Dati e Algoritmi (Pietracaprina)

Esercizi svolti sui Grafi

Problema 1 *Progettare e analizzare un algoritmo efficiente basato sulla* `BFS` *che, dato un grafo* $G = (V, E)$ *non diretto, restituisca un ciclo, se esiste, o* `null`*, se non esiste.*

Soluzione. Osserviamo che visitando tutte le componenti connesse di G con la `BFS`, se nessun arco viene etichettato come `CROSS EDGE` non ci possono essere cicli (infatti i `DISCOVERY EDGE` formano un albero per ogni componente connessa), mentre un arco (u, v) è etichettato come `CROSS EDGE` (u, v), esiste un ciclo, che può essere ottenuto aggiungendo all'arco (u, v) tutti i `DISCOVERY EDGE` incontrati risalendo da u e da v al loro primo antentato comune nell'albero dei `DISCOVERY EDGE` radicato nella sorgente da cui è iniziata la visita della loro componente connessa. L'algoritmo è basato su questa osservazione. Assumiamo che per ciascun vertice $v \in V$ siano disponibili i seguenti campi (oltre a v`.ID`): v`.level`, in cui verrà memorizzato il livello di v nella `BFS` della sua componente connessa, e v`.parent`, in cui verrà memorizzato il padre di v, se esiste, nel `BFS` tree della sua componente connessa. Si supponga di modificare `BFS`(G, s) come segue: quando si esamina un vertice $v \in L_i$ e si inserisce un vicino w nel livello L_{i+1}, etichettando l'arco (v, w) come `DISCOVERY EDGE`, allora, dopo aver impostato w.ID a 1, si imposta w.level $\leftarrow i + 1$, e w.parent $\leftarrow v$. L'algoritmo è il seguente:

Algoritmo `FindCycle`(G)

```
Input:  grafo G = (V, E) non diretto
Output: Un ciclo in G (lista di archi), se esiste, o null se non
        esiste
forall v ∈ V do
    v.ID ← 0;
    v.parent ← null;
forall e ∈ V do e.label ← null;
forall v ∈ V do if (v.ID = 0) then BFS(G,v);
C ← lista vuota;

forall e ∈ E do
    if (e.label = CROSS EDGE) then
        C.add(e);
        Siano u e v i vertici su cui e incide;
        if (u.level = v.level + 1) then
            C.add((u, u.parent));
            u ← u.parent;
        if (v.level = u.level + 1) then
            C.add((v, v.parent));
            v ← v.parent;
        while (u ≠ v) do
            C.add((u, u.parent)); C.add((v, v.parent));
            u ← u.parent;
            v ← v.parent;
        return C;
return null;
```

È facile vedere che le modifiche richieste alla `BFS` non ne alterano la complessità. Di con-

seguenza, i primi tre cicli forall eseguono sostanzialmente a una visita di tutto il grafo e, come visto a lezione, hanno una complessità $\Theta(n+m)$, dove n è il numero di vertici ed m il numero di archi. L'ultimo ciclo forall esegue una scansione di tutti gli archi (ad esempio sfruttando la lista L_E) e appena trova un arco marcato come `CROSS EDGE` fa un numero costante di operazioni e poi un ciclo while con al più n iterazioni, ciascuna di complessità costante. Se ne deduce che la complssità finale è $\Theta(n+m)$. (Si osservi che le `BFS` potrebbero essere interrotte appena si marca un arco come `CROSS EDGE`.) □

Problema 2 *Sia $G=(V,E)$ un grafo con n vertici ed m archi. Progettare un algoritmo che conti le coppie di vertici $u,v \in V$, tali che $u \neq v$ ed esiste un cammino da u a v, analizzandone la complessità. Per avere punteggio pieno la complessità deve essere $O(n+m)$.*

Soluzione. Si ricorda che da un insieme di K oggetti si possono formare esattamente $K(K-1)/2$ coppie distinte. L'idea è di visitare tutte le componenti connesse di G usando una `BFS` modificata che per ogni componente connessa ne determina il numero di vertici K e aggiunge il valore $K(K-1)/2$ al conteggio delle coppie di vertici connessi da un cammino (cioè raggiungibili uno dall'altro), mantenuto tramite un contatore `count`. Supponiamo di modificare `BFS`(G,v) definendo un contatore C, inizializzato a 1 (per tenere traccia di v), che viene incrementato ogni volta in cui si visita un vertice $w \neq v$ impostando w.`ID` a 1, 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` $u,v \in V$`, con` $u \neq v$`, collegate da cammini`

```
count ← 0;
forall v ∈ V do v.ID ← 0;
forall e ∈ V do e.label ← null;
forall v ∈ V do
    if (v.ID = 0) then
        K ← BFS(G, v);
        if (K > 1) then count ← count+K(K − 1)/2;
return count
```

La correttezza dell'algoritmo è immediata. Per quanta riguarda la complessità, è facile vedere che le modifiche richieste alla `BFS` non ne alterano la complessità. Di conseguenza, l'algoritmo equivale sostanzialmente a una visita di tutto il grafo e, come visto a lezione, ha complessità $\Theta(n+m)$. □

Problema 3 *Sia $G=(V,E)$ 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 vertice arbitrario $s \in V$. 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 $i \geq 0$.*

Soluzione. Come base dell'induzione consideriamo $i=0$ e $i=1$. Il livello L_0 contiene il solo vertice s e quindi $|L_0|=1 \leq c\cdot(c-1)^{-1}$. Il livello L_1 contiene i c vicini di s, e quindi $|L_1|=c=c\cdot(c-1)^0$. Fissiamo un indice $i \geq 1$ e supponiamo, come ipotesi induttiva, che per ogni $0 \leq j \leq i$ si abbia $|L_j| \leq c\cdot(c-1)^{j-1}$. 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 $L_{i+1} \leq (c-1) \cdot |L_i|$. Applicando l'ipotesi induttiva, otteniamo

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

□

Problema 4 *Sia $G = (V, E)$ un grafo con n vertici ed m archi. Progettare un algoritmo che restituisce, se esiste, un vertice $v \in V$ 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`.

