ESERCIZI DI
ALGORITMI E STRUTTURE DATI 2
Parte 3

Livio Colussi
Dipartimento di Matematica Pura ed Applicata
Università di Padova

16 giugno 2006

Esercizio 29 Dimostrare che non esiste nessuna struttura dati S che permetta di eseguire in tempo costante tutte e tre le operazioni MAKE(S), INSERT(S, x) ed EXTRACTMIN(S) (sia caso pessimo che ammortizzato).

Soluzione. Possiamo usare una tale struttura dati per ordinare un array nel seguente modo:

SORT(A, n)
  MAKE(S)
  for $i \leftarrow 1$ **to** n **do** INSERT($S, A[i]$)
  for $i \leftarrow 1$ **to** n **do** $A[i] \leftarrow$ EXTRACTMIN(S)

Se MAKE(S), INSERT(S, x) ed EXTRACTMIN(S) richiedessero tempo costante l'algoritmo $Sort$ richiederebbe tempo $O(n)$. Impossibile perché $O(n \log n)$ è un limite inferiore per l'ordinamento.

Esercizio 30 Sono dati un insieme $p_1, p_2, \ldots, p_n$ di n punti ed un insieme di m connessioni dirette $c_1 = (p_{i_1}, p_{j_1}), c_2 = (p_{i_2}, p_{j_2}), \ldots, c_m = (p_{i_m}, p_{j_m})$. Descrivere un algoritmo efficiente che utilizza una struttura dati per insiemi disgiunti per determinare se tutti i punti sono connessi tra loro.

Soluzione.

CONNESSI(p, n, c, m)
  for $i \leftarrow 1$ **to** n **do** MAKESET(p_i)
  $nsets \leftarrow n$
  for $k \leftarrow 1$ **to** m **do**
    $p \leftarrow$ FINDSET(p_{i_k}) , $q \leftarrow$ FINDSET(p_{j_k})
    if $p \neq q$ **then**
      UNION(p, q), $nsets \leftarrow nsets - 1$
  if $nsets = 1$ **then return** $true$
  else return $false$

Esercizio 31 Sia $T = \{1, 2, \ldots, n\}$ ed $S = \{o_1, o_2, \ldots, o_{n+m}\}$ una sequenza di operazioni contenente n operazioni INSERT(x) che inseriscono ogni intero x in T una e una sola volta ed m operazioni EXTRACTMIN che estraggono m degli interi inseriti. Le operazioni possono essere in un ordine qualsiasi con il solo vincolo

che se o_i è una EXTRACTMIN la sequenza $\{o_1, o_2, \ldots, o_{i-1}\}$ deve contenere più INSERT(x) che EXTRACTMIN. Una possibile sequenza con $n = 9$ ed $m = 6$ è:

$$S = \{4, 8, 0, 3, 0, 9, 2, 6, 0, 0, 0, 1, 7, 0, 5\}$$

in cui le operazioni INSERT(x) sono indicate con il numero $x \in T$ che viene inserito mentre le operazioni EXTRACTMIN sono indicate con uno 0. Si chiede un algoritmo efficiente che data la sequenza di operazioni $S = \{o_1, o_2, \ldots, o_{n+m}\}$ calcoli la sequenza $X = \{x_1, x_2, \ldots, x_m\}$ degli m interi estratti. Ad esempio, con la sequenza

$$S = \{4, 8, 0, 3, 0, 9, 2, 6, 0, 0, 0, 1, 7, 0, 5\}$$

il risultato dovrebbe essere

$$X = \{4, 3, 2, 6, 8, 1\}$$

Soluzione. Raggruppiamo in $m + 1$ insiemi disgiunti $I_1, I_2, \ldots, I_m, I_{m+1}$ gli interi $1, 2, \ldots, n$ inseriti mettendo in I_1 gli interi inseriti prima della prima EXTRACTMIN, in I_{m+1} quelli inseriti dopo l'ultima EXTRACTMIN e, per $2 \leq i \leq m$, mettendo in I_i quelli inseriti tra la $(i-1)$-esima e la i-esima EXTRACTMIN. Siccome alcuni degli insiemi possono essere vuoti aggiungiamo a ciascun insieme I_i un elemento fittizio di valore $n + i$. Inoltre, nel rappresentante di ciascun insieme memorizziamo l'indice dell'insieme stesso.

MINIMOOFFLINE(S, X, m, n)
/* Precondizione: S è l'array di interi che rappresenta la sequenza di n INSERT(x) (indicate con l'intero x inserito) e di m EXTRACTMIN (indicate con l'intero 0). */
"raggruppa gli interi inseriti e quelli fittizi negli insiemi $I_1, I_2, \ldots, I_{m+1}$"
for $i \leftarrow 1$ **to** n **do**
"trova l'indice j dell'insieme I_j che contiene i"
if $j \leq m$ **then**
$X_j \leftarrow i$
"trova il primo $h > j$ con X_h non ancora calcolato"
$I_h \leftarrow I_j \cup I_h$
/* Postcondizione: $X_1, \ldots, X_m$ sono gli interi estratti. */

Dettagli implementativi:

Gli interi da raggruppare (quelli inseriti e quelli fittizzi) sono gli $m + n + 1$ interi $1, 2, \ldots, n, n + 1, \ldots, n + m + 1$.

Per ragrupparli negi insiemi disgiunti $I_1, I_2, \ldots, I_{m+1}$ useremo le foreste di insiemi disgiunti.

Per ogni intero $1, 2, \ldots, n, n+1, \ldots, n+m+1$ dobbiamo quindi prevedere i campi $rank$ e p ed un campo $indice$ in cui memorizzare l'indice dell'insieme di appartenenza (aggiungiamo tale campo a tutti ma ci preoccuperemo di mantenere aggiornato soltanto quello del rappresentante).

