Algoritmi e Strutture Dati II

Esercizi sulla visita di un grafo

Alberto Dennunzio e Giancarlo Mauri

Esercizio 1

Scrivere un algoritmo che, dato un grafo $G = (V, E)$ e un vertice $s \in V$, conta i vertici raggiungibili da s (escluso s stesso).

Soluzione. Si può eseguire l'algoritmo di visita in ampiezza del grafo dato (algoritmo BFS a pag. 455 del libro di testo) ed utilizzare l'informazione contenuta nel vettore d delle distanze alla fine della visita. Precisamente, i vertici $u \in V \setminus \{s\}$ raggiungibili da s sono quelli per cui si ha $d[u] \neq \infty$ alla fine della visita. Pertanto, un algoritmo che risolve il problema proposto è il seguente:

```
COUNT–REACHABLE(G, s)
   BFS(G, s)
   n := 0
   for ogni u ∈ V \ {s} do
         if d[u] ≠ ∞ then do
               n := n + 1
   return n
```

Esercizio 2

Scrivere un algoritmo che, dati un grafo $G = (V, E)$ non orientato ed un vertice $v \in V$, stampa l'elenco dei vertici che hanno distanza pari da v (e quindi stampa anche v).

Soluzione. L'idea è quella di eseguire l'algoritmo BFS sul grafo dato con v vertice sorgente e di utilizzare l'informazione contenuta nel vettore delle distanze d alla fine della visita. Precisamente, i vertici $u \in V$ a distanza pari da v sono quelli tali che alla fine della visita $d[u] \neq \infty$ (cioè sono raggiungibili da v) e $d[u]$ è divisibile per 2. Pertanto, un algoritmo che risolve il problema proposto è il seguente:

```
PRINT–EVEN-DISTANCE(G)
BFS(G, v)
for ogni vertice u ∈ V do
     if d[u] ≠ ∞ and d[u] mod 2 = 0 then do
          print u
```

Esercizio 3

Scrivere un algoritmo che, dati un grafo $G = (V, E)$, un vertice $s \in V$ e un intero $k > 0$, effettua la visita in ampiezza di G sino a scoprire i vertici che distano k da s. Si richiede che al termine dell'esecuzione dell'algoritmo i vertici v con $d[v] \leq k$ abbiano colore nero, mentre gli altri vertici abbiano colore bianco.

Soluzione. È sufficiente modificare l'algoritmo BFS di visita in ampiezza di un grafo in modo tale che un vertice scoperto v venga colorato di grigio ed inserito nella coda Q solo se $d[v] < k$, mentre non venga inserito in Q e sia colorato di nero se $d[v] = k$. Si ricorda infatti che nella coda vengono inseriti i vertici grigi appena scoperti. A partire da essi saranno scoperti successivamente vertici a distanza maggiore. Quindi al fine di non scoprire vertici a distanza maggiore di k è sufficiente non inserire in coda i vertici appena scoperti che sono a distanza k dalla sorgente. Pertanto, un algoritmo che risolve il problema proposto è il seguente:

```
BFS-UNTIL-DISTANCE(G, s, k)
   for ogni vertice u ∈ V \ {s} do
         color[u] := WHITE
         d[u] := ∞
         π[u] := NIL
   color[s] := GRAY
   d[s] := 0
   π[s] := NIL
   Q := {s}
   while Q ≠ ∅ do
         u := head[Q]
         for ogni v ∈ Adj[u] do
              if color[v] = WHITE  then do
                   π[v] := u
                   d[v] := d[u] + 1
                   if d[v] < k then do
                        color[v] := GRAY
                        ENQUEUE(Q, v)
                   else do
                        color[v] := BLACK
         DEQUEUE(Q)
         color[u] := BLACK
```

Esercizio 4.

Scrivere un algoritmo che stabilisce se un grafo $G = (V, E)$ non orientato è connesso.

Soluzione. Si può eseguire l'algoritmo BFS sul grafo dato con vertice sorgente s scelto casualmente nell'insieme V ed utilizzare l'informazione contenuta nel vettore d delle distanze alla fine della visita. Il grafo è connesso se tutti i vertici $u \in V \setminus \{s\}$ sono raggiungibili da s ossia se per tutti i vertici $u \in V \setminus \{s\}$ si ha $d[u] \neq \infty$ alla fine della visita. Il seguente algoritmo ritorna *true* se e solo se il grafo G è connesso:

```
IS–CONNECTED(G)
   s :=RANDOM(V)
   BFS(G, s)
   for ogni u ∈ V \ {s} do
         if d[u] = ∞ then do
               return false
   return true
```

Esercizio 5.

