Dati e Algoritmi 1: A. Pietracaprina

Grafi

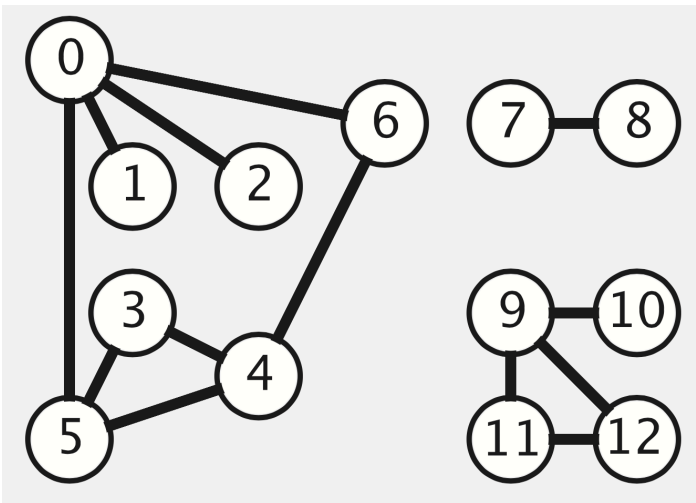
Definizione di Grafo

Definizione

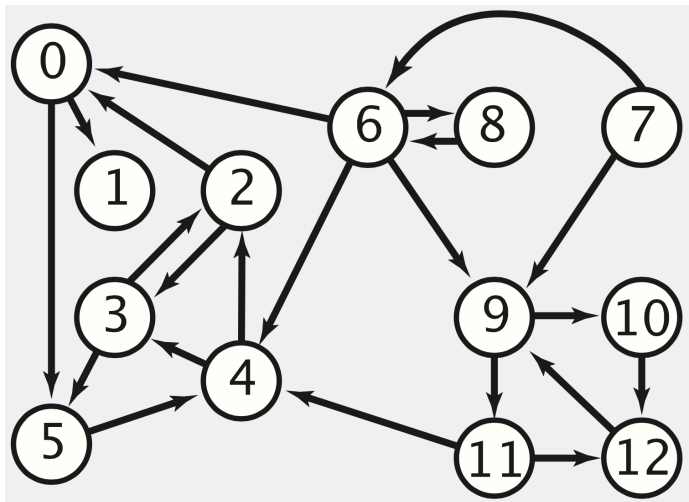
Grafo $G = (V, E)$:

- $V \equiv$ insieme di vertici (o nodi)
- $E \equiv$ collezione di archi (coppie di vertici).

Il grafo si dice diretto se ogni arco $(u, v) \in E$ è una coppia ordinata $(u \to v)$, altrimenti si dice non diretto $(u - v)$



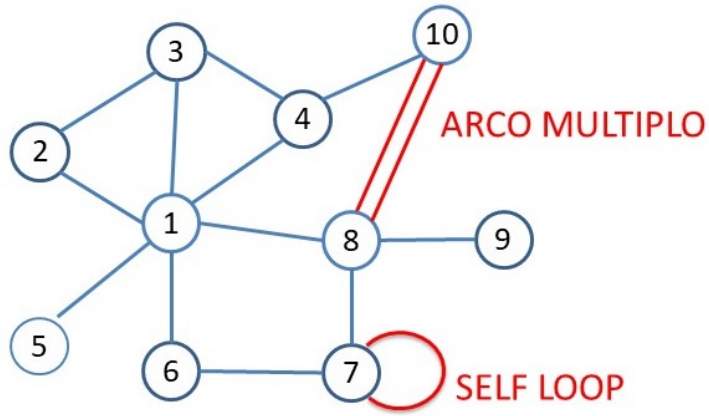
Grafo *non diretto*



Grafo *diretto*

Osservazioni sulla definizione

- La definizione ammette la presenza di archi multipli tra due vertici (per questo E è una *collezione* e non un *insieme*), e di archi (u, u) (self loop).



- Un grafo semplice è un grafo senza archi multipli e senza self loop
- Per alcune applicazioni agli archi (e a volte ai vertici) sono associati dei pesi.

In questo corso studiamo grafi semplici non diretti e non pesati

Terminologia

- In Inglese:
  - nodi/vertici: nodes/vertices ([GTG14] usa vertices)
  - archi: edges/arcs ([GTG14] usa edges). Di solito si usa edges per grafi non diretti e arcs per grafi diretti.
- Dato un arco $e = (u, v) \in E$, si dice che e è incidente su u e v e che u e v sono adiacenti.
- I vicini di un vertice v sono tutti i vertici u tali che $(v, u) \in E$.
- Il grado di un vertice $v \in V$ (degree(v)) è il numero di archi di E incidenti su v.

Esempi reali di grafi

- Reti sociali (ad es. Facebook, Twitter, LinkedIn ...)

Esempi reali di grafi (continua)

- Reti stradali (ad es. per navigatori)

Esempi reali di grafi (continua)

- Reti biologiche

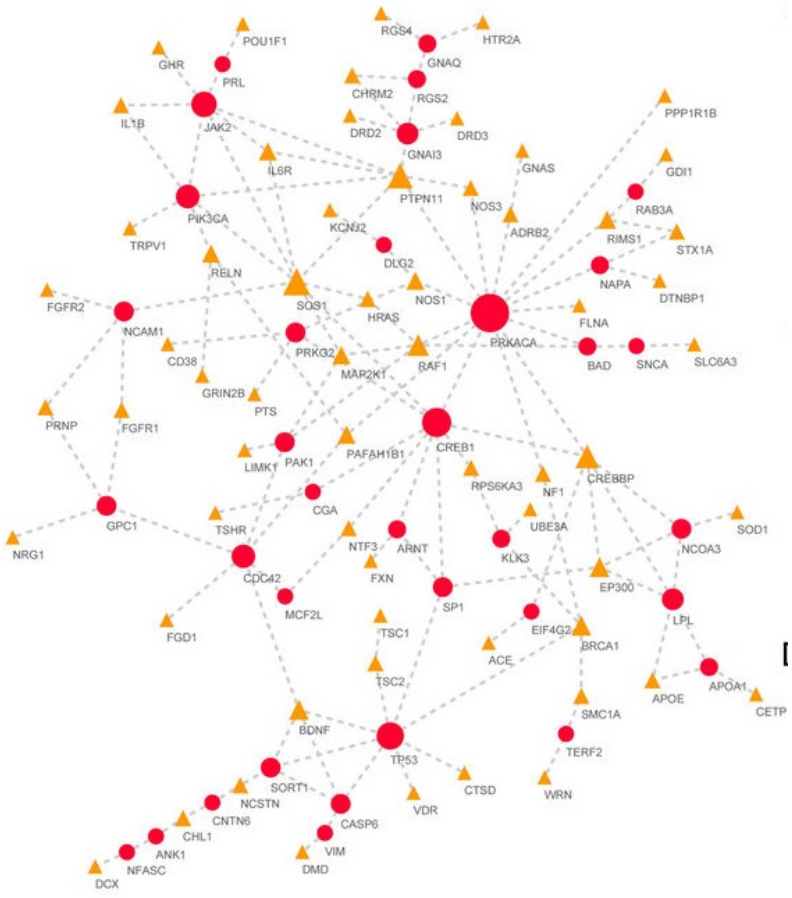


Figure: Network of IQ-related genes

Esempi reali di grafi (continua)

- Internet
- Web
- Reti di comunicazione per supercomputer
- Reti peer-to-peer
- Reti wireless, reti di sensori

Concetti fondamentali: cammino

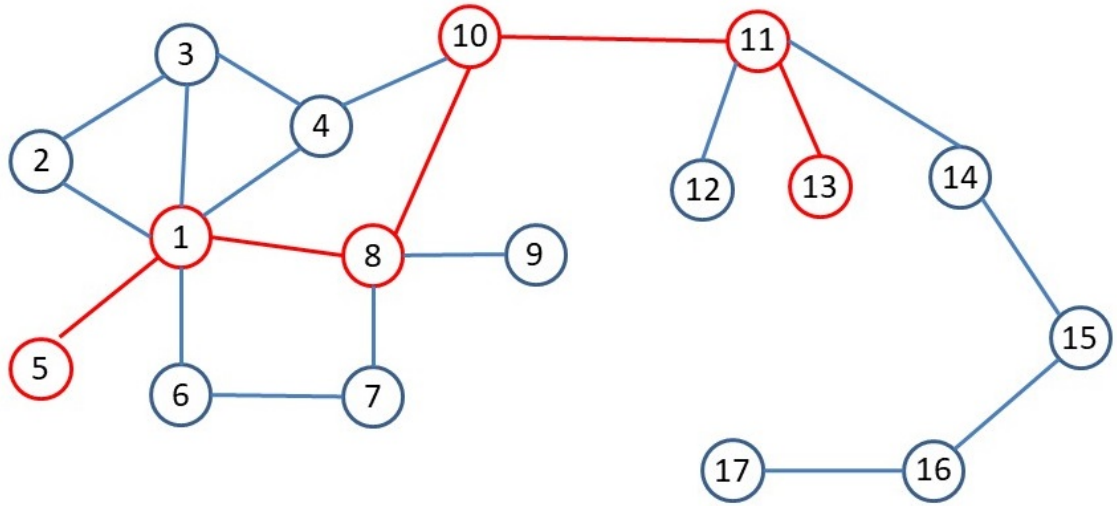


Figure: Grafo G=(V,E)

Cammino semplice (simple path): $u_1, u_2, \ldots, u_k$ con tutti i vertici distinti, e $(u_i, u_{i+1}) \in E$ per $1 \leq i < k$. Nell'esempio: 5,1,8,10,11,13.

La lunghezza del cammino è il numero di archi (u_i, u_{i+1}), ovvero $k - 1$. Nel caso gli archi siano pesati, la lunghezza è la somma dei pesi degli archi (u_i, u_{i+1}).

Oss.: l'aggettivo *semplice*, spesso omesso, indica che i vertici tutti distinti. Altrimenti si parla solo di *cammino*. Ad es., 5,1,8,7,6,1,4 non è semplice.

