Progettazione di algoritmi

Discussione dell'esercizio [file]

Si hanno n file di dimensioni $s_1, \dots, s_n$ e un disco di capacità C. Vogliamo trovare un sottoinsieme dei file che può essere memorizzato sul disco e che massimizza lo spazio usato. Più precisamente, vogliamo trovare un sottoinsieme F dei file tale che (1) $S(F) \leq C$ e (2) per ogni sottoinsieme di file X con $S(X) \leq C$, si ha $S(X) \leq S(F)$, dove $S(X)$ è la somma delle dimensioni dei file in X.

Potremmo tentare di scegliere i file in ordine di dimensione crescente, cioè scegliamo prioritariamente i file più piccoli:

```
FILE_MIN(S: array delle dimensioni degli n file, C: capacità disco)
    F <- insieme vuoto
    Ordina l'array S in modo non decrescente
    R <- C
    FOR i <- 1 TO n DO
        IF S[i] <= R THEN
            F <- F ∪ {i}
            R <- R - S[i]
    RETURN F
```

L'algoritmo è corretto? Un momento di riflessione è sufficiente per intuire che ci sono istanze in cui non trova l'ottimo. Infatti, potrebbe esserci un file piccolo che una volta scelto impedisce di sceglierne uno più grande che sfrutta meglio lo spazio del disco. Ad esempio, se i file hanno dimensioni 1, 10 e il disco ha capacità 10, FILE_MIN sceglie il primo file di dimensione 1 e deve lasciare fuori il file da 10 che invece è la scelta ottima. Su questa istanza il fattore di approssimazione è $10/1 = 10 >> 1$. Quindi FILE_MIN ha fattore di approssimazione almeno pari a 10. Può essere peggiore? Sicuramente sì. Ad esempio con file di dimensioni 1, 100 e disco di capacità 100, il fattore sale a 100. In generale, se i file hanno dimensioni 1, M e il disco ha capacità M, il fattore di approssimazione è M. Quindi FILE_MIN non ha nessun limite al fattore di approssimazione che può essere peggiore di qualsivoglia costante.

Consideriamo allora l'algoritmo speculare FILE_MAX che sceglie i file in ordine di dimensione descrescente:

```
FILE_MAX(S: array delle dimensioni degli n file, C: capacità disco)
    F <- insieme vuoto
    Ordina l'array S in modo non crescente
    R <- C
    FOR i <- 1 TO n DO
        IF S[i] <= R THEN
            F <- F ∪ {i}
```

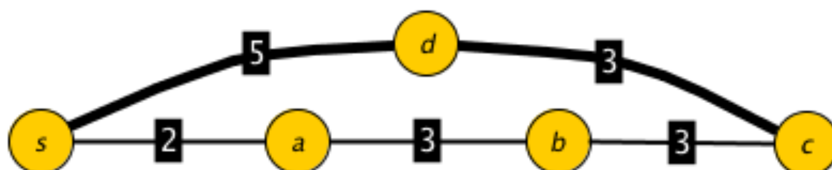
```
            R <- R - S[i]
    RETURN F
```

