

Esercizi per il Corso di Algoritmica 2

(a.a. 2013/14)

Roberto Grossi
Dipartimento di Informatica, Università di Pisa
grossi@di.unipi.it

16 dicembre 2013

Sommario

Vengono raccolti i problemi discussi in classe e collegati agli argomenti a lezione. Tali problemi vengono approfonditi e osservati da più punti di vista, anche sbagliati, in quanto l'errore è funzionale all'apprendimento di situazioni complesse. La motivazione risiede nel fatto che interessa sviluppare il percorso che conduce alla soluzione (piuttosto che la soluzione stessa), sotto la guida del docente in base alle idee proposte dagli studenti. La soluzione non viene qui fornita per i motivi suddetti: è preferibile venire a ricevimento dal docente.

1. [Sottografi K_4 e C_4] Descrivere un algoritmo per elencare e contare, dato un grafo non orientato, tutti i suoi sottografi di tipo K_4 (grafo completo di 4 vertici) e C_4 (ciclo di vertici che non sono collegati da altri archi tra di loro). Valutare e motivare la complessità della soluzione proposta.
2. [Clique massimali] Una clique massimale è un sottoinsieme S dei vertici del grafo tale che S induce un grafo completo mentre per ogni altro vertice $u \notin S$ vale che $S \cup \{u\}$ non induce un grafo completo (massimalità). Descrivere un algoritmo che elenchi tutte le clique massimali di un grafo non orientato con un costo che dipende dalla dimensione dell'uscita (output-sensitive): se C è l'insieme delle clique massimali del grafo con n vertici e m archi, il costo temporale deve essere limitato superiormente da $|C| \times (m + n)^{O(1)}$.

Suggerimento: dati gli insiemi R e P , le chiamate ricorsive generano tutte le clique massimali che contengono sicuramente R e possibilmente alcuni vertici di P . Nell'albero della ricorsione, il nodo α che corrisponde a tale chiamata con R e P avrà tali clique nelle foglie discendenti. Sia k il numero di tali foglie, ossia il numero di clique massimali generate a partire da α durante il ciclo `for` sui vertici in P . Possiamo pagare $O(k(m+n)^{O(1)})$ tempo per decidere se una chiamata ricorsiva in tale ciclo `for` porterà sicuramente a una nuova clique massimale: basta vedere se R unito a ciò che rimane di P non è coperto dalle clique generate fino a quel momento dal ciclo `for`.