Concetti fondamentali: ciclo

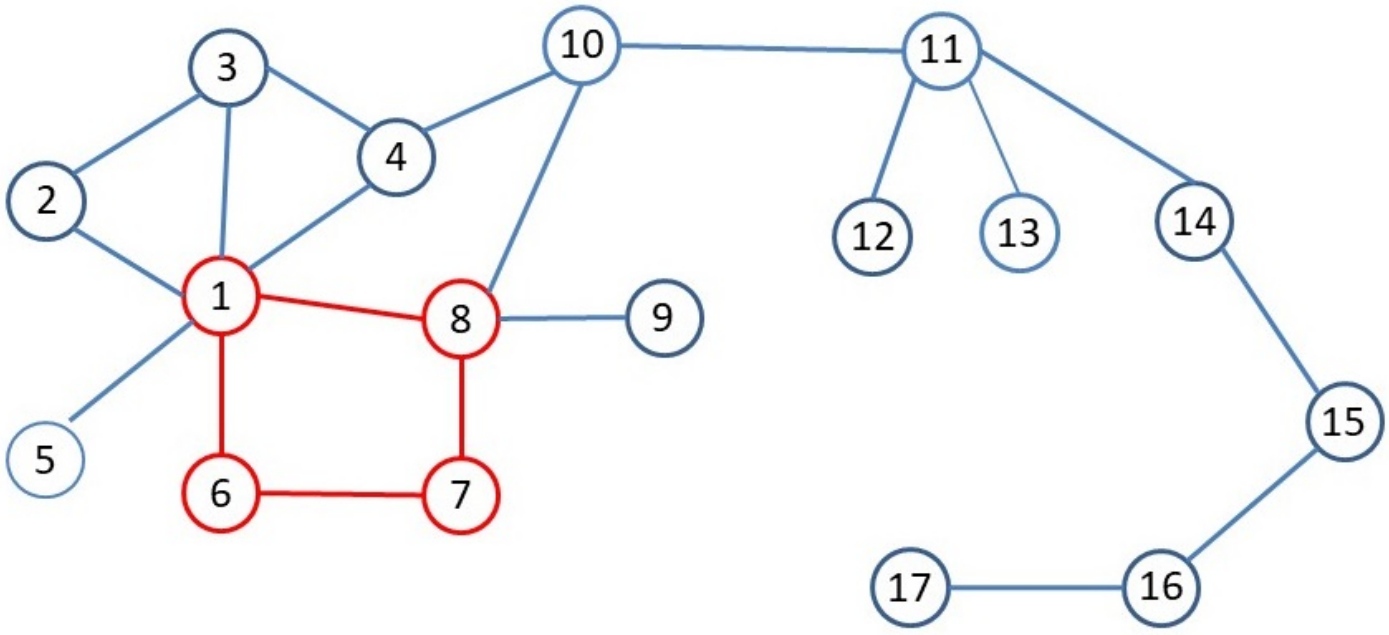


Figure: Grafo G=(V,E)

Ciclo semplice (simple cycle): cammino semplice $u_1, u_2, \ldots, u_k$ con $u_1 = u_k$. Nell'esempio: 1,6,7,8,1.

La lunghezza del ciclo è quella del cammino $u_1, u_2, \ldots, u_k$.

Oss.: Senza l'aggettivo *semplice* si ammette la presenza di vertici ripetuti, oltre ai due estremi: ad es., 6,1,3,4,1,6

Concetti fondamentali: sottografo

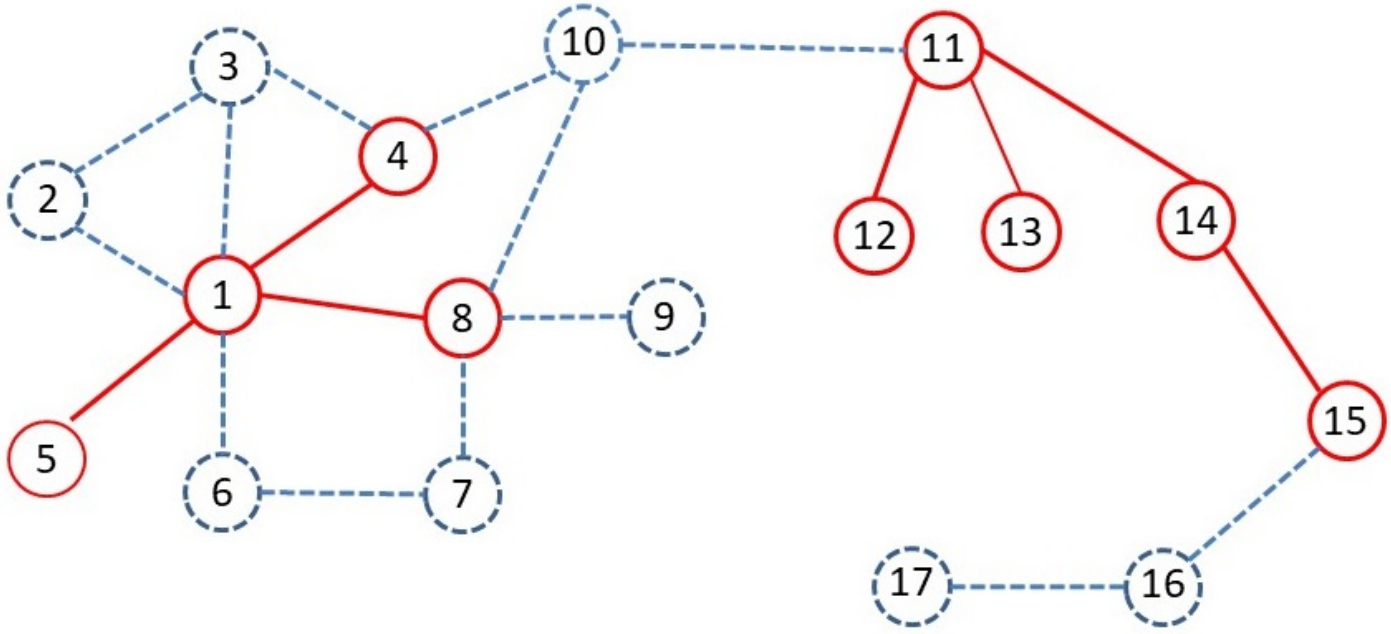


Figure: Grafo G=(V,E)

Sottografo (subgraph): $G' = (V', E')$ con $V' \subseteq V$, $E' \subseteq E$, e tale che gli archi di E' incidono solo su V'. Nell'esempio: vertici e archi in rosso.

Concetti fondamentali: sottografo di copertura

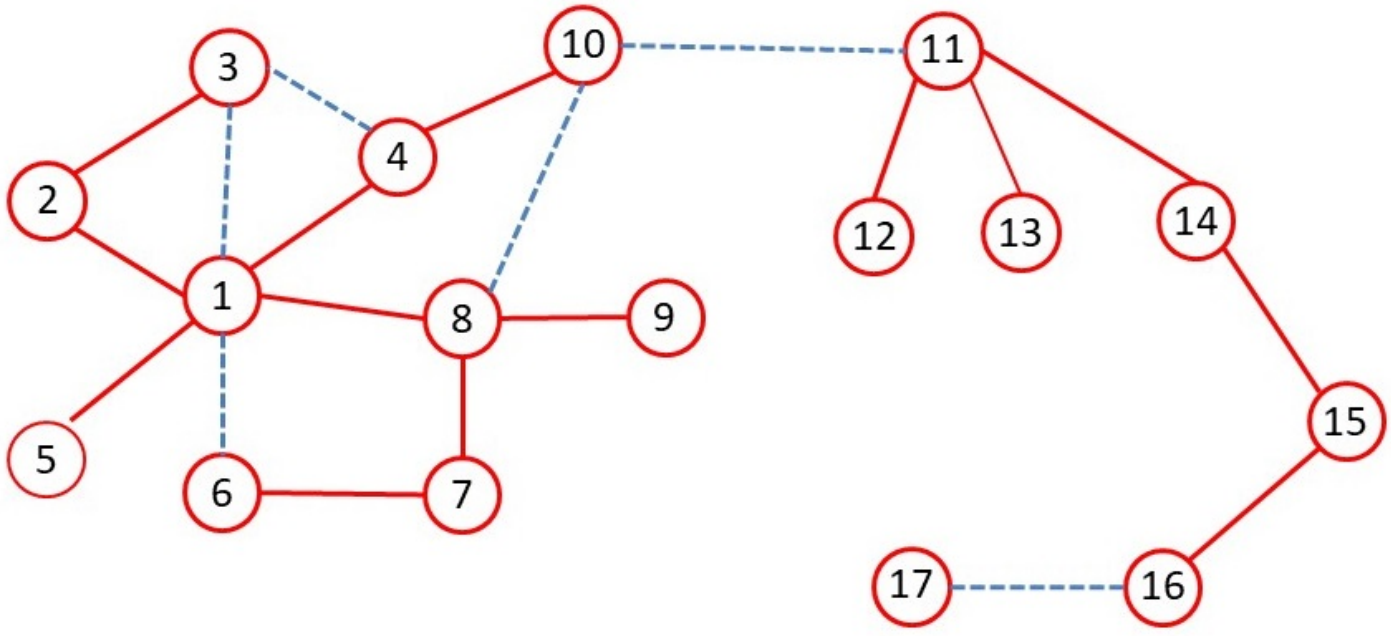


Figure: Grafo G=(V,E)

Sottografo di copertura (spanning subgraph): sottografo $G' = (V', E')$ con $V' = V$. Nell'esempio: vertici e archi in rosso.