Soluzione. Si noti che da un vertice $v \in V$ sono raggiungibili $\geq n/2$ altri vertici se e solo se la componente connessa di v contiene almeno $n/2 + 1$ vertici. L'idea è quindi quella di determinare, tramite `BFS` modificata, la cardinalità di ciascuna componente connessa e, appena se ne trova una di cardinalità almeno $n/2 + 1$, restituire un vertice arbitrario di essa (ad es., quello da cui è iniziata la visita). Supponiamo di modificare `BFS`(G, v) definendo una variabile `cardinality`, inizializzata a 0, che viene incrementata ogni volta in cui si visita un vertice w e si imposta w.`ID` a 1, e il cui valore viene restituito in output. L'algoritmo richiesto è il seguente.

Algoritmo `LargeComponent`(G)

Input: `grafo` $G = (V, E)$ `non diretto`
Output: `vertice` $v \in V$ `da cui sono raggiungibili` $\geq n/2$ `vertici, o null se un tale vertice non esiste`

```
forall v ∈ V do v.ID ← 0;
forall e ∈ V do e.label ← null;
forall v ∈ V do
    if (v.ID = 0) then
        K ← BFS(G, v);
        if (K ≥ n/2 + 1) then return v;
return null
```

La correttezza dell'algoritmo discende immediatamente dalla osservazione fatta all'inizio. Per quanta riguarda la complessità, è facile vedere che le modifiche richieste alla `BFS` non ne alterano la complessità. Di conseguenza, l'algoritmo equivale sostanzialmente a una visita di tutto il grafo e, come visto a lezione, ha complessità $\Theta(n + m)$. □

Problema 5 *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.*

Soluzione. Si osservi anzitutto che G è bipartito se e solo se ciascuna sua componente connessa è bipartita. Basterà quindi controllare per ogni componente connessa se essa è bipartita oppure no. Si ricordi che, nell'esecuzione della `BFS`, ogni `DISCOVERY EDGE` collega due vertici in livelli adiacenti, mentre ogni `CROSS EDGE` collega due vertici che appartengono allo stesso livello o a livelli adiacenti. Si consideri l'invocazione `BFS`(G, v) per un qualche $v \in V$, e sia C_v la componente connessa di v. E' facile vedere che C_v è bipartita se e solo se

non esistono `CROSS EDGE` tra vertici dello stesso livello. Infatti, se non ci sono `CROSS EDGE` tra vertici dello stesso livello, tutti gli archi connettono vertici in livelli adiacenti (uno di indice pari e uno di indice dispari). In questo caso, definendo X_v come l'insieme di vertici appartenenti a livelli di indice pari e Y_v come l'insieme di vertici appartenenti a livelli di indice dispari, si ha una corretta bipartizione. Se invece esiste un `CROSS EDGE` tra vertici dello stesso livello la componente connessa non può essere bipartita. Infatti, se lo fosse e i suoi vertici fossero suddivisi in X_v e Y_v, con $v \in X_v$, si avrebbe che, in base ai `DISCOVERY EDGE`, tutti i vertici appartenenti a livelli di indice pari dovrebbero essere in X_v e tutti i vertici appartenenti a livelli di indice dispari dovrebbero essere in Y_v, escludendo la possibilità di `CROSS EDGE` tra vertici dello stesso livello. In virtù di questa osservazione, per controllare se C_v è bipartita è sufficiente invocare `BFS`(G, v) e verificare che non ci siano `CROSS EDGE` tra vertici dello stesso livello.

Assumiamo che per ciascun vertice $v \in V$, oltre al campo v`.ID`, sia disponibile un campo v`.level`, in cui verrà memorizzato il livello di v nella `BFS` della sua componente connessa. Assumiamo anche di avere la lista L_E degli archi (che può essere scansionata tramite un iteratore) e che, dato un arco $e = (u, v)$, i vertici u e v possano essere ottenuti in tempo $O(1)$. Si supponga di modificare `BFS`(G, s) come segue: quando si esamina un vertice $v \in L_i$ e si inserisce un vicino w nel livello L_{i+1}, etichettando l'arco (v, w) come `DISCOVERY EDGE`, allora dopo aver impostato w.ID a 1, si imposta w.level $\leftarrow i + 1$. L'algoritmo è il seguente:

Algoritmo `isBipartite`(G)

Input: `Grafo` $G = (V, E)$ `non diretto`
Output: `TRUE/FALSE se` G `è/non è bipartito`
forall $v \in V$ **do** v`.ID` $\leftarrow 0$`;`
forall $e \in V$ **do** e`.label` $\leftarrow$ `null;`
forall $v \in V$ **do**
  if `(`v`.ID` $= 0$`)` **then** `BFS`(G, v)`;`
forall $e \in E$ **do**
  `Siano` u `e` v `i vertici su cui` e `incide;`
  if `(`u`.level` $= v$`.level)` **then return** `FALSE;`
return `TRUE;`

