Algoritmi e strutture dati in C++

Lezione 1 di Programmazione 2

Introduzione

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2007-8

Indice

algoritmi:
correttezza, semplicità, efficienza

cosa intendiamo per **algoritmo**?

definizione informale: **metodo** per la soluzione di un problema,
adatto a essere implementato in forma di programma

tutti gli algoritmi operano su **strutture dati**

astratte, o **tipi di dati astratti (ADT)**:
appropriate alla descrizione degli algoritmi

concrete, o **implementazioni di ADT**:
appropriate alla stesura dei relativi programmi

principali criteri di **qualità** degli algoritmi

correttezza: conditio sine qua non

semplicità: utile a dimostrare la correttezza

efficienza: utile in pratica per la **scalabilità**

algoritmi e programmi:
efficienza o semplicità?

legge di Pareto:

nei sistemi con un numero molto grande di parti
più dell'80% dei fenomeni si deve a meno del 20% di esse

in grandi sistemi software, l'efficienza di **alcuni** algoritmi (<20%)
è di cruciale importanza per le prestazioni del sistema

efficienza di algoritmi ≠ efficienza di codifica

... la prima è molto più **redditizia**!

efficienza e semplicità spesso in **conflitto...** (che fare?)

combinazione ottimale: **algoritmo efficiente, codifica semplice**

valutazione delle prestazioni di un algoritmo:

analitica: può essere difficile, ma è rigorosa

empirica: sopperisce alle difficoltà di analisi

un esempio: il problema della connettività

formulazione matematica concisa: problema di

decisione di una relazione di equivalenza finita

cioè, più precisamente:

data una relazione binaria simmetrica R, specificata da un insieme finito di coppie, e una coppia in input,

decidere se la coppia appartiene a R^*, la chiusura riflessiva e transitiva di R, ossia se, nel grafo non orientato che rappresenta R, esiste un cammino che **connette** gli elementi della coppia data

il problema ha molte **applicazioni** pratiche, ad es.:

costruzione di reti (di calcolatori, elettriche, etc.)

problemi di unificazione
(nella deduzione automatica, implementazione di linguaggi dichiarativi, etc.)

il problema della connettività: precisazione

il problema proposto in (Sedgewick, 2003, Cap. 1) è un po' diverso:

input: una sequenza di coppie

si vuole un algoritmo che, per ogni coppia c_n, decida se i suoi elementi sono **connessi da un cammino** nel grafo (non orientato) costruito dalle coppie precedenti $\{c_m \mid m<n\}$:

se lo sono, si passa alla coppia successiva

altrimenti si produce in **output** c_n
(e si passa alla coppia successiva)

esempio	
in	out
1-3	1-3
2-4	2-4
3-0	3-0
1-0	
4-2	
0-2	0-2
2-1	

problemi di connettività: variazioni sul tema

varianti del problema proposto in (Sedgewick, 2003):

più complessa: se la coppia **p-q** è connessa, produrre in output uno dei (o tutti i) modi in cui è connessa

esempio: 4°, 5°, 7° input del precedente

esempio	
in	out
1-0	(1-3-0)
4-2	(2-4)
2-1	(1-3-0-2)

più "semplice" (secondo (Sedgewick, 2003)):
date **M** connessioni su **N** oggetti, dire se tutte le coppie di oggetti sono connesse

N.B. è più "semplice" l'output, ma si richiede pur sempre un algoritmo per il problema di decisione enunciato sopra!

idea per la soluzione: **output = albero di copertura?**

tutte le coppie degli **N** oggetti sono connesse sse l'output dell'algoritmo per il problema di connettività come posto in (Sedgewick, 2003) consta di **N-1** coppie

progetto di soluzioni: operazioni astratte *union, find*

idea essenziale: rappresentare le **componenti connesse** del grafo, cioè le classi di equivalenza di R^*

ogni nodo **p** del grafo appartiene a una e una sola componente connessa: C_p

operazione **find**: determina C_p, dato **p**

connettere due nodi fra i quali non c'è cammino nel grafo ne "fonde" le rispettive componenti connesse:

operazione **union**: unisce le componenti connesse C_p, C_q

operazioni astratte facilmente **riusabili** per molti altri problemi

soluzione astratta al problema proposto in (Sedgewick, 2003, Cap. 1):

per ogni coppia **p-q** in input:
find C_p, **find** C_q,
se $C_p \neq C_q$ allora **union**(C_p, C_q), **output p-q**

algoritmo *quick-find*

un algoritmo **semplice** (non troppo: non richiede memorizzazione di tutte le coppie in input)

si abbiano **N** nodi

struttura dati di supporto all'algoritmo: un **array id[N]** che soddisfi l'**invariante:**

se c_n = **p-q** è la **n**-sima coppia in input, allora **id[p]** = **id[q]** sse $C_p = C_q$ nel grafo costruito dalle coppie precedenti $\{c_m \mid m<n\}$

inizializzazione: **id[i]** ← **i**, **0 ≤ i < N**

implementazione di **union**(C_p,C_q):
t ← **id[p]** ; per **0 ≤ i < N**: se **id[i]** = **t** allora **id[i]** ← **id[q]**

implementazione di **find**: immediata, grazie all'**invariante**

efficienza dell'algoritmo quick-find : non entusiasmante ...
esegue almeno **MN** istruzioni, per **M** operazioni di union

un programma C++ per *quick-find*

problema della connettività: soluzione 1

```
#include <iostream>
static const int N = 10000;
int main()
{ int i, p, q, id[N];
  for (i = 0; i < N; i++) id[i] = i ;
  while (cin >> p >> q)
  {
    if (id[p] == id[q]) continue;
    int t = id[p];
    for (i = 0; i < N; i++)
      if (id[i] == t) id[i] = id[q];
    cout << " " << p << " " << q << endl;
  }
}
```