Come struttura dati di supporto per memorizzare gli interi da raggruppare usiamo tre array $rank$, p e $indice$ di $m+n+1$ elementi ciascuno. Per ogni intero $x = 1, \ldots, m+n+1$, $rank[x]$, $p[x]$ e $indice[x]$ sono i valori dei campi $rank$, p e $indice$ relativi a tale intero x.

Correttezza dell'algoritmo:

MINIMOOFFLINE(S, X, m, n)

/* Precondizione: S è l'array di interi che rappresenta la sequenza di n INSERT(x) (indicate con l'intero x inserito) e di m EXTRACTMIN (indicate con l'intero 0). */

"raggruppa le INSERT(x) in insiemi $I_1, I_2, \ldots, I_{m+1}$"

for $i \leftarrow 1$ **to** n **do**

/* Ogni $x < i$ estratto è stato memorizzato al posto giusto nell'array X. Se $i \leq x \leq n$ e $x \in I_k$ allora:

a) x è stato inserito prima della k-esima EXTRACTMIN,

b) x non può essere estratto da una EXTRACTMIN che precede la k-esima e

c) la k-esima EXTRACTMIN estrae un intero $y \geq i$.

*/

"trova l'indice j dell'insieme I_j che contiene i"

/* Ogni $x < i$ estratto è stato memorizzato al posto giusto nell'array X. Se $i \leq x \leq n$ e $x \in I_k$ allora:

a) x è stato inserito prima della k-esima EXTRACTMIN,

b) x non può essere estratto da una EXTRACTMIN che precede la k-esima,

c) la k-esima EXTRACTMIN estrae un intero $y \geq i$ e

d) $i \in I_j$

*/

/* Se $j = m+1$ allora i non può essere estratto e quindi è vera la postcondizione di for. */

if $j \leq m$ **then**

/* Se $j \leq m$ allora i è stato inserito prima di e_j ed è il minimo elemento inserito prima della j-esima EXTRACTMIN e che possa essere estratto da tale operazione. Quindi la j-esima EXTRACTMIN estrae i. */

$X_j \leftarrow i$

/* Ogni $x < i + 1$ estratto è stato memorizzato al posto giusto nell'array X. Se $i \leq x \leq n$ e $x \in I_k$ allora:
a) x è stato inserito prima della k-esima EXTRACTMIN,
b) x non può essere estratto da una EXTRACTMIN che precede la k-esima,
c) la k-esima EXTRACTMIN estrae un intero $y \geq i$ e
d) $i \in I_j$ e $j \leq m$.
*/

"trova il primo $h > j$ con X_h non ancora calcolato"
$I_h \leftarrow I_j \cup I_h$

/* Ogni $x < i + 1$ estratto è stato memorizzato al posto giusto nell'array X. Se $i + 1 \leq x \leq n$ e $x \in I_k$ allora:
a) x è stato inserito prima della k-esima EXTRACTMIN,
b) x non può essere estratto da una EXTRACTMIN che precede la k-esima e
c) la k-esima EXTRACTMIN estrae un intero $y \geq i + 1$.
*/

/* Postcondizione: $X_1, \ldots, X_m$ sono gli interi estratti. */

Esercizio 32 Il minimo antenato comune di due nodi u e v in un albero T è il nodo w di massima profondità che è antenato sia di u che di v. Nel problema dei minimi comuni antenati off-line sono dati un albero T con n nodi ed una lista

$$L = \{(u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m)\}$$

di m coppie di nodi. Si chiede di calcolare la lista

$$W = \{w_1, w_2, \ldots, w_m\}$$

dei minimi comuni antenati di ogni coppia.

Esiste un algoritmo, dovuto a Robert E. Tarjan, che risolve il problema in tempo (quasi) lineare con una sola percorrenza dell'albero. Assumiamo che ogni nodo dell'albero abbia i seguenti campi:

- $child$ puntatore al primo figlio;

- $sibling$ puntatore al fratello;

- $ancestor$ puntatore ad un suo antenato nell'albero;

- $color$ che all'inizio è $white$ per tutti i nodi e alla fine è $black$.

oltre ai campi necessari per raggruppare i nodi dell'albero in foreste di insiemi disgiunti:

- $rank$ rango del nodo nella foresta di insiemi disgiunti;

- p puntatore al padre in una foresta di insiemi disgiunti (non al padre nell'albero T).

L'algoritmo di Tarjan è descritto dalla seguente funzione ricorsiva che deve essere chiamata sulla radice dell'albero.

MINCOMANT(u)
  MAKESET(u), $ancestor[u] \leftarrow u$
  $v \leftarrow child[u]$
  while $v \neq nil$ **do**
    MINCOMANT(v)
    UNION(u, v), $ancestor[\text{FINDSET}(u)] \leftarrow u$
    $v \leftarrow sibling[v]$
  $color[u] \leftarrow black$
  for "ogni $(u, v_i) \in L$" **do**
    if $color[v_i] = black$ **then**
      $w_i \leftarrow ancestor[\text{FINDSET}(v_i)]$

dimostrare che esso è corretto e valutarne la complessità.

Soluzione.

Correttezza. Consideriamo la generica chiamata MINCOMANT(u) e sia $r = x_0, \ldots, x_k = u$ il cammino dalla radice r dell'albero T al nodo u su cui viene richiamata la funzione $MinComAnt$.

Sia $Inv(u)$ la seguente asserzione sugli antenati $x_0, \ldots, x_{k-1}$ di u:

"Per ogni $j = 0, \ldots, k-1$ il nodo x_j e tutti i nodi che stanno nei sottoalberi radicati nei figli di x_j che precedono il figlio x_{j+1} costituiscono un insieme disgiunto S_j il cui rappresentante ha il puntatore $ancestor$ che punta ad x_j. Inoltre $color[z] = black$ per tutti i nodi $z \in S_j$ escluso il nodo x_j per il quale $color[x_j] = white$."

