coppia $G = (V, E)$ dove V è un insieme di nodi (node) o vertici (vertex), mentre E è un insieme di coppie ordinate (u, v) di nodi detti archi (edges).

Definizione 9.1.2 (Grafo non orientato, *undirected*)**.** Un grafo non orientato è una coppia $G = (V, E)$ dove è un insieme di nodi (node) o vertici (vertex), mentre E è un insieme di coppie non ordinate (u, v) dette archi (edges).



(a) Grafo diretto

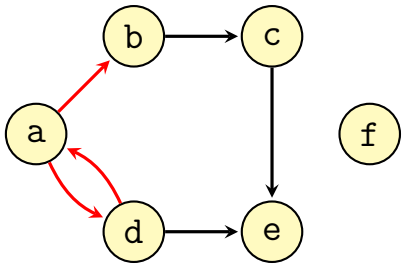


(b) Grafo indiretto

9.1.2 Terminologia

Proprietà (Adiacenza)**.** *Un vertice v è detto adiacente a u se esiste un arco (u, v).*

Proprietà (Incidenza)**.** *Un arco (u, v) è detto incidente da u a v.*



- (a, b) è incidente da a a b
- (a, d) è incidente da a a d
- (d, a) è incidente da d a a
- b è adiacente ad a
- d è adiacente ad a
- a è adiacente a d

Nota. In un grafo non orientato, la relazione di adiacenza è simmetrica.

9.1.3 Ragionamenti sulla complessità

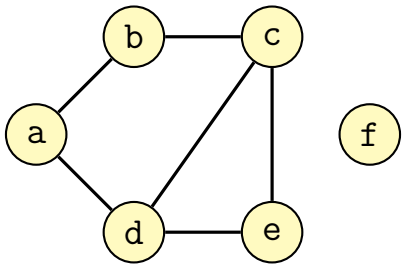
Definiamo il numero di nodi con $n = |V|$, ed il numero di archi con $m = |E|$. C'è una relazione precisa fra n ed m. In un grafo non orientato $m \leqslant \frac{n(n-1)}{2} = \mathcal{O}(n^2)$, mentre in un grafo orientato $m \leqslant n^2 - n = \mathcal{O}(n^2)$. Questi ordini di grandezza ci serviranno a valutare quale algoritmo utilizzare in base al numero di possibili archi. La complessità viene quindi espressa in termini sia di n che di m, ad esempio $\mathcal{O}(n + m)$.

9.1.4 Casi speciali

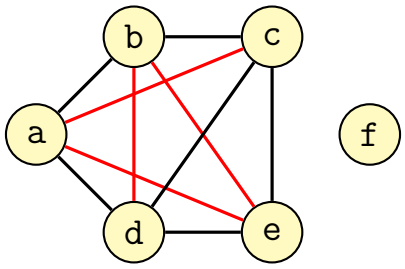
Completezza di un grafo

Un grafo con un arco fra tutte le coppie di nodi è detto *completo*. Informalmente (non c'è accordo sulla definizione) parleremo di:

- grafo *sparso* se ha "pochi archi"; ad esempio grafi con m pari a $\mathcal{O}(n)$ o $\mathcal{O}(n \log n)$ sono considerati tali;
- grafo *denso* se ha "tanti archi"; ad esempio grafi con m pari a $\Omega(n^2)$.



(a) Grafo sparso



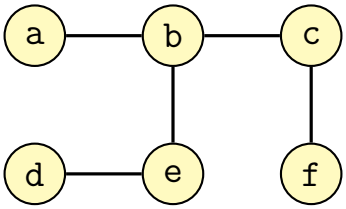
(b) Grafo denso

Alberi radicati

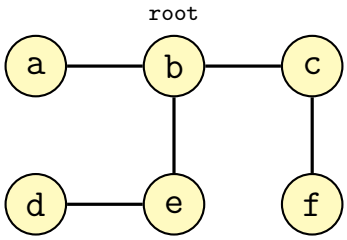
Definizione 9.1.3 (Albero non radicato, libero)**.** Un albero libero (free tree) è un grafo connesso con $m = n - 1$, dove non viene identificata una radice.

Definizione 9.1.4 (Albero radicato)**.** Un albero radicato (rooted tree) è un grafo connesso con $m = n - 1$ nel quale uno dei nodi è designato come radice.

Definizione 9.1.5 (Foresta)**.** Un insieme di alberi è un grafo detto foresta.



(a) Albero libero



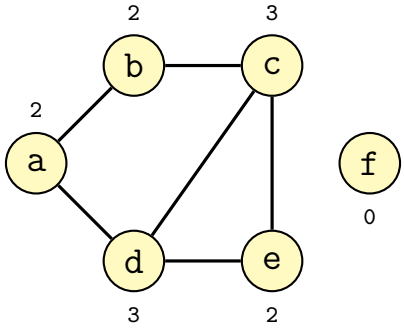
(b) Albero radicato

Definizioni

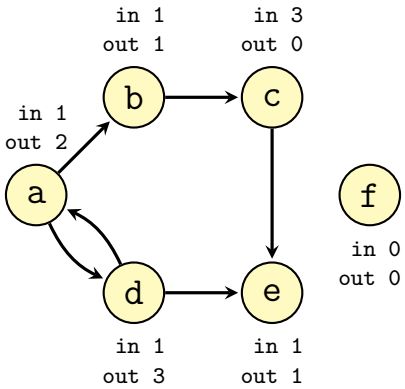
Definizione 9.1.6 (Grado, *degree*)**.** In un grafo non orientato il grado di un nodo è il numero di archi incidenti su di esso.

Definizione 9.1.7 (Grado entrante, *in-degree*)**.** In un grafo orientato il grado entrante di un nodo è il numero di archi entranti su di esso.

Definizione 9.1.8 (Grado uscente, *out-degree*)**.** In un grafo orientato il grado uscente di un nodo è il numero di archi uscenti da esso.



(a) Grado di un grafo indiretto



(b) Grado di un grafo diretto

Definizione 9.1.9 (Cammino, *path*)**.** In un grafo $G = (V, E)$, un cammino C di lunghezza k è una sequenza di nodi $u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leqslant i \leqslant k - 1$.

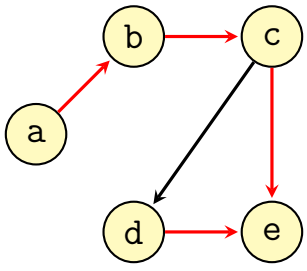


Figura 9.5: Cammino in un grafo diretto, a, b, c, e, d è un cammino di lunghezza 4

Nota. Un cammino è detto semplice se tutti i suoi nodi sono distinti.

9.1.5 Specifica

Nella versione più generale, il grafo è una struttura di dati dinamica che permette di aggiungere e rimuovere nodi e archi. La specifica che utilizzeremo non prevede la rimozione dei nodi dal grafo.

Algoritmo 1: Specifica della struttura dati GRAPH

```
Graph                                          // crea un grafo vuoto
SET V                                          // restituisce l'insieme di tutti i nodi
int size                                       // restituisce il numero di nodi
SET adj(NODE u)                                // restituisce l'insieme di nodi adiacenti a u
insertNode(NODE u)                             // aggiunge il nodo u al grafo
deleteNode(NODE u)                             // rimuove il nodo u dal grafo
insertEdge(NODE u, NODE v)                     // aggiunge l'arco (u,v) al grafo
deleteEdge(NODE u, NODE v)                     // rimuove l'arco (u,v) dal grafo
```

9.1.6 Memorizzazione

Esistono due diversi modi per memorizzare un grafo:

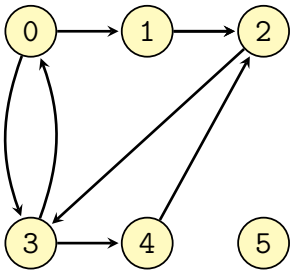
1. **matrici di adiacenza**: utilizza una matrice contenente un bit per indicare la presenza di ciascun arco: questo permette di controllare in tempo costante se un determinato arco è presente. La matrice di adiacenza viene ottenuta nel seguente modo:

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

2. **lista di adiacenza**: una lista delle adiacenze presenti fra i nodi.

Grafo orientato

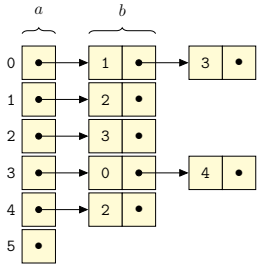
Memorizzare un grafo orientato attraverso matrice di adiacenza occupa uno spazio pari a n^2 bit, mentre se si utilizza una lista di adiacenza vengono occupati $an + bm$ bit.



(a) Grafo orientato

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	1	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

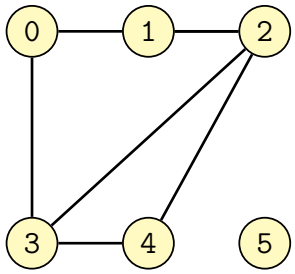
(b) Matrice di adiacenza



(c) Lista di adiacenza

Grafo non orientato

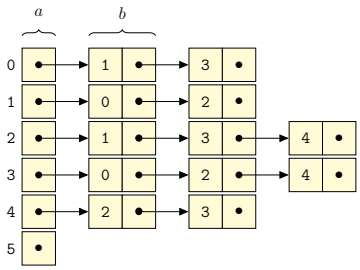
Se il grafo non è orientato ed utilizziamo una matrice di adiacenza per memorizzarlo, è sufficiente memorizzare solo la metà superiore, occupando uno spazio pari a $n(n-1)/2$ bit, mentre con lista di adiacenza dobbiamo raddoppiare i puntatori occupando $an + 2 \cdot bm$ bit.



(a) Grafo non orientato

	0	**1**	**2**	**3**	**4**	**5**
0		1	0	1	0	0
1			1	0	0	0
2				1	1	0
3					1	0
4						0
5						

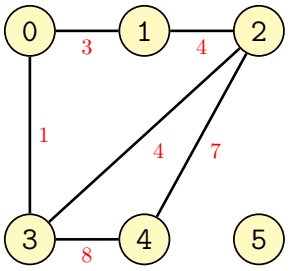
(b) Matrice di adiacenza



(c) Lista di adiacenza

Grafo pesato

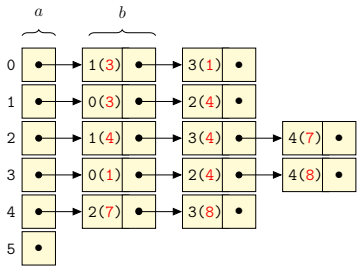
Infine se il grafo è pesato ed usiamo le matrici per rappresentarlo possiamo memorizzare il peso al posto del valore booleano, l'assenza dell'arco è quindi segnalata da un valore particolare (come -1 o ∞ in base alla procedura che vogliamo risolvere). Mentre se utilizziamo una lista di adiacenza memorizzeremo semplicemente il peso.