Scrivere un algoritmo che stabilisce se un grafo $G = (V, E)$ non orientato è un albero.

Soluzione. Si ricorda che un albero è un grafo non orientato, connesso e privo di cicli. Per stabilire se G è un albero occorre quindi stabilire se è connesso e privo di cicli. Per controllare se G è connesso si può utilizzare la procedura IS–CONNECTED(G) illustrata nell'esercizio 4. Si può evitare di controllare la presenza di cicli sfruttando la seguente proprietà (si veda il teorema B.2 a pag. 927 del libro di testo): un grafo $G = (V, E)$ non orientato è un albero se e solo se G è connesso e $|E| = |V| - 1$. Pertanto una volta stabilito che il grafo G dato è connesso si potrà concludere che esso è un albero se e solo se $|E| = |V| - 1$. Si ricordi infine che in un grafo non orientato $|E| = \frac{1}{2} \sum_{u \in V} |Adj[u]|$. Il seguente algoritmo ritorna *true* se e solo se il grafo G è un albero:

```
IS–TREE(G)
   connected := IS–CONNECTED(G)
   if connected = true then do
         if |E| = |V| − 1 then do
               return true
         else do
               return false
   else do
         return false
```

Esercizio 6.

Modificare l'algoritmo DFS di visita di un grafo *orientato* G in maniera tale che stampi ogni arco di G specificandone il tipo (ossia se è un arco dell'albero, un arco in avanti, un arco all'indietro o un arco di attraversamento).

Soluzione. Un arco (u, v) è dell'albero se v è scoperto esplorando (u, v) e quindi se durante la visita $color[v] = \text{WHITE}$. Un arco (u, v) è all'indietro se connette u ad un suo antenato v e quindi se durante la visita $color[v] = \text{GRAY}$. Un arco (u, v) è in avanti se connette u ad un suo discendente v e quindi se durante la visita $color[v] = \text{BLACK}$ e $d[u] < d[v]$. Se esplorando (u, v) si ha $color[v] = \text{BLACK}$ e $d[u] > d[v]$ allora (u, v) è un arco di attraversamento. Una possibile soluzione al problema dato è quindi:

```
PRINT–EDGE–TYPE(G)
  for ogni vertice u ∈ V do
        color[u] := WHITE
        π[u] := NIL
  time := 0
  for ogni vertice u ∈ V do
        if color[u] = WHITE then do
             PRINT–EDGE–TYPE–VISIT(u)

PRINT–EDGE–TYPE–VISIT(u)
  color[u] := GRAY
  time := time + 1
  d[u] := time
  for ogni v ∈ Adj[u] do
        if color[v] = WHITE then do
             print u,v ``arco dell'albero''
             π[v] := u
             PRINT–EDGE–TYPE–VISIT(v)
        else do
             if color[v] = GRAY then do
                  print u,v ``arco all'indietro''
             else do
                  if d[u] < d[v] then do
                       print u,v ``arco in avanti''
                  else do
                       print u,v ``arco di attraversamento''
  color[u] := BLACK
  time := time + 1
  f[u] := time
```

Esercizio 7.

Modificare l'algoritmo DFS di visita di un grafo *non orientato* G in maniera tale che stampi ogni arco di G specificandone il tipo.

Soluzione. Un grafo non orientato può avere solo archi dell'albero o archi all'indietro. Per specificare il tipo di un arco si può procedere in modo simile al caso di un grafo orientato. Durante l'esplorazione di un arco (u, v) si stampa il suo tipo a seconda che il colore di v sia bianco o grigio. Nel secondo caso, per affermare che l'arco è all'indietro, occorre anche aver controllato che v non sia padre di u. Infatti se v ha colore grigio ed è il padre di u l'arco (v, u) è un arco dell'albero ed è stato etichettato tale quando u è stato scoperto. Pertanto l'arco (u, v), che coincide con (v, u), non va considerato arco all'indietro. Una possibile soluzione al problema dato è quindi:

```
PRINT–EDGE–TYPE–NO(G)
   for ogni vertice u ∈ V do
          color[u] := WHITE
          π[u] := NIL
   time := 0
   for ogni vertice u ∈ V do
          if color[u] = WHITE  then do
               PRINT–EDGE–TYPE–VISIT–NO(u)

PRINT–EDGE–TYPE–VISIT–NO(u)
   color[u] := GRAY
   time := time + 1
   d[u] := time
   for ogni v ∈ Adj[u] \ {π[u]} do
          if color[v] = WHITE  then do
               print u,v ''arco dell'albero''
               π[v] := u
               PRINT–EDGE–TYPE–VISIT–NO(v)
          else do
               print u,v ''arco all'indietro''
   color[u] := BLACK
   time := time + 1
   f[u] := time
```

Esercizio 8.