L'algoritmo è corretto? Anche per FILE_MAX la risposta è negativa. Infatti, consideriamo tre file di dimensioni 8, 6, 4 e un disco di capacità 10. L'algoritmo FILE_MAX sceglie il primo file e poi non può aggiugerne altri e quindi usa spazio 8, mentre la soluzione ottima è scegliere i file da 6 e da 4 usando spazio 10. Già questo primo esempio ci dice che il fattore di approssimazione di FILE_MAX è maggiore o uguale a 10/8 = 5/4 > 1. Può essere peggiore? Sempre con tre file potremmo avere le dimensioni 6, 5, 5 e sempre disco da 10. In questo caso il fattore è 10/6 = 5/3 > 5/4. Dovrebbe essere chiaro ormai lo schema di queste istanze: disco di capacità $2M$ e file di dimensioni $M+1$, M, M. In questi casi il fattore di approssimazione è $2M/(M+1)$. Facendo crescere M possiamo farlo avvicinare a 2 quanto vogliamo. Può essere peggiore di 2? Se la somma delle dimensioni di tutti i file non supera la capacità del disco allora FILE_MAX, ovviamente, trova l'ottimo. Supponiamo allora che la somma delle dimensioni superi la capacità del disco (assumiamo che i file di dimensione maggiore della capacità non ci siano). Quando l'algoritmo FILE_MAX incontra il primo file che non può essere scelto quanto può essere grande la capacità residua R? Non può essere $R \geq C/2$ perchè se lo fosse il primo file che non può essere scelto avrebbe dimensione $s > C/2$ e quindi il file precedente che invece è stato scelto avrebbe dimensione $s' \geq s > C/2$. Ma allora la capacità residua sarebbe $R \leq C - s' < C/2$ in contraddizione con l'ipotesi $R \geq C/2$. Sapendo che quindi i file scelti usano almeno la metà del disco, abbiamo che la soluzione prodotta da FILE_MAX ha valore S pari ad almeno $C/2$, inoltre, siccome il valore ottimo non può superare la capacità del disco, il fattore di approssimazione è minore di $C/(C/2) = 2$. Concludiamo che il fattore di approssimazione di FILE_MAX è 2 (più precisamente arbitrariamente vicino a 2).

Esercizi di preparazione alla prova intermedia

Esercizio [sempre connesso] Descrivere un algoritmo che, dato un grafo non diretto e connesso G, trova un ordinamento dei nodi $v_1,v_2,\ldots,v_n$ tale che eliminando i nodi in quell'ordine lascia sempre il grafo connesso. L'algoritmo deve avere complessità $O(n + m)$.

Esercizio [super-minimo] Un cammino da un nodo u a un nodo v si dice *super-minimo* se ha peso minimo tra tutti i cammini da u a v e inoltre tra tutti i cammini di peso minimo da u a v ha il minimo numero di archi. Dato un grafo pesato G tale che i pesi sono tutti interi positivi, si vogliono trovare i cammmini super-minimi da un nodo s. Ad esempio, nel grafo qui sotto



vogliamo il cammino (s, d, c) e non quello di pari peso ma più lungo (s, a, b, c). Mostrare

come modificare i pesi degli archi del grafo *G* ottenendo un grafo *G'* in modo tale che applicando Dijkstra a *G'* si ottengono i cammini super-minimi di *G*. Motivare la risposta.

Esercizio [arco massimo] Dato un grafo *G* pesato e connesso, dimostrare che se un arco *e* appartiene a un ciclo *C* in cui *e* ha peso massimo rispetto agli archi di *C*, allora esiste un MST di *G* che non contiene l'arco *e*.

Esercizio [Universa] La rete viaria della città Universa consiste di varie piazze collegate fra loro da strade. La particolarità di Universa è che tutte le strade sono a senso unico. La piazza Unica è la piazza più importante e si sa che essa è raggiungibile da qualsiasi altra piazza della città. Il sindaco vuole selezionare un insieme di strade la cui lunghezza totale sia minima e che permettano da una qualsiasi piazza di raggiungere la piazza Unica. Gli urbanisti pensano di trovare questo insieme di strade tramite un algoritmo di Prim modificato che partendo dalla piazza Unica ad ogni passo sceglie sempre la strada entrante di lunghezza minima. L'algoritmo degli urbanisti produce sempre un insieme di strade che garantisce la raggiungibilità della piazza Unica da una qualsiasi altra piazza? L'insieme selezionato ha lunghezza totale minima? **Motivare bene le risposte**.