(a) Grafo pesato

	0	**1**	**2**	**3**	**4**	**5**
0		3	0	1	0	0
1			4	0	0	0
2				4	7	0
3					8	0
4						0
5						

(b) Matrice di adiacenza pesata



(c) Lista di adiacenza pesata

Dettagli sull'implementazione

Nel seguito, se non diversamente specificato, assumeremo che:

- l'implementazione sia basata su vettori di adiacenza (statici o dinamici);
- l'accesso alle informazioni abbia costo costante (ossia che la classe NODE sia equivalente a **int**);
- le operazioni per aggiungere nodi e archi abbiano costo ammortizzato $\mathcal{O}(1)$;
- dopo l'inizializzazione, il grafo sia statico.

Iterazione su nodi e archi

Negli algoritmi che seguiranno utilizzeremo questi schemi di codice per iterare su nodi ed archi.

Algoritmo 1: Schemi di iterazione per nodi ed archi

```
// iterare sui nodi
foreach u ∈ G.V do
    { Esegui operazioni sul nodo u }

// iterare sugli archi
foreach u ∈ G.V do
    { Esegui operazioni sul nodo u }
    foreach v ∈ G.adj(u) do
        { Esegui operazioni sull'arco u, v }
```

Complessità dell'iterazione Quest'operazione costa $\mathcal{O}(m+n)$ con le liste di adiacenza, mentre $\mathcal{O}(n^2)$ con le matrici di adiacenza.

Riassumendo

Le matrici sono ideali per grafi *densi*, occupano uno spazio $\mathcal{O}(n^2)$ e iterare su tutti gli archi costa $\mathcal{O}(n^2)$, verificare se u è adiacente a v richiede tempo costante $\mathcal{O}(1)$.

Le liste di adiacenza sono ideali per grafi *sparsi*, occupano uno spazio $\mathcal{O}(n+m)$ e iterare su tutti gli archi costa $\mathcal{O}(n+m)$, verificare se u è adiacente a v richiede tempo lineare $\mathcal{O}(n)$.

9.2 Visite dei grafi

Dato un grafo $G = (V, E)$ e un vertice $r \in V$ di partenza (che prende il nome di *radice* o di *sorgente*), si vuole visitare una e una volta sola tutti i nodi del grafo che possono essere raggiunti da r.

In ampiezza La visita in ampiezza effettua una visita dei nodi per livelli: prima visita la radice, poi i nodi a distanza uno dalla radice, poi i nodi a distanza due e così via... Una possibile applicazione di questa visita è quella di calcolare i cammini più brevi da una singola sorgente.

In profondità La visita in profondità effettua una visita ricorsiva: per ogni nodo adiacente, si visita il nodo e tutti i suoi nodi adiacenti ricorsivamente. Delle possibili applicazioni sono l'ordinamento topologico, la verifica della ciclicità e le componenti connesse e fortemente connesse.

9.2.1 Visita in ampiezza

Un approccio ingenuo alla visita di un grafo potrebbe essere il seguente:

Algoritmo 2: Primo tentativo di visita di un grafo

```
visita(GRAPH G)
    foreach u ∈ G.V do
        { visita nodo u }
        foreach v ∈ G.adj(u) do
            { visita arco (u, v) }
```

Ma la struttura del grafo non viene presa in considerazione, poiché si itera su tutti i nodi e gli archi senza alcun criterio. Un possibile approcio potrebbe essere quello di sfruttare l'algoritmo delle visite sugli alberi

Algoritmo 3: Algoritmo adatto all'attraversamento degli alberi

```
BFSTraversal(GRAPH G, int r)
    QUEUE Q ← Queue
    Q.enqueue(r)
    while not Q.isEmpty do
        NODE u ← Q.dequeue
        { visita il nodo u }
        foreach u ∈ G.adj(u) do
            Q.enqueue(v)
```

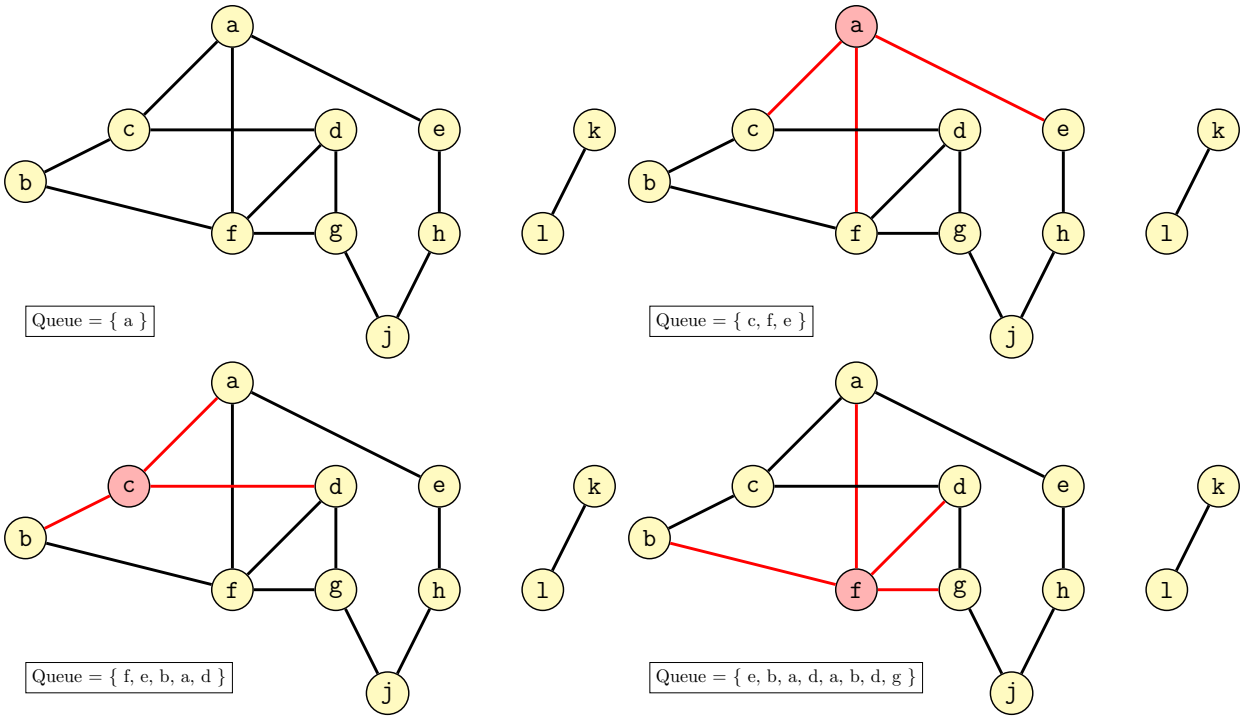


Figura 9.9: Esempio di attraversamento di un grafo tramite la procedura di visita di un albero

Dall'esempio possiamo notare come i nodi vengano reinseriti all'infinito all'interno della coda e questo non permette all'algoritmo di terminare. Negli alberi, data la loro struttura, abbiamo la sicurezza che non visiteremo mai lo stesso nodo più di una volta. Abbiamo bisogno di un meccanismo che ci permetta di evitare che questo avvenga. Possiamo farlo memorizzando i nodi che abbiamo già visitato.

Algoritmo 4: Algoritmo adatto all'attraversamento dei grafi

```
visita(GRAPH G, NODE r)
    SET S ← Set  // insieme generico, da specificare (STACK, QUEUE)
    S.insert(r)  // inserisco il nodo, da specificare
    // ho visitato il nodo
    { marca il nodo r come "scoperto" }
    // fintanto che l'insieme non è vuoto
    while S.size > 0 do
        // la politica di rimozione dipende dal problema da risolvere
        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
                // serve a non inserire il nodo più di una volta
                { marca il nodo v come "scoperto" }
                S.insert(v)  // inserisce il nodo nell'insieme, da specificare
```

Gli obiettivi della visita in ampiezza sono:

- visitare i nodi a distanze crescenti dalla sorgente: visitare quindi i nodi a distanza k prima di quelli a distanza $k+1$;
- calcolare il cammino più breve da r a tutti gli altri nodi, dove il cammino più breve è il percorso con il minor numero di archi;
- generare un albero in ampiezza (*breadth-first*): l'albero in ampiezza è un albero contenente tutti i nodi raggiungibili da r, tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u nel grafo.

Algoritmo 5: Procedura specializzata per la visita in ampiezza di un grafo

```
// visitare tutti i nodi a distanza k prima di visitare i nodi a distanza k+1
bfs(GRAPH G, NODE r)
    QUEUE S ← Queue  // creo una pila
    S.enqueue(r)  // inserisco la radice

    // inizializzazione
    boolean[] visitato ← boolean[1...G.n]  // della dimensione del no. di nodi
    foreach u ∈ G.V − {r} do  visitato[u] ← false  // devo ancora visitarli
    visitato[r] ← true  // radice visitata

    // visita del grafo
    while not S.isEmpty do
        NODE u ← S.dequeue  // rimuovo un nodo
        { esamina il nodo u }
        foreach v ∈ G.adj(u) do // per ciascun nodo adiacente "v"
            { esamina l'arco (u, v) }
            if not visitato[v] then // se non ho ancora visitato "v"
                visitato[v] ← true  // marcalo come visitato
                S.enqueue  // inseriscilo nella coda
```

9.2.2 Cammini più brevi

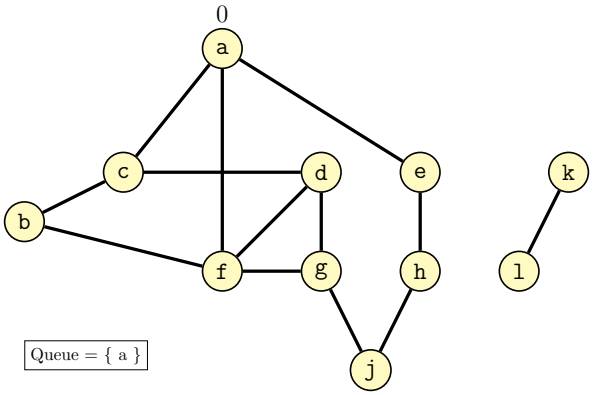
Vediamo ora un'applicazione della visita in ampiezza: la ricerca dei cammini più brevi e lo facciamo tramite un esempio particolare: calcolare il numero di erdös.

