Lezione 11

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- visita di un grafo in ampiezza usando una coda implementata come vettore
- visita di un grafo in profondità usando una pila implementata come vettore
- visita di un grafo in profondità usando un algoritmo ricorsivo
- calcolo delle componenti connesse di un grafo, sfruttando la visita
- calcolo delle componenti fortemente connesse di un grafo, sfruttando la visita

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

In questo esercizio si vogliono affrontare due problemi legati fra loro:

1. la determinazione delle componenti connesse di un grafo non orientato;
2. la determinazione delle componenti fortemente connesse di un grafo orientato.

Un terzo problema, la determinazione del sottoinsieme di nodi o vertici raggiungibili da un nodo o vertice dato, cioè il problema della visita, si dimostra utile come sottoproblema per entrambi.

Formato di ingresso e di uscita I grafi vengono descritti come nella lezione precedente, cioè come semplice elenco di lati o di archi. Ciascuno di questi è rappresentato come una coppia di indici numerici, separati da una virgola e racchiusi fra parentesi tonde. Gli indici corrispondono ai vertici o nodi del grafo e sono compresi fra 1 e n. Per semplicità si suppone che non ci siano nodi isolati, per cui ogni nodo compare in almeno un arco dell'elenco. Per esempio, il grafo usato nei lucidi corrisponde a:

```
(1,2) (1,4) (2,4) (2,5) (3,6) (4,5)
```

Il problema fondamentale di questo formato è che non consente di distinguere tra grafi orientati e non orientati. Assumeremo che l'utente sappia a priori di che tipo di grafo si tratti. Nel caso dei grafi non orientati, quindi, le coppie vanno intese come non ordinate, mentre in quello dei grafi orientati come coppie ordinate.

La stampa delle componenti connesse deve cominciare con una riga che ne indichi il numero, seguito dalla parole chiave `componenti`:

```
2 componenti
```

Si stampa quindi un'altra riga con il vettore di marcatura racchiuso fra parentesi quadre e preceduto dalle parole chiave `C =`.

```
C = [ 1 1 2 1 1 2 ]
```

Infine, si stampano tante righe quante sono le componenti, riportando gli indici dei vertici che ne fanno parte, preceduti dalle parole chiave `U[`c`] =`, dove c è l'indice progressivo della componente via via considerata.

```
U[1] = 1 2 4 5
U[2] = 3 6
```

Per le componenti fortemente connesse, i formati di ingresso e di uscita sono gli stessi, salvo che le coppie di nodi nel file di ingresso vanno interpretate come archi orientati.

Traccia della risoluzione La lezione scorre i lucidi fino a pagina 6, prendendo le mosse dal problema della ricerca delle componenti connesse in un grafo non orientato. Questo problema è riconducibile a un sottoproblema più semplice, che è la visita di un grafo a partire da un vertice, per determinare i vertici da esso raggiungibili. Consideriamo un grafo non orientato. Essendo la relazione di connessione fra vertici simmetrica e transitiva, l'insieme U_v dei vertici visitati a partire dalla sorgente v coincide con la componente connessa che contiene v. Infatti:

- la componente connessa è contenuta in U_v perché tutti i suoi vertici sono per definizione raggiungibili da v;
- U_v è contenuto nella componente connessa perché
  - per simmetria, tutti i vertici raggiungibili da v sono anche co-raggiungibili;
  - per transitività, tutte le coppie di vertici sono vicendevolmente raggiungibili e co-raggiungibili passando per v.

Avendo un algoritmo per identificare una singola componente connessa, per determinarle tutte basta iterativamente:

1. scegliere una sorgente non ancora marcata
2. costruire la componente che la contiene

fermandosi quando tutti i vertici del grafo sono stati marcati.

Passiamo alle strutture dati utilizzate:

- il grafo viene trasformato in un grafo orientato simmetrico, rappresentato con il vettore delle *forward star*, dato che di ogni nodo occorre conoscere gli adiacenti, cioè le destinazioni degli archi uscenti dal nodo stesso;
- la soluzione viene rappresentata con un *vettore di marcatura* C, che fornisce per ogni indice $v \in V$ l'indice intero C_v che identifica la componente a cui appartiene v; questo indice sarà un numero progressivo[1].

Scartiamo invece come inefficienti le idee di rappresentare la soluzione con n_c tabelle o liste o vettori di incidenza binari (uno per ogni componente): tabelle e liste renderebbero inefficiente il test di appartenenza, i vettori di incidenza richiederebbero molto più spazio.