Sia $Pre(u)$ la seguente asserzione:

"Per ogni coppia (u_i, v_i) in L tale che $u_i, v_i \in \cup_{j=1}^{k-1} S_j$ e $color[u_i] = color[v_i] = black$ è stato assegnato correttamente a w_i il puntatore al minimo comune antenato di u_i e v_i.

Inoltre $color[z] = white$ per ogni $z \notin \cup_{j=1}^{k-1} S_j$."

e $Post(u)$ l'asserzione:

"Il nodo $x_k = u$ e tutti i suoi discendenti costituiscono un insieme disgiunto S_k il cui rappresentante ha il puntatore $ancestor$ che punta ad x_k e $color[z] = black$ per tutti i nodi $z \in S_k$ compreso il nodo $x_k = u$.

Per ogni coppia (u_i, v_i) in L tale che $u_i, v_i \in \cup_{j=1}^{k} S_j$ e $color[u_i] = color[v_i] = black$ è stato assegnato correttamente a w_i il puntatore al minimo comune antenato di u_i e v_i.

Inoltre $color[z] = white$ per ogni $z \notin \cup_{j=1}^{k} S_j$."

Quando viene effettuata la chiamata principale MINCOMANT(r) sulla radice dell'albero la precondizione $Inv(r)$ **and** $Pre(r)$ è banalmente vera. Se quando termina la chiamata principale MINCOMANT(r) risulta vera la postcondizione $Inv(r)$ **and** $Post(r)$ allora tutti i nodi hanno $color = black$ e quindi per ogni coppia in L è stato assegnato correttamente a w_i il puntatore al minimo comune antenato di u_i e v_i.

Dobbiamo dimostrare che se è vera $Inv(u)$ **and** $Pre(u)$ quando viene effettuata una generica chiamata MINCOMANT(u) allora è vera $Inv(u)$ **and** $Post(u)$ quando tale chiamata termina. Per fare questa dimostrazione possiamo assumere induttivamente che le chiamate ricorsive soddisfino la medesima condizione.

Il seguente è la funzione $MinComAnt$ annotata con le asserzioni che servono per tale verifica.

MINCOMANT(u)
/* $Inv(u)$ **and** $Pre(u)$ */
MAKESET(u), $ancestor[u] \leftarrow u$

$v \leftarrow child[u]$

while $v \neq nil$ **do**

/* $Inv(v)$ **and** $Pre(v)$ */

MINCOMANT(v)

/* $Inv(v)$ **and** $Post(v)$ */

UNION(u, v), $ancestor[\text{FINDSET}(u)] \leftarrow u$

$v \leftarrow sibling[v]$

/* $Inv(u)$.
Il nodo $x_k = u$ e tutti i suoi discendenti costituiscono un insieme disgiunto S_k il cui rappresentante ha il puntatore $ancestor$ che punta ad x_k e $color[z] = black$ per tutti i nodi $z \in S_k$ escluso il nodo $x_k = u$ per cui $color[u] = white$.
Per ogni coppia (u_i, v_i) in L tale che $u_i, v_i \in \cup_{j=1}^{k} S_j$ e $color[u_i] = color[v_i] = black$ è stato assegnato correttamente a w_i il puntatore al minimo comune antenato di u_i e v_i.
$color[z] = white$ per ogni $z \notin \cup_{j=1}^{k} S_j$. */

$color[u] \leftarrow black$

for "ogni $(u, v_i) \in L$" **do**

if $color[v_i] = black$ **then**

$w_i \leftarrow ancestor[\text{FINDSET}(v_i)]$

/* $Inv(u)$ **and** $Post(u)$ */

Complessità. Possiamo assumere che le operazioni sugli insiemi disgiunti richiedano tempo costante.

Sia n il numero di nodi di T ed m il numero di coppie in L.

La funzione MINCOMANT(u) viene richiamata una e una sola volta su ogni vertice u dell'albero. Infatti essa viene richiamata soltanto su vertici di colore $white$, prima di terminare li colora $black$ e alla fine di tutto l'algoritmo i nodi sono tutti di colore $black$. Quindi il numero totale di chiamate è n e dunque le istruzioni interne al ciclo while vengono eseguite al più n volte.

Supponiamo che la lista L sia suddivisa in liste delle adiacenze L_u contenenti tutti i v_i tali che $(u, v_i) \in L$. In caso contrario lo possiamo fare in tempo $O(m)$.

Il ciclo for esplora ciascuna lista L_u al più una sola volta. Siccome la somma delle lunghezze delle liste L_u è $2m$ l'esecuzione di tutti i cicli for di tutte le chiamate richiede tempo $O(m)$.

Pertanto l'intero algoritmo richiede tempo $O(m + n)$.

Esercizio 33 Sia $G = (V, E)$ un grafo orientato con $n = |V|$ vertici ed $m = |E|$ archi e supponiamo che puntatori e interi richiedano 32 bit. Dire per quali valori

di n ed m le liste delle adiacenze richiedono meno memoria della matrice delle adiacenze.

Soluzione. La memoria richiesta con le liste delle adiacenze è $32(n + 2m)$ mentre quella richiesta con la matrice delle adiacenze è n^2. Le liste delle adiacenze richiedono quindi meno memoria quando $32(n + 2m) < n^2$ ossia quando

$$m < \frac{n(n - 32)}{64}$$

Ecco, in funzione di n, i valori massimi di m per cui sono convenienti le liste:

n	m
1-32	nessuno
33	0
34	1
35	1
40	5
50	14
100	106
200	525
500	3656
1000	15125
$n \to \infty$	$n^2/64$

Esercizio 34 Il trasposto di un grafo orientato $G = (V, E)$ è il grafo $G^T = (V, E^T)$ in cui $E^T = \{uv : vu \in E\}$. Descrivere due algoritmi efficienti per calcolare G^T usando rispettivamente la rappresentazione con liste delle adiacenze e la rappresentazione con matrice delle adiacenze.

Soluzione.

