

Principi di progettazione di algoritmi

Lezione 3 di Fondamenti di Informatica

Docenti: Marina Madonia & Giuseppe Scollo

Università di Catania

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica, I livello, AA 2008-09

Indice

1. Principi di progettazione di algoritmi
2. correttezza, semplicità, efficienza
3. efficienza o semplicità?
4. il problema della connettività
5. prima variazione sul tema
6. altre variazioni sul tema
7. progetto di soluzioni: operazioni astratte
8. algoritmo quick-find
9. strutture dati astratte: sequenze, alberi
10. algoritmo quick-union
11. algoritmo quick-union pesata
12. prospettiva metodologica
13. esercizi

correttezza, semplicità, efficienza

principali criteri di **qualità** degli algoritmi

correttezza: *conditio sine qua non*

semplicità: utile a dimostrare la correttezza

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

l'esercizio 2 della lezione precedente suggerisce un promettente metodo di sviluppo di algoritmi per la soluzione di problemi di una certa complessità:

partendo dalla descrizione del problema, ideare una soluzione ad alto livello di **astrazione**, in cui cioè si assuma che l'esecutore possa eseguire come primitive operazioni complesse, delle quali occorre solo preoccuparsi che siano ben definite (e utili alla soluzione del problema ;)

successivamente procedere al **raffinamento** dell'algoritmo, affrontando il problema della realizzazione di quelle operazioni che, considerate primitive in prima approssimazione, risultano non esserlo per le effettive capacità dell'agente di calcolo

a ben vedere, la realizzazione di ciascuna delle operazioni da considerare nella fase del raffinamento costituisce a sua volta un problema al quale si cerca una soluzione algoritmica... si può dunque **reiterare** il metodo a ciascuno dei problemi della seconda fase, abbassando gradualmente il livello di astrazione in successivi passi di raffinamento, finché non rimane alcuna operazione che non sia effettivamente eseguibile

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 \neq 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

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.)

prima variazione sul tema

un algoritmo per il problema proposto può servire, fra l'altro, a risolverne 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	

altre variazioni sul tema

altre varianti del problema proposto:

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

esempio: $4^0, 5^0, 7^0$ input del precedente

più "semplice" (secondo (Sedgewick, 2002)):

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 la precedente prima variante del problema di connettività consta di $N-1$ coppie

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

progetto di soluzioni: operazioni astratte

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 alla prima variante del problema:

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: quale struttura dati concettuale di supporto all'algoritmo, introduciamo una sequenza di variabili id_i ($i=1,\dots,N$) 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: $id_i \leftarrow i, 1 \leq i \leq N$

implementazione di $union(C_p, C_q)$:

$t \leftarrow id_p$; per $1 \leq i \leq N$: se $id_i = t$ allora $id_i \leftarrow id_q$

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

efficienza dell'algoritmo *quick-find*: non entusiasmante ...

esegue almeno MN istruzioni, per M operazioni di $union$

strutture dati astratte: sequenze, alberi

cosa è esattamente una sequenza?

non un "insieme totalmente ordinato", perché ogni elemento di un insieme vi occorre un sola volta, mentre in una sequenza si possono avere più occorrenze di uno stesso elemento

a ben vedere, una sequenza è una **funzione**, definita su un dominio (totalmente ordinato) di **indici**

le sequenze sono strutture dati di frequente impiego negli algoritmi; lo sono anche gli **alberi**:

applicazioni: organigrammi, sistemi di file, strutture di libri, alberi sintattici, termini,

...

naturalmente associati allo sviluppo di algoritmi per raffinamenti successivi

cosa è esattamente un albero?

albero libero: grafo (non orientato) connesso privo di cicli

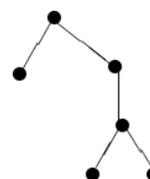
N.B. equivalentemente: grafo in cui esiste un unico cammino fra ogni coppia di vertici

albero (radicato): albero con un vertice designato, detto **radice**

sebbene il grafo di un albero sia generalmente non orientato, la designazione di un vertice radice induce naturalmente un ordinamento parziale dei vertici, grazie all'assenza di cicli, dunque un orientamento degli archi

