Dati e Algoritmi I (Pietracaprina)

Esercizi svolti sui Grafi

Problema 1 *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)`.

Soluzione. Siano $n = |V|$, $m = |E|$, e $V = \{1, 2, \ldots, n\}$. Si denoti con v_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:

$$\{(v_j, v_{j+1}) \ : \ 1 \leq j < k\}.$$

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` $|E'| = k - 1$

```
DFS(G, 1);
u ← 1;
for v ← 2 to n do
    if (L_V[v].ID = 0) then
        G.addArc(u, v);
        u ← v;
        DFS(G, 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 $\Theta(n + m)$. □

Problema 2 *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. Eseguendo la DFS su tale grafo (e assumendolo connesso) si riesce a scovare il Minotauro ovunque esso sia, 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 3 *Progettare e analizzare un algoritmo efficiente basato sulla BFS che dato un grafo $G = (V, E)$ non diretto restituisca un ciclo, se esiste, o l'indicazione* `NO CYCLE`*, altrimenti.*

Soluzione. Osserviamo che G contiene un ciclo se e solo se, visitando tutte le sue componenti connesse con la BFS, uno degli archi viene etichettato come `CROSS EDGE`. L'algoritmo descritto di seguito è basato su questa osservazione: prima cerca un `CROSS EDGE` (u, v) e, se lo trova, crea un ciclo aggiungendo a tale arco tutti i `DISCOVERY EDGE` incontrati risalendo da u e v al loro primo antentato comune. Assumiamo di rappresentare i vertici di G con gli interi $1, 2, \ldots, n$ e supponiamo che per ciascun vertice $v \in V$ siano disponibili i seguenti campi: $L_V[v]$`.ID`, utilizzato dall'algoritmo BFS e inizializzato a 0, $L_V[v]$`.level`, destinato a contenere il livello di v nella BFS della sua componente connessa e inizializzato a `null`), e $L_V[v]$`.parent`, destinato a contenere il padre di v, se esiste, nel BFS tree della sua componente connessa e inizializzato a `null`. Si supponga di modificare `BFS`(G, v) in modo che quando un vertice w viene inserito nel livello L_i si imposti $L_V[w]$.level $= i$, e $L_V[w]$.parent $= x$, dove x (se esiste) è il vertice analizzando i cui vicini si scopre w. L'algoritmo è il seguente:

Algoritmo `FindCycle`(G)

Input: `grafo` $G = (V, E)$ `non diretto`
Output: `un ciclo` C `in` G`, o NO CYCLE se` G `è aciclico`
for $v \leftarrow 1$ **to** n **do**
  if $(L_V[v]$.ID$= 0)$ **then** `BFS`(G, v);
forall $e \in E$ **do**
  if $(e = (u, v)$ `è marcato come CROSS EDGE`$)$ **then**
    $C \leftarrow \{e\}$;
    if $(L_V[u]$.level $= L_V[v]$.level $+ 1)$ **then**
      `aggiungi` $(u, L_V[u]$`.parent`$)$ `a` C;
      $u \leftarrow L_V[u]$`.parent`;
    if $(L_V[v]$.level $= L_V[u]$.level $+ 1)$ **then**
      `aggiungi` $(v, L_V[v]$`.parent`$)$ `a` C;
      $v \leftarrow L_V[v]$`.parent`;
    while $(u \neq v)$ **do**
      `aggiungi` $(u, L_V[u]$`.parent`$)$ `e` $(v, L_V[v]$`.parent`$)$ `a` C;
      $u \leftarrow L_V[u]$`.parent`;
      $v \leftarrow L_V[v]$`.parent`;
    return C;
return `NO CYCLE`;

È facile vedere che le modifiche richieste alla BFS non ne alterano la complessità. Di conseguenza, il ciclo for equivale sostanzialmente a una visita di tutto il grafo e, come visto a lezione, ha complessità $\Theta(n + m)$, dove n è il numero di vertici ed m il numero di archi. Il 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 possono essere interrotte appena si marca un arco come `CROSS EDGE`.) □

Problema 4 *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.)

Soluzione. 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 raggiungibili uno dall'altro. Assumiamo di rappresentare i vertici di G con gli interi $1, 2, \ldots, n$ e supponiamo di modificare `BFS`(G, v) definendo una variabile `cardinality` inizializzata a 0 che viene incrementata ogni volta in cui si visita 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
```

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 5 *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 6 *Sia $G=(V,E)$ un grafo con n vertici e m archi. Progettare 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`*.*

Soluzione. Si noti che da un vertice $i \in V$ sono raggiungibili $\geq n/2$ altri vertici se e solo se la componente connessa di i 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). Assumiamo di rappresentare i vertici di G con gli interi $1, 2, \ldots, n$ e supponiamo di modificare `BFS`(G, v) definendo una variabile `cardinality` inizializzata a 0 che viene incrementata ogni volta in cui si visita un vertice 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`

for $v \leftarrow 1$ **to** n **do**
  if $(L_V[v]$`.ID` $= 0)$ **then**
    $K \leftarrow$ `BFS`(G, v);
    if $(K \geq n/2+1)$ **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 7 *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 eseguendo la BFS da un vertice v, 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. In virtù di questa osservazione, per controllare se una componente connessa è bipartita sarà sufficiente invocare la BFS a partire da un suo vertice arbitrario v e verificare che non ci siano `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 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.