TRASPONIMATR(A, n)
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow i + 1$ **to** n **do**
      $t \leftarrow A[i, j]$, $A[i, j] \leftarrow A[j, i]$, $A[j, i] \leftarrow t$

TRASPONIADJ(Adj, n)
  for $i \leftarrow 1$ **to** n **do** $Adj1[i] \leftarrow nil$
  for $i \leftarrow 1$ **to** n **do**

while $Adj[i] \neq nil$ **do**
$j \leftarrow$ POP$(Adj[i])$, PUSH$(Adj1[j], i)$
for $i \leftarrow 1$ **to** n **do** $Adj[i] \leftarrow Adj1[i]$

Esercizio 35 Con la matrice delle adiacenze molti algoritmi richiedono tempo $O(n^2)$. Ci sono però alcune eccezioni quali il seguente problema della celebrità:

> "Dato un grafo orientato $G = (V, E)$ con n vertici determinare se esiste un vertice avente grado entrante $n - 1$ (conosciuto da tutti) e grado uscente 0 (non conosce nessuno)."

Trovare un algoritmo che risolve il problema in tempo $O(n)$ usando la matrice delle adiacenze.

Soluzione.

CELEBRITA(M, n)
$j \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
/* Nessun k minore di i e diverso da j è una celebrità */
if $M_{i,j} = 0$ **then**
/* i conosce j e quindi i non è una celebrità. Passo al successivo. */
else
/* i non conosce j e quindi j non è una celebrità ma i può esserlo. */
$j \leftarrow i$
/* j è l'unico che può essere una celebrità. Effettuo il controllo. */
for $i \leftarrow 1$ **to** n **do**
if $i \neq j$ **and** $(M_{i,j} = 0$ **or** $M_{j,i} = 1)$ **then return** nil
return j

Esercizio 36 Aggiungere alla ricerca in profondità in un grafo orientato la stampa degli archi incontrati con la loro classificazione.

Soluzione.

CLASSIFICAARCHI(G)

```
for "Ogni u ∈ V(G)" do color[u] ← bianco
t ← 1
for "Ogni u ∈ V(G)" do
    if color[u] = bianco then CLASSIFICAARCHIRIC(u, t)
```

```
CLASSIFICAARCHIRIC(u, t)
color[u] ← grigio , d[u] ← t , t ← t + 1
for "Ogni v ∈ Adj[u]" do
    if color[v] = bianco then
        "Stampa uv arco d'albero."
        p[v] ← u , CLASSIFICAARCHIRIC(v)
    else
        if color[v] = grigio and v ≠ p[u] then
            "Stampa uv arco all'indietro."
        else
            if d[v] < d[u] then
                "Stampa uv arco trasversale."
            else
                "Stampa uv arco in avanti."
color[u] ← nero , f[u] ← t , t ← t + 1
```

Esercizio 37 Mostrare che, se il grafo su cui si effettua la ricerca in profondità è non orientato, vi sono soltanto archi d'albero ed archi all'indietro.

Soluzione. Supponiamo esista un arco uv in avanti. Siccome il grafo è non orientato l'arco uv coincide con l'arco vu che è un arco all'indietro. Siccome un arco viene classificato nella prima delle categorie a cui appartiene uv è un arco all'indietro e non un arco in avanti.

Supponiamo esista un arco uv trasversale. Allora v deve essere scoperto e finito prima che u sia scoperto, ma questo non è possibile perché esiste l'arco vu e quindi u viene scoperto prima che v sia finito.

Esercizio 38 Scrivere un algoritmo che dato un grafo non orientato $G = (V, E)$ connesso trova un cammino in G che attraversa tutti gli archi una e una sola volta in ognuna delle due direzioni. L'algoritmo deve avere complessità $O(n+m)$. Dire come, avendo a disposizione un numero sufficiente di monetine, si possa usare tale algoritmo per cercare l'uscita in un labirinto.

Soluzione. In una ricerca in profondità vengono attraversati tutti gli archi d'albero in entrambe le direzioni. Siccome, oltre agli archi d'albero ci sono soltanto archi all'indietro basta aggiugere l'attraversamento andata e ritorno di tutti gli archi all'indietro incontrati per ottenere il cammino cercato.

Possiamo utilizzare l'algoritmo precedente per percorrere tutti i corridoi di un labirinto fino a che non troviamo una uscita oppure abbiamo percorso tutti i corridoi senza trovarla.

Avendo a disposizione un numero sufficiente di monetine poniamo una monetina all'inizio e due monetine alla fine di ogni corridoio percorso.

Partendo da un incrocio qualsiasi scegliamo un corridoio senza monetina all'inizio, poniamo una monetina al suo inizio, lo percorriamo e, se in esso non troviamo l'uscita del labirinto, poniamo due monetine alla fine e ripetiamo l'operazione partendo dall'incrocio successivo.

Se ci troviamo in un incrocio in cui nessun corridoio è privo di monetine scegliamo il corridoio con due monetine (quello da cui siamo arrivati), togliamo una delle due monetine e ripercorriamo il corridoio lasciando la monetina che avevamo posto all'inizio e ripetiamo l'operazione partendo dall'incrocio in cui siamo tornati.

Se arriviamo ad un incrocio in cui tutti i corridoi hanno esattamente una monetina abbiamo percorso tutto il labirinto e ci fermiamo (disperati perché il labirinto non ha uscite).

Esercizio 39 Mostrare con un controesempio che se vi è un cammino da u a v in un grafo orientato e u viene scoperto prima di v non necessariamente v è discendente di u nella foresta di ricerca in profondità.

Soluzione.

Sia G il grafo orientato di vertici u, v, w e archi uv, vu, vw e supponiamo che la ricerca in profondità parta dal vertice v e, tra i vertici adiacenti a v visiti prima il vertice u e poi il vertice w. L'albero di ricerca in profondità ha radice v con figli u e w. Dunque w non è discendente di u anche se u viene scoperto prima di w e vi è un cammino da u a w in G. Notare che, quando viene scoperto u il cammino da u a w non è bianco.