Concetti fondamentali: grafo connesso

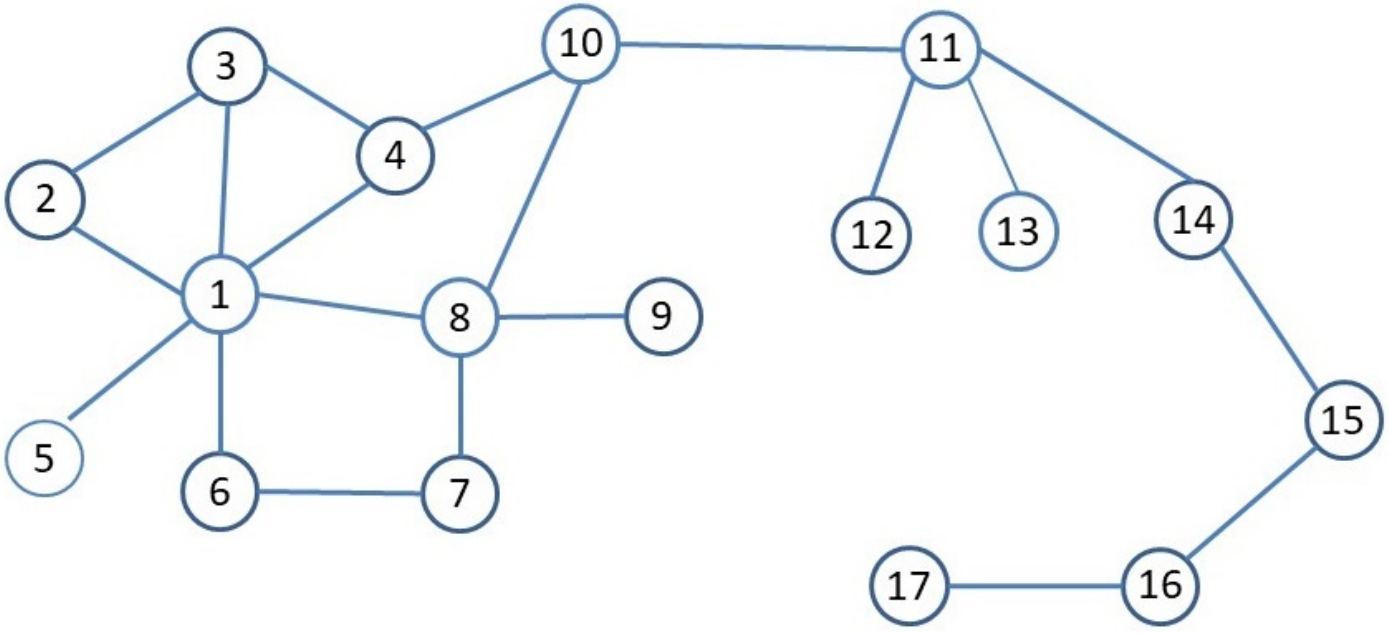


Figure: Grafo G=(V,E)

Un grafo $G = (V, E)$ si dice connesso (connected) se per ogni $u, v \in V$ esiste un cammino che inizia in u e termina in v.

Concetti fondamentali: grafo disconnesso

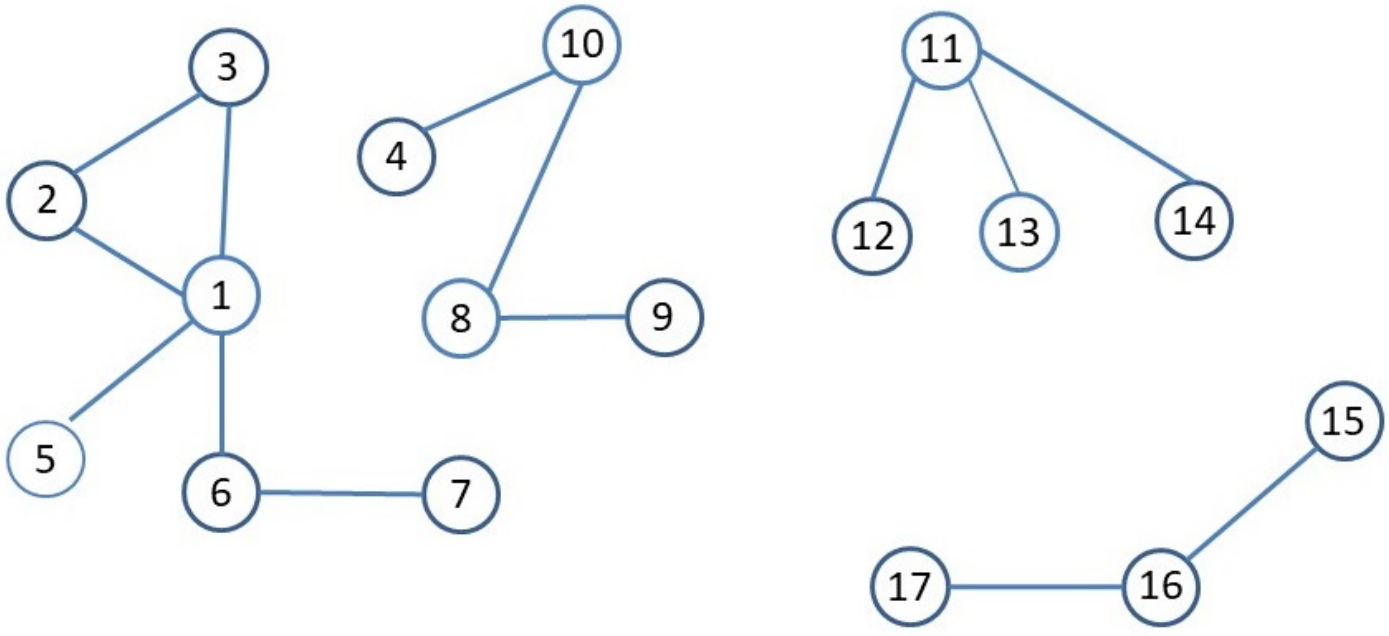


Figure: Grafo G=(V,E)

Un grafo $G = (V, E)$ si dice disconnesso (disconnected) se non è connesso, ovvero esistono due vertici $u, v \in V$ per i quali non esiste un cammino che inizia in u e termina in v.

Concetti fondamentali: componenti connesse

Le componenti connesse (connected component) di un grafo $G = (V, E)$ sono una partizione di G in sottografi $G_i = (V_i, E_i)$, per $1 \leq i \leq k$, tali che

- $G_i = (V_i, E_i)$ è connesso, per $1 \leq i \leq k$
- $V = V_1 \cup V_2 \cup \cdots \cup V_k$ (*partizione* di V)
- $E = E_1 \cup E_2 \cup \cdots \cup E_k$ (*partizione* di E)
- $\forall i \neq j$, non esistono archi in E tra V_i e V_j

Osservazioni:

- $\{G_i : 1 \leq i \leq k\}$ sono sottografi connessi massimali
- G connesso $\Rightarrow k = 1$
- la partizione di G in componenti connesse è univoca

Esempio

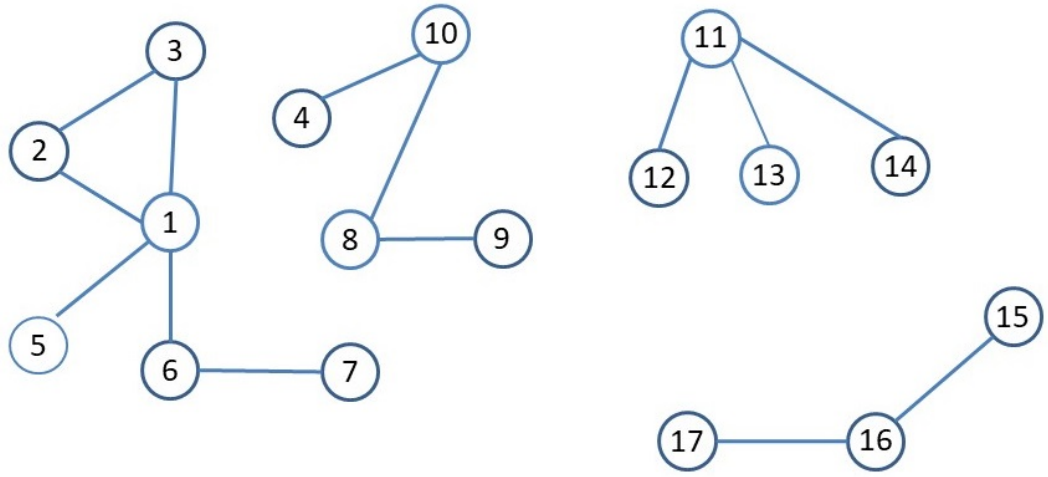


Figure: Grafo G=(V,E)

Nella figura le componenti connesse sono 4 con

- $V_1 = \{1, 2, 3, 5, 6, 7\}$
- $V_2 = \{4, 8, 9, 10\}$
- $V_3 = \{11, 12, 13, 14\}$
- $V_4 = \{15, 16, 17\}$

Concetti fondamentali: albero

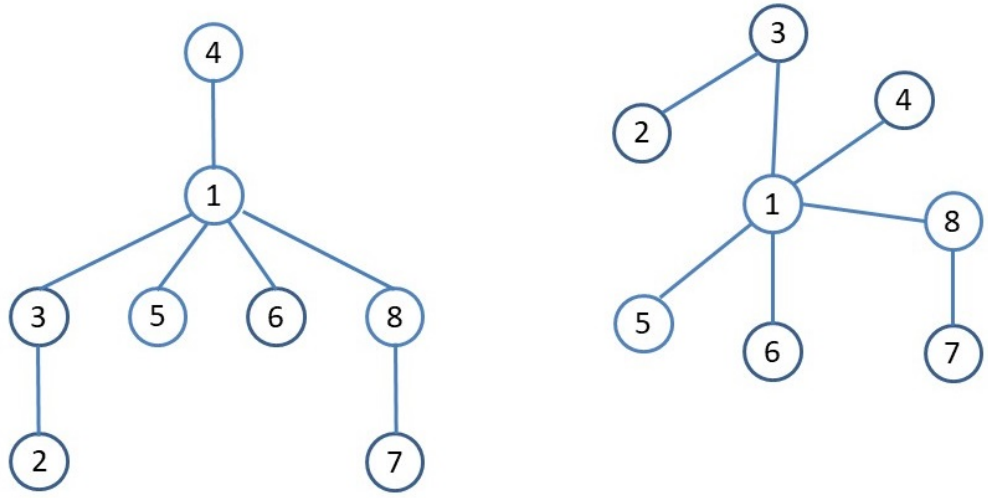
Un albero radicato (rooted tree) è un grafo $G = (V, E)$ tale che

- Esiste un vertice *radice* $r \in V$
- Per ogni $u \in V$, con $u \neq r$, esiste un unico padre $p(u) \in V$, e si ha che $E = \{(u, p(u)) \ : \ u \in V, u \neq r\}$
- Per ogni $u \in V$ andando di padre in padre si raggiunge r

Un albero libero (free tree) è un grafo $G = (V, E)$ connesso e senza cicli

Una foresta (forest) è un grafo $G = (V, E)$ senza cicli, ovvero un insieme di alberi liberi disgiunti. Si noti che un albero è anche una foresta ma non vale il viceversa