Modificare l'algoritmo DFS di visita di un grafo *orientato* G in maniera tale che stabilisca se G è acicilico, ossia se non contiene cicli.

Soluzione. Un grafo orientato è aciclico se non contiene nessun arco all'indietro. Per risolvere il problema, è quindi sufficiente controllare che non vi siano archi all'indietro. Per fare questo, si può utilizzare una variabile booleana che alla fine della visita avrà valore *true* se e solo se il grafo è acicilico. La si inizializza a *true* e le si assegna valore *false* non appena si incontra un arco all'indietro durante la visita del grafo. Il seguente algoritmo restituisce valore *true* se e solo se il grafo dato è privo di cicli.

```
Is–Acyclic(G)
   for ogni vertice u ∈ V do
         color[u] := WHITE
         π[u] := NIL
   time := 0
   acyclic := true
   for ogni vertice u ∈ V do
         if color[u] = WHITE  then do
              Is–Acyclic–Visit(u)
   return acyclic

Is–Acyclic–Visit(u)
   color[u] := GRAY
   time := time + 1
   d[u] := time
   for ogni v ∈ Adj[u] do
         if color[v] = WHITE  then do
              π[v] := u
              Is–Acyclic–Visit(v)
         else do
              if color[v] = GRAY  and acyclic = true then do
                   acyclic := false
   color[u] := BLACK
   time := time + 1
   f[u] := time
```

Si noti che l'algoritmo proposto non interrompe la visita quando scopre la presenza di un ciclo. È possibile aggiungere dei controlli per interrompere la visita non appena viene scoperto un ciclo.

Esercizio 9.

Modificare l'algoritmo DFS di visita di un grafo *non orientato* G in maniera tale che stabilisca se G è acicilico, ossia se non contiene cicli.

Soluzione. Si tratta del problema dell'esercizio precedente per il caso di grafi non orientati. Si procede nello stesso modo tenendo però presente l'osservazione sugli archi all'indietro di grafi non orientati (si veda l'Esercizio 7).

```
Is–Acyclic–NO(G)
   for ogni vertice u ∈ V do
         color[u] := WHITE
         π[u] := NIL
   time := 0
   acyclic := true
   for ogni vertice u ∈ V do
         if color[u] = WHITE  then do
              Is–Acyclic–Visit–NO(u)
   return acyclic

Is–Acyclic–Visit–NO(u)
   color[u] := GRAY
   time := time + 1
   d[u] := time
   for ogni v ∈ Adj[u] \ {π[u]} do
         if color[v] = WHITE  then do
              π[v] := u
              Is–Acyclic–Visit–NO(v)
         else do
              if color[v] = GRAY  and acyclic = true then do
                   acyclic := false
   color[u] := BLACK
   time := time + 1
   f[u] := time
```

Esercizio 10.

Scrivere un algoritmo che determina il numero di componenti connesse di un grafo $G = (V, E)$ non orientato.

Soluzione. Per risolvere il problema dato, è sufficiente modificare l'algoritmo di visita in profondità per quanto concerne la procedura DFS(G). Si ricorda che ogni volta che DFS(G) invoca la chiamata DFS–Visit(u) viene individuata una nuova componente connessa del grafo. Quindi, contare le componenti connesse di G equivale a contare quante volte viene invocata la procedura DFS–Visit(u) da parte di DFS(G). Pertanto una soluzione al problema dato è la seguente:

```
COUNT–CONNECTED–COMPONENTS(G)
   for ogni vertice u ∈ V do
         color[u] := WHITE
         π[u] := NIL
   time := 0
   n := 0
   for ogni vertice u ∈ V do
         if color[u] = WHITE  then do
               n := n + 1
               DFS–VISIT(u)
   return n
```

Esercizio 11.

Scrivere un algoritmo che, dato un grafo $G = (V, E)$ non orientato, permetta di individuare la componente connessa di cui ogni vertice $u \in V$ fa parte.

