Università degli Studi di Verona

Dipartimento di Informatica

Ca' Vignal 2
Strada le Grazie 15
37134 Verona - Italia
Tel. +39 045 802 7069
Fax +39 045 802 7068

Esame di ALGORITMI: Corso di Laurea in Informatica
13 giugno 2017

1. Indicare la veridicità di ognuna delle seguenti affermazioni.

V F

□ □ E' possibile ordinare un array di numero razionali in tempo cubico.

□ □ Non è possibile ordinare in tempo lineare un array di razionali in [0, 10] con numeratore limitato.

□ □ Bucket sort non è stabile.

□ □ Heap sort ordina in loco.

□ □ E' possibile trovare il mediano di un array non ordinato in tempo logaritmico.

2. La soluzione all'equazione di ricorrenza $T(n) = 27T(n/3) + n^2 \log n / 2$ è

□ $O(\log n)$ □ $\Omega(n \log^2 n)$ □ $O(n^2 \log n)$ □ $O(n^3)$ □ $\Theta(5n^3)$

3. Il problema di verificare se un grafo è bipartito appartiene a

□ $O(V^2)$ □ $\Omega(VE)$ □ $\Theta(V+E)$ □ $O(V \log E)$ □ $\Omega(E^2)$

4. Indicare la veridicità di ognuna delle seguenti affermazioni.

V F

□ □ Le matrici di adiacenza non possono essere usate per rappresentare grafi sparsi.

□ □ Se in un grafo orientato tutti gli archi hanno lo stesso peso, non negativo, allora i cammini minimi possono essere calcolati in tempo O(V+E).

□ □ Un grafo con un solo arco è sempre bipartito.

□ □ Il problema del flusso massimo non può essere espresso come problema di programmazione lineare..

□ □ L'algoritmo di Johnson per i cammini minimi tra tutte le coppie non funziona quando esistono archi negativi.

5. Indicare la veridicità di ognuna delle seguenti affermazioni.

V F

□ □ In un grafo aciclico esiste al più un nodo che può raggiungere tutti gli altri nodi.

□ □ La chiusura transitiva di un grafo aciclico è un grafo aciclico.

□ □ Se un grafo è orientato, allora la sua matrice di adiacenza è simmetrica.

□ □ In una rete di flusso un flusso è massimo se e solo se non esistono tagli non saturi.

□ □ Se un algoritmo funziona sui grafi orientati allora funziona anche sui grafi non orientati.

Soluzione crocette

Es 1

1. V posso ordinare un qualsiasi array per confronto in tempo Θ(nlog(n)), quindi anche uno di razionali
2. V Nel caso di numeratori compresi in [0, 1) e denominatori illimitati, ho un numero illimitato di possibili valori. Non posso quindi utilizzare algoritmi lineari.
3. F Bucket Sort è stabile
4. V La costruzione dello heap viene fatta sullo stesso array che si sta ordinando
5. F è possibile trovare il mediano di un array in tempo lineare (con l'algoritmo SELECT)

Es 2

Applico il "master theorem": vale che $n^2 \log n = O(n^{log_3 27 - \epsilon})$

Mi trovo quindi nel caso numero 1, concludo che quindi

$$T(n) = \Theta(n^3)$$

Pertanto, vale anche che $T(n) = \Omega(nlog^2 n)$, $T(n) = O(n^3)$ e ovviamente $T(n) = \Theta(5n^3)$

Es 3

Per verificare se un grafo è bipartito, posso utilizzare una visita di tipo DFS, in cui vado a colorare alternatamente i nodi di un colore o dell'altro. Pertanto la complessità è $\Theta(V + E)$ e dunque anche $O(V^2)$.

Es 4

1. F Posso usare le matrici di adiacenza anche per grafi sparsi, seppur non sia una soluzione efficiente in termini di memoria
2. V In tal caso posso usare l'algoritmo BFS per calcolare i cammini minimi
3. F Può essere che l'unico arco sia un auto arco
4. F Nel libro vi è l'esempio di come un flusso massimo possa essere espresso come programmazione lineare
5. F L'algoritmo di Johnson non funziona quando esistono cicli negativi, in caso di archi negativi funziona perché viene aggiunta una costante per far si che gli archi non siano più negativi

Es 5

1. V Se da più di un nodo posso raggiungere tutti gli altri nodi, banalmente ho un ciclo
2. V Vedi definizione di chiusura transitiva
3. F Se un grafo è NON orientato la sua matrice di adiacenza è simmetrica
4. F Un flusso può essere massimo e possono esistere tagli non saturi, banalmente se abbiamo dei colli di bottiglia nel sistema per forza avremmo tagli non saturi
5. V Un grafo non orientato non è altro che un grafo orientato con due archi in direzioni opposto

Esercizio

[il testo è scritto per come lo ricordo]

Per produrre un'automobile si devono completare più fasi, ognuna delle quali richiede un certo tempo. Possiamo eseguire più fasi in parallelo, tuttavia alcune fasi richiedono che prima sia stata ultimata la fase precedente (es non posso montare l'autoradio se prima non ho completato la carrozzeria).

Vogliamo un algoritmo che sia in grado di calcolare in quanto tempo al minimo possiamo assemblare l'automobile, supponendo di poter eseguire in parallelo tutte le fasi che ci è consentito fare.

Rappresentazione del problema:

costruiamo un DAG (Directed Acyclic Graph), in cui ogni vertice rappresenta una fase dell'assemblaggio dell'auto. Ogni arco invece punta vero la fase dalla quale il vertice sorgente dipende. Il peso di ogni arco invece indica il tempo di completamento della fase al quale l'arco punta.

Es: l'arco (u, v) di peso 5 significa che prima di eseguire l'operazione u devo aver completato l'operazione v, la quale può essere completata in 5 ore.

A questo punto non si deve fare altro che trovare il cammino di costo massimo che va dal nodo finale ad uno dei nodi iniziali. Questo si può fare in più modi.

Sicuramente quello più veloce da scrivere è il seguente, prendo il grafo trasposto (inverto la direzione degli archi), aggiungo un supernodo iniziale e lo collego con archi di costo 0 ad ogni nodo iniziale, cambio i pesi degli archi e li metto tutti negativi, ed applico l'algoritmo DAG-SHORTEST-PATH per calcolare il cammino di costo minimo (ma che sarà massimo, avendo invertito i pesi) fra il nodo S e il nodo finale.

La procedura DAG-SHORTEST-PATH ha complessità O(V+E) e quindi l'algoritmo pure.

Altra soluzione, si poteva presentare un algoritmo custom ricorsivo simile a DFS che va a trovare direttamente il cammino di costo massimo (tuttavia era accettabilissimo anche dire uso DAG-SHORTEST-PATH senza scrivere una riga di codice)

Alessandro Righi