Per quanta riguarda la complessità, è facile vedere che le modifiche richieste alla `BFS` non ne alterano la complessità. Di conseguenza, l'algoritmo equivale sostanzialmente a una visita di tutto il grafo e, come visto a lezione, ha complessità $\Theta(n + m)$. □

Problema 6 *La* teoria dei 6 gradi di separazione *fu formulata per la prima volta nel 1929 dallo scrittore ungherese Frigyes Karinthy nel racconto omonimo pubblicato nel volume Catene.*

a. *Trovare l'enunciato di tale teoria.*

b. *Definire un problema computazionale su grafo tale che la sua risoluzione possa servire a confermare o confutare la teoria, usando Facebook come insieme di dati di input.*

c. *Progettare e analizzare un algoritmo efficiente per il problema definito nel punto precedente.*

Soluzione.

a. La teoria afferma che ogni persona può essere collegata a qualunque altra persona o cosa attraverso una catena di conoscenze e relazioni con non più di 5 intermediari.

b. Sia $G = (V, E)$ il grafo (non diretto) di Facebook in cui i vertici rappresentano i profili e gli archi le amicizie. Assumiamo che G connesso (in realtà non lo è) e definiamo la *separazione* tra due profili $u, v \in V$ come il numero di archi nel cammino più breve da u a v in G. Per confermare o confutare la teoria dei 6 gradi di separazione, possiamo per determinarne la massima separazione tra due profili in G. La teoria sarà confermata se la massima separazione risulta minore o uguale a 6, e sarà confutata altrimenti.

c. Si noti che la separazione tra due profili $u, v \in V$ non è altro che la distanza $d(u, v)$ tra u e v in G. Definiamo max-sep(v) la massima separazione tra v e un qualsiasi altro vertice u. (Il valore max-sep(v) è anche chiamato *eccentricità di* v.) L'esercizio chiede quindi di determinare il valore

$$\max_{v \in V} \text{max-sep}(v).$$

Si osservi che per un qualsiasi profilo $v \in V$ l'esecuzione di `BFS`(G, v) partiziona gli altri profili in base alla loro separazione da v. In particolare, il livello L_i, con $i > 0$, conterrà tutti e soli i profili con separazione i da v. Se `BFS`(G, v) ha generato $k+1$ livelli non vuoti $(L_0, L_1, \ldots, L_k)$ significa che max-sep$(v) = k$. Supponiamo di modificare `BFS`(G, v) in modo che alla fine restituisca l'indice dell'ultimo livello non vuoto generato. Assumiamo anche di avere la lista L_E degli archi, che può essere scansionata tramite un iteratore. Per trovare la massima separazione tra due profili si può usare il seguente algoritmo.

Algoritmo `MaxSeparation`(G)

Input: `grafo` $G = (V, E)$ `non diretto e connesso`
Output: $\max_{v \in V}$ `max-sep`(v)

`max-sep` $\rightarrow 0$;
forall $v \in V$ **do**
  forall $u \in V$ **do** u.`ID` $\leftarrow 0$;
  forall $e \in E$ **do** e.`label` $\leftarrow$ `null`;
  `max-sep` $\rightarrow \max\{$`max-sep`, `BFS`$(G, v)\}$
return `max-sep`

Si noti che l'inizializzazione dei campi `ID` e `label` fatta all'inizio di ciascuna iterazione del forall esterno assicura che ciascuna invocazione della `BFS` visiti tutto il grafo. Di conseguenza, la viene invocata esattamente una volta a partire dal ciascun vertice. Poichè G è connesso, ogni invocazione costa $\Theta(n + m) = \Theta(m)$, dato che $m \geq n - 1$. Quindi, la complessità di tutto l'algoritmo è $\Theta(nm)$.

□

Problema 7 *Si consideri l'esecuzione di* `DFS`(G, s)*, e sia* T *lo spanning tree di* C_s*, radicato in* s*, costituito dagli archi etichettati come* `DISCOVERY EDGE` *da tale esecuzione. Supponiamo che durante una delle varie chiamate ricorsive* `DFS`(G, v) *si etichetti un arco* (v, w) *come* `BACK EDGE`*. Dimostrare che* w *è antenato di* v *in* T*.*

Soluzione. Quando `DFS`(G, v) esamina w come vicino di v, il fatto che etichetti l'arco (v, w) come `BACK EDGE`, implica che l'arco è stato trovato non ancora etichettato e w.`ID` $= 1$. Questo

significa che DFS(G, w) deve essere stata già invocata ma non conclusa, altrimenti (v, w) avrebbe già una etichetta diversa da `null` quando `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. □

Problema 8 *Sia $G = (V, E)$ un grafo non diretto con $k > 1$ componenti connesse. Progettare un algoritmo basato sulla DFS 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)`.

Soluzione. Siano $n = |V|$, $m = |E|$. Si denoti con u_j un vertice arbitrario dell'j-esima componente connessa di G, per $1 \leq j \leq k$. Si aggiungano a E i seguenti $k - 1$ archi:

$$\{(u_j, u_{j+1}) \ : \ 1 \leq j < k\}.$$

Si vede facilmente che con tale aggiunta il grafo diventa connesso. L'algoritmo richiesto è il seguente.