Esempio

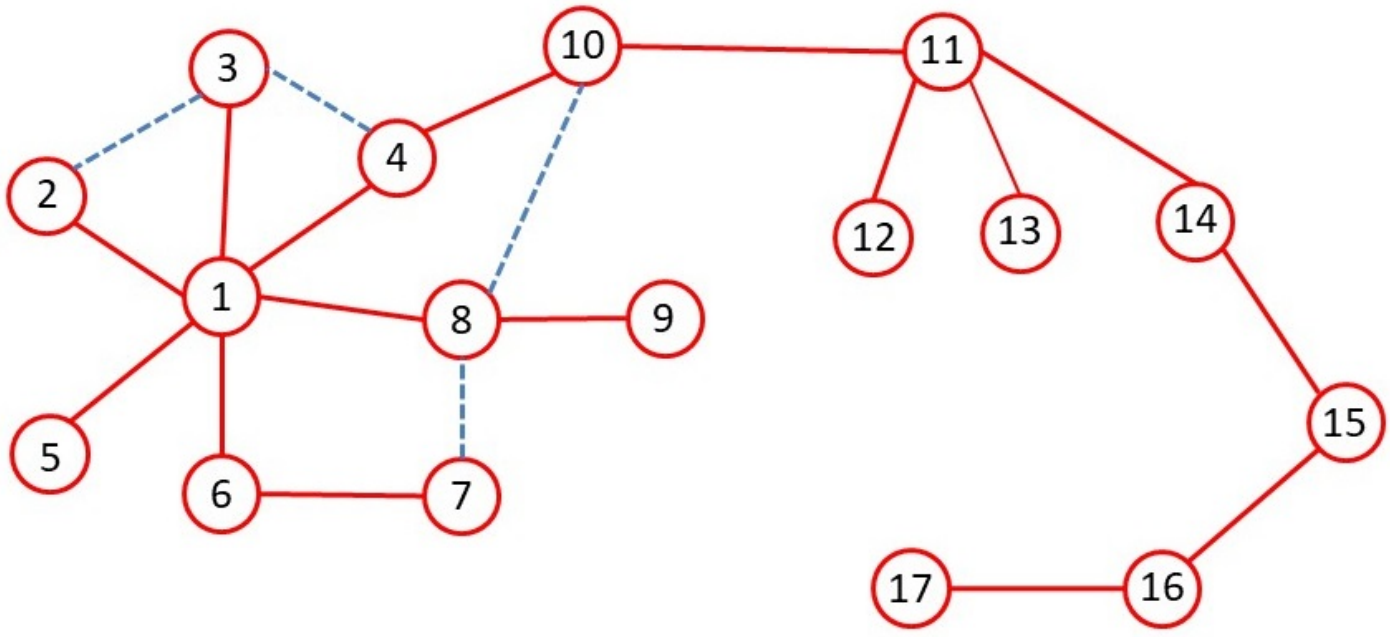


Rooted Tree Free Tree

Osservazioni

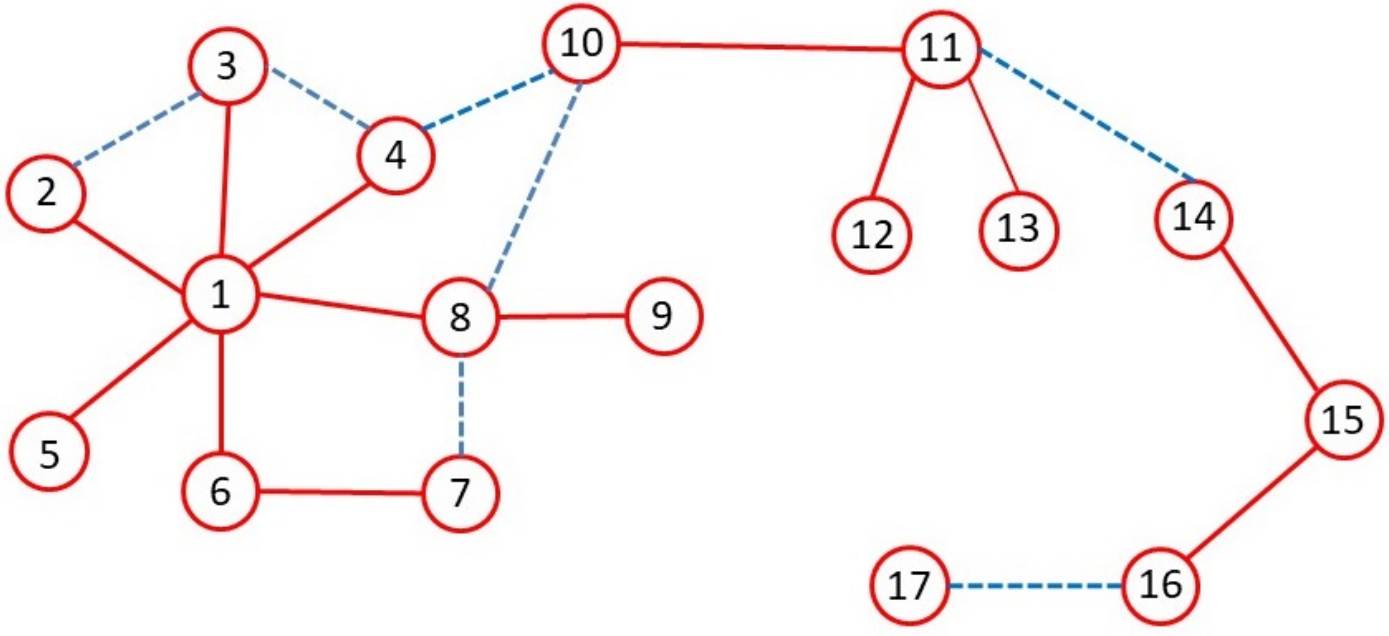
- Il concetto di albero radicato è lo stesso di quello studiato all'inizio del corso.
- I concetti di albero radicato e albero libero sono *equivalenti:* ogni albero radicato è un grafo connesso senza cicli, e ogni albero libero, può essere visto come albero radicato scegliendo opportunamente una radice e determinando di conseguenza le relazioni padre-figlio.

Concetti fondamentali: spanning tree



Uno spanning tree di un grafo G è uno spanning subgraph connesso e senza cicli (free tree). Nell'esempio: vertici e archi in rosso. Esiste solo se G è connesso.

Concetti fondamentali: spanning forest



Una spanning forest di un grafo G è uno spanning subgraph senza cicli. Nell'esempio: vertici e archi in rosso.

Primitive Importanti

- Traversal: Esplorazione sistematica del grafo (es.: crawling)
- Connettività: Verifica se il grafo è connesso (es.: reti wireless)
- Identificazione delle componenti connesse: (es.: reti wireless)
- Ricerca di cammini minimi: (es.: navigatore)
- Ricerca di minimum spanning tree: (es.: broadcast efficiente)
- Stima della distanza media/massima: (es.: Facebook)

Proprietà dei Grafi

Proposizione

Sia $G = (V, E)$ un grafo non diretto e semplice con $|V| = n$ vertici e $|E| = m$ archi. Valgono le seguenti proprietà:

1. $\sum_{v \in V} degree(v) = 2m$
2. $m \leq \binom{n}{2}$, e quindi $m \in O\left(n^2\right)$
3. Se G è un albero, allora $m = n - 1$
4. Se G è connesso, allora $m \geq n - 1$
5. Se G è senza cicli (cioè una foresta) $m \leq n - 1$

Proprietà dei Grafi (continua)

Dimostrazione

1. $\sum_{v \in V} degree(v) = 2m$. Ogni arco è contato esattamente 2 volte nella sommatoria.
2. $m \leq \binom{n}{2}$. Dato che il grafo è semplice, E è un sottoinsieme di tutte le $\binom{n}{2}$ possibili coppie di vertici.
3. Se G è un albero, allora $m = n - 1$. Consideriamo G come un rooted tree (grazie all'equivalenza tra rooted tree e free tree). Allora E rappresenta *relazioni padre-figlio*, che sono $n - 1$ (ogni nodo non radice ha un unico padre).

Proprietà dei Grafi (continua)

Dimostrazione (continua)

4. Se G è connesso, allora $m \geq n - 1$.

   Si supponga di eseguire il seguente ciclo:

   While($\exists$ `ciclo` C) **do** `elimina da` G `un arco di` C.

   Si ha che:

   - Alla fine di ogni iterazione, G è connesso
   - Alla fine del ciclo, G è connesso e senza cicli (ovvero un free tree) con $m' = n - 1 \leq m$ archi.

Proprietà dei Grafi (continua)

Dimostrazione (continua)

5. Se G è senza cicli (cioè una foresta) $m \leq n - 1$.

   Si supponga che G sia composto da $k \geq 1$ alberi disgiunti, e che l'i-esimo albero abbia n_i vertici ed m_i archi. Quindi si ha che $n = \sum_{i=1}^{k} n_i$ e $m = \sum_{i=1}^{k} m_i$. Ma dato che $m_i = n_i - 1$, in quanto si tratta di alberi, si ha che

$$m = \sum_{i=1}^{k} m_i = \sum_{i=1}^{k} (n_i - 1) = n - k \leq n - 1.$$

□

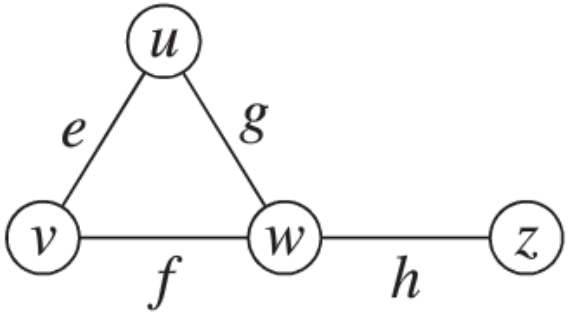
Rappresentazione di Grafi

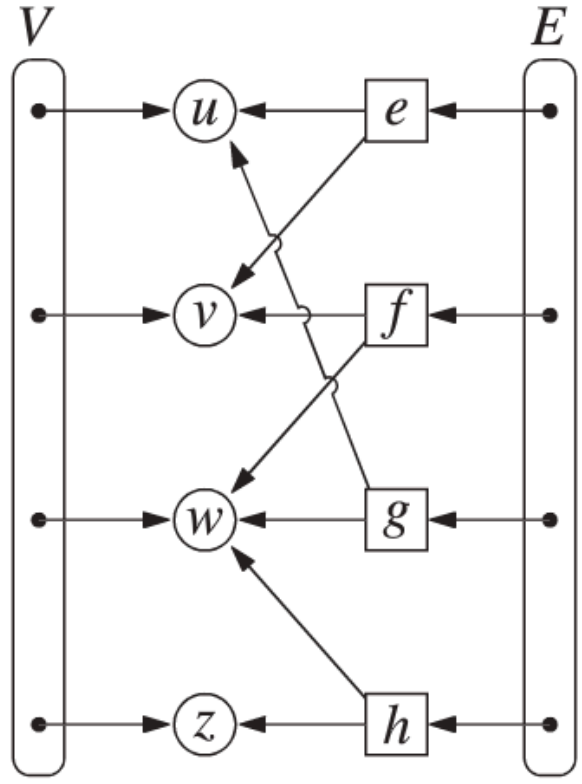
Sia $G = (V, E)$ con n vertici ed m archi. Se non diversamente specificato, assumeremo che i vertici siano associati agli interi $1, 2, \ldots, n$.