Esercizio 40 Mostrare come un vertice u di un grafo orientato possa essere l'unico vertice di un albero della foresta di ricerca in profondità pur avendo sia archi entranti che archi uscenti.

Soluzione.

Sia G il grafo orientato di vertici u, v, w e archi uv, vw.

Supponiamo che la ricerca in profondità parta dal vertice w costruendo un albero della foresta di ricerca in profondità costituito soltanto dal vertice w.

Supponiamo inoltre che il secondo vertice da cui parte la ricerca in profondità sia v. Viene allora costruito un albero della foresta di ricerca in profondità costituito soltanto dal vertice v e questo benché v abbia sia l'arco entrante uv che l'arco uscente vw.

Esercizio 41 Mostrare che se in un grafo non orientato G esiste l'arco uv allora i due vertici u e v stanno in uno stesso albero della foresta di ricerca in profondità.

Soluzione. Abbiamo visto che in un grafo non orientato ci sono soltanto archi d'albero ed archi all'indietro. In entrambi i casi i due vertici u e v sono uno discendente dell'altro e quindi stanno nello stesso albero.

Esercizio 42 Scrivere un algoritmo che determina se un grafo $G = (V, E)$ non orientato è aciclico in tempo $O(n)$ indipendentemente dal numero m di archi del grafo.

Soluzione.

```
CICLI(G)
    for "Ogni u ∈ V(G)" do color[u] ← bianco
    trovato ← false
    for "Ogni u ∈ V(G)" do
        if color[u] = bianco then CICLIRIC(u, trovato)
        if trovato then break
    if not trovato then
        "Stampa: Il grafo non contiene cicli."


CICLIRIC(u, trovato)
    color[u] ← grigio
    for "Ogni v ∈ Adj[u]" do
        if color[v] = bianco then
            CICLIRIC(v, trovato)
            if trovato then break
        else
            /* L'arco uv è un arco all'indietro che forma un ciclo. */
```

"Stampa: Il grafo contiene un ciclo."
$trovato \leftarrow true$, **break**

$color[u] \leftarrow nero$

L'algoritmo ha complessità proporzionale al numero di archi visitati. Gli archi visitati sono soltanto gli archi d'albero ed al più un solo arco all'indietro (l'algoritmo termina non appena trova un arco all'indietro). Gli archi d'albero possono essere al più $n - 1$ e quindi l'algoritmo ha complessità $O(n)$.

Esercizio 43 Come si modifica il numero di componenti fortemente connesse aggiungendo un arco? Trovare un esempio in cui il numero di componenti fortemente connesse non cambia, un esempio in cui il numero di componenti fortemente connesse diminuisce di 1 ed un esempio in cui il numero di componenti fortemente connesse da 10 diventa 1.

Soluzione.

L'aggiunta di un arco riunisce due componenti fortemente connesse C_1 e C_2 se prima dell'aggiunta dell'arco vi era un cammino da C_2 a C_1 e l'aggiunta dell'arco crea un cammino da C_1 a C_2.

Il numero di componenti fortemente connesse non cambia se l'arco appartiene alla stessa componente fortemente connessa oppure a due componenti fortemente connesse C_1 e C_2 per le quali non esiste né un cammino da C_1 a C_2 né un cammino da C_2 a C_1.

Se le componenti fortemente connesse $C_1, C_2, \ldots, C_{10}$ sono connesse da un cammino che parte da C_1 ed arriva a C_{10} passando per tutte le componenti intermedie $C_2, \ldots, C_9$ e l'arco aggiunto crea un cammino da C_{10} a C_1 tutte e 10 le componenti vengono riunite in un'unica componente fortemente connessa.

Esercizio 44 Un grafo orientato $G = (V, E)$ è semiconnesso se per ogni due vertici u e v esiste o un cammino da u a v oppure un cammino da v a u. Trovare un algoritmo efficiente per verificare se un grafo è semiconnesso. Suggerimento: Ordinare topologicamente il grafo delle componenti fortemente connesse.

Soluzione.

L'algoritmo consiste nei seguenti passi:

1. Calcola le componenti fortemente connesse del grafo G. (Tempo richiesto $O(n + m)$.)

2. Costruisci il grafo orientato D delle componenti fortemente connesse. Tale grafo è aciclico, ossia un DAG. (Tempo richiesto $O(n + m)$.)

3. Ordina topologicamente il grafo diretto aciclico D delle componenti fortemente connesse. Sia $C_1, C_2, \ldots, C_k$ l'ordine ottenuto. (Tempo richiesto $O(n+m)$.)

4. Controlla se per ogni $i = 1, \ldots, k-1$ vi è un arco da C_i a C_{i+1} in D. Se è così il grafo G è semiconnesso, altrimenti non lo è. (Tempo richiesto $O(n+m)$.)

Complessità: $O(n+m)$.

Correttezza: Supponiamo che per ogni $i = 1, \ldots, k-1$ vi sia un arco da C_i a C_{i+1} in D e siano $u, v \in V$ due vertici qualsiasi. Uno dei due vertici, diciamo u, appartiene ad una componente fortemente connessa C_i che precede la componente fortemente connessa C_j che contiene l'altro vertice v. Allora nel grafo G vi è un cammino da u a v.

Viceversa, assumiamo che non ci sia un arco tra due componenti fortemente connesse consecutive C_i e C_{i+1} e siano u un vertice in C_i e v un vertice in C_{i+1}.

Un cammino da v ad u dovrebbe contenere un arco da una componente fortemente connessa ad una precedente. Quindi non ci sono cammini da v ad u.

Un cammino da u a v dovrebbe contenere un arco da C_i ad una componente fortemente connessa precedente oppure un arco che arriva in C_{i+1} da una componente fortemente connessa successiva a C_{i+1}. Quindi non ci sono neppure cammini da u a v e il grafo non è semiconnesso.