Paulo Erdös è stato uno dei matematici più prolifici al mondo (1500+ articoli e 500+ co-autori). Definiamo il numero di erdos nel seguente modo:

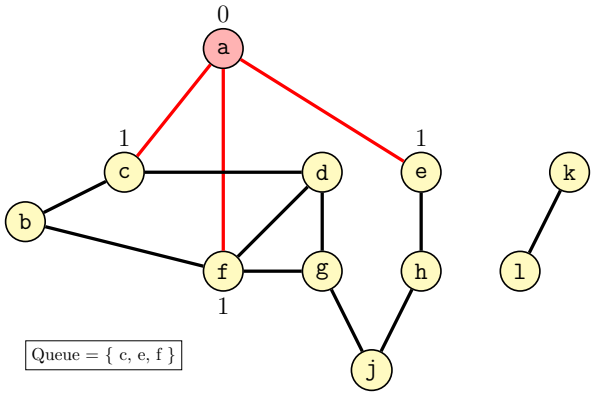
- Erdös ha valore $erdos = 0$;
- i co-autori di Erdös hanno $erdos = 1$;
- se X è co-autore di qualcuno con $erdos = k$ e non è co-autore con qualcuno con $erdos < k$, allora X ha $erdos = k + 1$;
- le persone non raggiunte da questa definizione hanno $erdos = +\infty$.

Algoritmo 6: Ricerca dei cammini minimi più brevi dalla radice

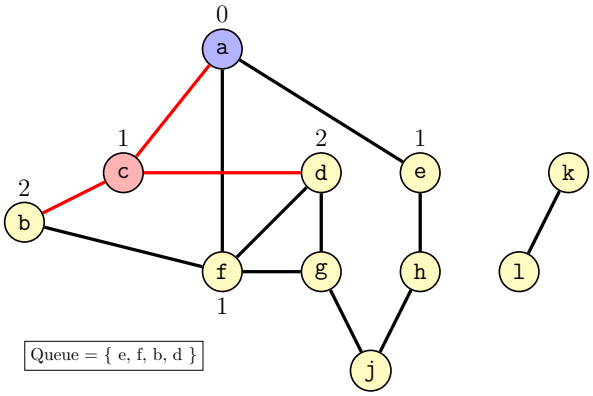
```
// il camminimo più breve fra due vertici viene memorizzato tramite il vettore dei padri p
erdos(GRAPH G, NODE r, int[] erdos, NODE parent)
    // struttura di supporto
    QUEUE S ← Queue // creo una pila
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    foreach u ∈ G.V − {r} do erdos[u] ← ∞ // nodi non ancora raggiunti
    erdos[r] ← true // erdõs ha distanza 0 da se stesso
    parent[r] ← nil // per la stampa del cammino

    // visita del grafo
    while not S.isEmpty do
        NODE u ← S.dequeue
        foreach u ∈ G.adj(u) do
            { esamina l'arco (u, v) }
            if erdos[v] == ∞ then
                // il nodo non è stato ancora stato scoperto
                erdos[v] ← erdos[u] + 1 // gli assegno un livello di erdõs+1
                parent[v] ← u // memorizzo il padre del nodo attuale nel v. dei padri
                S.enqueue(v) // è la prima volta che lo raggiungo quindi lo metto in coda
```



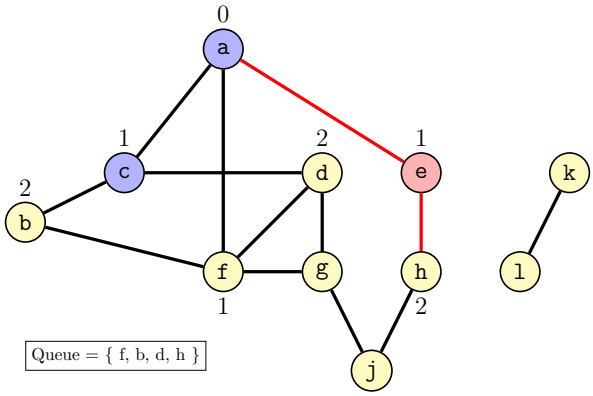
(a) Assegno al nodo a distanza 0, lo aggiungo alla coda



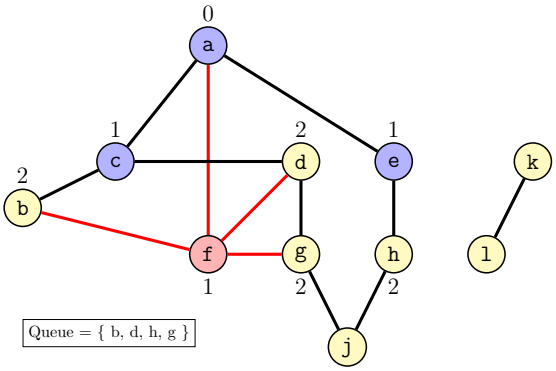
(b) Estraggo il nodo a, assegno ai suoi nodi adiacenti non ancora visitati (c, f, e) distanza $a + 1 = 1$ e li aggiungo alla coda



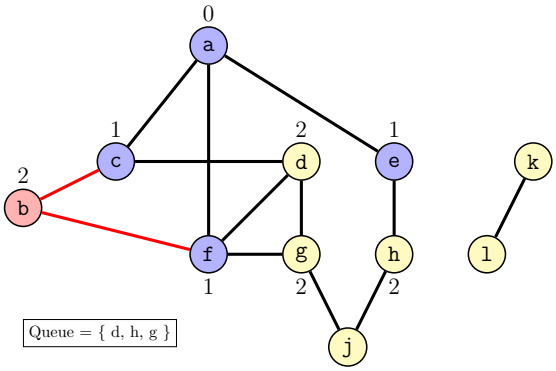
(c) Estraggo il nodo c, assegno ai suoi nodi adiacenti non ancora visitati (b, d) distanza $c + 1 = 2$ e li aggiungo alla coda



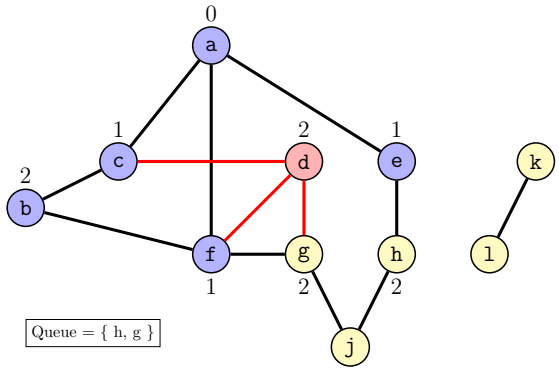
(d) Estraggo il nodo e, assegno ai suoi nodi adiacenti non ancora visitati (h) distanza $e + 1 = 2$ e lo aggiungo alla coda



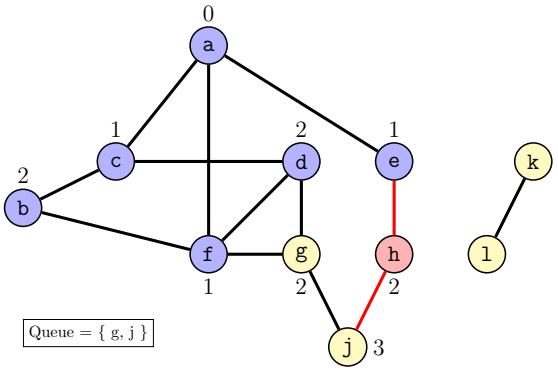
(e) Estraggo il nodo f, assegno ai suoi nodi adiacenti non ancora visitati (b, g) distanza $f + 1 = 2$ e li aggiungo alla coda



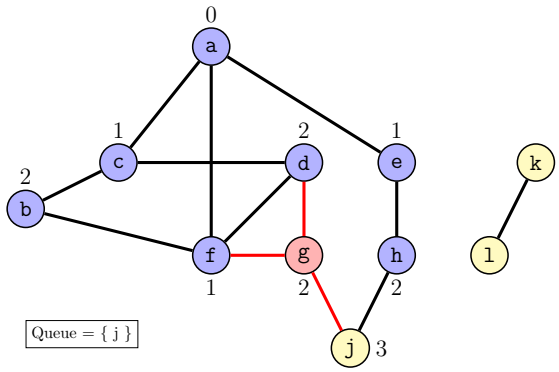
(f) Estraggo il nodo b, il quale non ha nodi adiacenti non ancora visitati



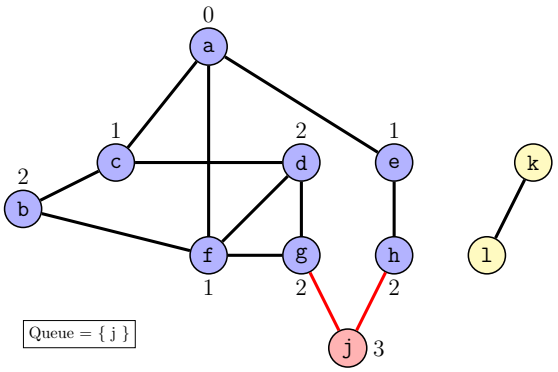
(g) Estraggo il nodo d, il quale non ha nodi adiacenti non ancora visitati



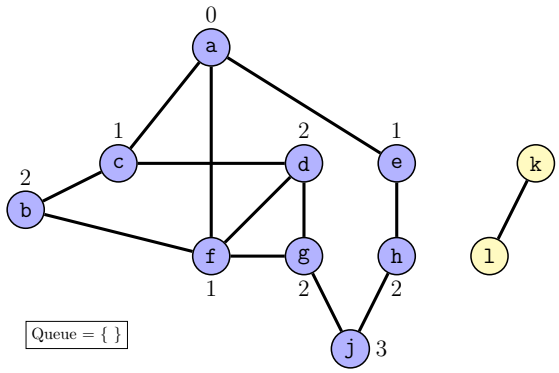
(h) Estraggo il nodo h, assegno al suo nodo adiacente non ancora visitato (j) distanza $h + 1 = 3$ e lo aggiungo alla coda



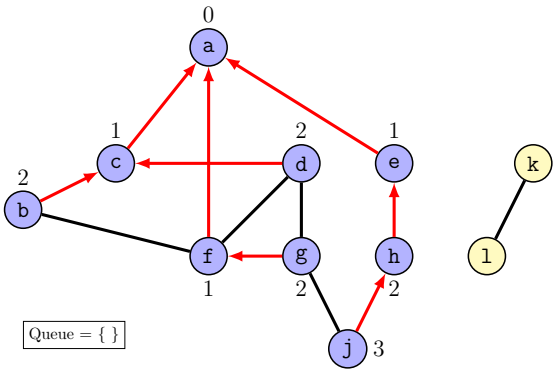
(i) Estraggo il nodo g, il quale non ha nodi adiacenti non ancora visitati



(j) Estraggo il nodo j, il quale non ha nodi adiacenti non ancora visitati



(k) Questo è l'albero risultante



(l) Rappresentazione dell'albero di copertura bfs, attraverso il vettore dei padri

Figura 9.10: Esempio di visita di un grafo tramite la procedura di visita in ampiezza

L'albero "di copertura" con radice r viene memorizzato tramite vettore dei padri (che nell'algoritmo di erdos abbiamo chiamato *parent*).