La versione iniziale del codice, `compconn0.c`, fornisce già le funzioni per interpretare la linea di comando (che richiede il nome del file che contiene il grafo) e per caricare il grafo stesso. Quest'ultima merita un'osservazione preliminare. Il formato che abbiamo già usato per codificare un grafo in un file di testo (lista degli archi) è ambiguo, perché non permette di distinguere tra grafi orientati e non orientati[2]. Per il calcolo delle componenti fortemente connesse, siccome esso è possibile solo per grafi orientati, assumeremo che l'utente fornisca al programma solo grafi di

[1] Potrebbe anche essere l'indice del vertice usato per individuare la componente.

[2] La convenzione di racchiudere i lati fra parentesi graffe e gli archi fra parentesi tonde è decisamente poco usata.

questo tipo. Per il calcolo delle componenti connesse, invece, il risultato è lo stesso per entrambi i tipi di grafo, dato che la definizione ignora l'orientamento. Siccome la rappresentazione del grafo con il vettore delle *forward-star* è intrinsecamente orientata, il programma interpreterà il grafo come non orientato e lo trasformerà automaticamente nel corrispondente grafo orientato simmetrico, sostituendo ogni lato letto con una coppia di archi opposti[3].

Prima fase (`compconn1.c`) La prima fase dell'esercizio si svolge come sempre in modalità *top-down*, realizzando le funzioni:

- `ComponentiConnesse`, per determinare le componenti connesse con l'algoritmo descritto nei lucidi, basato sulla riduzione al problema della visita;
- `StampaComponenti`, per stampare a video le componenti stesse, nel formato su indicato.

È necessario decidere dove allocare e dove inizializzare il vettore C. Diverse soluzioni sono ugualmente corrette: questa volta lo faremo nel `main`, come se si trattasse della creazione di una soluzione vuota. La funzione `ComponentiConnesse` scorrerà i vertici del grafo e ne eseguirà una visita a partire da ciascun vertice `s` non ancora visitato, marcando nel vettore C i vertici via via raggiunti con un indice progressivo c. Per far ciò, assumeremo di avere una funzione `visita`. Al termine, c fornirà il numero di componenti connesse del grafo. La stampa delle componenti verrà realizzata in modo abbastanza inefficiente, ma semplice, scorrendo il vettore C una volta per ogni componente e stampando solo i vertici della componente corrente[4].

Seconda fase (`compconn2.c` e `intqueue.c`) Possiamo quindi procedere con i lucidi da pagina 7 a pagina 11, ricapitolando il funzionamento degli algoritmi di visita in generale e di visita in ampiezza (*BFS*) in particolare. Questo algoritmo rappresenta l'insieme dei vertici raggiunti, ma non ancora usati per estendere la visita con una coda. Tale struttura astratta può essere implementate come:

- una tabella;
- una lista a puntatori (in tutte le otto varianti viste nella lezione dedicata);
- una lista con vettori e indici (in tutte le otto varianti viste nella lezione dedicata;
- un semplice vettore con due indici che marcano le posizioni di aggiunta ed eliminazione.

Quest'ultima implementazione è di gran lunga la più efficiente sia rispetto allo spazio sia rispetto al tempo. Seguendo sempre l'approccio delle strutture dati astratte, costruiremo librerie indipendenti dal programma in modo da nascondere l'implementazione. Questo consente di cambiare implementazione semplicemente sostituendo la libreria senza modificare il programma. Inoltre, dal punto di vista didattico il codice diventa molto leggibile e chiaro. Come già discusso, c'è uno svantaggio in termini di efficienza, dato che operazioni banali come la cancellazione di un elemento o la verifica di vuotezza (singole istruzioni) diventano chiamate di funzione, con

[3] Questo crea un piccolo problema se il file descrive invece un grafo orientato con qualche coppia di archi opposti: caricandolo, tali archi verranno raddoppiati. Siccome questo non produce errori nel risultato finale, per semplicità ignoreriamo il problema, anche se la presenza di archi duplicati rende l'algoritmo meno efficiente.

[4] È un buon esercizio per il progetto d'esame realizzare una versione efficiente di questa procedura, che scorra una volta sola il vettore C, usando un'opportuna struttura dati ausiliaria.

tutto il corollario di valutazione degli argomenti, allocazione di spazio sulla pila di sistema, copia dei parametri, e deallocazione finale dello spazio dalla pila di sistema.