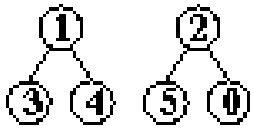
algoritmo *quick-union*

strutture ad albero nell'array **id[N]**

nell'esecuzione di **quick-find**:

possiamo modificarle per rendere più efficiente l'implementazione di **union**

input: 3-4, 4-1, 5-2, 0-2



nuovo **invariante:**
$C_p = C_q$ nel grafo costruito dalle coppie precedenti **p-q** sse $id^{\omega}p = id^{\omega}q$

inizializzazione: come in **quick-find**

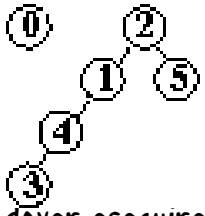
implementazione di **find**

si complica: $i \leftarrow id^{\omega}p$, $j \leftarrow id^{\omega}q$

implementazione di **union**(C_p,C_q)

immediata: **id[i]** ← **id[j]**

input: 3-4, 4-1, 5-2, 3-5



efficienza dell'algoritmo **quick-union** : non sempre buona, può dover eseguire più di **MN/2** istruzioni (per **N** nodi e **M** coppie in input), se **M>N**

un programma C++ per *quick-union*

problema della connettività: soluzione 2

si ottiene con semplici modifiche del programma C++ già visto per l'algoritmo **quick-find**:

si aggiunga la dichiarazione della variabile **`int j`**

si sostituisca il corpo del ciclo **`while`**, esclusa l'ultima istruzione (che resta invariata), con il codice seguente:

```
for (i = p; i != id[i]; i = id[i]) ;
for (j = q; j != id[j]; j = id[j]) ;
if (i == j) continue;
id[i] = j;
```

algoritmo *quick-union pesata*

quick-union : cammini lunghi da foglia a radice, anche fino a **N-1** passi

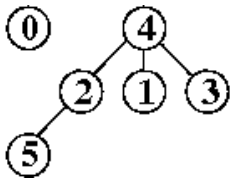
(!) possiamo intervenire su questo per ottenere una **find** più rapida

l'invariante di **quick-union** permane, ma fra due nodi $id^{\omega}p \neq id^{\omega}q$ da connettere per **union**, manteniamo la **radice dell'albero più grande**

per tener traccia della dimensione dell'albero che nell'array **id[N]** sottende ciascun nodo, usiamo l'array ausiliario **sz[N]**

inizializzazione: come in **quick-union**, ma estesa all'array ausiliario **sz**

input: 3-4, 4-1, 5-2, 3-5



implementazione di **find** : invariata

implementazione di **union**(C_p,C_q) : si complica un po' per tener conto di, e aggiornare, **sz**

efficienza di **quick-union pesata** : molto migliore della precedente, per la riduzione della lunghezza dei cammini da **O(N)** a **O(lg N)**

programma C++ per *quick-union pesata*

problema della connettività: soluzione 3

```
#include <iostream>
static const int N = 10000;
int main()
{ int i, j, p, q, id[N], sz[N];
  for (i = 0; i < N; i++) { id[i] = i ; sz[i] = 1; }
  while (cin >> p >> q)
  {
    for (i = p; i != id[i]; i = id[i]) ;
    for (j = q; j != id[j]; j = id[j]) ;
    if (i == j) continue;
    if (sz[i] > sz[j])
         { id[j] = i; sz[i] += sz[j]; }
    else { id[i] = j; sz[j] += sz[i]; }
    cout << " " << p << " " << q << endl;
  }
}
```

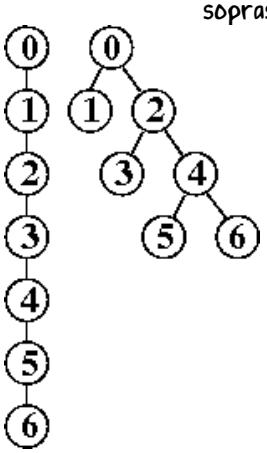
varianti dell'algoritmo *quick-union pesata*

ci si può attendere che ulteriori miglioramenti dell'efficienza si possano ottenere riducendo la lunghezza dei cammini

compressione dei cammini: si modificano puntatori visitati durante una **find**, con varie tecniche:

completa: ogni nodo visitato viene fatto puntare alla radice del suo albero, al termine della **union**

per dimezzamento: nei cammini visitati, si considerano due connessioni per volta, e si assegna al puntatore sottostante il valore di quello soprastante



prestazioni:

quanto sono vantaggiose queste tecniche?
ripagano il loro **costo computazionale** ?
come possiamo valutarne la convenienza nel **caso medio** e nel **caso peggiore** ?

valutazione empirica delle prestazioni

test: generazione casuale di **M** coppie per **N** nodi, fino a connetterli tutti

legenda:
F: quick-find
U: quick-union
W: quick-union pesata
P: quick-union pesata con compressione dei cammini completa
H: quick-union pesata con compressione dei cammini per dimezzamento

N	M	F	U	W	P	H
1000	6206	14	25	14	5	3
2500	20236	82	210	13	15	12
5000	41913	304	1172	46	26	25
10000	83857	1216	4577	91	73	50
25000	309802			219	208	216
50000	708701			469	387	497
100000	1708701			1071	1106	1096

Confronto empirico delle prestazioni per diversi valori di M e N