Algoritmo 7: Stampa del cammino

```
// stampa il cammino da r a s nell'ordine corretto
stampaCammino(NODE r, NODE s, NODE[] parent)
    // parent: vettore dei padri
    if r == s then // se è la radice
        stampa s // la stampo
    else if parent[s] == nil then
        stampa "nessun cammino da r a s"
    else
        // la chiamata ricorsiva prima della stampa per stampare in ordine
        stampaCammino(r, parent[s], p)
        stampa s
```

9.2.3 Complessità della visita in ampiezza

Ognuno degli n nodi viene inserito nella coda al massimo una volta, ed ogni volta che un nodo viene estratto tutti i suoi archi vengono analizzati una volta sola, per una complessità totale di $\mathcal{O}(m+n)$. Il numero di archi analizzati è quindi

$$m = \sum_{u \in V} out_d(u)$$

dove $out_d(u)$ è il grado uscente (*out-degree*) del nodo u.

9.2.4 Visita in profondità

Molto spesso la visita in profondità è solo una parte di una soluzione ad un altro problema, viene utilizzata per esplorare un intero grafo, e non solo i nodi raggiungibili da una singola sorgente.

L'output dell'algoritmo non è più un singolo albero, ma una foresta *depth-first* (ossia un insieme di alberi *depth-first*).

La struttura dati utilizzata non è più una coda, bensì uno *stack* implicito (attraverso la ricorsione) o esplicito (vedremo un algoritmo nel seguito).

Algoritmo 8: Visita in profondità, ricorsiva con stack implicito

```
// genera un albero depth-first
dfs(GRAPH G, NODE u, boolean[] visitato)
    visitato[u] = true // ho visitato il nodo
(1) { esamina il nodo u (caso pre-visita) }
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u, v) }
        if not visitato[v] then // se non l'ho ancora visitato
            // chiamata ricorsiva
            dfs(G, v, visitato) // lo visito ricorsivamente
(2) { esamina il nodo u (caso post-visita) }
```

Analisi della complessità Questo algoritmo ha la stessa complessità della visita in ampiezza: $\mathcal{O}(n+m)$ con il grafo implementato tramite liste di adiacenza, e $\mathcal{O}(n^2)$ con matrice di adiacenza.

Si presenta però un problema, mentre con gli alberi possiamo assumere che la loro profondità sia limitata, con i grafi non possiamo fare lo stesso discorso. Eseguire una visita in profondità basata su chiamate ricorsive può essere rischioso (errore di stack overflow) in grafi molto grandi e connessi (in particolare se non sono

orientati, in quanto la visita analizza tutti i nodi). È possibile che la profondità raggiunta sia troppo grande per la dimensione dello stack del linguaggio. In tali casi, si preferisce utilizzare una visita in ampiezza, oppure in profondità ma basata su stack esplicito.

Algoritmo 9: Visita in profondità, iterativa con stack esplicito, visita in *pre*-ordine

```
// effettua una visita in profondità iterativa
dfs(Graph G, Node r)
    Stack S ← Stack
    S.push(r)  // inserisco la radice nella pila
    boolean[] visitato ← new boolean[1...G.size]
    foreach u ∈ G.V − {r} do visitato[u] ← false

    visitato[r] ← true  // marco la radice come visitata
    while not S.isEmpty do
        Node u ← S.pop  // estraggo un nodo
        if not visitato[v] then // se non l'ho ancora visitato
            { esamina il nodo u in pre-ordine }
            visitato[v] ← true  // lo segno come visitato
            foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
                { esamina l'arco (u, v) }
                S.push(v)  // lo inserisco nella pila
```

La procedura si ottiene semplicemente sostituendo una pila alla coda utilizzata nella visita in ampiezza.

Un nodo può essere inserito nella pila più volte (tante volte quanti sono gli archi entranti in quel nodo, il numero di archi entranti in tutti i nodi è limitato superiormente dal numero di archi m, quindi $\mathcal{O}(m)$, in quanto il controllo se un nodo è già stato inserito viene fatto all'estrazione, non all'inserimento come avveniva in precedenza.

Complessità della visita in ampiezza con stack esplicito Inserimenti ed estrazioni sono pari al numero degli archi, quindi $\mathcal{O}(m)$, visitare gli archi costa $\mathcal{O}(m)$ e le visite dei nodi costano $\mathcal{O}(n)$, per una complessità risultante di $\mathcal{O}(m+n)$, invariata rispetto agli algoritmi precedenti.

Visita in post-ordine È possibile effettuare una visita in profondità con una procedura con stack esplicito e visita in *post*-ordine ma abbiamo bisogno di aggiungere due "flag": discovery e finish.

Quando un nodo viene scoperto viene inserito nello stack con il tag discovery; quando un nodo viene estratto dalla coda con tag discovery: viene re-inserito con il tag finish e tutti i suoi vicini vengono inseriti; Quando un nodo viene estratto dalla coda con tag finish, viene effettuata la post-visita. Non vedremo il codice poiché è complicato e i dettagli non sono interessanti.

9.2.5 Componenti connesse

Vogliamo identificare le componenti connesse di un grafo. Prima di tutto vogliamo capire se è connesso, dopodiché vogliamo sapere quante sono le sue componenti connesse. Il motivo per cui vogliamo farlo è che molti algoritmi che operano sui grafi iniziano decomponendo il grafo nelle sue componenti connesse per poi ri-comporre i risultati assieme. Sono definite due tipologie di problemi:

- componenti connesse (*Connected Components, CC*);
- componenti fortemente connesse (*Strongly Connected Components, SCC*)

Entrambi i problemi sono definiti su grafi non orientati.

Per capire meglio il problema abbiamo bisogno di qualche definizione:

Definizione 9.2.1 (Raggiungibilità di un nodo)**.** Un nodo v è raggiungibile da un nodo u se esiste almeno un cammino da u a v.

Definizione 9.2.2 (Grafo non orientato connesso)**.** Un grafo non orientato $G = (V, E)$ è connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.

Definizione 9.2.3 (Componente connessa)**.** Un grafo $G' = (V', E')$ è una componente connessa di G se e solo se G' è un sottografo connesso e massimale di G.

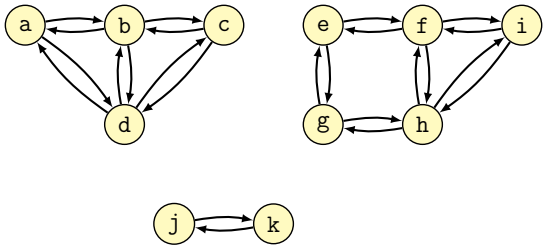
Definizione 9.2.4 (Sottografo)**.** G' è un sottografo di G $(G' \subseteq G) \Leftrightarrow V' \subseteq V$ e $E' \subseteq E$

Definizione 9.2.5 (Grafo massimale)**.** G' è massimale se e solo se non esiste nessun altro sottografo G'' di G tale che G'' è connesso e più grande di G' (ad esempio $G' \subseteq G'' \subseteq G$)

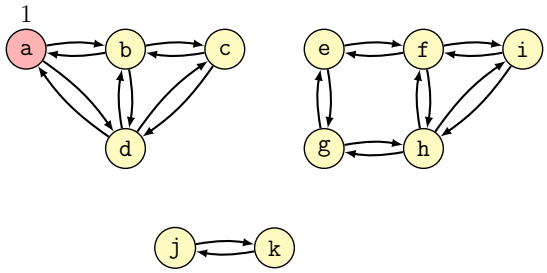
Vogliamo quindi verificare se un grafo è connesso oppure no, ed identificare le sue componenti connesse. Per farlo utilizzeremo di un vettore (che chiameremo id), il quale conterrà l'indentificatore alla quale la componente appartiene ($id[u]$ è l'indentificatore della c.c. a cui appartiene u). Un grafo risulta connesso se, al termine della visita in profondità, tutti i suoi nodi risultano marcati. Altrimenti la visita deve ricominciare da capo da un nodo non marcato, identificando una nuova componente.

Algoritmo 10: Identifica le componenti connesse di un grafo non orientato

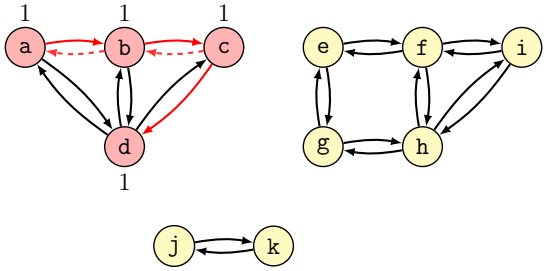
```
// parte iterativa
int[] cc(GRAPH G, STACK S)
    // creo un vettore della dimensione dei nodi del grafo
    int[] id ← new int[1...G.size]
    // inizializzo il vettore
    foreach u ∈ G.V do id[u] ← 0
    int counter = 0 // contatore delle componenti connesse
    foreach u ∈ G.V do // per ogni nodo del grafo
        if id[u] == 0 then // ho trovato una nuova componente connessa
            counter++ // aggiorno il contatore
            // effettuo una chiamata ricorsiva sul nodo scoperto
            ccdfs(G, counter, u, id)
    // restituisco l'identificativo della componente connessa
    return id
// visita ricorsiva di ciascuna componente
ccdfs(GRAPH G, int counter, NODE u, int[] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u:       il nodo che sto visitando
    // id:      l'identificativo della componente
    // memorizzo l'identificativo della cc
    id[v] ← counter
    foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
        if id[v] == 0 then // non è ancora stato visitato
            // v:  il nodo su cui vado ad operare
            ccdfs(G, counter, v, id) // lo visito ricorsivamente
```