Algoritmo `MakeConnected`(G)

Input: `Grafo` $G = (V, E)$ `non diretto con` k `componenti connesse`
Output: `Grafo` $G' = (V, E \cup E')$ `connesso con` $|E'| = k - 1$

```
forall v ∈ V do v.ID ← 0;
forall e ∈ V do e.label ← null;
u ← vertice arbitrario di V;
DFS(G, u);
forall v ∈ V do
    if (v.ID = 0) then
        G.addArc(u, v);
        DFS(G, v);
        u ← v;
```

La correttezza dell'algoritmo è giustificata dalla precedente osservazione. Per quanto riguarda la complessità, l'algoritmo equivale sostanzialmente a una visita di tutto il grafo e, come visto a lezione, ha complessità $\Theta(n + m)$.

Osservazione: all'algoritmo non serve conoscere il numero k di componenti connesse. □

Problema 9 *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.*

Soluzione. Rappresentiamo L come grafo non diretto nel modo seguente. I vertici del grafo sono: il punto di ingresso s, gli incroci di L in cui si possono fare diverse scelte, e i

punti terminali di strade senza uscita in cui si è costretti a tornare indietro. Gli archi del grafo sono tutte le strade che collegano direttamente coppie di incroci. Realisticamente l'ing. Teseo può percorrere il labirinto a partire da s soltanto spostandosi lungo gli archi, e quindi muovendosi da un vertice a un vertice adiacente. Assumiamo che ogni vertice u sia una sorta di rotonda stradale in cui gli archi incidenti vengono esaminati in senso antiorario. Con questa assunzione, è sufficiente che Teseo esegua una DFS sul grafo con i seguenti adattamenti

- Se trova il Minotauro lo uccide.
- La prima volta che Teseo entra in un vertice v appende un a *marca di ingresso* vicino all'arco da cui è entrato.
- Quando Teseo attraversa un arco $e = (u, v)$ da u a v, opera come segue:
  - Se non trova alcuna marca di ingresso in v (quindi v non è stato ancora visitato) appende una marca di ingresso in v vicino a e, che equivale a etichettare e come DISCOVERY EDGE e a invocare DFS(L, v).
  - Se trova una marca di ingresso in v ed era arrivato a u per la prima volta da v (perchè in u c'è la marca di ingresso vicino all'arco e), allora significa che ha finito l'esame di tutti gli archi incidenti su u. Questo equivale alla fine della chiamata DFS(L, u), e Teseo prosegue a esaminare il prossimo arco incidente su v.
  - Se trova una marca di ingresso in v ma non era arrivato a u per la prima volta da v, allora Teseo torna su u e prosegue a esaminare il prossimo arco incidente su u. Questo equivale a trovare e già etichettato o etichettarlo come BACK EDGE.
  - Appena ritorna a s esce.

Teseo avrà successo nella sua missione dato che tutti i vertici vengono visitati e tutti gli archi vengono attraversati. (In effetti la DFS è stata inventata nel 19-esimo secolo dal matematico Francese Trémaux proprio come metodo di risoluzione di labirinti.) □

Problema 10 (Esercizio C-14.56 in [GTG14]) *Sia $G = (V, E)$ un grafo non diretto. Un insieme di vertici $I \subseteq V$ è un* Independent Set (IS) *rispetto a G, se E non contiene archi (u, v) con $u, v \in I$, e si dice* Maximal *se appena si aggiunge a esso un vertice $w \in V - I$, si perde la proprietà di IS. In altre parole, I è un* Maximal Independent Set (MIS) *se per ogni $w \in V - I$ esiste $v \in I$ tale che $(v, w) \in E$.*

a. Dimostrare che in ogni grafo G esiste sempre un MIS.

b. Progettare e analizzare un algoritmo efficiente per trovare un MIS in un grafo G.

Soluzione.

a. Si consideri un algoritmo che parte da un IS I vuoto, ed esegue un ciclo in cui ogni iterazione aggiunge a I un vertice v che non è ancora in I e che non è adiacente ad alcun vertice di I, se un tale v esiste. L'algoritmo termina quando non si trovano più vertici da aggiungere a I, ovvero quando $I = V$, oppure qualsiasi vertice non in I è adiacente a qualche vertice in I. In entrambi i casi, I è un MIS dato per costruzione non ci sono archi tra i soi vertici, e nessuno dei vertici in $V - I$ può essere aggiunto a I senza violare la proprietà di IS. L'algoritmo dimostra che esiste sempre un MIS.

b. Un'implementazione banale dell'algoritmo del punto precedente sembrerebbe richiedere, al caso pessimo, un'intera scansione di tutto V per ogni iterazione del ciclo. Tuttavia si può fare meglio eseguendo la scansione dei vertici una sola volta e aggiungendo ciascun vertice v all'IS I corrente se, nel momento in cui v viene esaminato, esso non ha vicini in I. Assumiamo di avere a disposizione, per ogni $v \in V$, un campo binario $v.\texttt{IS}$ che alla fine varrà 1 per tutti e soli i vertici del MIS. L'algoritmo è il seguente:

Algoritmo `MIS`(G)

Input: `Grafo` $G = (V, E)$ `non diretto`
Output: `MIS` I `per` G
forall $v \in V$ **do** $v.\texttt{IS} \leftarrow 0$;
$I \longleftarrow \emptyset$;
forall $v \in V$ **do**
  $v.\texttt{IS} \leftarrow 1$;
  forall $e \in G.\texttt{incidentEdges}(v)$ **do**
    $u \leftarrow G.\texttt{opposite}(v, e)$;
    if $(u.\texttt{IS} = 1)$ **then** $v.\texttt{IS} \leftarrow 0$;
  if $(v.\texttt{IS} = 1)$ **then** `aggiungi` v `a` I;
return I

La correttezza discende facilmente dal seguente invariante che vale alla fine di ogni iterazione del secondo ciclo forall:

- L'insieme I è un IS
- Nessuno dei vertici già esaminati e non inseriti in I può essere aggiunto a I senza violare la proprietà di IS.