STRUTTURE di BASE: Liste di vertici e archi. Per comodità notazionale le rappresentiamo come array.

- Lista di vertici L_V: per ogni $v \in V$, $L_V[v]$ contiene tutte le informazioni su v
- Lista di archi L_E: per ogni $e = (u, v) \in E$, $L_E[e]$ contiene tutte le informazioni su e e il collegamento a $L_V[u]$ e $L_V[v]$.

Esempio

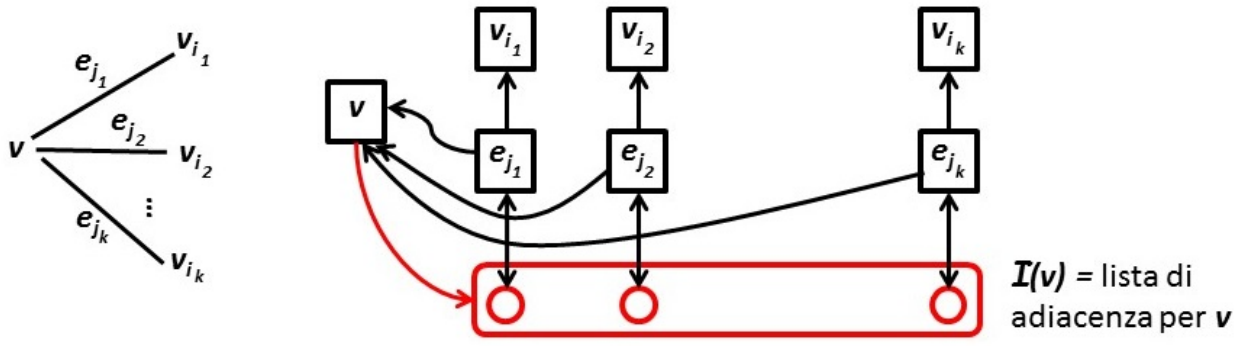




Rappresentazione di Grafi (continua)

Per permettere un accesso più diretto agli archi si usa una delle due strutture seguenti, in aggiunta alle strutture di base L_V e L_E:

- Liste di Adiacenza (Adjacency List): Per ogni vertice $v \in V$ si ha una lista $I(v)$ di puntatori agli archi (elementi di L_E) incidenti su v.



L'uso delle liste di adiacenza, che è il più frequente, consente l'accesso veloce ai vicini di un vertice e richiede occupando *spazio lineare nella taglia del grafo*.

Rappresentazione di Grafi (continua)

- Matrice di Adiacenza (Adjacency Matrix) A: matrice $n \times n$ tale che i vertici sono in corrispondenza 1-1 con righe e colonne e

$$A[i_1, i_2] \doteq \begin{cases} null & \text{se } (i_1, i_2) \notin E \\ \text{puntatore a } e = (i_1, i_2) \in L_E & \text{se tale arco esiste} \end{cases}$$

L'uso della matrice di adiacenza, consente di testare velocemente la presenza di un arco e accedere alle sue informazioni, ma richiede *spazio quadratico nei vertici che può risultare superlineare nella taglia del grafo*. Per questo motivo è una rappresentazione utilizzata soprattutto nel caso di grafi densi (con numero di archi quadratico nei vertici).

Implementazione Java

Le seguenti librerie software in `Java` supportano la gestione e l'utilizzo di grafi nelle applicazioni.

- JGraphT (`http://jgrapht.org`)

- JUNG - Java Universal Network/Graph Framework (`http://jung.sourceforge.net`)

Graph Traversal di $G = (V, E)$

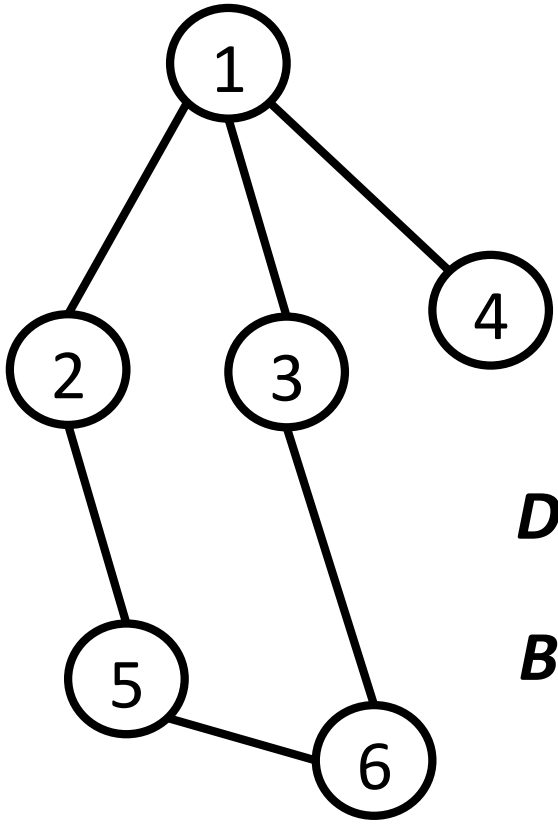
Procedura sistematica per esplorare G a partire da un vertice s visitando tutti i vertici.

- Depth-First Search (DFS): dopo la visita di un vertice si visita un vicino, poi un vicino del vicino, ecc.
- Breadth-First Search (BFS): dopo la visita di un vertice si visitano tutti i vicini prima di passare ai vicini dei vicini

Osservazioni:

- Come nel caso delle viste degli alberi, DFS e BFS costituiscono dei design pattern in cui l'operazione di *visita* può essere opportunamente istanziata in modo per risolvere specifici problemi: ad es. connettività, identificazione di spanning tree, ecc.
- Anche scorrendo le liste L_V e L_E si ottiene un'esplorazione completa del grafo. Tuttavia la non sistematicità di tale esplorazione la rende poco sfruttabile per la risoluzione di problemi.

Esempio



DFS: 1 ⟶ 2 ⟶ 5 ⟶ 6 ⟶ 3 ⟶ 4

BFS: 1 ⟶ 2 ⟶ 3 ⟶ 4 ⟶ 5 ⟶ 6

Depth-First Search: Algoritmo DFS

- Algoritmo ricorsivo che a partire da un vertice s "visita" tutti i vertici della componente connessa $C_s \subseteq G$ contenente s, toccando tutti i vertici e gli archi di C_s.
- Oltre alle strutture di base L_V e L_E assumiamo di usare le liste di adiacenza che inducono un ordine di visita dei vicini di ogni vertice.
- Ogni vertice v ha un campo $L_V[v]$.`ID` che vale 1, se v è stato visitato, e 0 altrimenti.
- Ogni arco e ha un campo $L_E[e]$.`label` che vale `null` se e non ha ancora etichetta, oppure riporta una delle due etichette `DISCOVERY EDGE` o `BACK EDGE`
- In un qualsiasi istante dell'esecuzione dell'algoritmo diremo che un vertice u è discoverable da un vertice v se esiste un cammino da v a u fatto di vertici non ancora visitati.

Depth-First Search: Algoritmo DFS (continua)

Algoritmo DFS(G, v) (*prima invocazione:* $v = s$)

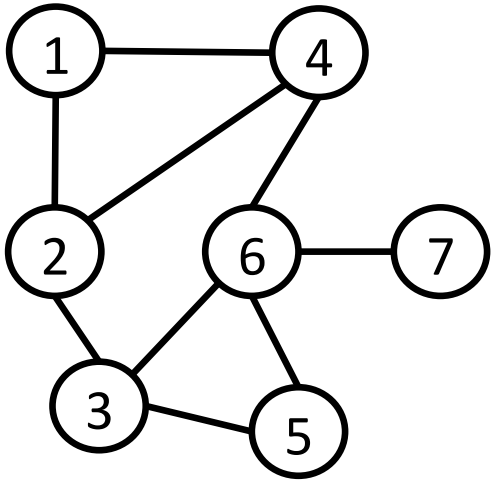
Input: grafo $G = (V, E)$ non diretto, vertice $v \in V$ non visitato
Output: tutti i vertici discoverable da v visitati e gli archi incidenti su essi etichettati come DISCOVERY o BACK EDGE

```
visita il vertice v; L_V[v].ID ← 1;
forall e ∈ G.incidentEdges(v) do
    if (L_E[e].label = null) then
        w ← G.opposite(v,e);
        if (L_V[w].ID = 0) then
            L_E[e].label ← DISCOVERY EDGE;
            DFS(G,w);
        else L_E[e].label ← BACK EDGE;
```

Osservazioni

- `incidentEdges(`v`)` restituisce un iteratore ai vicini di v.
- `opposite(`v`,`e`)` restituisce il vertice di e opposto a v.
- Si assume che nella prima invocazione `DFS(`G`,`s`)` tutti i vertici siano non visitati ($L_V[v]$`.ID=`0 per ogni $v \in V$), e tutti gli archi non etichettati ($L_E[e]$`.label=null` per ogni $e \in E$)
- In una generica invocazione `DFS(`G`,`v`)` alcuni vertici della componente connessa di v possono essere già visitati e alcuni archi già etichettati.