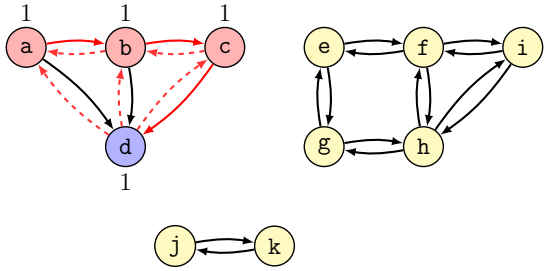
(a) Stato iniziale



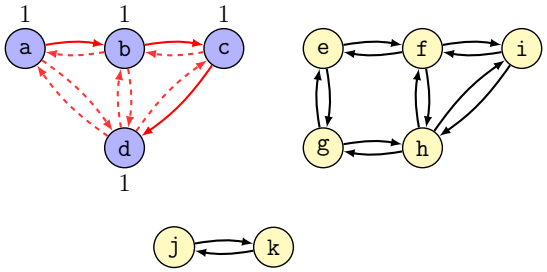
(b) Partiamo dal nodo a per comodità, gli assegniamo il valore 1



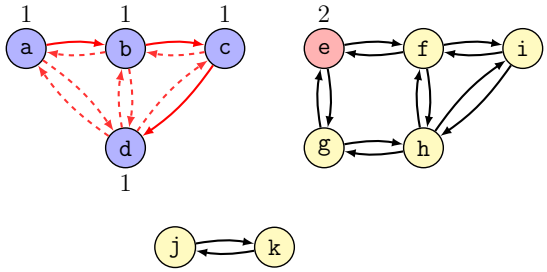
(c) Visito tutti i nodi adiacenti al nodo a



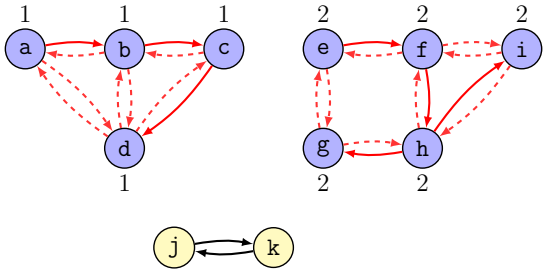
(d) Una volta arrivati al nodo d abbiamo già visitato tutti i suoi possibili vicini



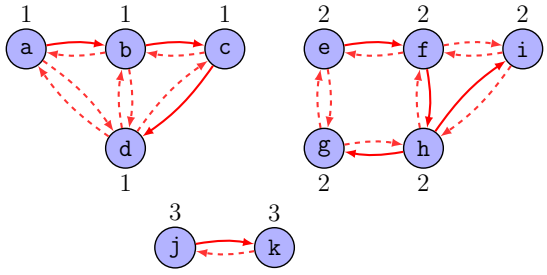
(e) Abbiamo completato la visita della prima componente connessa



(f) Abbiamo ricontrollato se potevamo ripartire da uno qualsiasi dei nodi della prima componente ma aveva già un id assegnato, partiamo da e per comodità



(g) Completo così anche la seconda componente...



(h) ...e la terza

Figura 9.11: Esempio di identificazione delle componenti connesse in un grafo non orientato

9.3 Verifica ciclicità

Grafi aciclici non orientati

Definizione 9.3.1 (ciclo, *cycle*)**.** In un grafo orientato $G = (V, E)$, un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1, \ldots u_k$ tale che $(u_i, u_{i+1} \in E)$ (un cammino) per $0 \leqslant i \leqslant k-1$ e $u_0 = u_k$ (il primo e l'ultimo nodo coincidono, ossia è chiuso).

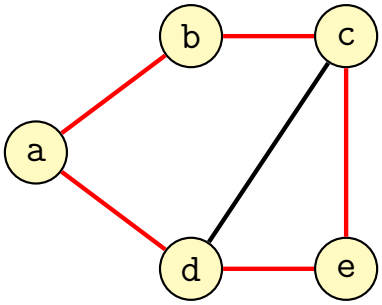


Figura 9.12: $k > 2$ esclude cicli banali composti da coppie di archi, i quali sono onnipresenti nei grafi non orientati. Questo grafo contine 3 cicli, uno di lunghezza 3, uno di lunghezza 4 ed uno di lunghezza 5.

Definizione 9.3.2 (Grafo aciclico)**.** Un grafo non orientato che non contiene cicli è detto aciclico.

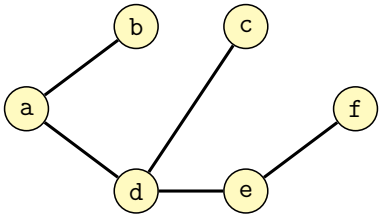


Figura 9.13: Un grafo aciclico.

Un grafo che contiene cicli è detto *ciclico*, altrimenti viene detto *aciclico*. Dato un grafo non orientato, vogliamo poter essere in grado di determinare se contiene cicli o meno. Dobbiamo scrivere quindi un algoritmo che restituisca **true** se il grafo contiene un ciclo, **false** altrimenti.

Nota. Se è un grafo connesso ed è aciclico, allora è un albero.

L'idea è che se visito un nodo che ho già visitato, allora ho identificato un ciclo.

Algoritmo 11: Ricerca di un ciclo in un grafo non orientato

```
// restituisce true se trova un ciclo
boolean hasCycleRec(GRAPH G, NODE u, NODE p, boolean[] visited)
    // G:       grafo esplorato
    // u:       nodo da esaminare
    // p:       nodo da cui provengo (padre)
    // visited: vettore dei nodi visitati
    visited[u] ← true  // lo visito per la prima volta
    foreach v ∈ G.adj(u) – {p} do // visito tutti i suoi vicini
        // G.adj(u) – {p}:  non considero il nodo da cui provengo (è un grafo orientato)
        if visited[v] then // ho già visitato il nodo
            return true  // ho trovato un ciclo
            // altrimenti effettuo una visita ricorsiva sul nodo vicino v
        else if hasCycleRec(G, v, u, visited) then
            // se una qualsiasi delle sottochiamate ritorna vero, allora ho trovato un ciclo
            return true
    // non ho trovato alcun ciclo
    return false
// ricerca di un ciclo per grafi disconnessi
boolean hasCycle(GRAPH G)
    boolean[] visited ← new boolean[]1G.size  // creo il vettore
    foreach u ∈ G.V do // lo inizializzo
        visited[u] ← false
    foreach u ∈ G.V do // per ciascun nodo appartenente al grafo
        if not visited[u] then // il primo nodo non sarà stato visitato
            if hasCycleRec(G, v, null, visited) then
                // effettuo una visita ricorsiva sul nodo vicino v
                return true
    return false
```

La parte non ricorsiva dell'algoritmo ci permettere di cercare cicli in grafi non connessi, in quanto stiamo osservando grafi e non alberi (i quali sono connessi).

Grafi aciclici orientati

Cosa succede se iniziamo a considerare i grafi orientati?

Definizione 9.3.3 (ciclo, *cycle*)**.** In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k \geqslant 2$ è una sequenza di nodi $u_0, u_1, \ldots u_k$ tale che $(u_i, u_{i+1} \in E)$ (un cammino) per $0 \leqslant i \leqslant k-1$ e $u_0 = u_k$ (il primo e l'ultimo nodo coincidono, ossia è chiuso).

La definizione è identica a parte che ora consideriamo come cicli anche i cammini composti da due soli nodi $(k \geqslant 2)$

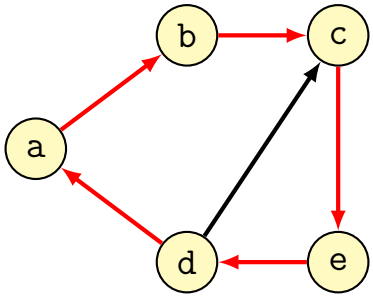
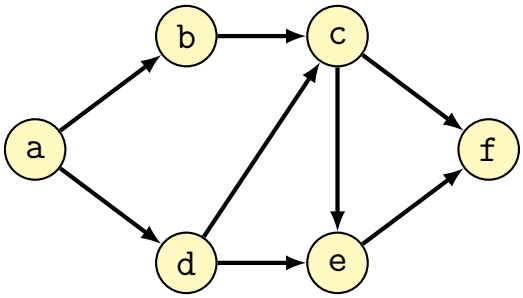


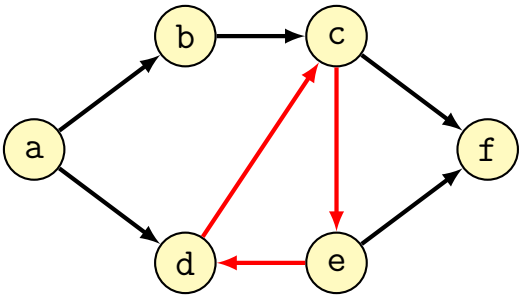
Figura 9.14: a, b, c, e, d, a è un cammino di lunghezza 5

Nota. Un ciclo è detto semplice se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo)

Definizione 9.3.4 (Grafo diretto aciclico, *Directed Acyclic Graph*)**.** Un grafo orientato che non contiene cicli è detto DAG (Directed Acyclic Graph)



(a) Un grafo aciclico.



(b) Un grafo ciclico. Il ciclo è dato dai nodi c, e, f.

Figura 9.15: Esempio di grafo aciclico e ciclico

I *Directed Acyclic Graph*, DAG d'ora in poi, rappresentano una classe di problemi ben precisi e vedremo degli algoritmi che trattano nello specifico questi grafi. Il problema è il medesimo: dato un grafo non orientato, vogliamo poter essere in grado di determinare se contiene cicli o meno. Dobbiamo scrivere quindi un algoritmo che restituisca **true** se il grafo contiene un ciclo, **false** altrimenti.

L'algoritmo precedente riesce a risolvere anche queto problema? Riesci a pensare ad un grafo orientato per cui l'algoritmo appena visto non si comporta correttamente?

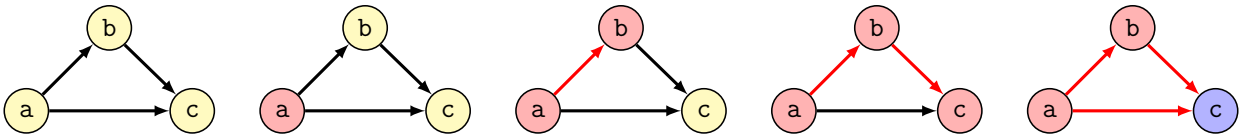
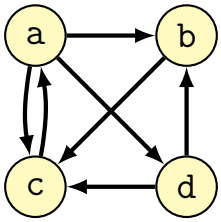


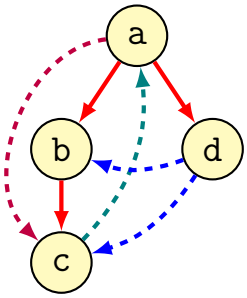
Figura 9.16: Se in un grafo esistono due cammini che portano allo stesso nodo l'algoritmo precedente dirà erroneamente che esiste un ciclo.