Esercizio 45 Supponiamo che il grafo G abbia più alberi di connessione minimi. Mostrare che due alberi di connessione minimi T e T' non solo hanno lo stesso costo totale ma anche i costi dei singoli archi sono gli stessi. In altre parole se $w_1 \leq w_2 \leq \ldots \leq w_n$ è la lista ordinata dei costi degli archi di T e $w'_1 \leq w'_2 \leq \ldots \leq w'_n$ è la lista ordinata dei costi degli archi di T' allora $w_i = w'_i$ per ogni i.

Soluzione. Siano $a_1, a_2, \ldots, a_n$ gli archi di T con costi $w_1 \leq w_2 \leq \ldots \leq w_n$ e $a'_1, a'_2, \ldots, a'_n$ gli archi di T' con costi $w'_1 \leq w'_2 \leq \ldots \leq w'_n$.

Se $a_i = a'_i$ per ogni i la cosa è ovvia.

Altrimenti sia i il primo indice per cui $a_i \neq a'_i$.

Mostriamo intanto che $w'_i = w_i$.

Supponiamo infatti $w'_i > w_i$. L'arco a_i non appartiene a T' in quanto gli archi di T' di indice minore di i sono uguali ai corrispondenti archi di T e gli archi di indice maggiore o uguale ad i hanno peso maggiore del peso di a_i.

Se aggiungiamo l'arco a_i a T' si forma un ciclo che contiene sicuramente un arco a'_j che non appartiene a T. w'_j non può essere maggiore di w_i altrimenti sostituendo a'_j con a_i in T' si otterrebbe un albero di connessione di costo inferiore.

Ma tutti gli archi in T' di costo minore o uguale di w_i sono in comune con T. Assurdo.

Il caso $w'_i < w_i$ è simmetrico. Quindi $w'_i = w_i$.

Facciamo vedere che possiamo trovare una sequenza $a''_1, a''_2, \ldots, a''_n$ di archi che formano un albero di connessione minimo avente la stessa sequenza di costi di T' ma avente in comune con la sequenza $a_1, a_2, \ldots, a_n$ degli archi di T almeno i primi $i+1$ archi.

Distinguiamo i seguenti casi:

1. $w_i = w'_i$ e $a_i = a'_j$ è un arco di T' (ovviamente con indice $j > i$). In questo caso scambiando a'_i con a'_j nella sequenza ordinata $a'_1, a'_2, \ldots, a'_n$ otteniamo un riordino $a''_1, a''_2, \ldots, a''_n$ degli archi di T' a cui corrisponde la stessa sequenza di costi. Inoltre i primi $i+1$ archi di $a''_1, a''_2, \ldots, a''_n$ coincidono con i primi $i+1$ archi di $a_1, a_2, \ldots, a_n$.

2. $w_i = w'_i$ e a_i non è in T'. In questo caso aggiungendo l'arco a_i a T' si forma un ciclo che contiene sicuramente un arco a'_j che non appartiene a T. w'_j non può essere minore di w_i perchè tutti gli archi di costo minore sono in comune e non può essere maggiore altrimenti sostituendo a'_j con a_i in T' si otterrebbe un albero di costo inferiore.

   Quindi $w'_j = w_i = w'_i$. Se nella sequenza $a'_1, a'_2, \ldots, a'_n$ scambiamo di posto a'_i con a'_j e sostituiamo l'arco a'_j con a_i otteniamo la sequenza $a''_1, a''_2, \ldots, a''_n$ degli archi di un albero di connessione minimo T'' a cui corrisponde la stessa sequenza di costi $w'_1 \leq w'_2 \leq \ldots \leq w'_n$. Inoltre i primi $i+1$ archi di $a''_1, a''_2, \ldots, a''_n$ coincidono con i primi $i+1$ archi di $a_1, a_2, \ldots, a_n$.

Esercizio 46 Supponiamo che i costi degli archi siano interi compresi tra 1 ed $n = |V|$. Come si può usare questo fatto per diminuire la complessità dell'algoritmo di Kruskal?

Soluzione.

KRUSKAL(G)
/* G grafo pesato sugli archi */
$A \leftarrow \emptyset$
for "ogni $u \in V[G]$" **do**
    MAKESET(u)
"ordina gli archi in ordine di costo non decrescente"
/* in questo caso si può fare in tempo $O(n)$ */

for "ogni $a = (u, v) \in A[G]$ in ordine di costo" **do**
  if FINDSET$(u) \neq$ FINDSET(v) **then**
    UNION(u, v), $A \leftarrow A \cup \{a\}$
return A

Esercizio 47 Sia $G = (V, E)$ un grafo orientato i cui archi rappresentano dei canali di trasmissione. Ad ogni arco (u, v) è associato un coefficiente di affidabilità $R(u, v)$ che è un valore compreso tra 0 ed 1 ($0 < R(u, v) \leq 1$). $R(u, v)$ rappresenta la probabilità che un messaggio spedito da u lungo il canale (u, v) venga ricevuto correttamente da v. Scrivere un algoritmo che calcola i cammini più affidabili da un vertice s ad ogni altro vertice.

Soluzione.

L'affidabilità di un cammino è il prodotto delle affidabilità degli archi che lo compongono.

L'affidabilità di un cammino è massima se il suo logaritmo è massimo e il logaritmo dell'affidabilità di un cammino è la somma dei logaritmi delle affidabilità degli archi che lo compongono.

Definiamo $W(u, v) = -\log(R(u, v))$ come peso di un arco. Notiamo che $W(u, v) > 0$ per ogni arco uv. L'affidabilità di un cammino è massima se la somma dei pesi dei suoi archi è minima. Siccome $W(u, v) > 0$ per ogni arco uv possiamo utilizzare l'algoritmo di Dijkstra per trovare i cammini minimi da s rispetto ai pesi $W(u, v)$. Tali cammini minimi sono anche cammini di massima affidabilità.