La seconda fase dell'esercizio comporta quindi la realizzazione della funzione `BFS` per la visita in ampiezza, sulla falsariga dello pseudocodice riportato nei lucidi. È disponibile una libreria `intqueue.h`, che implementa una coda di interi. I concetti di base di questa implementazione sono richiamati nei lucidi alle pagine 12 e 13. Nel file `intqueue0.c`, le definizioni delle funzioni sono inizialmente vuote: usando la libreria, la visita crea una componente diversa per ogni vertice del grafo (il risultato è scorretto, ma si sta avvicinando a quello finale). Si deve quindi procedere a riempire le funzioni, completando la libreria `intqueue.c`.

Vale la pena di osservare che, rispetto alla definizione astratta di coda riportata nelle dispense del modulo di teoria (e rispetto alla definizione generale di lista adottata in una delle precedenti lezioni di laboratorio), le funzioni fondamentali che manipolano una coda la ricevono per indirizzo e non restituiscono alcun risultato. Questo è banalmente dovuto al fatto che, in genere, nel linguaggio C si preferisce passare alle funzioni le strutture complesse per indirizzo, per limitare le operazioni di copia dei dati. Analogamente, e per lo stesso motivo, si preferisce restituire un risultato di tipo struttura con il passaggio per indirizzo, anziché con l'istruzione `return`. Lo stesso succederà in seguito per la libreria di gestione delle pile.

Analisi di complessità dell'algoritmo *BFS* L'algoritmo di visita in ampiezza si può descrivere con il seguente pseudocodice.

```
 1: procedure BFS(V, E, s, C, c)
 2:     Q := ∅;
 3:     C[s] := c;
 4:     Enqueue(s, Q);
 5:     while not IsEmpty(Q) do
 6:         v := Front(Q);
 7:         Dequeue(Q);
 8:         for all w ∈ Adj(v) do
 9:             if C[w] = 0 then
10:                 C[w] := c;
11:                 Enqueue(w, Q);
12:             end if
13:         end for
14:     end while
15: end procedure
```

La creazione di una coda vuota, la marcatura della sorgente e il suo inserimento in coda richiedono tempo costante, $\Theta(1)$. Il ciclo alle righe ciclo 5–14 è abbastanza complicato da valutare, perché non indica chiaramente il numero di iterazioni: bisogna ricavarlo con un ragionamento. Ad ogni iterazione, si estrae un vertice dalla coda. Questo vertice era entrato in coda o prima del ciclo (se è la sorgente) o dentro il ciclo soddisfacendo la condizione alla riga 9 (`if (C[w] == 0)`). In entrambi i casi, si tratta di vertici appena raggiunti dalla visita. Appena un vertice viene inserito nella coda il corrispondente indice nel vettore C viene marcato. Quindi un vertice uscito dalla coda è marcato, non viene mai smarcato e non rientrerà mai più nella coda grazie alla condizione alla riga 9. Questo significa che ogni iterazione del ciclo $5-14$ è associata a un diverso vertice raggiunto dalla visita. I vertici raggiunti dalla visita sono tutti e soli quelli della componente connessa, per cui la complessità temporale del ciclo corrisponde a una sommatoria con indice $v \in U_s$. Dentro il ciclo, l'estrazione del vertice dalla coda (righe 6 e 7) richiede tempo costante,

mentre il ciclo più interno (righe 8–13) scorre i vertici adiacenti, per mezzo della lista degli archi uscenti. Le operazioni interne a questo ciclo (test di assenza della marcatura, marcatura e inserimento nella coda) richiedono tempo costante, $\Theta(1)$. Riassumendo:

$$T = \Theta(1) + \sum_{v \in U_s} \left(\Theta(1) + \sum_{a \in \Delta_v^+} \Theta(1) \right)$$

Se consideriamo per ogni vertice della componente tutti gli archi uscenti, stiamo sostanzialmente considerando tutti gli archi della componente, dato che non solo l'origine, ma anche la destinazione di tali archi appartiene alla componente:

$$\sum_{v \in U_s} \sum_{a \in \Delta_v^+} 1 = \sum_{a \in A(U_s)} 1 = |A(U_s)|$$

da cui

$$T = \Theta(1) + \Theta(n_s) + \Theta(m_s) = \Theta(n_s + m_s)$$

dove indichiamo per brevità con $n_s = |U_s|$ e $m_s = |A(U_s)|$ il numero di nodi e il numero degli archi della componente che include il nodo s.