La verifica dell'invariante è lasciata come esercizio. Per quanto riguarda la complessità, sia n il numero di vertici ed m il numero di archi di G. Il primo ciclo for richiede $O(n)$ operazioni, mentre ogni iterazione v del secondo ciclo for richiede un numero di operazioni proporzionale al grado del vertice v. Dato che la somma dei gradi di tutti i vertici è $O(m)$ si deduce che la complessità dell'algoritmo è $O(n + m)$.

□

Problema 11 *Un grafo non diretto G si dice* biconnected *se non esiste alcun vertice la cui rimozione, unitamente alla rimozione degli archi incidenti, disconnette il grafo. Far vedere che aggiungendo al più n archi a un grafo $G = (V, E)$ non diretto, connesso, e con $n = |V| \geq 3$ vertici, lo si può rendere biconnected (se non lo è inizialmente).*

Soluzione. Sia $V = \{v_1, v_2, \ldots, v_n\}$, dove i vertici sono indicizzati in base alla loro posizione nella lista L_V. Si supponga di aggiungere a E gli archi $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)$, se non già presenti. Tali archi formano un ciclo che tocca tutti i vertici, e di nuovi, rispetto a E, che ne sono al più n. Chiaramente, la rimozione di un qualsiasi vertice v_i, e degli archi in esso incidenti, non disconnette il grafo in quanto gli altri vertici rimangono connessi almeno dal cammino identificato dai seguenti archi:

$$(v_{i+1}, v_{i+2}) \cdots (v_n, v_1)(v_1, v_2) \cdots (v_{i-2}, v_{i-1}).$$

□

Problema 12 *Sia $G = (V, E)$ un grafo non diretto che rappresenta una rete sociale con n vertici ed m archi. Ogni vertice $u \in V$ ha un campo u.*`influencer` *che vale 1 se u è un influencer e 0 altrimenti. Un vertice x non influencer si dice* influenzabile *se esiste un influencer y e un cammino tra x e y. Progettare in pseudocodice un algoritmo* `Influenzabili` *che conti il numero di vertici influenzabili in G, e analizzarne la complessità. Per avere punteggio pieno la complessità deve essere $O(n + m)$.*

Soluzione. Si osservi che se in una componente connessa è presente un influencer, allora tutti i vertici non influencer di quella componente connessa sono influenzabili. In base a questa osservazione, possiamo risolvere il problema come segue. Si modifica `BFS`(G, s) in modo che determini il numero di influencer e il numero di non influencer nella componente connessa di s, restituendoli in output. A tale fine è sufficiente utilizzare due variabili n_1 e n_2 inizializzate a 0, e, quando si visita un vertice u, incrementare n_1 o n_2 di 1, a seconda che u sia influencer oppure no. Adesso, su ciascuna componente connessa, si invoca la `BFS` modificata e, se essa contiene almeno un influencer, allora il numero di non influencer della componente connessa viene aggiunto al conteggio degli influenzabili. Lo pseudocodice è il seguente.

Algoritmo `Influenzabili`(G)

Input: `Grafo` $G = (V, E)$ `non diretto con vertici etichettati come influencer/non influencer`
Output: `Numero di vertici di` V `influenzabili`

```
count ← 0;
forall v ∈ V do v.ID ← 0;
forall e ∈ V do e.label ← null;
forall v ∈ V do
    if (v.ID = 0) then
        (r1, r2) ← BFS(G, v);
        if (r1 > 0) then count ← count+r2;
return count
```

È immediato vedere che le modifiche richieste alla `BFS` non ne alterano la complessità. Dato che la `BFS` viene invocata al più una volta per ogni componente connessa (l'algoritmo ha la stessa struttura del pattern di visita completa di un grafo) la complessità è $\Theta(n + m)$. □

Problema 13 *Sia $G = (V, E)$ un grafo non diretto con n vertici ed m archi.*

a. *Progettare un algoritmo che dato G e un intero $k > 0$, per ogni vertice $v \in V$ salva in una variabile v.*`numNeighbors` *il numero di vertici a distanza $\leq k$ da v.*

b. *Analizzare la complessità dell'algoritmo.*

Soluzione.

a. Modifichiamo `BFS`(G, s) in modo che accumuli in una variabile `num-k`, inizializzata a 0, il numero di vertici che durante la visita sono inseriti nei livelli L_i, con $0 \leq i \leq k$. Questi sono tutti e soli i vertici che hanno distanza $\leq k$ da s. Al termine della vista `BFS`(G, s) imposterà la variabile s.`numNeighbors` al valore accumulato in `num-k`. A questo punto, è sufficiente invocare la `BFS` modificata da ciascun vertice di $v \in V$, resettando a 0 i campi `ID` di tutti i vertici, e a `null` i campi `label` di tutti gli archi, prima di ogni

invocazione (come nell'algoritmo sviluppato per il Problema 6) Lo pseudocodice è il seguente.