Esercizio 48 Modificare l'algoritmo di Bellman e Ford:

BELLMANFORD(G, s)
  /* G grafo pesato sugli archi */
  INIZIALIZZA(G, s)
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    for "ogni $uv \in E[G]$" **do**
      RILASSA(G, u, v)
  for "ogni $uv \in E[G]$" **do**
    if $d[v] > d[u] + w(uv)$ **then**
      return $false$
  return $true$

in modo che esso ponga $d[v] = -\infty$ per ogni vertice v che è raggiungibile da s con un cammino che contiene un ciclo negativo.

Soluzione.

BELLMANFORD(G, s)
    /* G grafo pesato sugli archi */
    INIZIALIZZA(G, s)
    for $i \leftarrow 1$ **to** $n - 1$ **do**
        for "ogni $uv \in E[G]$" **do**
            RILASSA(G, u, v)
    for $i \leftarrow 1$ **to** $n - 1$ **do**
        for "ogni $uv \in E[G]$" **do**
            if $d[v] > d[u] + w(uv)$ **then**
                /* Il cammino da s a v contiene un ciclo negativo. */
                $d[v] \leftarrow -\infty$

Esercizio 49 Descrivere un algoritmo che dato un grafo orientato aciclico pesato sugli archi ed un vertice s calcola i cammini massimi da s ad ogni altro vertice.

Soluzione.

Basta cercare i cammini minimi da s ad ogni altro vertice rispetto ai pesi $w'(uv) = -w(uv)$.

Esercizio 50 Sono date un'insieme di task $a_1, \ldots, a_n$ con tempi di esecuzione $t_1, \ldots, t_n$. Tali task si possono eseguire in parallelo utilizzando un numero sufficiente di processori. L'esecuzione di tali task deve però rispettare dei vincoli di propedeuticità rappresentati mediante coppie (a_i, a_j) il cui significato è

"a_i deve essere finita prima di iniziare l'esecuzione di a_j."

Descrivere un algoritmo efficiente che calcola il tempo minimo necessario per eseguire tutte le attività.

Soluzione.

Sia $G = (V, E)$ il grafo diretto avente le attività come vertici e come archi le coppie (a_i, a_j) relative alle propedeuticità.

Il grafo $G = (V, E)$ deve essere chiaramente aciclico (altrimenti il processo si blocca).

L'algoritmo è il seguente:

TEMPOESECUZIONE(G, t)

/* G grafo pesato sui vertici */

"ordina topologicamente le attività."

/* $a_1, \ldots, a_n$ sono in ordine topologico */

for $i \leftarrow 1$ **to** n **do** $f_i \leftarrow t_i$

for $i \leftarrow 2$ **to** n **do**

  for $j \leftarrow 1$ **to** $i-1$ **do**

    if "esiste l'arco (a_j, a_i)" **and** $f_i < f_j + a_i$ **then**

      $f_i \leftarrow f_j + a_i$

$TempoEs \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

  if $TempoEs < f_i$ **then** $TempoEs \leftarrow f_i$

Esercizio 51 Una scatola rettangolare a d dimensioni è definita dalle dimensioni $(x_1, x_2, \ldots, x_d)$ dei suoi lati. La scatola X di lati $(x_1, x_2, \ldots, x_d)$ puo essere racchiusa nella scatola Y di lati $(y_1, y_2, \ldots, y_d)$ se esiste una permutazione π degli indici $(1, 2, \ldots, d)$ tale che

$$x_{\pi(1)} \leq y_1, x_{\pi(2)} \leq y_2, \ldots, x_{\pi(d)} \leq y_d$$

1. Verificare che la relazione "essere racchiudibile" è transitiva.

2. Trovare un metodo efficiente per testare se una scatola X è racchiudibile in un'altra scatola Y.

3. Trovare un algoritmo efficiente che dato un insieme $(S_1, S_2, \ldots, S_n)$ di n scatole di dimensione d, trova la più lunga sequenza $(S_{i_1}, S_{i_2}, \ldots, S_{i_k})$ di scatole racchiudibili una nell'altra.

Soluzione.

1. Supponiamo X racchiudibile in Y e Y racchiudibile in Z. Sia π la permutazione tale che $x_{\pi(i)} \leq y_i$ per ogni i e π' la permutazione tale che $y_{\pi'(i)} \leq z_i$ per ogni i. Sia π'' la permutazione prodotto $\pi''(i) = \pi'(\pi(i))$.

   Allora $x_{\pi''(i)} \leq z_i$ per ogni i e dunque X è racchiudibile in Z.

2. Ordinare i lati $(x_1, x_2, \ldots, x_d)$ e $(y_1, y_2, \ldots, y_d)$ delle due scatole in ordine di lunghezza decrescente. Se $x_i \leq y_i$ per ogni i la scatola X è racchiudibile nella scatola Y, altrimenti non lo è. Tempo richiesto $O(d \log d)$.

3. Ordinare i lati di ogni scatola per lunghezza decrescente ed ordinare quindi le scatole in ordine lessicografico rispetto alle sequenze delle lunghezze dei lati. Tempo richiesto $O(nd \log d + dn \log n) = O(dn(\log n + \log d)$.

   Costruire il grafo G avente le scatole $(S_1, S_2, \ldots, S_n)$ come vertici ed un arco da S_i ad S_j se la scatola S_i è racchiudibile nella scatola S_j. Tale grafo è chiaramente aciclico. Tempo richiesto $O(dn^2)$.

   Ordinare topologicamente il grafo G. Tempo richiesto $O(n^2)$.