Esempio

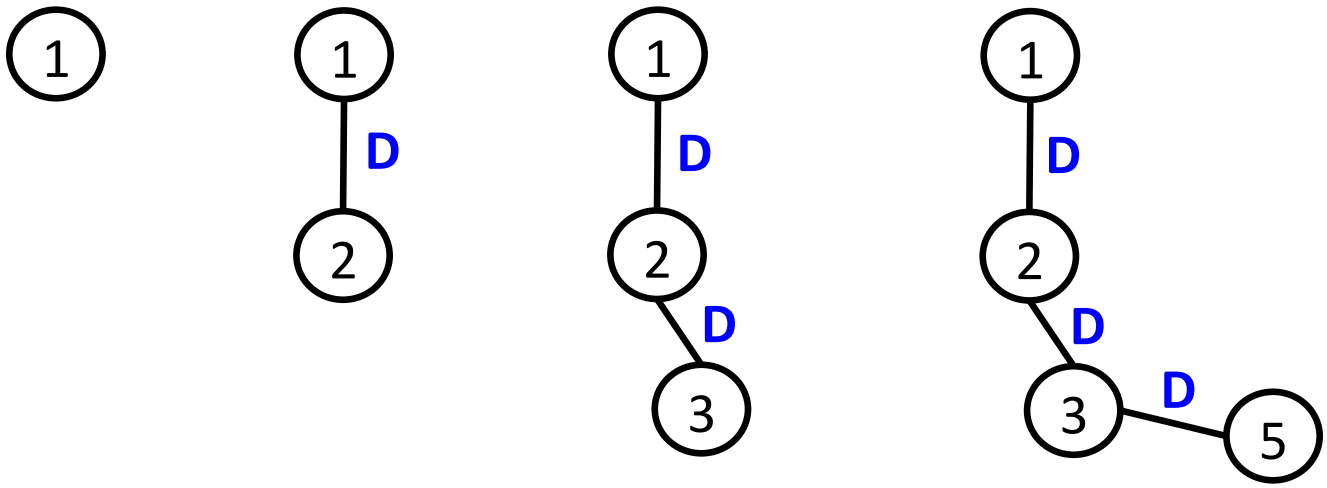


s = 1

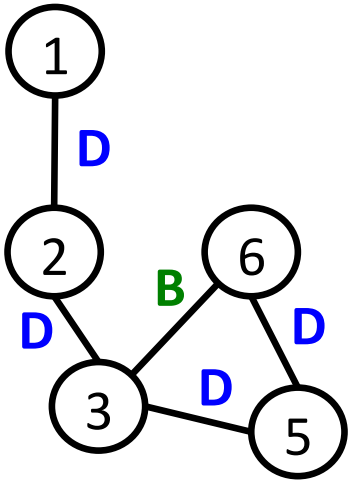
Liste di adiacenza (ordine crescente)

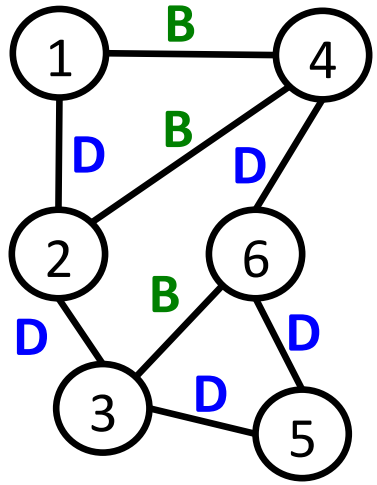
1: 2 4
2: 1 3 4
3: 2 5 6
4: 1 2 6
5: 3 6
6: 3 4 5 7
7: 6

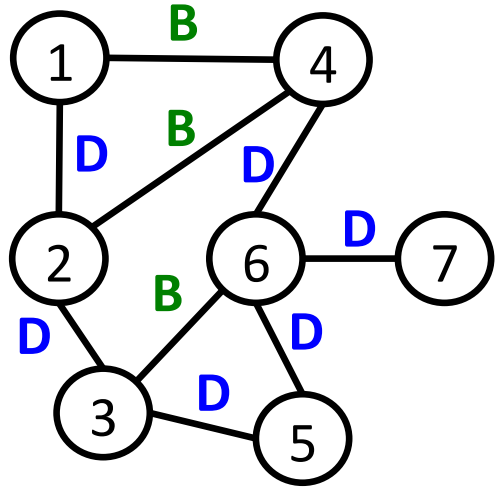
D = DISCOVERY EDGE
B = BACK EDGE



Esempio (continua)







D = DISCOVERY EDGE

B = BACK EDGE

Analisi di DFS(G,s)

- $G = (V, E)$ grafo non diretto, $s \in V$
- $C_s \subseteq G$: componente connessa di G contenente s.

Proposizione (simile a Proposizione 14.12 [GTG14])

Si supponga di eseguire DFS(G,s) quando nessuno dei vertici/archi di C_s è stato visitato/etichettato. Alla fine dell'esecuzione si ha che:

1. tutti i vertici di C_s sono visitati e tutti gli archi di C_s sono etichettati come DISCOVERY o BACK EDGE;
2. i DISCOVERY EDGE formano uno spanning tree T di C_s radicato in s.

Dimostrazione

1. Si noti anzitutto che un vertice u viene visitato solo quando si invoca `DFS(`G`,`u`)`. Per assurdo, supponiamo che esista un vertice $v \in C_s$ che non sia stato visitato. Dato che v ed s sono nella stessa componente connessa, deve esistere un cammino

$$s = u_0 - u_1 - u_2 - \cdots - u_\ell = v,$$

con $u_j \in C_s$, per ogni j. Dato che sicuramente s è visitato, se v non è visitato, nel cammino deve esistere un primo vertice non visitato u_i. Ma questo è un assurdo perchè `DFS(`G`,`u_{i-1}`)` deve essere eseguita e, trovando u_i non visitato, al suo interno invocherebbe `DFS(`G`,`u_i`)`.

Chiaramente, se `DFS` è invocata su tutti i vertici di C_s, allora tutti gli archi incidenti su tali vertici (ovvero tutti gli archi di C_s) devono essere alla fine etichettati come `DISCOVERY` o `BACK EDGE`

Dimostrazione (continua)

2. (Sul libro di testo la prova è meno rigorosa.) Dal punto precedente sappiamo che `DFS(`G`,`v`)` è invocata su tutti i vertici $v \in C_s$. È facile vedere che su ciascun vertice viene invocata una sola volta e che per ogni $v \in C_s$, con $v \neq s$, deve esistere un vertice u tale che: l'arco (u, v) esiste e viene etichettato come `DISCOVERY EDGE` e `DFS(`G`,`v`)` viene invocata da `DFS(`G`,`u`)`. In questo caso diciamo che v viene "scoperto" da u e definiamo u padre di v.

   Di conseguenza, per ogni $v \in C_s$, con $v \neq s$:

   - Esiste un unico padre
   - Risalendo di padre in padre si risale attraverso invocazioni di `DFS` indietro nel tempo e ci si può fermare solo su `DFS(`G`,`s`)`.

   $\Rightarrow$ I `DISCOVERY EDGE` formano un rooted tree (radice s) che tocca tutti i vertici di C_s, ed è quindi uno spanning tree. □

Complessità di DFS(G, s)

Definiamo

$$\begin{aligned} n_s &= \text{numero di vertici in } C_s \\ m_s &= \text{numero di archi in } C_s \end{aligned}$$

Albero della ricorsione per DFS(G, s):

- I nodi corrispondono alle invocazioni DFS(G, v), esattamente una per ogni vertice di C_s
- Il costo attribuito al nodo associato a DFS(G, v) (escludendo le invocazioni ricorsive al suo interno) è $\Theta(\text{degree}(v))$

$$\Rightarrow \text{Complessità} \in \Theta\left(\sum_{v \in C_s} \text{degree}(v)\right) \in \Theta(m_s)$$

Osservazione: Poichè C_s è connesso $m_s \geq n_s - 1 \Rightarrow m_s \in \Omega(n_s)$

Corollario

Se $G = (V, E)$ è connesso, DFS(G, s) ha complessità $\Theta(|E|)$, $\forall s \in V$.

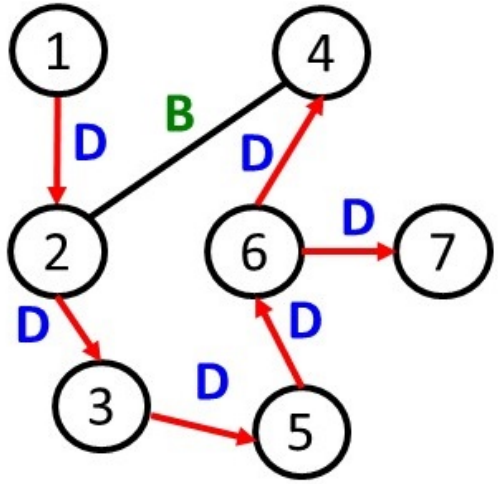
Osservazione

(Esercizio C-14.37 [GTG14]) Sia T lo spanning tree di C_s, radicato in s, costituito dagli archi etichettati come `DISCOVERY EDGE` dopo l'esecuzione di $\texttt{DFS}(G, s)$. Supponiamo che durante una delle varie chiamate ricorsive $\texttt{DFS}(G, v)$, fatta durante l'esecuzione dell'algoritmo, si etichetti un arco (v, w) come `BACK EDGE`. Allora w deve essere un antenato di v in T in base alle seguenti considerazioni:

- Per etichettare (v, w) come `BACK EDGE` deve essere $L_V[w].\texttt{ID} = 1$ e quindi DFS(G, w) deve essere stata già invocata.
- Tuttavia DFS(G, w) deve essere iniziata ma non conclusa, altrimenti (v, w) avrebbe già una etichetta diversa da `null` quando $\texttt{DFS}(G, v)$ lo esamina.
- Ne consegue che esiste una sequenza di vertici $w = w_1, w_2, \ldots, w_k = v$ tali che DFS(G, w_{i+1}) è invocata direttamente da DFS(G, w_i), per $1 \leq i < k$, e quindi $(w_1, w_2)(w_2, w_3), \cdots, (w_{k-1}, w_k)$ è un cammino di discovery edge da w a v, ovvero w è antenato di v.

Questa osservazione giustifica la locuzione `BACK EDGE`.

Esempio



Spanning tree dei discovery edge:

- il verso delle frecce rappresenta le relazioni padre-figlio
- Es.: (4,2) è marcato come BACK EDGE e 2 è antenato di 4

Visita di tutto il grafo

Si noti che l'algoritmo DFS(G, s) visita solo la componente connessa di s.

Il seguente pseudocodice può essere utilizzato come *design pattern* per estendere la visita a tutto il grafo, nel caso esso non sia connesso.

```
for v ← 1 to n do L_V[v].ID ← 0;
for v ← 1 to n do
    if (L_V[v].ID = 0) then
        DFS(G, v);
```

Visita di tutto il grafo (analisi)

Sia c il numero di componenti connessi di G. Si osservi che

- Nel secondo ciclo for, l'invocazione `DFS`(G, v) verrà fatta *esattamente* c volte su vertici di componenti connesse distinte, dato che ogni invocazione `DFS`(G, v) imposta il campo `ID` di ciascun vertice della componente connessa di v a 1.