3. [Cammini tra due vertici] Prendendo il metodo della partizione binaria visto a lezione per elencare tutti i cammini semplici tra due vertici distinti s e t di un grafo non orientato, mostrare come modificarlo in modo da garantire che ogni chiamata ricorsiva conduca ad almeno un nuovo cammino elencato (vogliamo quindi evitare chiamate ricorsive che non generino alcun cammino da elencare). Valutare e motivare la complessità della soluzione proposta.
4. [Cammini più brevi tra due vertici] Descrivere un algoritmo per elencare tutti i cammini più brevi (shortest path) tra due vertici distinti s e t di un grafo non orientato (volendo, pesato o meno). Valutare e motivare la complessità della soluzione proposta.
5. [Minimal feedback vertex set] Nell'algoritmo per elencare i minimal feedback vertex set (MFVS) visto a lezione, dimostrare che il meta-grafo risultante è fortemente connesso, ossia, per ogni due meta-vertici esiste sempre un cammino diretto che li collega, dove ogni arco è rappresentato dalla funzione successore μ : questa proprietà serve a garantire che da qualunque meta-vertice partiamo, riusciamo a raggiungere tutti gli altri meta-vertici. Dimostrare se la funzione successore μ sia iniettiva o meno; in base a ciò, mostrare come elencare tutti i MFVS attraverso una visita di tutti i meta-vertici senza però costruire e memorizzare esplicitamente il meta-grafo. Valutare e motivare la complessità della soluzione proposta. Suggerimento: non riuscendo a dimostrare se μ sia iniettiva o meno, descrivere come generare tutti i MFVS ipotizzando che (i) μ sia iniettiva e (ii) μ non sia iniettiva (qui usare un dizionario e una coda di priorità per memorizzare i MFVS/meta-vertici).
6. [Cammino/ciclo euleriano] Descrivere un algoritmo per trovare, in tempo lineare, un cammino o un ciclo euleriano in un grafo non orientato e connesso (suggerimento: utilizzare il fatto che nei vertici di grado pari è possibile sempre uscire una volta che si è entrati per trovare un cammino parziale, che viene esteso attraverso cicli che devono essere collegati al cammino perché il grafo è connesso).
7. [Ricoprimento minimo di vertici] Per il problema del minimum vertex cover (MVC), trovare un contro-esempio che mostri come la strategia greedy di scegliere sempre il vertice di grado massimo e iterando sul grafo residuo ottenuto cancellando tale vertice e i suoi archi incidenti, non garantisca una 2-approssimazione. Generalizzare tale argomento per mostrare che tale strategia greedy non garantisce una r -approssimazione per una qualunque costante prefissata $r > 1$.
8. [Approssimazione per MAX-SAT] Per il problema MAX-SAT della soddisfacibilità di una formula booleana, si consideri il seguente algoritmo di approssimazione per massimizzare il numero di clausole soddisfatte in una data formula: Sia F la formula data, x_1, x_2, \dots, x_n le variabili booleane in essa contenute, e c_1, c_2, \dots, c_m le sue clausole. Scegli i valori booleani casuali b_1, b_2, \dots, b_n , ossia ciascun $b_i \in \{0, 1\}$ ($1 \leq i \leq n$). Calcola il numero m_0 di clausole soddisfatte dall'assegnamento tale che $x_i := b_i$ ($1 \leq i \leq n$). Calcola il numero m_1 di clausole soddisfatte dall'assegnamento tale che $x_i := \bar{b}_i$ ($1 \leq i \leq n$), dove \bar{b}_i indica la negazione di b_i . Se $m_0 > m_1$,

restituisce l'assegnamento $x_i := b_i$ ($1 \leq i \leq n$); altrimenti, restituisce l'assegnamento $x_i := \bar{b}_i$ ($1 \leq i \leq n$).

Dimostrare che il suddetto algoritmo è una r -approssimazione per MAX-SAT, indicando anche il valore di $r > 1$ (e motivando l'utilizzo di tale valore). Discutere se, in generale, la scelta di b_1, b_2, \dots, b_n possa influenzare o meno il valore di r , motivando le argomentazioni addotte. Facoltativo: creare un'istanza di MAX-SAT in cui il suddetto algoritmo ottiene un costo che è r volte più piccolo del costo ottimo per una data scelta dei valori di b_1, b_2, \dots, b_n .

9. [Approssimazione per knapsack] Nell'algoritmo greedy di approssimazione per il problema dello zaino (knapsack) visto a lezione, mostrare che quando vale la condizione $\bar{a}_j = b$ (vedere Teorema 2.1 nel capitolo disponibile sul sito) tale algoritmo ottiene una soluzione di costo ottimo.
10. [Approssimazione per MAX-CUT] Il problema MAX-CUT è NP-hard ed è definito come segue per un grafo non orientato $G = (V, E)$. Una partizione di nodi $(C, V - C)$ con $C \subseteq V$ si chiama "cut" o *taglio*. Un arco $e = (v, w)$ con $v \in C$ e $w \in V - C$ si chiama arco di taglio (ricordando che (v, w) e (w, v) denotano lo stesso arco in un grafo non orientato). Il numero di archi di taglio definisce la dimensione del cut $(C, V - C)$. Poiché cambiando taglio, può cambiare la sua dimensione, il problema richiede di trovare il taglio di dimensione *massima* e quindi gli archi di taglio corrispondenti. Dimostrare che il seguente algoritmo randomizzato è una 2-approssimazione in valore atteso, ossia che il numero medio di archi di taglio così individuati è in media almeno la metà di quelli del taglio massimo. (1) Per ogni nodo $v \in V$, lancia una moneta equiprobabile: se viene testa, inserisci v in C ; altrimenti (viene croce), inserisci v in $V - C$. (2) Inizializza T all'insieme vuoto. Per ogni arco $(v, w) \in E$, tale che $v \in C$ e $w \in V - C$, aggiungi (v, w) all'insieme T . Restituisci C e T come soluzione approssimata.
11. [Famiglia di funzioni hash uniformi] Mostrare che la famiglia di funzioni hash $H = \{h(x) = ((ax + b) \% p) \% m\}$ è (quasi) "pairwise independent", dove $a, b \in [m]$ con $a \neq 0$ e p è un numero primo sufficientemente grande ($m + 1 \leq p \leq 2m$). La pairwise independent è la k -wise independent vista a lezione dove $k = 2$.
12. [Count-min sketch: prodotto scalare] Mostrare e analizzare come utilizzare il paradigma del count-min sketch per approssimare il prodotto scalare $\sum_{k=1}^n F_a[k] * F_b[k]$.
13. [Count-min sketch: interval query] Mostrare e analizzare come utilizzare il paradigma del count-min sketch per rispondere in modo approssimato alle interval query (i, j) per calcolare $\sum_{k=i}^j F[k]$. Suggerimento: ridurre tale query alla stima di $t \leq 2 \log n$ contatori c_1, c_2, \dots, c_t . Notare che per avere una probabilità al più δ che $\sum_{l=1}^t c_l > \sum_{k=i}^j F[k] + 2\epsilon \log n \|F\|$, non basta dire che è al più δ la probabilità di errore di ciascun contatore c_l : occorre infatti vedere ciascun contatore come il vero valore più il residuo, come visto nel count-min sketch di base, e sommare assieme i t valori ottenendo V e i t