Analisi di complessità per la ricerca delle componenti connesse L'algoritmo di ricerca delle componenti connesse si può invece descrivere con il seguente pseudocodice.

```
1: procedure ComponentiConnesse(V, E)
2:     c := 0;
3:     for all s ∈ V do
4:         if C[s] = 0 then
5:             c := c + 1;
6:             Visita(V, E, s, C, c);
7:         end if
8:     end for
9: end procedure
```

L'azzeramento iniziale delle marcature richiede tempo lineare, $\Theta(n)$. Il successivo ciclo scorre i vertici, che sono n. Nel caso pessimo, esegue (apparentemente) un test di assenza di marcatura, l'incremento dell'indice della componente e una visita. In totale, si concluderebbe che:

$$T = \Theta(n) + \sum_{v \in V} (\Theta(1) + \Theta(n_s + m_s)) = \Theta\left(n_s^2 + n_s m_s\right)$$

ma questa analisi, corretta per quel che riguarda la stima per eccesso di complessità, non lo è per quanto riguarda la stima per difetto: $T \in O\left(n_s^2 + n_s m_s\right)$, ma $T \notin \Omega\left(n_s^2 + n_s m_s\right)$. Infatti, ogni volta che si esegue una visita, tutti i vertici della componente vengono marcati, e quindi il test di assenza di marcatura viene soddisfatto solo una volta per ogni componente.

$$T = \Theta(n) + \sum_{v \in V} \Theta(1) + \sum_{c=1}^{nc} \Theta(n_c + m_c)$$

dove si è indicato con nc il numero di componenti e con n_c e m_c il numero di vertici e di archi della componente di indice c (per comodità, abbiamo cambiato notazione rispetto all'analisi dell'algoritmo di visita). È ovvio che la somma di tali due numeri su tutte le componenti dà, rispettivamente, il numero totale di vertici e di archi del grafo.

$$T = \Theta(n) + \Theta(n) + \Theta(n + m) = \Theta(n + m)$$

che è chiaramente molto meglio della rozza stima per eccesso da cui si era partiti.

Terza fase (`compconn3.c` e `intstack.c`) Nella terza fase dell'esercizio, sostituiamo la `BFS` con la visita in profondità (`DFS`), descritta nelle pagine $14 - -16$ dei lucidi. Tale algoritmo è praticamente identico alla `BFS`: confrontando i relativi pseudocodici, ci si rende conto che cambiano solo la struttura dati usata per conservare l'insieme dei vertici raggiunti e ancora non usati per proseguire la visita, e i nomi delle funzioni usate per:

- aggiungere elementi all'insieme (`Enqueue` anziché `Push`)
- togliere elementi dall'insieme (`Dequeue` anziché `Pop`)
- accedere all'unico elemento leggibile dell'insieme (`Front` anziché `Top`)

È disponibile la libreria `intstack.h`, sempre con un file `intqueue0.c` di definizioni vuote (che genera lo stesso risultato sbagliato di prima). Il file va completato in base ai meccanismi descritti alle pagine 17–18 dei lucidi.

Si può condurre un'analisi di complessità per la visita in profondità *DFS* e per la ricerca delle componenti connesse che danno luogo allo stesso risultato ottenuto più sopra, ancora una volta perché le funzioni che manipolano la pila sono molto simili a quelle usare per manipolare la coda, e hanno anch'esse complessità costante.

Quarta fase (`compconn4.c` Si può quindi realizzare la versione ricorsiva della visita in profondità (`DFSricorsiva`), che ha forti analogie con quella iterativa, come illustrato nei lucidi a pagina 19. A questo proposito è interessante disegnare passo per passo su un piccolo esempio quanto avviene nella pila di sistema, paragonando il comportamento della funzione ricorsiva con quello della funzione iterativa corrispondente. Questa ha una pila di numeri interi esplicitamente dichiarata nell'area di memoria riservata alla funzione `DFS`, mentre la funzione ricorsiva ha solo una cella intera `s`, dedicata alla sorgente. Però le chiamate ricorsive della funzione stessa creano sulla pila di sistema una sequenza di blocchi di memoria identici, che contengono le sei informazioni `grafo *pG`, `int s`, `vint C`, `int c`, `int w` e `posarco pa`. In particolare, la cella `s` si trova ripetuta a intervalli regolari lungo la pila di sistema e contiene esattamente gli stessi indici interi contenuti nella pila esplicita quando si esegue la funzione iterativa. La versione ricorsiva ha il vantaggio dell'eleganza ed essenzialità. Ovviamente, occupa molta più memoria sulla pila di sistema e richiede più tempo, a causa del meccanismo di allocazione, copia e deallocazione implicito nelle chiamate. La complessità asintotica rimane, tuttavia, la stessa.