- Sia m_j il numero di archi della j-esima componente connessa, per $1 \leq j \leq c$, e si noti che $m = \sum_{j=1}^{c} m_j$. Il costo aggregato di tutte le invocazioni di `DFS` è $O\left(\sum_{j=1}^{c} m_j\right) = O(m)$, mentre il costo delle altre operazioni fatte dall'algoritmo è $O(n)$.

  $\Rightarrow$ La complessità è $O(n + m)$.

Applicazioni della DFS: connettività

Dato $G = (V, E)$ vogliamo determinare il numero di componenti connesse (se tale numero è 1, allora G è connesso).

Sia DFS(G,v,k) una modifica di DFS(G, v)) che sostituisce l'istruzione

$$L_V[v].\texttt{ID} \leftarrow 1$$

con

$$L_V[v].\texttt{ID} \leftarrow k.$$

Il seguente algoritmo calcola il numero di componenti connesse di G e assegna ai vertici di ciascuna componente uno stesso identificativo.

```
for v ← 1 to n do L_V[v].ID ← 0;
k ← 0;
for v ← 1 to n do
    if (L_V[v].ID = 0) then
        k ← k + 1;
        DFS(G, v, k);
return k
```

Applicazioni della DFS: connettività (analisi)

Sia c il numero di componenti connesse di G. Si osservi che

- Ragionando come nell'analisi precedente vediamo facilmente che nel secondo ciclo for l'invocazione $\texttt{DFS}(G, v, k)$ verrà fatta esattamente c volte su vertici di componenti connesse distinte. Di conseguenza, alla fine del ciclo si avrà $k = c$ ($k = 1$ se G è connesso).

  $\Rightarrow$ L'algoritmo è corretto.

- Sempre ragionando analogamente all'analisi precedente si dimostra facilmente che la complessità è $O(n + m)$.

Applicazioni della DFS: spanning tree

Vogliamo trovare uno spanning tree di un grafo $G = (V, E)$ connesso.

È sufficiente eseguire `DFS`(G, s) a partire da qualsiasi vertice s e restituire i discovery edge come archi dello spanning tree.

La correttezza è conseguenza immediata dell'analisi di `DFS` fatta in orecedenza, mentre la complessità è $O(m)$, dato che si invoca `DFS` una sola volta.

Applicazioni della DFS: s-t reachability

Dato $G = (V, E)$ e due vertici $s, t \in V$ vogliamo determinare, se esiste, un cammino da s a t.

L'algoritmo è il seguente (una specifica più dettagliata è lasciata come esercizio):

- Per ciascun vertice $w \in V$ usiamo un campo aggiuntivo $L_V[w]$.`parent`.
- Modifichiamo `DFS`(G,v) in modo che quando etichetta un arco (v, w) come `DISCOVERY EDGE` imposti $L_V[w]$.`parent` a v (ovvero v è padre di w nell'albero dei discovery edge).
- Eseguiamo `DFS`(G, s). Alla fine, se t non è stato visitato si dice in output che non esiste un cammino da s a t, altrimenti, partendo da t e risalendo di padre in padre si costruisce il cammino e lo si restituisce in output.

È facile vedere che l'algoritmo è corretto e ha complessità $O(m_s)$, dove m_s è il numero di archi nella componente connessa di s.

Applicazioni della DFS: ciclicità

Dato $G = (V, E)$ vogliamo determinare un ciclo, se esiste.

L'algoritmo è il seguente (una specifica più dettagliata è lasciata come esercizio):

- Per ciascun vertice $u \in V$ usiamo un campo aggiuntivo $L_V[u]$.`parent` e per ciascun arco $e \in E$ usiamo un campo aggiuntivo $L_E[e]$.`ancestor`.
- Facciamo le seguenti modifiche a `DFS`(G,v):
  - quando etichetta un arco (v, u) come `DISCOVERY EDGE` imposta $L_V[u]$.`parent` a v.
  - quando etichetta un arco $e = (v, w)$ come `BACK EDGE` imposta $L_E[e]$.`ancestor` a w, per rappresentare il fatto che w è antenato di v nell'albero dei discovery edge (come osservato in precedenza).

Applicazioni della DFS: ciclicità (continua)

- Eseguiamo la DFS su ciascuna componente connessa del grafo (come per in calcolo del numero di componenti connesse).
- Eseguiamo un ciclo su tutti gli archi. Appena si trova un arco $e = (v, w)$ etichettato come `BACK EDGE` e con $L_E[e].\texttt{ancestor} = w$ antenato di v, si costruisce un ciclo aggiungendo a (v, w) gli archi incontrati risalendo da v a w di padre in padre, e lo si restituisce in output. Se invece non si trova alcun `BACK EDGE` si restituisce in output che il grafo è aciclico.

La complessità è $O(n + m)$, dato che si invoca la DFS una volta per ogni componente connessa.

Nei lucidi precedenti abbiamo dimostrato la seguente proposizione:

Proposizione 14.14 [GTG14]

Dato $G = (V, E)$, con $|V| = n$ e $|E| = m$, i seguenti problemi possono essere risolti in tempo $O(m + n)$ usando la DFS:

1. testare se G è connesso;
2. trovare le componenti connesse di G;
3. trovare uno spanning tree di G, se G è connesso:
4. trovare un cammino tra 2 vertici s e t, se esiste (s-t reachability);
5. trovare un ciclo, se esiste.

Esercizi

Esercizio

Sia $G = (V, E)$ un grafo non diretto con $k > 1$ componenti connesse. Progettare un algoritmo che aggiunga $k - 1$ archi a G per renderlo connesso e analizzarne la complessità. Si assuma di poter aggiungere a E un arco $(u, v) \notin E$ in tempo costante invocando il metodo `G.addArc(u,v)`.

Ing. TESEO
Vs
MINOTAURO

Esercizio

Si consideri un labirinto L che ha un unico punto di ingresso s e al cui interno è nascosto il terribile Minotauro. L'ing. Teseo deve entrare in L, trovare il Minotauro, ucciderlo, e uscire da L. Trovare un'opportuna rappresentazione del labirinto come grafo e far vedere come, sfruttando l'algoritmo DFS, l'ing. Teseo può compiere con successo la sua missione. Si assuma che il labirinto sia connesso, nel senso che ogni punto di esso sia raggiungibile da s.

Breadth-First Search: Algoritmo BFS

- Algoritmo iterativo che a partire da un vertice s "visita" tutti i vertici della componente connessa $C_s \subseteq G$ contenente s, toccando tutti i vertici e gli archi di C_s, e partizionando i vertici in livelli L_i in base alla loro distanza i da s.
- Oltre alle strutture di base L_V e L_E assumiamo di usare le liste di adiacenza che inducono un ordine di visita dei vicini di ogni vertice.
- Ogni vertice v ha un campo $L_V[v]$.`ID` che vale 1, se v è stato visitato, e 0 altrimenti.
- Ogni arco e ha un campo $L_E[e]$.`label` che vale `null` se e non ha ancora etichetta, oppure riporta una delle due etichette `DISCOVERY EDGE` o `CROSS EDGE`

Breadth-First Search: Algoritmo BFS (continua)

Algoritmo `BFS`(G, s)

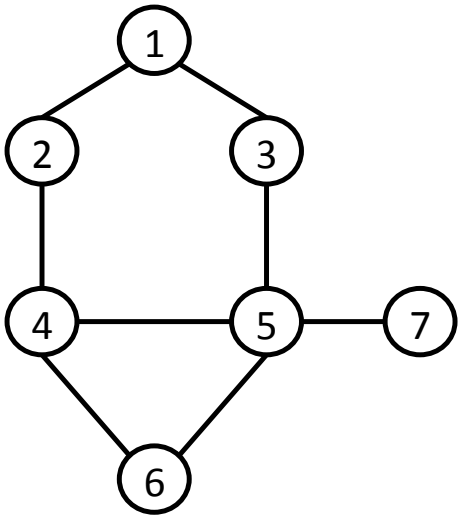
Input: grafo $G = (V, E)$ non diretto, vertice $s \in V$ non visitato

Output: tutti i vertici nella componente connessa di s visitati e gli archi etichettati come `DISCOVERY` o `CROSS EDGE`

Breadth-First Search: Algoritmo BFS (continua)

```
visita il vertice s; L_V[s].ID ← 1;
crea una collezione L_0 contenente s;
i ← 0;
while (!L_i.isEmpty()) do
    crea una collezione di vertici L_{i+1} vuota;
    forall v ∈ L_i do
        forall e ∈ G.incidentEdges(v) do
            if (L_E[e].label = null) then
                w ← G.opposite(v,e);
                if (L_V[w].ID = 0) then
                    L_E[e].label ← DISCOVERY EDGE;
                    visita il vertice w; L_V[w].ID ← 1;
                    inserisci w in L_{i+1};
                else L_E[e].label ← CROSS EDGE;
    i ← i + 1;
return;
```

Esempio



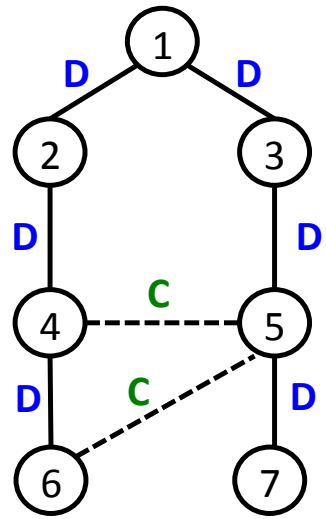
s = 1

Liste di adiacenza
(ordine crescente)

1: 2 3
2: 1 4
3: 1 5
4: 2 5 6
5: 3 4 6 7
6: 4 5
7: 5

D = DISCOVERY EDGE
C = CROSS EDGE

L_0 →
L_1 →
L_2 →
L_3 →



Analisi di BFS

- $G = (V, E)$ grafo non diretto, $s \in V$
- $C_s \subseteq G$: componente connessa di G contenente s.