residui ottendo X e poi procedere con la disuguaglianza di Markov per ottenere che δ è un limite superiore alla probabilità che $V + X > \sum_{k=i}^j F[k] + 2\epsilon \log n \|F\|$ (cioè vedere la somma dei contatori come fosse un unico meta-contatore).

14. [Count-min sketch: estensione] Estendere l'analisi vista a lezione permettendo di incrementare e decrementare i contatori con valori arbitrari. Invece di avere le operazioni $F[i]++$ e $F[i]--$, l'elemento generico dello stream contiene una coppia (i, v) dove i è una item e v è un intero qualsiasi: l'operazione diventa $F[i] = F[i] + v$. Notare che l'incremento e il decremento unitari di $F[i]$ possono essere ora visti come $(i, 1)$ e $(i, -1)$.
15. [Ordinamento in memoria esterna] Nel modello EMM (external memory model), mostrare come implementare il k -way merge ($(k+1)B \leq M$), ossia la fusione di k sequenze individualmente ordinate e di lunghezza totale N , con costo I/O di $O(N/B)$ dove B è la dimensione del blocco. Minimizzare e valutare il costo di CPU. Analizzare il costo del merge sort (I/O complexity, CPU complexity) che utilizza tale k -way merge.
16. [Limite inferiore per la permutazione] Estendere l'argomentazione utilizzata per il limite inferiore al problema dell'ordinamento in memoria esterna a quello della permutazione: dati N elementi e_1, e_2, \dots, e_N e un array π contenente una permutazione degli interi in $[1, 2, \dots, N]$, disporre gli elementi secondo la permutazione in π . Dopo tale operazione, la memoria esterna deve contenerli nell'ordine $e_{\pi[1]}, e_{\pi[2]}, \dots, e_{\pi[N]}$.
17. [Limite inferiore per la ricerca] Considerare il miglior algoritmo A possibile (passato, presente o futuro) che effettua la ricerca di una chiave mediante confronti in un insieme statico S di N elementi che obbediscono a una relazione d'ordine $<$. L'algoritmo può effettuare un'elaborazione preliminare (preprocessing) per organizzare a suo piacimento le chiavi di S utilizzando la relazione $<$. Mostrare che la successiva ricerca di una qualsiasi chiave effettuata da A mediante confronti con $<$ richiede $\Omega(\log_B N)$ I/O (trasferimenti di blocchi di taglia B) nel modello EMM.
18. [Esecuzione della permutazione in memoria esterna] Dati due array A e π , dove A contiene N elementi (non importa quali) e π contiene una permutazione di $\{1, \dots, N\}$, descrivere e analizzare nel modello EMM un algoritmo ottimo per costruire un terzo array C di N elementi tale che $C[\pi[i]] = A[i]$ per $1 \leq i \leq N$.
19. [MapReduce] Utilizzare il paradigma scan&sort (per esempio, mediante la MapReduce) per calcolare la distribuzione dei gradi in ingresso delle pagine Web, ipotizzando di avere già tali pagine a disposizione.
20. [Ricerca nel suffix array in memoria esterna] Ipotizzare di avere già costruito il suffix array SA per il testo T , entrambi di N elementi. Progettare e analizzare algoritmi efficienti per rispondere alle seguenti query di una stringa pattern P che viene fornita ogni volta on-line da un utente (dove P può variare ad ogni query mentre T rimane lo stesso e possiamo quindi sfruttare SA): (1) verifica se P occorre in T (un booleano); (2) conta il numero di occorrenze di P in T (un intero non negativo); (3) elenca tutte