Tutti e tre gli algoritmi possono essere modificati per restituire (invece del vettore marcatura C o in aggiunta ad esso) l'albero di visita, cioè l'albero formato dai soli archi utilizzati nel corso della visita. Tale albero è descritto alla pagina 20 dei lucidi, ed è un buon esercizio aggiungerne la costruzione ai codici realizzati in precedenza.

Quinta fase (`compfortconn1.c`) Consideriamo ora la ricerca delle componenti fortemente connesse su un grafo orientato. Anche questo problema è riducibile al problema della visita, ma questa volta si richiede di calcolare sia l'insieme U_v^+ dei nodi raggiungibili da v sia l'insieme U_v^- dei nodi co-raggiungibili da v. Infatti, la componente fortemente connessa che contiene v coincide con l'intersezione di tali due insiemi, dato che la relazione di connessione forte fra nodi è simmetrica e transitiva e:

- la componente fortemente connessa è contenuta in $U_v^+ \cap U_v^-$ perché tutti i suoi nodi sono per definizione raggiungibili e co-raggiungibili da v;

- $U_v^+ \cap U_v^-$ è contenuto nella componente connessa perché
  - tutti i vertici dell'intersezione sono raggiungibili e co-raggiungibili da v;
  - per transitività, tutte le coppie di vertici sono vicendevolmente raggiungibili e co-raggiungibili passando per v.

Per realizzare l'algoritmo, possiamo riutilizzare il codice scritto per implementare la visita del grafo, ma la visita va ripetuta due volte: sul grafo dato e sul grafo trasposto. Occorre quindi definire una funzione che costruisca il grafo trasposto a partire da quello dato. Inoltre, la visita applicata sui due grafi deve restituire due vettori diversi, di cui poi bisogna calcolare l'intersezione per marcare gli elementi di C. Siccome i due vettori sono in generale più grandi rispetto alla componente, vanno ripuliti ogni volta prima di cominciare la visita (contrariamente al caso delle componenti connesse, in cui una visita restituisce esattamente la componente cercata).

Sarebbe anche possibile non costruire fisicamente il grafo trasposto, realizzando una procedura di visita che usi gli archi all'indietro anziché in avanti per la seconda delle due chiamate. Affinché questa procedura sia efficiente, però il grafo deve essere rappresentato come vettore di *backward star*, oltre che come vettore di *forward star*. Bisogna quindi aggiornare la libreria che descrive il grafo, aumentando l'occupazione di memoria in misura analoga a quanto abbiamo fatto costruendo un secondo grafo a partire dal primo.

Analisi di complessità per la ricerca delle componenti fortemente connesse L'algoritmo di ricerca delle componenti fortemente connesse si può descrivere con il seguente pseudocodice.

```
1:  procedure ComponentiFortementeConnesse(N, A)
2:      A^T := GrafoTrasposto(A);
3:      c := 0;
4:      for all s ∈ N do
5:          if C[s] = 0 then
6:              c := c + 1;
7:              Visita(N, A, s, U^+, c);
8:              Visita(N, A^T, s, U^-, c);
9:              C := C + (U^+ ∩ U^-);
10:         end if
11:     end for
12: end procedure
```

Purtroppo, in questa analisi non si può ripetere esattamente quanto detto per le componenti connesse. Per prima cosa, occorre costruire il grafo trasposto. Questo comporta di:

1. creare e riempire un vettore di *forward star* orientate in modo opposto (dalla destinazione all'origine), in tempo $\Theta(n + m)$;
2. per ogni nodo non ancora marcato, visitare sia il grafo dato sia il grafo trasposto
3. intersecare i sottoinsiemi ottenuti con le due visite.

Gli insiemi generati dalle due visite contengono la componente fortemente connessa che include il nodo corrente, ma non coincidono con essa: solo la loro intersezione coincide con essa. Nonè quindi più vero che si esegue una sola visita per ogni

componente. La stima grezza $\Theta\left(n^2 + nm\right)$ è l'unica possibile[5]. Fortunatamente, esistono algoritmi per il calcolo delle componenti fortemente connesse che riescono a raggiungere la complessità $\Theta\left(n + m\right)$, ma non vengono trattati in questo corso.

[5] A rigore, si può migliorare la stima considerando che per ogni nodo le due visite toccano solo nodi e archi della componente (non fortemente, ma semplicemente) connessa che lo contiene, e quindi $\Theta\left(n\left(n_s + m_s\right)\right)$.