Algoritmo `CountNeighbors`(G, k)

Input: `Grafo` $G = (V, E)$ `non diretto con` n `vertici ed` m `archi, intero` $k \in [0, n)$
Output: $\forall v \in V$, v`.numNeighbors = numero vertici a distanza` $\leq k$ `da` v

forall $v \in V$ **do**
  forall $u \in V$ **do** u.`ID` $\leftarrow 0$;
  forall $e \in E$ **do** e.`label` $\leftarrow$ `null`;
  `BFS`(G, v)

b. È immediato vedere che le modifiche richieste alla `BFS` non ne alterano la complessità. Quindi, ciascuna iterazione del ciclo forall esterno richiede $O(n + m)$ operazioni. Di conseguenza, la complessità dell'algoritmo è $O(n(n + m)) = O(n^2 + nm)$.

□

Problema 14 *Sia $G = (V, E, w)$ un grafo non diretto e pesato, e sia s un vertice di V. Dimistrare che alla fine di ciascuna iterazione del ciclo while di* `ShortestPaths`(G, s)*, vale il seguente invariante. Per ogni $v \in V$:*

- *se v`.D` $= +\infty$, allora v`.parent = null`.*
- *se v`.D` $< +\infty$, il cammino ottenuto risalendo da v di "*`parent` *in* `parent`*" è un cammino da s a v di lunghezza v`.D`.*

Soluzione. L'inizializzazione assicura che l'invariante sia vero all'inizio del ciclo. Si osservi che la prima proprietà è assicurata dal fatto che v`.parent` può essere impostato a un valore diverso da `null` solo se v`.D` diventa $< +\infty$. Per quanto riguarda la seconda proprietà, essa viene mantenuta in ciascuna iterazione in base alla seguente osservazione. Se in una iterazione, v`.D` viene impostato a u`.D` $+ w(u, v)$ e v`.parent` viene impostato a u, la proprietà continua a valere in quanto esiste l'arco (u, v) e dall'iterazione precedente sappiamo che il cammino ottenuto risalendo da u di "`parent` in `parent`" è un cammino da s a u di lunghezza u`.D`. □

Problema 15 *Sia $G = (V, E, w)$ un grafo non diretto con n vertici ed m archi e con pesi non negativi sugli archi.*

a. Progettare una modifica di `ShortestPaths`(G, s) *in modo che la complessità sia $O(n + m_s \log n_s)$, dove n_s è il numero di vertici e m_s il numero di archi nella componente connessa di s (C_s).*

b. Dire se l'algoritmo modificato restituisce comunque le distanze corrette anche per i vertici non in C_s.

Soluzione.

a. È facile vedere che nella versione di `ShortestPaths`(G, s) presentata a lezione, nel momento in cui si estrae dalla Priority Queue Q una entry $(u$`.D`$, u)$ con u`.D` $= +\infty$, ovvero

associata a un vertice $u \notin C_s$, da quel momento in poi tutte le altre entry estratte avranno chiave uguale a $+\infty$, ovvero saranno relative a vertici che non appartengono a C_s, e non ci saranno altri aggiornamenti di Q. In altre parole, le entry associate ai vertici di C_s sono estratte prima di quelle associate a vertici non in C_s. Allora si può modificare `ShortestPaths`(G, s) in modo che Q sia inizializzata solo con la entry $(0, s)$, e che, la prima volta che per un vertice v si aggiorna il campo v`.D` a un valore $< +\infty$, la entry $(v$`.D`$, v)$ venga inserita in Q. In questo modo, le entry relative a vertici di C_s sono estratte da Q nello stesso ordine con cui vengono estratte nell'algoritmo originale e gli aggiornamenti dei campi `D` sono gli stessi (e nello stesso ordine) di quelli dell'algoritmo originale. Lo pseudocodice dell'algoritmo modificato è il seguente

```
s.D ← 0; s.parent ← null;
forall v ∈ V − {s} do
    v.D ← +∞;
    v.parent ← null
Q ← Priority Queue contenente solo la entry (0, s);
while (!Q.isEmpty()) do
    (u.D, u) ← Q.removeMin();
    forall (u, v) ∈ E do
        if (u.D + w(u, v) < v.D) then
            oldvD ← v.D;
            v.D ← u.D + w(u, v);
            v.parent ← u;
            if (oldvD < +∞) then (Q.decreaseKey(v.D, v));
            else (Q.insert(v.D, v));
```

Riguardo alla complessità, il primo ciclo forall richiede $O(n)$ operazioni e l'inizializzazione di Q richiede $\Theta(1)$ operazioni. Il ciclo while esegue esattamente n_s iterazioni, in ciascuna delle quali viene estratta una entry relativa a un vertice distinto di C_s. Si noti che Q conterrà solo entry relative a vertici di C_s, (quindi $|Q| \leq n_s$), e che per ogni vertice di C_s, la relativa entry viene inserita/estratta in/da Q solo una volta. Ne consegue che il contributo del ciclo while alla complessità è quello relativo a n_s estrazioni da Q, $n_s - 1$ inserimenti in Q, e m_s edge-relaxation (una per ogni arco di C_s), ciascuna delle quali richiede tempo al più proporzionale a quello di una invocazione al metodo `decreaseKey`. Si supponga di implementare Q tramite Heap. Come visto nell'analisi di `ShortestPaths`, il metodo `decreaseKey` può essere implementato con un numero di operazioni logaritmico nel numero di entry in Q. Dato che in ogni momento $|Q| \leq n_s$, il costo complessivo del ciclo while è $O((n_s + m_s) \log n_s) = O(m_s \log n_s)$, e la complessità dell'algoritmo modificato è $O(n + m_s \log n_s)$, come richiesto.

b. L'algoritmo modificato restituisce comunque le distanze corrette anche per i vertici non in C_s, in quanto esse sono impostate a $+\infty$ all'inizio dell'algoritmo, e non sono mai modificate.