Soluzione. Per risolvere il problema dato, è sufficiente numerare le componenti connesse del grafo dato e assegnare ad ogni vertice $u \in V$ il numero della componente connessa di cui u fa parte. Per numerare le componenti connesse basta contarle modificando la procedura DFS(G) come illustrato nell'esercizio precedente. Per individuare la componente connessa di cui ogni vertice fa parte, si introduce un vettore cc e quando un vertice u è scoperto si assegna il numero della componente connessa attualmente visitata memorizzando tale valore in $cc[u]$. L'assegnamento viene quindi aggiunto all'inizio dell'usuale procedura DFS-VISIT(u).

Esercizio 12.

Scrivere un algoritmo che, dati un grafo $G = (V, E)$ non orientato e un intero $k > 0$, stabilisca se G è una foresta composta da k alberi.

Soluzione. Per risolvere il problema dato, occorre controllare che il grafo G sia aciclico e composto da k componenti connesse. Pertanto la soluzione è una combinazione delle soluzioni ai problemi presentati negli esercizi 9 e 10.

Esercizio 13.

Scrivere un algoritmo che, dato un grafo $G = (V, E)$ non orientato, calcola il numero di componenti connesse che contengono un numero pari di vertici.

Soluzione. Si tratta di contare quanti vertici sono presenti in ogni componente connessa del grafo e testare se tale numero è pari incrementando in caso affermativo una variabile inizializzata a 0 e introdotta nell'usuale procedura DFS(G) per contare le componenti con vertici in numero pari. Per contare quanti sono i vertici in una componente connessa è possibile far ritornare ricorsivamente tale valore all'usuale DFS-Visit, che verrà modificata e rinominata DFS–Count–Visit. Precisamente, dato un vertice u, DFS–Count–Visit(u) ritorna il numero di vertici scoperti a partire da u (compreso u). Per ritornare tale valore, tale procedura incrementa una variabile (*count*), mediante chiamate ricorsive, del numero DFS–Count–Visit(v) di tutti i vertici scopribili a partire da quei vertici $v \in Adj[u]$ che sono immediatamente scopribili da u. La procedura ritornerà quindi il valore di *count* incrementato di 1 per conteggiare anche il vertice u. Pertanto, una soluzione al problema dato è la seguente:

```
Even–Connected–Component(G)
   ecc := 0
   for ogni vertice u ∈ V do
         color[u] := WHITE
   for ogni vertice u ∈ V do
         if color[u] = WHITE then do
              if DFS–Count–Visit(u) mod 2 = 0 then do
                   ecc := ecc + 1
   return ecc
```

```
DFS–Count–Visit(u)
   color[u] := GRAY
   count := 0
   for ogni v ∈ Adj[u] do
         if color[v] = WHITE then do
              count := count+ DFS–Count–Visit(v)
   color[u] := BLACK
   return count + 1
```

Esercizio 14.

Scrivere un algoritmo che, dato un grafo $G = (V, E)$ non orientato, calcola il numero di vertici di ogni componenti connessa.

Soluzione 1. Come nell'esercizio precedente, per contare quanti sono i vertici in una componente connessa è possibile far ritornare ricorsivamente tale valore all'usuale DFS-Visit, che verrà modificata e rinominata DFS–Count–Visit. Precisamente, dato un vertice u, DFS–Count–Visit(u) ritorna il numero di vertici scoperti a partire da u (compreso u). Per ritornare tale valore, tale procedura incrementa una variabile (*count*), mediante chiamate ricorsive, del numero DFS–Count–Visit(v) di tutti i vertici scopribili a partire da quei vertici $v \in Adj[u]$ che sono immediatamente scopribili da u. La procedura ritornerà quindi il valore di *count* incrementato di 1 per conteggiare anche il vertice u. Pertanto, una soluzione al problema dato è la seguente:

```
DFS(G)
   for ogni vertice u ∈ V do
         color[u] := WHITE
   for ogni vertice u ∈ V do
         if color[u] = WHITE  then do
              nv = DFS–Count–Visit(u)
              # Ho a disposizione il numero di vertici nv...

DFS–Count–Visit(u)
   color[u] := GRAY
   count := 0
   for ogni v ∈ Adj[u] do
         if color[v] = WHITE  then do
              count := count+ DFS–Count–Visit(v)
   color[u] := BLACK
   return count + 1
```

