Dati e Algoritmi I (Pietracaprina)

Scritto del 12/02/2020 (Seconda Parte)

SOLUZIONI

Problema. Sia $S = S[0] \dots S[n-1]$ una sequenza di n chiavi intere in $[1, K]$. Il seguente algoritmo restituisce la chiave più grande tra quelle con $\geq \gamma$ occorrenze in S. Se ogni chiave ha meno di γ occorrenze in S, restituisce -1.

```
key ← −1;
B[1...K] ← array inizializzato con tutti 0;
for i ← 0 to n − 1 do
    B[S[i]] ← B[S[i]] + 1;
    if (B[S[i]] ≥ γ) AND (S[i] > key) then key ← S[i];
return key;
```

Identificare un opportuno invariante che valga alla fine di ciascuna iterazione del **for**, e che sia utile per provare la correttezza dell'algoritmo. (La prova di correttezza non è richiesta.)

Soluzione. Alla fine della iterazione i del ciclo **for** vale il seguente invariante

- $B[\ell]$ = numero di occorrenze di ℓ in $S[0 \dots i]$.
- `key` è la minima chiave ℓ con $B[\ell] \geq \gamma$, se almeno una tale chiave esiste, altrimenti `key` $= -1$.

□

Problema. Sia T un albero binario proprio dove ogni nodo $v \in T$ memorizza un valore v`.val`. Diciamo che T soddisfa la *semi-heap property* se la seguente proprietà vale *per ogni nodo* $v \in T$:

$$v.\texttt{val} \geq \frac{1}{2} \max\{u.\texttt{val} : u \in T_v\},$$

dove T_v denota il sottoalbero con radice v. Progettare un algoritmo *ricorsivo* `semiHeap` per determinare se T soddisfa la *semi-heap property*, e analizzarne la complessità.

Soluzione. Sviluppiamo un algoritmo `semiHeap(T,v)` che, quando invocato su un nodo $v \in T$, restituisce il valore massimo nei nodi di T_v e un flag binario che dice se T_v soddisfa la semi-heap property. Chiaramente, il flag restituito dalla invocazione `semiHeap(T,T.root()` dice se T soddisfa la semi-heap property. Lo pseudocodice è il seguente

Algoritmo `semiHeap`(T, v)

Input: `Albero` T`, nodo` $v \in T$
Output: $\max\{u.\texttt{val} : u \in T_v\}$`, flag = 1 se` $\texttt{T}_v$ `soddisfa la semi-heap property, e 0 altrimenti`

```
if (T.isExternal(v)) then return (v.val,1);
(ML,fL) ← semiHeap(T,T.left(v));
(MR,fR) ← semiHeap(T,T.right(v));
M ← max{maxL, maxR, v.val};
if ((v.val ≥ (1/2) M) AND fL AND fR) then  return (M,1);
else return (M,0);
```

Se invocato con $v =$ `T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo richiede tempo $O(1)$. La complessità risulta essere $O(n)$, dove n è il numero di nodi di T. □

Problema. Sia $G = (V, E)$ un grafo in cui $V = \{1, 2, \ldots, n\}$ e $E \subseteq \{(i, i+1) : 1 \leq i < n\}$. (Ad es., $n = 10$ e $E = \{(1, 2), (4, 5), (5, 6), (6, 7), (8, 9), (9, 10)\}$.) Progettare un algoritmo `Diametro` che in tempo $O(n)$ trovi il diametro del grafo, ovvero, la massima distanza tra due vertici di una stessa componente connessa, dove la distanza tra due vertici u, v è data dal numero di archi nel cammino minimo che connette u a v.

Soluzione. Per come è definito, il grafo è un insieme di catene disgiunte, ciascuna delle quali è del tipo $(i, i+1, \ldots, i+k)$ dove $1 \leq i \leq i+k \leq n$. Per trovare il diametro basterà quindi una scansione dei vertici da 1 a n tenendo traccia della catena più lunga incontrata. L'algoritmo è il seguente

Algoritmo `Diametro`(G)

Input: `Grafo` $G = (V, E)$ `con` $V = \{1, 2, \ldots, n\}$ `e` $E \subseteq \{(i, i+1) : 1 \leq i < n\}$
Output: `Diametro di` G

```
diam ← 0;
curr ← 0;
for i ← 2 to n do
    if (i ∈ G.incidentEdges(i − 1)) then
        curr ← curr+1;
        if (curr > diam) then diam ← curr;
    else curr ← 0;
return diam
```

Si noti che dato che ogni vertice ha grado al più 2, la verifica $i \in G$.incidentEdges$(i-1)$ può essere fatta con $O(1)$ operazioni. Di conseguenza ciascuna iterazione del for richiede $O(1)$ operazioni e quindi l'algoritmo ha complessità $O(n)$. □