$m < n$ se m precede n nel cammino dalla radice a n

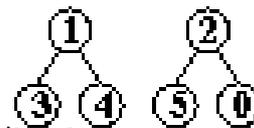
terminologia: **padre**, **figli**, **foglie**, **sottoalbero**, ...



algoritmo *quick-union*

alberi nella sequenza *id*
nell'esecuzione di *quick-find*: ne sono
radici i punti fissi della funzione *id*,
cioè gli *n* tali che $id_n = n$

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



possiamo modificarli per una più efficiente esecuzione di *union*

nuovo invariante: (id^w_p : radice dell'albero a cui appartiene *p*)

$C_p = C_q$ nel grafo costruito dalle coppie precedenti *p*-*q* sse $id^w_p = id^w_q$

inizializzazione: come in *quick-find*

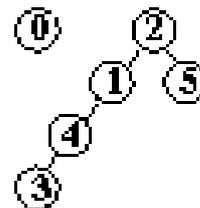
input: 3-4, 4-1, 5-2, 3-5

implementazione di *find*

si complica: $i \leftarrow id^w_p, j \leftarrow id^w_q$

implementazione di *union*(C_p, C_q)

immediata: $id_i \leftarrow id_j$



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 $M > N$

algoritmo *quick-union pesata*

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

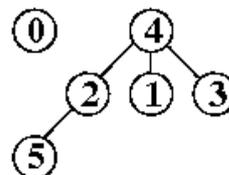
(!) possiamo intervenire su questo per ottenere una *find* più rapida
l'invariante di *quick-union* permane, ma fra due nodi $id^w_p \neq id^w_q$ da
connettere per *union*, manteniamo la radice dell'albero più grande
per tener traccia della dimensione dell'albero che nella sequenza *id* sottende
ciascun nodo, usiamo una sequenza ausiliaria $sz_i, i = 1, \dots, N$

inizializzazione: come in *quick-union*, ma estesa
alla sequenza ausiliaria *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, proporzionale non più a *N* bensì a $\lg N$, anche nel caso
peggiore

prospettiva metodologica

che "lezione di metodo" trarre dallo studio condotto sul problema della connettività?

punti salienti dello schema di studio degli algoritmi union-find:

specificazione chiara ed esatta dell'enunciato del problema

analisi dell'enunciato

per l'identificazione delle **operazioni astratte** intrinseche del problema

sviluppo di un primo algoritmo

semplice e di immediata correttezza, in termini delle **operazioni astratte**

implementazione semplice in un primo programma

con una codifica delle operazioni astratte su **strutture dati** atte allo scopo

miglioramento dell'efficienza per raffinamenti successivi

valutandone l'efficacia con **analisi formali e/o valutazioni empiriche**

uso di rappresentazioni astratte di strutture dati e algoritmi impiegati

per la **progettazione ad alto livello** delle successive versioni

analisi prestazionale (caso peggiore e medio) per trarne **garanzie di efficienza**

esercizi

1. descrivere un metodo semplice per contare il numero di componenti connesse che si hanno al termine dell'esecuzione di un algoritmo per la prima variante del problema della connettività, che usi le operazioni **union** e **find**
2. nella presentazione dell'algoritmo di quick union si definisce id^w_p come il punto fisso della funzione **id** che è radice dell'albero a cui appartiene **p**, ma non si descrive in dettaglio come trovarlo: fornire una tale descrizione in pseudocodice
3. nella presentazione dell'algoritmo di quick union pesata si accenna alle modifiche, rispetto all'algoritmo precedente, riguardanti l'inizializzazione e l'esecuzione dell'operazione **union**, per soddisfare la specifica della sequenza **sz**; tuttavia non si descrive il dettaglio di tali modifiche: descriverle in pseudocodice
4. modificare la descrizione dei dettagli dell'algoritmo di quick-union pesata, prodotta in risposta all'esercizio precedente, in modo da usare l'**altezza** degli alberi (massima lunghezza dei cammini dai nodi alla radice) invece del loro peso (numero di vertici) quale criterio di scelta della radice del nuovo albero nell'esecuzione dell'operazione **union**