Soluzione 2. Per contare quanti sono i vertici in una componente connessa è possibile utilizzare una variabile globale *nv* inizializzata a 0 e incrementata all'interno della DFS-Visit, che verrà modificata e rinominata DFS–Count–Visit. Precisamente, ogni volta che un nuovo vertice viene scoperto (ossia quanto la DFS–Count–Visit viene invocata), è possibile incrementare tale variabile globale. Una volta ritornati nella funzione DFS, la variabile globale *nv* contiene il numero di vertici della componente connessa appena visitata. Pertanto, una soluzione al problema dato è la seguente:

```
DFS(G)
   for ogni vertice u ∈ V do
         color[u] := WHITE
   for ogni vertice u ∈ V do
         if color[u] = WHITE  then do
              nv = 0
              DFS–Count–Visit(u)
              # Ho a disposizione il numero di vertici nv...

DFS–Count–Visit(u)
   color[u] := GRAY
   nv := nv + 1
   for ogni v ∈ Adj[u] do
         if color[v] = WHITE  then do
```

```
            DFS–COUNT–VISIT(v)
    color[u] := BLACK
```

Soluzione 3. Per contare quanti sono i vertici in una componente connessa è possibile utilizzare la variabile *time*. Subito prima e subito dopo la chiamata DFS-Visit all'interno della DFS è possibile salvarsi il valore corrente della variabile *time* nelle variabili *inizio* e *fine*. In questo modo, una volta terminata la visita della componente connessa è possibile calcolare il numero di vertici come $\frac{fine-inizio}{2}$ (è necessario dividere per due in quanto la variabile *time* viene incrementata due volte per ogni vertice). Pertanto, una soluzione al problema dato è la seguente:

```
DFS(G)
    for ogni vertice u ∈ V do
          color[u] := WHITE
    time := 0
    for ogni vertice u ∈ V do
          if color[u] = WHITE  then do
                inizio = time
                DFS–VISIT(u)
                fine = time
                nv = fine − inizio
                # Ho a disposizione il numero di vertici nv...

PRINT–EDGE–TYPE–VISIT(u)
    color[u] := GRAY
    time := time + 1
    for ogni v ∈ Adj[u] do
          if color[v] = WHITE  then do
                DFS–VISIT(v)
    color[u] := BLACK
    time := time + 1
```

Esercizio 15.

Scrivere un algoritmo che, dato un grafo $G = (V, E)$ non orientato, calcola il numero di archi di ogni componenti connessa.

Soluzione. Per contare quanti sono gli archi in una componente connessa è possibile utilizzare una variabile globale *na* inizializzata a 0 all'interno della DFS e incrementata all'interno della DFS-Visit, che verrà modificata e rinominata DFS–Count–Visit. Precisamente, ogni volta che un nuovo vertice u viene scoperto (ossia quanto la DFS–Count–Visit viene invocata), è possibile incrementare tale variabile globale per ogni arco uscente dal vertice corrente visitato (ossia per ogni vertice v appartenente alla lista di adiacenza di u). Una volta ritornati nella funzione DFS, la variabile globale na contiene il doppio del numero di archi della componente connessa appena visitata (in quanto il grafo è non orientato). Per ottenere il vero numero di archi è necessario dividere tale valore per 2.

Se, oltre al numero di archi, si tiene traccia anche del numero di vertici della componente connessa, è poi possibile stabilire se la componente connessa è un albero ($na = nv - 1$) oppure un grafo completo ($na = nv \cdot (nv - 1)/2$). Pertanto, una soluzione al problema dato è la seguente:

```
DFS(G)
   for ogni vertice u ∈ V do
         color[u] := WHITE
   for ogni vertice u ∈ V do
         if color[u] = WHITE  then do
              nv = 0
              na = 0
              DFS–Count–Visit(u)
              na := na/2
                   if na = nv − 1 then do
                        # questa componente connessa è un albero...
                   if na = nv · (nv − 1)/2 then do
                        # questa componente connessa è un grafo completo...
```

```
DFS–Count–Visit(u)
   color[u] := GRAY
   nv := nv + 1
   for ogni v ∈ Adj[u] do
         na := na + 1
         if color[v] = WHITE  then do
              DFS–Count–Visit(v)
   color[u] := BLACK
```

Partendo da questa soluzione, è poi possibile risolvere problemi come il seguente: scrivere un algoritmo che, dato un grafo $G = (V, E)$ non orientato, stabilisce se, tra le componenti connesse di G, ve ne sono esattamente 5 che sono alberi ed esattamente 3 che prese singolarmente sono grafi completi. Si noti che le soluzioni di questo problema comprendono anche le situazioni in cui il grafo è composto, oltre che da 5 componenti connesse che sono alberi e da 3 componenti connesse che sono (sotto)grafi completi, anche da altre componenti connesse che non sono n alberi n (sotto)grafi completi.