Assumiamo di rappresentare i vertici di G con gli interi $1, 2, \ldots, n$ e supponiamo che per ciascun vertice $v \in V$ siano disponibili i seguenti campi: $L_V[v]$`.ID`, utilizzato dall'algoritmo BFS e inizializzato a 0, e $L_V[v]$`.level`, destinato a contenere il livello di v nella BFS della sua componente connessa e inizializzato a `null`. Si supponga di modificare `BFS`(G, v) in modo

che quando un vertice w viene inserito nel livello L_i si imposti $L_V[w].\text{level} = i$. L'algoritmo è il seguente:

Algoritmo `isBipartite`(G)

Input: `Grafo` $G = (V, E)$ `non diretto`
Output: `TRUE/FALSE se` G `è/non è bipartito`

for $v \leftarrow 1$ **to** n **do**
  if $(L_V[v].\text{ID} = 0)$ **then** `BFS`(G, v);
`bipartite` $\leftarrow$ `TRUE ;`
forall $e = (u, v) \in E$ **do**
  if $(L_V[u].\texttt{level} = L_V[v].\texttt{level})$ **then** `bipartite` $\leftarrow$ `FALSE;`
return `bipartite;`

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 8 *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.*

Soluzione. 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 $(L_0, L_1, \ldots, L_k)$ significa che la massima separazione tra v e un altro profilo è k. Assumiamo di rappresentare i vertici di G con gli interi $1, 2, \ldots, n$ e supponiamo di modificare BFS(G, v) in modo che alla fine restituisca l'indice dell'ultimo livello (non vuoto) generato. 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 distanza (=separazione) tra 2 vertici`

`max-sep` $\rightarrow 0$;
for $v \leftarrow 1$ **to** n **do**
  `max-sep` $\rightarrow \max\{$`max-sep`, `BFS`$(G, v)\}$
return `max-sep`

BFS viene invocato 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 9 (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) 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 costituito da un vertice arbitrario, ed esegue ciclo in cui sino a quando esiste un vertice $v \notin I$ che non è adiacente ad alcun vertice di I lo aggiunge a I. Per costruzione, I rimane sempre un IS. L'algoritmo termina se $I = V$ oppure qualsiasi vertice non in I è adiacente a qualche vertice in I. In entrambi i casi I è un MIS dato che è un IS e non si possono aggiungere vertici a esso mantenedo 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, una scansione di tutti i vertici non in I e di tutti gli archi in essi incidenti in ciascuna iterazione del ciclo. È possibile però implementarlo in modo più efficiente con un'unica scansione di vertici e archi. L'idea è di fare una scansione di tutti i vertici e per ciascuno di essi determinare se può essere aggiunto all'IS corrente (che inizialmente è vuoto) e, in questo caso, aggiungerlo. Rappresentiamo, come sempre, i vertici tramite gli interi da 1 a n. Per ogni $v \in V$ usiamo un campo booleano $L_V[v]$.`IS` che alla fine varrà TRUE per tutti e soli i vertici del MIS. Lo pseudocodice è il seguente:

Algoritmo `MIS`(G)

```
Input: Grafo G = (V, E) non diretto
Output: MIS I per G
for v ← 1 to n do L_V[v].IS ← FALSE;
I ⟵ ∅;
for v ← 1 to n do
    L_V[v].IS ← TRUE;
    forall e = (v, u) ∈ G.incidentEdges(v) do
        if (L_V[u].IS = TRUE) then L_V[v].IS ← FALSE;
    if (L_V[v].IS = TRUE) then aggiungi v a I;
return I
```

La correttezza discende facilmente dal seguente invariante che vale alla fine di ogni iterazione $v \geq 0$ del secondo ciclo for ($v = 0$ è l'inizio del ciclo):

- L'insieme I è un IS
- Nessun vertice u, con $1 \leq u \leq v$ e $u \notin I$, 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 10 *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 $V = \{1, 2, \ldots, n\}$. Si supponga di aggiungere a E il seguente insieme di archi (se non già presenti in E):

$$\{(1, 2), (2, 3), \ldots, (n - 1, n), (n, 1)\}.$$

Con l'aggiunta di tali archi (che sono al più n dato che alcuni potrebbero già essere parte di E) si crea un ciclo che tocca tutti i vertici. Chiaramente, la rimozione di un qualsiasi vertice 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:

$$(i + 1, i + 2) \cdots (n, 1)(1, 2) \cdots (i - 1, i).$$

□