Classificazione degli archi

Ogni volta che si esamina un arco da un nodo marcato (visitato) ad un nodo non marcato (non visitato), tale arco viene detto arco dell'albero. Gli archi non inclusi nell'albero possono essere divisi in tre categorie:

- se u è un antenato di v in T, (u, v) è detto arco in avanti;
- se u è un discendente di v in T, (u, v) è detto arco all'indietro;
- altrimenti viene detto arco di attraversamento.





Algoritmo 11: Schema per visita dell'albero in profondità

```
// classifica i lati di un grafo
dfs-schema(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
    // time:    contatore
    // dt:      tempo di scoperta
    // ft:      tempo di fine
    esamina il nodo u (caso pre-visita)
    time++ // incremento il contatore
    dt[u] ← time // lo memorizzo nel vettore di scoperta
    // effettuo una visita in profondità
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u, v) (qualsiasi) } // qui si sviluppa la logica dell'algoritmo
        if dt[v] == 0 then // non ho ancora esaminato il nodo
            { esamina l'arco (u, v) (albero) }
            dfs-schema(G, v, time, dt, ft) // effettuo la chiamata ricorsiva
        else if dt[u] > dt[v] and ft[v] == 0 then
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita, allora ho
               trovato un arco all'indietro
            { esamina l'arco (u, v) (indietro) }
        else if dt[u] < dt[v] and ft[v] ≠ 0 then
            // se raggiungo un mio discendente e ho terminato la mia visita, allora ho trovato un
               arco in avanti
            { esamina l'arco (u, v) (avanti) }
        else
            // l'ultimo caso rimanente
            { esamina l'arco (u, v) (attraversamento) }
    { visita il nodo u (post-visita) }
    time++ // aggiorno il contatore
    ft[u] ← time // lo memorizzo nel vettore di fine
```

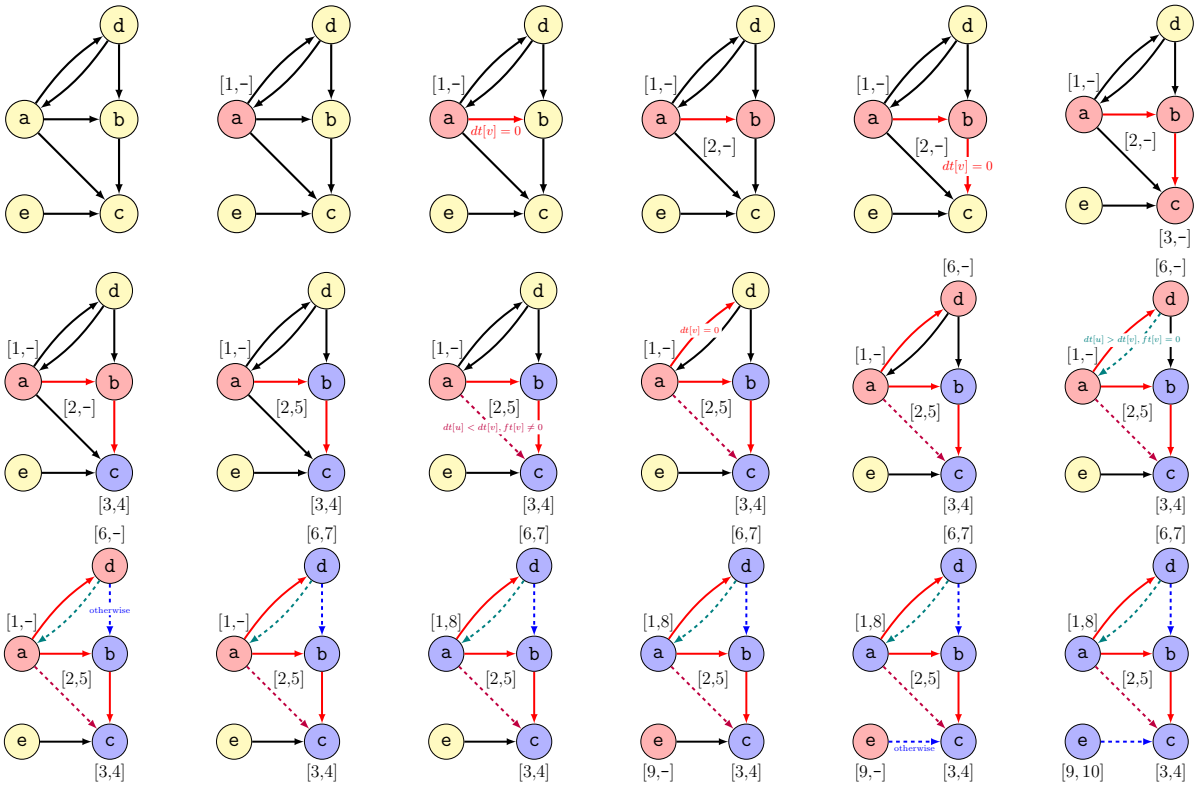


Figura 9.18: Visitati in ordine alfabetico per comodità

Osserviamo i numeri assegnati ai nodi; se li consideriamo come intervalli allora $a \subset b$ perché $[1,8] \subset [2,5]$. Osservando gli intervalli possiamo quindi dedurre le relazioni di discendenza fra i nodi.

Ma perché li classifichiamo? Perché possiamo dimostrare delle proprietà sul tipo di archi e usarle per costruire algoritmi migliori.

Teorema. *Data una visita in profondità di un grafo $G = (V, E)$, per ogni coppia di nodi $(u, v) \in V$, solo una delle condizioni seguenti è vera:*

- *Gli intervalli $[dt[u], ft[u]$ e $[dt[v], ft[v]]$ sono non-sovrapposti; u, v non sono discendenti l'uno dell'altro nella foresta* depth-first*;*
- *L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$; u è un discendente di v in un albero* depth-first*;*
- *L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$; v è un discendente di u in un albero* depth-first*.*

Teorema. *Un grafo orientato è aciclico se e solo se non esistono archi all'indietro nel grafo*

Dimostrazione. Abbiamo due casi:

1. se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e sia (u, v) un arco del ciclo. Il cammino che connette u a v verrà prima o poi visitato, e da v verrà scoperto l'arco all'indietro (v, u) (se esiste un ciclo prima o poi nella visita lo vado a toccare);
2. 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.

□

Sfruttando questa dimostrazione possiamo quindi semplificare l'algoritmo precedente per la ricerca di un ciclo in un grafo aciclico diretto.

Algoritmo 12: Ricerca di un ciclo in un grafo aciclico diretto

```
// applicabile solo ai DAG, in quanto non hanno archi all'indietro
boolean hasCycle(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
    // u:  il primo nodo che viene visitato
    time++ // aumento il contatore
    dt[u] ← time // memorizzo il tempo di scoperta
    foreach v ∈ G.adj(u) do
        if dt[v] == 0 then // non ho ancora scoperto questo nodo
            // effettuo una visita ricorsiva
            if hasCycle(G, v, time, dt, ft) then
                return true
        // logica dell'algoritmo
        else if dt[u] > dt[v] and ft[v] == 0 then
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita, allora ho
               trovato un arco all'indietro (un ciclo)
            return true
    time++ // aumento il contatore
    ft[u] ← time // memorizzo il tempo di fine
    // non ho trovato un ciclo
    return false
```

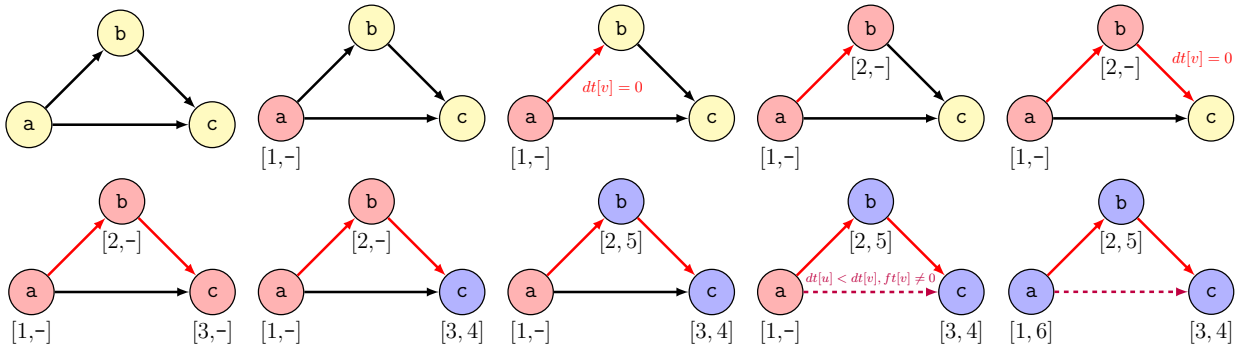


Figura 9.19: Questo è il particolare esempio sulla quale l'algoritmo precedente falliva

Nota. Se nello schema degli intervalli ci sono due intervalli sovrapposti, allora esiste un ciclo.

9.4 Ordinamento topologico

L'obiettivo è quello di scrivere un algoritmo che prende in input un DAG e che ne restituisca un possibile ordinamento topologico.

Definizione 9.4.1 (ordinamento topologico)**.** Dato un grafo diretto e aciclico (DAG) G, un ordinamento topologico di G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.

Nota. Se il grafo contiene un ciclo, non esiste un ordinamento topologico.

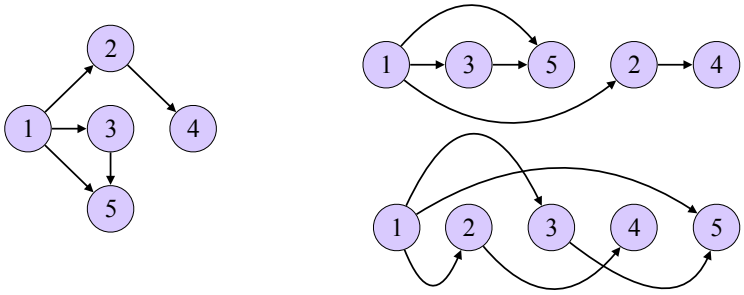


Figura 9.20: Possono esistere più ordinamenti topologici.

Un approccio banale potrebbe essere il seguente:

- trovo un nodo senza archi entranti;
- aggiungo questo nodo nell'ordinamento e lo rimuovo dal grafo insieme a tutti i suoi archi;
- ripeto questa procedura fino a quando tutti i nodi sono stati rimossi.

Si può fare meglio di così. Eseguiamo una visita in profondità nel quale l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista in *post*-ordine. E restituiamo la lista così ottenuta. Restituiamo in output la sequenza dei nodi ordinati per tempo decrescente di fine.