□

Problema 16 *Sia G un grafo diretto rappresentato tramite liste di adiacenza. Progettare un algoritmo che, in tempo lineare nella taglia del grafo, salva in ogni nodo $v \in V$ due campi* `v.outdeg` *e* `v.indeg` *uguali a, rispettivamente,* `outdegree(v)` *e* `indegree(v)`*.*

Soluzione. L'idea è di visitare tutti gli archi e, per ciascun arco (v, u), incrementare di 1 `v.outdeg` e `u.indeg`, inizialmente impostati a 0. Lo pseudocodice è il seguente.

Algoritmo `SetInOut`(G)

```
Input: Grafo G = (V, E) diretto con n vertici ed m archi
Output: ∀v ∈ V, v.indeg = indegree(v) e v.outdeg = outdegree(v)
forall v ∈ V do
    v.indeg ← 0;
    v.outdeg ← 0;
forall v ∈ V do
    forall e ∈ G.incidentEdges(v) do
        v.outdeg ← v.outdeg + 1;
        u ← G.opposite(v, e);
        u.indeg ← u.indeg + 1;
```

□

Problema 17 *Sia $G = (V, E)$ un grafo diretto. Modificare lo pseudocodice* `DFS`(G, s) *in modo che un arco (v, w), esaminato dalla chiamata* `DFS`(G, v)*, sia etichettato come* BACK EDGE *se w è antenato di v nello spanning tree definito dai* DISCOVERY EDGE*, e sia etichettato* ALTRO *negli altri casi.*

Soluzione. Sia T lo spanning tree definito dai DISCOVERY EDGE. È facile osservare che i discendenti di un vertice w di T sono tutti e soli i vertici scoperti durante l'esecuzione di `DFS`(G, w) e delle eventuali chiamate ricorsive fatte al suo interno. Quindi, per sapere se w è un antenato di v è sufficiente verificare se la chiamata `DFS`(G, w) ha terminato la sua esecuzione oppure no. A questo scopo, modifichiamo la DFS come segue. Supponiamo che ciascun vertice $w \in V$ abbia un flag w`.ON` inizializzato a 0. Tale flag sarà impostato a 1 appena `DFS`(G, w) inizia la sua esecuzione, e resettato a 0 quando `DFS`(G, w) termina la sua esecuzione. In questo modo, durante l'esecuzione di `DFS`(G, v) e l'esame dell'arco (v, w), se w è già stato visitato (cioè w`.ID` $= 1$) e (v, w) non ancora etichettato, l'arco sarà etichettato BACK EDGE se w`.ON` $= 1$, e ALTRO se w`.ON` $= 0$. È immediato vedere che la modifica alla DFS non ne altera la complessità. □

Problema 18 *Sia $G = (V, E)$ un grafo diretto con n vertici e m archi. Dato un vertice $s \in V$ si definisca E_s l'insieme di archi uscenti da* `reachable`(s)*. È noto che esiste un ciclo diretto in E_s se e solo se* `DFS`(G, s) *etichetta un arco come* BACK EDGE*. Usando questa proprietà, progettare e analizzare un algoritmo che dato $s \in V$ restituisce una lista L che contiene gli archi di un ciclo in E_s, se ne esiste uno, altrimenti è vuota.*

Soluzione. Dal Problema 17 sappiamo che la `DFS` può essere facilmente modificata in modo che identifichi i BACK EDGE. La modifichiamo ulteriormente come segue. Si supponga che ciascun vertice v abbia un campo v`.parent` inizializzato a `null`.

- Se un arco (v, w) è etichettato come DISCOVERY EDGE, si imposta w.`parent` a v.
- Se durante l'esecuzione di `DFS`(G, v) (e delle chiamate ricorsive al suo interno) almeno arco e viene etichettato come BACK EDGE, allora `DFS`(G, v) restituisce un tale arco, altrimenti restituisce `null`.

È facile vedere che le modifiche possono essere implementate senza alterare la complessità di `DFS`. A questo punto, grazie alla proprietà enunciata nel testo del problema, sappiamo che se `DFS`(G, s) non identifica alcun BACK EDGE, l'insieme di archi E_s non contiene cicli e si restituisce una lista L vuota. Altrimenti se `DFS`(G, s) restituisce un BACK EDGE (u, v) il ciclo da restiutire è ottenuto inserendo nella lista L l'arco (u, v) e tutti DISCOVERY EDGE incontrati risalendo da u a v di `parent` in `parent`. La scrittura dettagliata dello pseudocodice è lasciata come esercizio. Se si assume che i campi dei vertici e degli archi richiesti dall'algoritmo siano già opportunamente inizializzati, l'invocazione della `DFS` modificata e la costruzione del ciclo richiede $\Theta(|E_s|)$ operazioni. □

Problema 19 *Sia $G = (V, E)$ il grafo diretto di Instagram in cui i vertici rappresentano i profili e un arco (u, v) il fatto che il profilo u segue il profilo v. Progettare e analizzare un algoritmo che dato il profilo di un influencer $i \in V$ e un valore $k > 0$, restituisca il numero dei profili $u \in V$ tali che la distanza $d(u, i)$ del cammino minimo da u a i è $\leq k$. L'algoritmo deve avere complessità $O(n + m)$.*