   Sui vertici $(S_1, S_2, \ldots, S_n)$ presi in ordine topologico eseguire la seguente procedura:

   SEQMAX(G)
   /* i vertici $S_1, \ldots, S_n$ di G sono in ordine topologico */
   for $i \leftarrow 1$ **to** n **do**
   $l[i] \leftarrow 0$, $p[i] \leftarrow nil$
   for $i \leftarrow 2$ **to** n **do**
   for $j \leftarrow 1$ **to** $i - 1$ **do**
   if "esiste l'arco (S_j, S_i)" **and** $l[i] <= l[j]$ **then**
   $l[i] \leftarrow l[j] + 1$, $p[i] \leftarrow j$
   $lmax \leftarrow 0$, $Smax \leftarrow nil$
   for $i \leftarrow 1$ **to** n **do**
   if $lmax < d[i]$ **then**
   $lmax \leftarrow d[i]$, $Smax \leftarrow i$
   /* S_i è l'ultima scatola della sequenza più lunga, $lmax$ è la lunghezza di tale sequenza e le altre scatole della sequenza sono quelle che si incontrano a partire dalla scatola S_i e seguendo i puntatori $p[i]$. */

   Tempo richiesto $O(dn^2)$.

   Sommando tutti i tempi otteniamo una complessità pari a $O(d(n^2 + n \log d))$. Se assumiamo $\log d = O(n)$ allora il tempo totale è $O(dn^2)$.

Esercizio 52 Normalmente in una rete di flusso $G = (V, E)$ si considerano soltanto gli archi $uv \in E$ aventi capacità $c(uv)$ non nulla. Nondimeno, quando si voglia calcolare la rete residua $G_f = (V, E_f)$ rispetto ad un flusso $f(uv)$ possono comparire in E_f degli archi vu opposti di archi $uv \in E$ e che non erano presenti in E in quanto avevano capacità nulla.

Per facilitare la costruzione della rete residua è opportuno rappresentare la rete di flusso $G = (V, E)$ mettendo in E non solo gli archi uv con capacità $c(uv) > 0$ ma anche tutti gli archi opposti vu, anche quelli con capacità $c(vu) = 0$.

Supponiamo che la rete di flusso $G = (V, E)$ sia rappresentata mediante liste delle adiacenze e che un nodo x nella lista $Adj[u]$ dei vertici adiacenti al vertice u contenga i seguenti campi:

1. $v[x]$ è il vertice adiacente $v = v[x]$. Rappresenta l'arco $uv \in E$;
2. $c[x]$ è la capacità $c(uv)$ dell'arco uv;
3. $f[x]$ è il flusso $f(uv)$ che attraversa l'arco uv;
4. $next[x]$ è il nodo successivo nella lista.

Descrivere un algoritmo che in tempo $O(n+m)$ aggiunge alla rappresentazione mediante liste delle adiacenze di $G = (V, E)$ tutti gli archi vu di capacità nulla che sono opposti di un arco $uv \in E$ di capacità $c(uv) > 0$.

Soluzione.

La cosa non è semplice come potrebbe sembrare. Prima di aggiungere l'arco vu con capacità nulla occorre controllare che esso non sia già presente con capacità non nulla. Se la lista $Adj[v]$ non è ordinata occorre scorrerla tutta e questa operazione ripetuta per tutti gli archi richiede tempo $O(nm)$.

Dobbiamo quindi procedere in modo diverso. Osserviamo che se effettuiamo la trasposizione del grafo G le liste delle adiacenze del grafo trasposto G^T risultano ordinate. Naturalmente se ripetiamo l'operazione partendo dal grafo trasposto G^T otteniamo una rappresentazione di G con liste ordinate. Tutto questo si può fare in tempo $O(n + m)$.

A questo punto non rimane che porre a zero tutte le capacità del grafo trasposto e quindi, per ogni vertice u effettuare la riunione delle due liste ordinate $Adj[u]$ ed $Adj^T[u]$ con l'accortezza che se un vertice compare in entrambe dobbiamo tenere soltanto quello con capacità positiva.

```
TRASPONIRETE(Adj, n)
    for u ← 1 to n do  Adj^T[u] ← nil
    for u ← 1 to n do
        x ← Adj[u]
        while x ≠ nil do
            v ← v[x]
            NEW(y),  v[y] ← u ,  c[y] ← c[x] ,  f[y] ← −f[x]
            next[y] ← Adj^T[v] ,  Adj^T[v] ← y
            x ← next[x]
    /* Le liste Adj^T[u] del grafo trasposto sono ordinate in ordine decrescente. */
```

return Adj^T

AUMENTERETE(Adj, n)
$Adj^T \leftarrow$ TRASPONIRETE(Adj, n)
$Adj \leftarrow$ TRASPONIRETE(Adj^T, n)
/* Le liste $Adj[u]$ ed $Adj^T[u]$ sono ordinate in ordine decrescente. */
for $u \leftarrow 1$ **to** n **do**
$x \leftarrow Adj[u]\ ,\ xp \leftarrow nil\ ,\ y \leftarrow Adj^T[u]$
while $x \neq nil$ **and** $y \neq nil$ **do**
if $v[x] \geq v[y]$ **then**
$xp \leftarrow x\ ,\ x \leftarrow next[x]$
if $v[x] = v[y]$ **then** $y \leftarrow next[y]$
else
$z \leftarrow y\ ,\ y \leftarrow next[y]\ ,\ next[z] \leftarrow x$
$c[z] \leftarrow 0\ ,\ f[z] \leftarrow -f[z]\ ,\ xp \leftarrow z$
if $xp = nil$ **then** $Adj[u] \leftarrow z$ **else** $next[xp] \leftarrow z$
while $y \neq nil$ **do**
$z \leftarrow y\ ,\ y \leftarrow next[y]\ ,\ next[z] \leftarrow x$
$c[z] \leftarrow 0\ ,\ f[z] \leftarrow -f[z]\ ,\ xp \leftarrow z$
if $xp = nil$ **then** $Adj[u] \leftarrow z$ **else** $next[xp] \leftarrow z$
return Adj