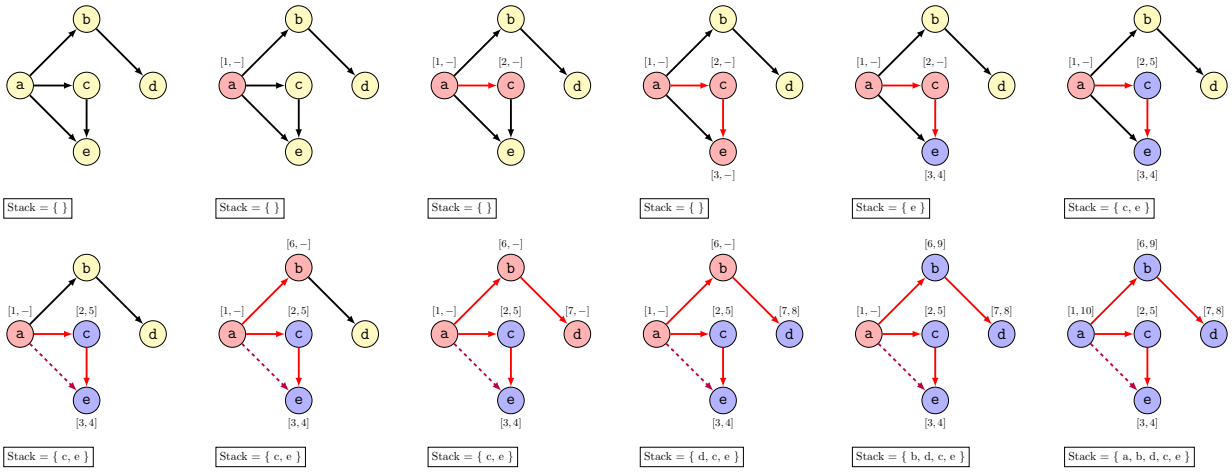


Figura 9.21: Esempio di ordinamento topologico

Algoritmo 13: Ordinamento topologico di un grafo orientato aciclico

```
// ritorna una pila in cui il primo elemento è il primo elemento dell'ordinamento
STACK topSort(GRAPH G)
    STACK S ← Stack
    boolean[] visited ← new boolean[1...G.size]
    foreach u ∈ G.V do
        visited[u] ← false
    foreach u ∈ G.V do // per ogni nodo del grafo
        if not visitato[u] then // se non l'ho visitato
            // effettua una chiamata ricorsiva
            ts-dfs(G, u, visitato, S)
    return S

// restituisce l'ordinamento topologico dei nodi di un DAG
int ts-dfs(GRAPH G, NODE u, boolean[] visitato, STACK S)
    visitato[u] ← true // imposta il nodo come visitato
    foreach v ∈ G.adj(u) do
        // è una grafo diretto aciclico quindi non ho bisogno di ricordarmi da dove sono venuto
        if not visitato[v] then
            // effettua una visita in profondità
            i ← ts-dfs(G, u, visitato, S)
    S.push(u) // aggiungi il nodo in testa alla pila
```

Quando termino tutte le chiamate ricorsive l'algoritmo restituisce un ordinamento topologico dei nodi del grafo dato in input; quando un nodo è "finito" tutti i suoi discendenti sono stati scoperti e aggiunti alla lista. Aggiungendolo in testa alla lista, il nodo si trova prima dei nodi a cui i suoi archi puntano, ossia i suoi discendenti.

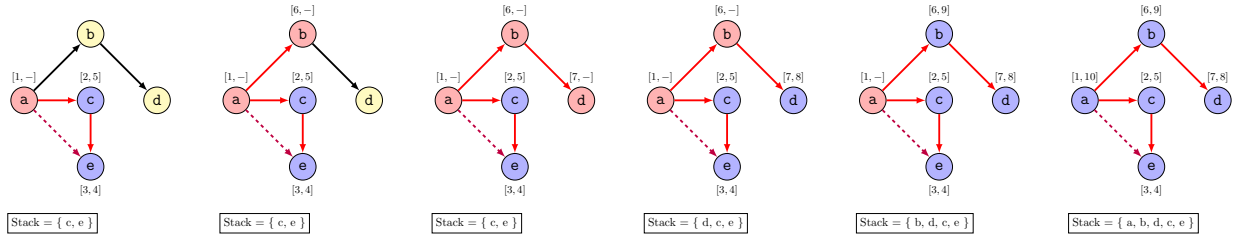


Figura 9.22: Esempio di ordinamento topologico alternativo al precendente, il quale dimostra informalmente che l'algoritmo funziona partendo da qualsiasi nodo

9.5 Componenti fortemente connesse

Definizione 9.5.1 (Grafo fortemente connesso)**.** Un grafo orientato $G = (V, E)$ è fortemente connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.

Definizione 9.5.2 (Componente fortemente connessa)**.** Un grafo $G' = (V', E')$ è una componente fortemente connessa di G se e solo se G' è un sottografo connesso e massimale di G.

Le definizioni di fortemente connessa è identica alla definizione di componente connessa, ma si opera su grafi orientati, mentre prima operavamo su grafi non orientati.

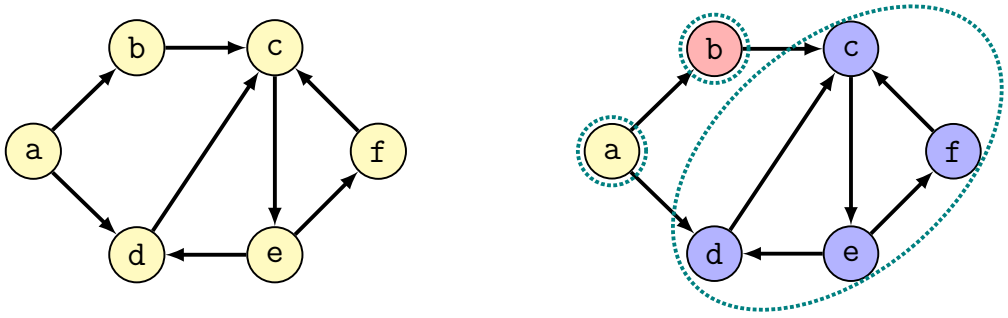


Figura 9.23: (a) e (b) sono componenti connesse massimali, (c, d, e, f) è una componente fortemente connessa.

Per definire le componenti fortemente connesse potremmo applicare l'algoritmo cc; purtroppo il risultato dipende dal nodo di partenza.

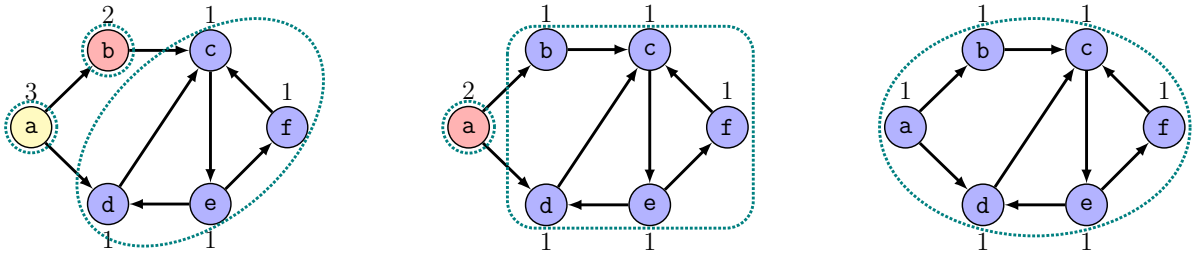


Figura 9.24: Il risultato dipende dal nodo di partenza.

9.5.1 L'algoritmo di Kosaraju

L'algoritmo di Kosaraju:

1. effettua una visita in profondità del grafo G
2. ne calcola il grafo trasposto G^T (ossia il grafo con la direzione degli archi invertiti)
3. esegue una visita in profondità sul grafo trasposto G^T utilizzando l'algoritmo cc, esaminando i nodi nell'ordine inverso di tempo di fine della prima visita;

Le componenti connesse (e i relativi alberi *depth-first*) rappresentano le componenti fortemente connesse di G.

Algoritmo 14: Algoritmo di Kosaraju

```
// indentifica le componenti fortemente connesse
int[] scc(GRAPH G)
    STACK S ← topSort(G)  // prima visita
    G^T ← transpose(G)  // trasposizione del grafo
    return cc(G^T, S)  // seconda visita
```

Restituisce un vettore di interi che associa ad ogni nodo l'id della sua componente fortemente connessa. Applicando l'algoritmo di ordinamento topologico *su un grafo generale*, siamo sicuri che:

- se un arco (u, v) non appartiene ad un ciclo, allora u viene listato prima di v nella sequenza ordinata;
- gli archi di un ciclo vengono listati in qualche ordine (che è ininfluente).

Utilizziamo quindi la procedura topSort per ottenere i nodi in ordine decrescente di tempo di fine.

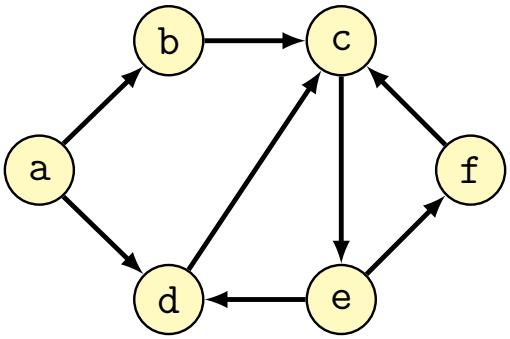
Nella topSort non calcoliamo nemmeno i tempi di fine, li utilizziamo semplicemente per dimostrare che i nodi vengono ordinati in ordine inverso di tempo di fine.

Definizione 9.5.3 (Grafo trasposto)**.** Dato un grafo orientato $G = (V, E)$, il grafo trasposto $G_t = (V, E_T)$ ha gli stessi nodi e gli archi orientati in senso opposto: $E_T = \{(u, v) \mid (v, u) \in E\}$

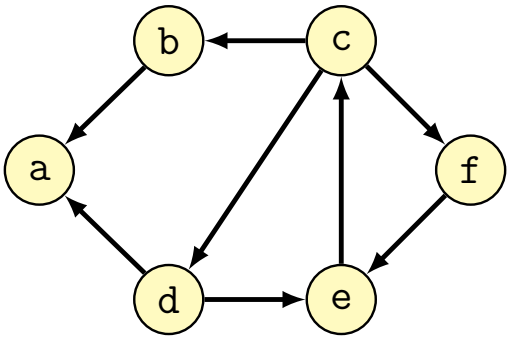
Algoritmo 15: Calcolo del grafo trasposto

```
// restituisce il grafo transposto
int[] transpose(GRAPH G)
    GRAPH G^T ← Graph  // creo il grafo
    foreach u ∈ G.V do
        G^T.insertNode(u)  // aggiungo gli stessi nodi
    foreach u ∈ G.V do
        foreach v ∈ G.adj(u) do
            G^T.insertEdge(v, u)  // li aggiungo in ordine inverso
    // restituisco il grafo trasposto
    return G^T
```

Complessità Il costo computazionale totale ammonta a $\mathcal{O}(m+n)$, in quanto aggiungere i nodi costa $\mathcal{O}(n)$, gli archi $\mathcal{O}(m)$ ed ogni operazione costa $\mathcal{O}(1)$.