Esercizio [sicurezza] Abbiamo una collezione di *n* server che devono essere collegati in rete fra loro. Ci sono *m* possibili collegamenti tra i server e ognuno di questi è o sicuro o insicuro (cioè il traffico sul collegamento potrebbe essere intercettato da persone non autorizzate). Vogliamo scegliere un insieme di collegamenti che permette di connettere tutti i server fra loro e allo stesso tempo minimizzi l'utilizzo di collegamenti insicuri. Descrivere un algoritmo che date le specifiche di tutti i collegamenti possibili (e della loro sicurezza) ritorni un sottoinsieme di questi che garantisca la connessione minimizzando l'uso di collegamenti insicuri. L'algoritmo deve avere complessità $O(n + m)$.

Esercizio [univoco] In un grafo diretto *G* un sottoinsieme *A* degli archi è detto *univoco* se non contiene 2 o più archi uscenti dallo stesso nodo. Descrivere un algoritmo greedy che preso in input un grafo diretto e pesato *G*, trovi un insieme univoco *A* di archi di *G* di peso massimo. Provare la correttezza dell'algoritmo proposto e valutare la complessità di una sua efficiente implementazione.

Esercizi per casa

Esercizio [cicli diretti] Si propone la seguente BFS modificata per determinare se esistono cicli in un grafo diretto *G*: durante la BFS se si incontra un adiacente (uscente) *v* del nodo corrente *u* tale che *v* è un antenato di *u* nell'albero della BFS allora termina ritornando `true`, altrimenti quando termina ritorna `false`. Supponendo tutti i nodi di *G* siano raggiungibili dal nodo di partenza *s*, questa BFS modificata determina sempre in modo corretto se ci sono cicli? **Motivare bene la risposta**.

Esercizio [resto] Supponiamo di avere monete con i seguenti tagli: 1,4,6,8,10. Consideriamo il seguente algoritmo che conta il numero di monete per formare un resto *r* preso in input:

```
RESTO(r: intero positivo)
```

```
    m <- 0
    WHILE r <> 0 DO
        sia n il massimo taglio in {1,4,6,8,10} minore o uguale a r
        r <- r - n
        m <- m + 1
    RETURN m
```

Dire se l'algoritmo ritorna sempre il numero minimo di monete, in caso affermativo dimostrare la correttezza altrimenti fornire un controesempio.

Esercizio [k-MST] Si consideri il seguente problema: dato un grafo connesso e pesato *G*, un suo nodo *s* ed un intero *k*, trovare un sottoalbero di peso minimo di *G* che contiene *s* ed ha esattamente *k* nodi. Quando *k* è uguale a *n*, il problema diventa equivalente a trovare un MST di *G*. Così, per risolvere il problema, in analogia con l'algoritmo di Prim, viene proposto il seguente algoritmo:

```
KMST(G: grafo connesso e pesato, s: nodo, k: intero)
    T <- insieme vuoto
    A <- {s}
    WHILE |A| < k DO
        Sia {u, v} un arco di peso minimo tra quelli con u in A e v non di A
        T <- T ∪ {{u, v}}
        A <- A ∪ {v}
    RETURN T
```

a) Provare che l'algoritmo produce un albero di *k* nodi.

b) Descrivere un'implementazione dell'algoritmo che abbia complessità O(*kn*).

c) Provare che l'algoritmo è corretto o fornire un controesempio.

Esercizio [teatro] All'acquisto dei biglietti per un'importante prima teatrale sono interessati *m* gruppi di persone. Ciascun gruppo ha intenzione di acquistare i biglietti solo se questi sono a sufficienza per tutti i membri del gruppo. Sapendo che il teatro ha una capienza di soli *n* posti bisogna selezionare i gruppi a cui vendere i biglietti in modo da massimizzare il numero di biglietti venduti. Viene proposto il seguente algoritmo greedy:

```
A <- gli m gruppi
SOL <- insieme vuoto
WHILE A non è vuto DO
    Estrai da A un gruppo g di cardinalità t massima
    IF n − t ≥ 0 THEN
        SOL <- SOL ∪ {g}
        n <- n − t
OUTPUT SOL
```

Provare la correttezza dell'algoritmo proposto o fornire un controesempio.