Soluzione. L'idea è la seguente. Sia $G^T = (V, E^T)$ il grafo ottenuto da G invertendo l'orientamento di ciascun arco. Chiaramente, se G è fortemente connesso anche G^T lo è. Inoltre, è immediato vedere che dato $i \in V$, un vertice u per cui la distanza $d(u, i)$ del cammino minimo da u a i è massima, è un qualsiasi vertice che nella BFS eseguita su G_T a partire da i, è inserito in un livello di indice $\leq k$. L'algoritmo richiesto deve quindi creare G^T (a questo fine è sufficiente visitare tutto il grafo come nella soluzione del Problema 16 e creare le liste di adiacenza per G^T), eseguire la `BFS`(G^T, i) modificata in modo che il suo output sia il numero totale di vertici nei livelli di indice $\leq k$, e restituire tale numero. La scrittura dello pseudocodice e l'analisi sono lasciati come esercizio. □

Problema 20 *Dimostrare le seguenti proprietà:*

a. Se un grafo diretto $G = (V, E)$ ha un ciclo diretto, non può avere un ordinamento topologico.

b. Dato un grafo semplice non diretto $G = (V, E)$ è possibile dare un orientamento ai suoi archi in modo che esso diventi un DAG.

Soluzione.

a. Sia $v_1 \to v_2 \to \cdots \to v_k \to v_{k+1} = v_1$ un ciclo diretto in G di lunghezza k, per un qualche $k \geq 2$. Se per assurdo esistesse un ordinamento topologico di G, avremmo che v_i dovrebbe occorrere prima di v_j, per ogni $0 < i < j \leq k + 1$, ma questo è impossibile che si verifichi per $i = 1$ e $j = k + 1$, dato che $v_1 = v_{k+1}$.

b. Sia $V = v_1, v_2, \ldots, v_n$, dove i vertici sono indicizzati in modo arbitrario. È sufficiente orientare ciascun arco dal vertice di indice minore a quello di indice maggiore. In questo modo non ci possono essere cicli, dato che un ciclo richiederebbe almeno un arco da un vertice di indice maggiore a uno di indice minore.

□

Problema 21 *Modificare l'algoritmo* `TopologicalSort`(G) *in modo che se il grafo di input non è un DAG restituisce* `null`*, altrimenti restituisce, come ora, l'ordinamento topologico dei suoi vertici.*

Soluzione. Sia $G = (V, E)$ un grafo diretto. È facile vedere che, se G non è un DAG, il ciclo while di `TopologicalSort`(G) termina lasciando alcuni vertici con il campo `indeg` maggiore di 0. Infatti, se G non è un DAG deve contenere un ciclo diretto, cioè una sequenza di vertici $v_1, v_2, \ldots, v_k = v_1$ tale che $(v_i, v_{i+1}) \in E$, per ogni $1 \leq i < k$. Durante l'esecuzione `TopologicalSort`(G) il campo `indeg` di ciascun v_i, che parte da un valore maggiore di 0, rimarrà sempre maggiore di 0[1]. I vertici il cui campo `indeg` non va mai a 0, non possono mai entrare in L, che quindi, alla fine, conterrà solo un sottoinsieme proprio di V. Per far in modo che l'algoritmo emetta l'output desiderato, è sufficiente sostituire l'istruzione **return** L alla fine dell'algoritmo, con la seguente:

$$\textbf{if } (|L| = |V|) \textbf{ return } L \textbf{ else return } \texttt{null}$$

□

Problema 22 *Sia $G = (V, E)$ un DAG i cui vertici rappresentano task e ogni arco (u, v) indica che v deve essere eseguito dopo u. Per ogni task v, un campo v.`length` contiene la durata del task. Progettare un algoritmo che in tempo $O(n + m)$ assegna a ciascun task v un tempo di inizio compatibile con le informazioni contenute in G, memorizzandolo in un campo v.`start`.*

Suggerimento: *usare l'algoritmo* `TopologicalSort`(G) *come primitiva.*

Soluzione. L'idea è di ottenere prima un ordinamento topologico dei task, diciamo $v_1, v_2, \ldots, v_n$, e poi assegnare a v_1 un tempo di inizio pari a 0, e a ciascun v_i, con $i > 1$, un tempo di inizio pari a quello di v_{i-1} più la durata del task v_{i-1}. Lo pseudocodice è il seguente:

Algoritmo `Scheduling`(G)
Input: DAG $G = (V, E)$ i cui vertici rappresentano task con durate associate
Output: Tempo di inizio v.`start` per ogni task $v \in V$
$L \leftarrow$ `TopologicalSort`(G);
$p \leftarrow L$.`first`(); /* p is a position */
$v \leftarrow p$.`getElement`(); /* v is a vertex */
v.`start` $\leftarrow 0$;
currentTime $\leftarrow v$.`length`;
while (L.`after`$(p) \neq$ `null`) **do**
  $p \leftarrow L$.`after`(p);
  $v \leftarrow p$.`getElement`();
  v.`start` $\leftarrow$ currentTime;
  currentTime $\leftarrow$ currentTime + v.`length`;
  $p \leftarrow L$.`after`(p);

[1] Altrimenti ci sarebbe un primo v_i il cui campo `indeg` va a 0, cosa non possibile perché il campo `indeg` di v_{i-1} sarebbe dovuto andare a zero prima.

È immediato vedere che la complessità rimane asintoticamente uguale a quella di `TopologicalSort`, cioè $O(n+m)$, dato che il ciclo while richiede $O(n)$ operazioni. □