Nota sul problema del maggioritario

02/03/2010

1 Definizioni

Definition 1 *Data una sequenza di valori interi $a_1 a_2 \dots a_n$ (dove lo stesso valore può ripetersi), si dice che il valore x è* maggioritario *in A se occorre per più di $n/2$ volte, ossia in notazione insiemistica: $|\{j \mid a_j = x\}| > n/2$*

Ogni sequenza può contenere al massimo un singolo elemento maggioritario. Il seguente problema algoritmico è di importanza fondamentale in molte applicazioni:

Problem 1 (del maggioritario) *Data una sequenza di interi di lunghezza n contenuta nell'array $A[1, n]$, determinare l'elemento maggioritario se presente o resituire $\perp$ nel caso in cui la sequenza sia priva di maggioritario.* [1]

2 Soluzione con numero lineare di confronti e spazio costante

Illustriamo una tecnica per il calcolo dell'elemento maggioritario dotata delle seguenti caratteristiche:

- Lavora *in tempo lineare* effettuando due scansioni dell'array A contenente la sequenza in input.
- Richiede un numero di celle di memoria *costante e indipendente da n.*

[1] Assumiamo che $\perp$ sia un valore speciale convenzionalmente usato per indicare l'assenza del maggioritario

La soluzione è basata sulla procedura `FindCandidate`, il cui pseudocodice è riportato in algorithm 1. La procedura `FindCandidate` non è sufficiente da sola a risolvere il problema del maggioritario tuttavia, grazie al seguente teorema 2 (che dimostreremo a breve), è immediato individuare l'eventuale maggioritario di A a partire dal risultato dell'esecuzione di `FindCandidate`(A) :

Theorem 2 *Se in A è presente un elemento maggioritario allora questo coincide con l'elemento contenuto nella variabile* `candidate` *al termine dell'esecuzione di* `FindCandidate`(A).

In virtù del teorema 2, se A è dotato di maggioritario questo potrá essere soltanto `candidate`, dunque basta effettuare una verifica del solo elemento `candidate` per stabilire se si tratti del maggioritario di A o meno. In definitiva, dopo aver eseguito `FindCandidate`(A), contiamo il numero di occorrenze di `candidate` in A con una ulteriore scansione e se queste superano $n/2$ possiamo restituire `candidate` come maggioritario di A, altrimenti possiamo concludere che A non ha maggioritario.

Algorithm 1 `FindCandidate`

Require: $A[1, n]$ array contenente la sequenza in input

```
1:  candidate ← NIL, candidato corrente
2:  counter ← 0
3:  for i = 0 to n do
4:    if A[i] = candidate then
5:      counter ← counter + 1
6:    else
7:      if counter > 0 then
8:        counter ← counter − 1
9:      else
10:       candidate ← A[i], counter ← 1
11:     end if
12:   end if
13: end for
14: return candidate
```

Si veda la figura 2 per un esempio di esecuzione della procedura `FindCandidate` su un array di 9 elementi. La figura mostra le successione di valori assunti da `candidate` e `counter` durante la scansione. La dimostrazione del teorema 2 si basa sul seguente lemma 3. D'ora in poi indicheremo con c e

Figura 2

r i valori contenuti rispettivamente nelle variabili `candidate` e `counter` al termine dell'esecuzione di `FindCandidate(A)`, e con U la sottosequenza ottenuta rimuovendo da A le ultime r occorrenze di c.

Lemma 3 *E' possibile raggruppare gli elementi di U in coppie costituite da elementi distinti.*[2]

Proof:

Dimostriamo il lemma in maniera costruttiva, ossia generando le coppie in cui U risulta partizionata via via che gli elementi di A sono esaminati dalla procedura `FindCandidate`. Consideriamo dunque una generica iterazione del ciclo `FOR` principale della procedura `FindCandidate`(A). Definiamo X come l' insieme delle ultime `counter` occorrenze dell'elemento `candidate` che precedono l'elemento corrente.

La costruzione garantisce come proprietà *invariante* che all'inizio di ogni iterazione tutti gli elementi di A già esaminati e non facenti parte di X sono stati accoppiati tra loro. Gli elementi contenuti in X restano invece temporaneamente spaiati, in attesa di scoprire un nuovo elemento di A con cui accoppiarli. Tale invariante è banalmente soddisfatto all'inizio della prima iterazione, dove $X = \emptyset$ e nessun elemento di A è stato ancora visto.

Per far si che l'invariante resti soddisfatto alla fine dell'iterazione corrente, la costruzione distingue due casi:

- ($A[i]$ = `candidate`): $A[i]$ diventa parte dell'insieme X all'iterazione successiva e nessuna nuova coppia viene formata
- ($A[i] \neq$ `candidate`): Se *counter* > 0 allora X è non vuoto, dal momento che contiene *counter* elementi. Dunque viene formata la coppia $(A[i], x_1)$ dove x_1 è l'elemento di X apparso per primo in A; tale coppia

[2] gli elementi di una coppia non occupano necessariamente posizioni adiacenti in A

è valida poichè $A[i] \neq candidate = x_1$. Se invece $counter = 0$ allora attualmente $X = \emptyset$ mentre dall'iterazione successiva $A[i]$ sarà il nuovo `candidate` e avremo $X = \{A[i]\}$; in tal caso $A[i]$ non viene inserito in nessuna coppia.

E' facile verificare che in entrambi i casi l'invariante di sopra resta preservato da un iterazione alla successiva. Alla fine dell'esecuzione di $FindCandidate(A)$ avremo che X consiste delle ultime r occorrenze di c, dunque tutti gli altri elementi di A, ossia quelli che formano la sottosequenza U, risultano partizionati in coppie di elementi distinti, come desiderato.

La figura 2 illustra il risultato della costruzione del lemma 3 su un array di dimensione 9, gli elementi di U inseriti in una stessa coppia sono congiunti da un arco nella figura. In tal caso, il valore finale di `counter` è 3 e l'elemento `candidate` è 0, dunque U consiste di tutti gli elementi della sequenza originale tranne le ultime 3 occorrenze di 0.

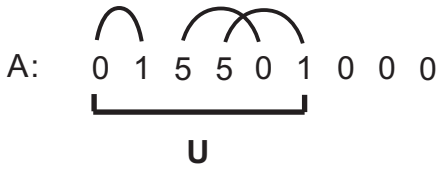

Figura 3

Il lemma 3 ci permette di provare immediatamente l'asserto del teorema 2. Infatti ammettiamo per assurdo che A contenga un elemento maggioritario $p \neq c$. Notare che tutte le occorrenze di p fanno parte di U. Ma, in base al lemma 3, U è suddiviso in coppie di elementi distinti, dunque p può occorrere al più una volta in ciascuna coppia, il che significa che ci sono al più $n/2$ occorrenze di p, contro il fatto che p sia maggioritario.

3 Esercizi

- Dare un'implementazione della funzione `FindCandidate` in linguaggio C, la cui segnatura è `int FindCandidate(int *A, int len)`. Sfruttando `FindCandidate`, si implementi una funzione per il calcolo del maggioritario in tempo lineare `int FindMajority(int *A, int`

`len, int *m)` che si comporti come segue: se `A` è dotato di maggioritario la funzione restituisce 1 e scrive il maggioritario all'indirizzo puntato da m; altrimenti la funzione restituisce 0.