le posizioni di T in cui P occorre (una lista di interi non negativi). Motivare che (3) implica (2) implica (1). Analizzare nell'EMM il costo dell'algoritmo proposto nelle seguenti situazioni: entrambi T e SA sono in memoria principale; solo uno tra T e SA è in memoria principale mentre l'altro è in memoria secondaria; entrambi T e SA sono in memoria secondaria.

21. [Costruzione del suffix array in memoria esterna] Utilizzando la costruzione del suffix array basata sul mergesort e la tecnica DC3 vista a lezione, progettare un algoritmo per EMM per costruire il suffix array di un testo di N simboli che abbia la stessa complessità del mergesort di N elementi in EMM.
22. [Navigazione implicita in vEB] Dato un albero completo memorizzato secondo il layout cache-oblivious (CO) di van Emde Boas (vEB) in modo implicito, ossia senza l'ausilio di puntatori (come succede nel classico heap binario implicito), trovare la regola per navigare in tale albero durante la ricerca di una chiave senza usare puntatori espliciti.
23. [Longest common prefix] Estendere la costruzione del suffix array SA basata sul mergesort e sulla tecnica DC3 vista a lezione, in modo da calcolare anche l'array LCP tale che $LCP[i]$ contiene il prefisso comune più lungo tra i due suffissi indicati da $SA[i]$ e $SA[i + 1]$, per $1 \leq i \leq N - 1$. La complessità asintotica deve rimanere la stessa del mergesort. Nota: è sufficiente la complessità nel modello RAM (ma va bene anche nell'EMM, per chi vuole).
24. [Costruzione del suffix tree] Ipotizzando di avere a disposizione il testo T , il suo suffix array SA e l'array LCP , mostrare come costruire il suffix tree per T in tempo lineare nel modello RAM. Suggerimento: come visto a lezione, il suffix tree viene costruito in preordine, con una "frontiera" che si espande verso destra man mano che le foglie vengono inserite da sinistra verso destra (la prima foglia inserita è quella che corrisponde al terminatore \$). Occorre scandire da sinistra verso destra SA per le foglie attaccandole al nodo interno di profondità in caratteri indicata in LCP : tale nodo interno è un antenato della foglia inserita al passo precedente e va creato se non esiste già.
25. [Lista invertita compressa] Data una sequenza crescente di n interi positivi i_1, i_2, \dots, i_n , contenuta in una lista invertita, la sua rappresentazione compressa differenziale è la sequenza S di $|S|$ bit ottenuti concatenando $\gamma(i_1), \gamma(i_2 - i_1), \gamma(i_3 - i_2), \dots, \gamma(i_n - i_{n-1})$, dove $\gamma(x)$ rappresenta il gamma code di Elias per la codifica istantanea dell'intero $x \geq 1$ in $2 \lfloor \log_2 x \rfloor + 1$ bit. Mostrare come aggiungere un'opportuna directory di spazio $O(|S|)$ bit per poter accedere velocemente, dato $j \in [2..n]$, alla codifica $\gamma(i_j - i_{j-1})$. Estendere tale approccio per ottenere il valore di i_j .
26. [Parsing di LZ77] Mostrare come utilizzare il suffix tree di un testo T per ottenere il parsing di T in una sequenza di frasi, come richiesto dall'algoritmo LZ77 di Lempel e Ziv.