(a) Grafo originale



(b) Grafo trasposto

Algoritmo 15: Identificazione delle componenti connesse alternativa

```
// parte iterativa
int[] cc(Graph G, Stack S)
    // creo un vettore della dimensione dei nodi del grafo
    int[] id ← new int[1...G.size]
    // inizializzo il vettore
    foreach u ∈ G.V do
        id[u] ← 0
    // contatore delle componenti connesse
    int counter ← 0
    while not S.isEmpty do // fintanto che la pila non è vuota
        u ← S.pop
        if id[u] == 0 then // ho trovato una nuova componente connessa
            counter++ // aggiorno il contatore
            // effettuo una chiamata ricorsiva sul nodo scoperto
            ccdfs(G, counter, u, id)
    // restituisco l'identificativo della componente connessa
    return id
// visita ricorsiva di ciascuna componente
ccdfs(Graph G, int counter, Node u, int[] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u:       il nodo che sto visitando
    // id:      l'identificativo della componente
    // memorizzo l'identificativo della cc
    id[v] ← counter
    foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
        if id[v] == 0 then // non è ancora stato visitato
            // v:  il nodo su cui vado ad operare
            ccdfs(G, counter, v, id) // lo visito ricorsivamente
```

Invece di esaminare i nodi in ordine arbitrario, questa versione di cc li esamina nell'ordine LIFO memorizzato nello stack.

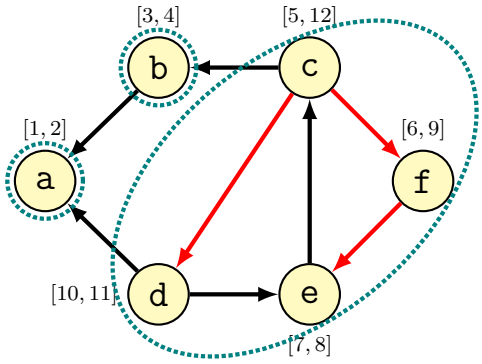


Figura 9.26: Identificazione delle componenti fortemente connesse; l'ordine in cui li visito è quello della pila `{ a, b, c, e, d, f }`

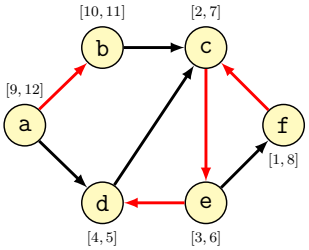
Algoritmo 16: Identificazione delle componenti fortemente connesse

```
// indentifica le componenti fortemente connesse
int[] scc(GRAPH G)
    STACK S ← topSort(G)  // prima visita
    G^T ← transpose(G)  // trasposizione del grafo
    return cc(G^T, S)  // seconda visita
```

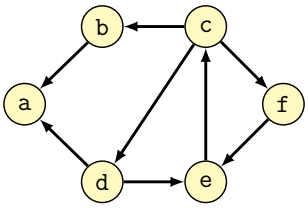
Complessità Ognuna delle fasi che compongono l'algoritmo:

1. visita in profondità della topSort;
2. la trasposizione del grafo di transpose;
3. la visita delle componenti connesse.

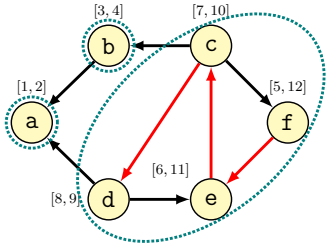
richiede un costo di $\mathcal{O}(m+n)$. Quindi la complessità è lineare nel numero di nodi e nel numero di archi, ossia $\mathcal{O}(m+n)$.



(a) La prima visita parte da f, la seconda dal nodo a



(b) Calcolo il grafo trasposto G^T



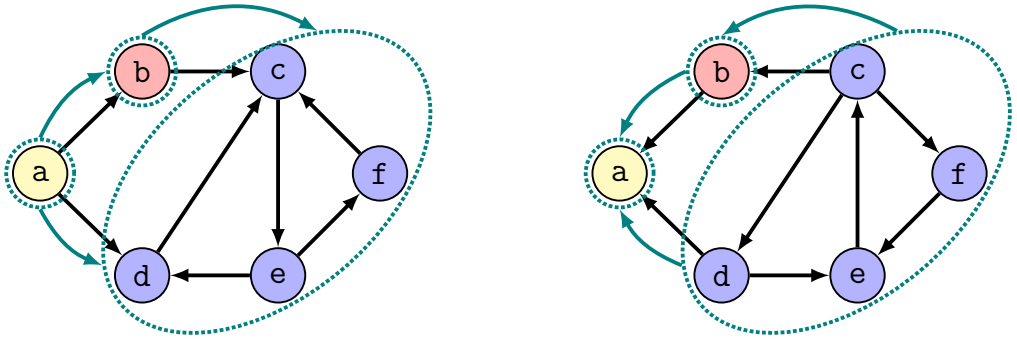
(c) Identificazione delle componenti fortemente connesse

Figura 9.27: Una seconda esecuzione dell'ordinamento topologico

9.5.2 Dimostrazione di correttezza

Definizione 9.5.4 (Grafo delle componenti)**.** Il grafo delle componenti si definisce come il grafo $C(G) = (V_c, E_c)$, dove:

- $V_c = \{C_1, C_2, \ldots, C_k\}$, dove C_i è la i-esima componente fortemente connessa del grafo G;
- $E_c = \{(C_i, C_j) \mid \exists (u_i, v_i) \in E : u_i \in C_i \wedge u_j \in C_j\}$



(a) Grafo delle componenti del grafo

(b) Grafo delle componenti del grafo trasposto

Figura 9.28

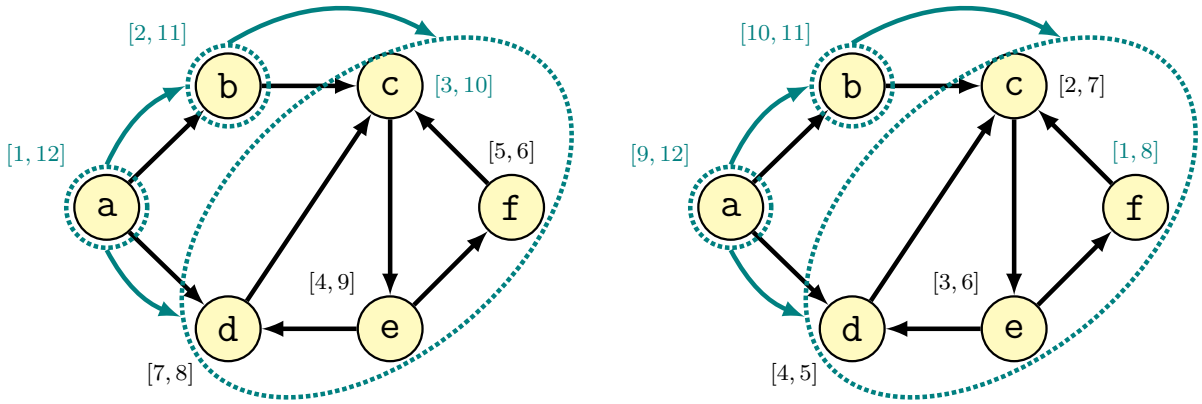
Quando si traspone un grafo fortemente connesso, l'insieme di nodi che compongono il grafo delle componenti connesse rimane lo stesso, mentre gli archi sono in direzione inversa.

Nota. Il grafo delle componenti è aciclico poiché se contenesse un ciclo, il ciclo stesso sarebbe una più grande componente fortemente connessa, il che è assurdo.

Se il grafo delle componenti è aciclico, allora posso farne un'ordinamento topologico. Possiamo definire quindi $dt(C) = \min\{dt(u) \mid u \in C\}$ e $ft(C) = \max\{ft(u) \mid u \in C\}$, i quali corrisponderanno ai tempo di inizio e di fine del primo nodo visitato in C.

Teorema. *Siano C e C' due distinte componenti fortemente connesse nel grafo orientato $G = (V, E)$. Se c'è un arco $(C, C') \in E_c$, allora $ft(C) > ft(C')$.*

La componente che finisce la visita per ultima è la componente dalla quale si possono raggiungere le altre componenti.



(a) Grafo delle componenti del grafo

(b) Grafo delle componenti del grafo trasposto

Corollario. Siano C_u e C_v due componenti fortemente connesse distinte del grafo orientato $G = (V, E)$. Se c'è un arco $(u, v) \in E_t$ tale che $u \in C_u$ e $v \in C_v$, allora $ft(C_u) < ft(C_v)$.

In generale:

$$(u, v) \in E_t \Rightarrow (v, u) \in E \Rightarrow (C_v, C_u) \in E_c \Rightarrow ft(C_v) > ft(C_u) \Rightarrow ft(C_u) < ft(C_v)$$

Nel nostro caso:

$$(b, a) \in E_t \Rightarrow (a, b) \in E \Rightarrow (C_a, C_b) \in E_c \Rightarrow \underset{12}{ft(C_a)} > \underset{11}{ft(C_b)} \Rightarrow \underset{11}{ft(C_b)} < \underset{12}{ft(C_a)}$$

Se la componente C_u e la componente C_v sono connesse da un arco $(u, v) \in E_t$, allora possiamo dedurne che $ft(C_u) < ft(C_v)$ (dal corollario) e che la visita di C_v inizierà prima della visita di C_u (dal teorema). Non esistendo cammini fra C_v e C_u in G_t (altrimenti il grafo sarebbe ciclico) la visita di C_v non raggiungerà C_u. In altre parole, l'algoritmo cc assegnerà correttamente gli indentificatori delle componenti ai nodi.

Algoritmo di Tarjan

L'algoritmo di Tarjan è preferito a quello di Kosaraju il quale, avendo comunque la medesima complessità computazionale ($\mathcal{O}(m + n)$), è preferito a quest'ultimo in quanto necessita di una sola visita e non richiede la trasposizione del grafo (al posto di una doppia visita e di memoria aggiuntiva).

Applicazioni

Gli algoritmi sulle componenti fortemente connesse possono essere utilizzati per risolvere il problema "2-satisfiability" (2-SAT), un problema di soddisfacibilità booleana con clausole composte da coppie di letterali.