Soluzioni

Soluzione esercizio [sempre connesso] Una soluzione molto semplice consiste nel considerare un albero di visita del grafo e eliminare i nodi iniziando da una foglia dell'albero e ad ogni passo eliminare una delle foglie dell'albero rimasto. In questo modo siamo sicuri che le eliminazioni non disconnettono il grafo perché non disconnettono l'albero. Quindi è sufficiente modificare una DFS per produrre la lista dei nodi nell'ordine in cui vanno eliminati:

```
VIS: array per marcare i nodi visitati, inizializzato a 0

ELIM(G: grafo non diretto e connesso, u: nodo, VIS: array, L: lista di nodi)
   VIS[u] <- 1
   FOR ogni adiacente v di u DO
       IF VIS[v] = 0 THEN
           ELIM(G, v, VIS, L)
   L.append(u)
```

La chiamata iniziale sarà `ELIM(G, u, VIS, L)`, dove `u` è un nodo qualsiasi di `G` e `L` è la lista vuota. Siccome un nodo u viene aggiunto alla lista L solo quando la visita da u termina, siamo sicuri che in L sono già stati aggiunti (e quindi vengono prima di u nell'ordinamento) tutti i nodi del sottoalbero di u. Perciò quando u viene eliminato sono già stati eliminati tutti i nodi del suo sottoalbero di visita e quindi u è una foglia dell'albero residuo. Chiaramente l'algoritmo ha complessità pari a quella di una DFS, cioè $O(n + m)$.

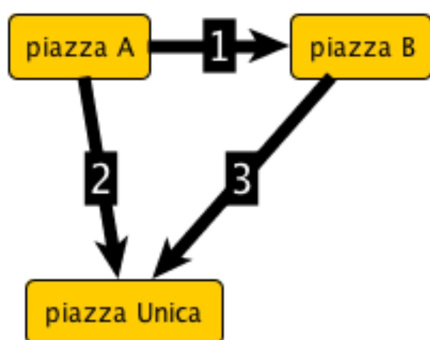
Soluzione esercizio [super-minimo] Dobbiamo modificare i pesi del grafo G ottenendo G' in modo tale che un cammino C di pari peso di un altro cammino C' in G ma che ha un minor numero di archi di C' pesi in G' meno di C, cioè $p(C) = p(C')$ ma $p'(C) < p'(C')$. Questo significa che il numero di archi di un cammino deve contribuire al peso di un cammino in G'. Il peso di un cammino in G' dovrebbe quindi essere formato da due contributi: il peso che ha in G e il suo numero di archi. Ovviamente non possiamo semplicemente sommare questi due contributi perchè altrimenti potremmo far pesare di meno un cammino breve che pesa molto rispetto a un cammino lungo che pesa poco. Solamente a parità di peso (in G), il numero di archi del cammino deve fare la differenza. Osserviamo che al massimo il contributo relativo al numero di archi è pari a n - 1, dato che un cammino può avere al massimo n - 1 archi. Sappiamo che tutti i pesi in G sono ineteri positivi, così se moltiplichiamo i pesi degli archi per n avremo che anche i pesi dei cammini sono moltiplicati per n. Questo significa che due cammini C e C' che avevano pesi "consecutivi", cioè $p(C') = p(C) + 1$, adesso avranno pesi $p'(C') = np(C') = np(C) + n = p'(C) + n$. Quindi sono distanziati di n e abbiamo lo spazio per aggiungere il contributo relativo al numero di archi. In conclusione per ogni arco e, il nuovo peso è $p'(e) = np(e) + 1$. Così per un qualsiasi cammino C, $p'(C) = np(C) + L(C)$, dove $L(C)$ è il numero di archi di C. In questo modo abbiamo la garanzia che per qualsiasi due cammini C e C' con $p(C) < p(C')$ si ha $p'(C) = np(C) + L(C) < np(C) + n + L(C') \leq p'(C')$. E se invece hanno lo stesso peso in G, in G' quello con il minor

numero di archi pesa di meno.