Proposizione (simile a Prop. 14.16 [GTG14])

Si supponga di eseguire `BFS(`G`,`s`)` quando nessuno dei vertici/archi di C_s è stato visitato/etichettato. Alla fine dell'esecuzione si ha che:

1. tutti i vertici di C_s sono visitati e tutti gli archi di C_s sono etichettati come `DISCOVERY` o `CROSS EDGE`;
2. i `DISCOVERY EDGE` formano uno spanning tree T di C_s radicato in s e chiamato *BFS tree*;
3. $\forall v \in L_i$ il cammino in T da s a v ha i archi e qualsiasi altro cammino in G da s a v ha $\geq i$ archi ($\Rightarrow i \equiv \mathit{distanza}(s, v)$);
4. se $(u, v) \in E$ e $(u, v) \notin T$ ($\Rightarrow (u, v)$ è un CROSS EDGE) gli indici dei livelli di u e v differiscono al più di 1

Dimostrazione (Esercizi C-14.46, C-14.47 [GTG14])

1. Come per la DFS
2. Come per la DFS
3. Si consideri il cammino $P : s = u_0 - u_1 - \cdots - u_i = v$, dove $u_j \in L_j$ viene "scoperto" da u_{j-1}, $\forall j : 1 \leq j \leq i$. Quindi (u_{j-1}, u_j) è un DISCOVERY EDGE, e, di conseguenza, P è un cammino in T.
   Per assurdo, se esistesse in G un cammino

$$P' : s = z_0 - z_1 - \cdots - z_t = v,$$

   con $t < i$, si avrebbe

Dimostrazione (continua)

$$\begin{aligned} s = z_0 \in \quad & L_0 \\ z_1 \in \quad & L_1 \\ z_2 \in \quad & L_1 \text{ o } L_2 \\ \vdots & \\ z_t = v \in \quad & L_1 \text{ o } \ldots \text{o } L_t \end{aligned}$$

$\Rightarrow \quad v \notin L_i$: assurdo

4. Per assurdo, se $u \in L_i$ e $v \in L_{i+k}$ con $k > 1$ v sarebbe scoperto a partire da u e quindi si avrebbe $v \in L_{i+1}$ e non $v \in L_{i+k}$ (assurdo). □

Complessità di BFS(G, s)

Definiamo

$$\begin{aligned} n_s &= \text{numero di vertici in } C_s \\ m_s &= \text{numero di archi in } C_s \end{aligned}$$

Supponiamo di rappresentare ogni L_i tramite una lista.

- $\forall v$ vertice in C_s viene eseguita *esattamente 1 iterazione* del primo ciclo **forall** ed esattamente $degree(v)$ iterazioni del secondo ciclo **forall**
- ciascuna iterazione del secondo ciclo **forall** richiede tempo $\Theta(1)$
- tutti gli accessi alle L_i richiedono tempo $\Theta(1)$

$\Rightarrow$ complessità di `BFS(`G`,`s`)` $\in \Theta(m_s)$

Corollario

Se $G = (V, E)$ è connesso, `BFS(`G`,`s`)` ha complessità $\Theta(|E|) \quad \forall s \in V$.

Applicazioni della BFS

Dato $G = (V, E)$, con $|V| = n$ e $|E| = m$, in modo del tutto analogo a quanto fatto con la DFS possiamo usare la BFS per risolvere i seguenti problemi in tempo $O(n + m)$:

- Visitare tutto G (anche nel caso non sia connesso);
- Determinare il numero di componenti connesse di G;
- Se G è connesso, trovare uno spanning tree di G.

Applicazioni della BFS: cammini minimi

Dato $G = (V, E)$ e due vertici $s, t \in V$ vogliamo determinare, se esiste, un *cammino di lunghezza minima* da s a t.

L'algoritmo è il seguente (una specifica più dettagliata è lasciata come esercizio):

- Per ciascun vertice $u \in V$ usiamo un campo aggiuntivo $L_V[u]$.`parent`.
- Modifichiamo `BFS`(G,s) in modo che quando etichetta un arco (v, u) come `DISCOVERY EDGE` imposti $L_V[u]$.`parent` a v (ovvero v è padre di u nell'albero dei discovery edge).
- Eseguiamo `BFS`(G, s). Alla fine, se t non è stato visitato si dice in output che non esiste un cammino da s a t, altrimenti, partendo da t e risalendo di padre in padre si costruisce il cammino e lo si restituisce in output.

È facile vedere che l'algoritmo è corretto (in virtù anche della proposizione precedente) e ha complessità $O(m_s)$, dove m_s è il numero di archi nella componente connessa di s.

Applicazioni della BFS: ciclicità

Dato $G = (V, E)$ vogliamo determinare un ciclo, se esiste.

Esercizio

Progettare e analizzare un algoritmo per risolvere il problema in tempo $O(n + m)$ basato sull'uso della BFS.

Suggerimento: sfruttare l'osservazione che esiste un ciclo se e solo se, visitando tutto G con la BFS, uno degli archi viene etichettato come `CROSS EDGE`.

Nei lucidi precedenti abbiamo dimostrato la seguente proposizione:

Proposizione

Dato $G = (V, E)$, con $|V| = n$ e $|E| = m$, i seguenti problemi possono essere risolti in tempo $O(m + n)$ usando la BFS:

1. testare se G è connesso;
2. trovare le componenti connesse di G;
3. trovare uno spanning tree di G, se G è connesso:
4. trovare un cammino minimo tra 2 vertici s e t, se esiste;
5. trovare un ciclo, se esiste.

Esercizio

Sia $G = (V, E)$ un grafo *non connesso* con n vertici ed m archi. Progettare un algoritmo che conti le coppie di vertici $u, v \in V$ tali che u e v sono raggiungibili uno dall'altro tramite cammini, analizzandone la complessità. Per avere punteggio pieno la complessità deve essere $O(n + m)$. (Si ricordi che da un insieme di K oggetti si possono formare $K(K-1)/2$ coppie distinte.)

Svolgimento

L'idea è usare un BFS modificata per determinare per ogni componente connessa di G la sua cardinalità K, aggiungendo il valore $K(K-1)/2$ al conteggio delle coppie di vertici raggiungibili uno dall'altro.

Supponiamo di aver modificato `BFS`(G, v) definendo una variabile `cardinality` inizializzata a 0 che viene incrementata ogni volta in cui si vista un vertice e il cui valore viene restituito in output.

L'algoritmo richiesto è il seguente.

Algoritmo `ReachablePairs`(G)

Input: Grafo $G = (V, E)$ non diretto e non connesso
Output: Numero coppie $u, v \in V$ raggiungibili uno dall'altro

```
count ← 0;
for v ← 1 to n do
    if (L_V[v].ID = 0) then
        K ← BFS(G, v);
        count ← count+K(K − 1)/2
return count
```

Analisi:

- La correttezza dell'algoritmo è immediata.
- Le modifiche apportate alla BFS non ne alterano la complessità, e quindi la complessità dell'algoritmo è $O(n + m)$

Esercizio

Sia G un grafo non diretto e connesso in cui ciascun vertice ha grado esattamente c, con $c > 2$ costante intera. Si consideri l'esecuzione di BFS(G, s) a partire da un arbitrario vertice $s \in V$. Dimostrare per induzione su i che il livello L_i generato da BFS(G, s) contiene $\leq c \cdot (c-1)^{i-1}$ vertici, per ogni $i \geq 0$.

Svolgimento

Base. $i = 0, 1$: banale.

Passo induttivo. Fissiamo $i \geq 1$ e supponiamo, come ipotesi induttiva, che $|L_j| \leq c \cdot (c-1)^{j-1}$ per ogni $0 \leq j \leq i$. I vertici del livello L_{i+1} sono tutti adiacenti a vertici del livello i e poiché ciascun vertice $v \in L_i$ ha c vicini, di cui però almeno uno è nel livello L_{i-1}, concludiamo che

$$|L_{i+1}| \leq (c-1) \cdot |L_i| \leq (c-1) \cdot c \cdot (c-1)^{i-1} = c \cdot (c-1)^i.$$

Esercizio

Sia $G = (V, E)$ un grafo con n vertici e m archi. Sviluppare un algoritmo che restituisce, se esiste, un vertice $i \in V$ da cui sono raggiungibili (con cammini) $\geq n/2$ altri vertici, e analizzarne la complessità. Se un tale vertice non esiste l'algoritmo restituisce null.

Esercizio C-14.49 [GTG14]

Un grafo $G = (V, E)$ si dice *bipartito* se l'insieme di vertici V può essere partizionato in due sottoinsiemi X e Y tali che ogni arco di E incide su un vertice di X e uno di Y. Progettare e analizzare un algoritmo efficiente che determini se un grafo non diretto G è bipartito.

Esercizio

Sia $G = (V, E)$ il grafo (non diretto) di Facebook in cui i vertici rappresentano i profili e gli archi le amicizie. Si assuma G connesso (in realtà non lo è). Si definisca la *separazione* tra due profili $u \neq v \in V$ come il numero di archi nel cammino più breve da u a v in G. Progettare e analizzare un algoritmo per determinarne la massima separazione tra due profili in G.

Esempio domande prima parte

- Definire rigorosamente le *componenti connesse* di un grafo non diretto G.
- Sapendo che un free tree con n vertici ha $m = n - 1$ archi, dimostrare che un grafo connesso con n vertici ha $m \geq n - 1$ archi.
- Sia $G = (V, E)$ un grafo non diretto e connesso. Dati due vertici $s, t \in V$ dire brevemente come trovare il cammino più breve da s a t, e quanto tempo è richiesto per trovarlo.
- Sia $G = (V, E)$ un grafo non diretto con n vertici ed m archi. Descrivere brevemente come determinare se G è connesso in tempo $O(n + m)$.

Riepilogo

- Definizioni e terminologia sui grafi.
- Concetti fondamentali: cammino, ciclo, grafo connesso, subgraph, spanning subgraph, alberi (radicati, liberi e loro equivalenza). spanning tree/forest.
- Relazione tra la somma dei degree e il numero di archi in un grafo
- Relazioni tra numero di vertici e numero di archi in alberi, foreste e grafi connessi
- Depth-First Search
  - Algoritmo
  - Analisi e proprietà
  - Problemi computazionali risolvibili tramite DFS
- Breadth-First Search
  - Algoritmo
  - Analisi e proprietà
  - Problemi computazionali risolvibili tramite BFS

Errata

Cambiamenti rispetto alla prima versione dei lucidi:

- Lucido 26: modificato il testo dei primi due capoversi.
- Lucido 40: reso più chiaro il primo capoverso.
- Lucidi 46 e 47: modificato lo pseudocodice e l'analisi facendo partire k da 0.
- Lucido 49: vertice u ridenominato w.
- Lucido 64 (secondo punto): sostituito BFS(G, v) con BFS(G, s)