Soluzione esercizio [arco massimo] Sia $e = \{u, v\}$ l'arco di peso massimo nel ciclo C. Sia T un qualsiasi MST che contiene l'arco e. Se eliminiamo da T l'arco e, dividiamo T in due parti disgiunte T_1 e T_2. Chiaramente un estremo di e appartiene a una delle parti e l'altro appartiene all'altra. Supponiamo che u appartiene a T_1 e v a T_2. Siccome il ciclo C contiene un cammino da u a v deve necessariamente contenere un arco e' che collega un nodo di T_1 a un nodo di T_2 (dato che ogni nodo di G o è in T_1 o in T_2). Definiamo quindi $T' = (T - \{e\}) \cup \{e'\}$. Chiaramente T' è un albero di copertura, inoltre $p(T') = p(T) - p(e) + p(e') \leq p(T)$, dato che e' è nel ciclo C e quindi $p(e') \leq p(e)$. Perciò T' è un MST che non contiene l'arco e.

Soluzione esercizio [Universa] La prima domanda chiede se l'algoritmo degli urbanisti (il Prim modificato) produce sempre un insieme di strade che permettono di raggiungere la piazza Unica da una qualsiasi piazza. La risposta è sì. Infatti, supponiamo per assurdo che esista una piazza z da cui non è possibile raggiungere la piazza Unica con le strade selezionate. Consideriamo il percorso P che per ipotesi esiste e che va da z verso la piazza Unica e sia z' la prima piazza attraversata da P che non è collegata alla piazza Unica tramite le strade selezionate. Questo significa che nessuna strada selezionata tocca z'. Dalla piazza successiva z'' di z' in P è invece possibile raggiungere la piazza Unica tramite le strade selezionate ma allora la strada (z'', z') poteva essere scelta dall'algoritmo e la sola ragione perchè non è stata scelta è che qualche altra strada che tocca z' è stata scelta, ma questo è in cotraddizione con l'assunzione che da z' la piazza Unica non è raggiungibile tramite le strade selezionate.

La seconda domanda chiede se le strade selezionate dall'algoritmo hanno lunghezza totale minima tra tutte le possibili scelte di strade che garantiscono la raggiungibilità della piazza Unica. Consideriamo il seguente esempio:



L'algoritmo seleziona le strade (piazza A, piazza Unica) e (piazza B, piazza Unica) di lunghezza totale 5, mentre la soluzione ottima sono le strade (piazza B, piazza Unica) e (piazza A, piazza B) di lunghezza totale 4. Quindi l'algoritmo non è corretto.

Soluzione esercizio [sicurezza] Vogliamo trovare un albero di copertura del grafo di tutti i possibili collegamenti tra server che minimizzi il numero di collegamenti insicuri. Per fare ciò possiamo assegnare agli archi che corrispondono a collegamenti sicuri peso 0 e agli archi che corrispondono a collegamenti insicuri il peso 1. In questo modo il peso di un

albero di copertura è proprio il numero di archi insicuri contenuti nell'albero. Un qualsiasi algoritmo per trovare un MST risolve il problema. Però dovremmo farlo in tempo O($n + m$) e la migliore implementazione che conosciamo (dell'algorimto di Prim) garantisce solamente tempo O(($n + m$)log$n$). Il nostro grafo ha pesi estremamente semplice o 0 o 1. Questo significa che i costi di aggiunta di un nuovo nodo all'albero possono essere solamente 0 o 1. Perciò possiamo mantenere due liste: `L0` per mantenere i nodi con costo 0 e `L1` per i nodi con costo 1. Inoltre dobbiamo mantenere anche un array che ci permette di conoscere il costo di un qualsiasi nodo e la sua eventuale posizione nella lista `L1`. Per rendere efficiente il passaggio dalla lista `L1` alla lista `L0` conviene che `L1` sia una lista bidirezionale. Ecco quindi una implementazione dell'algoritmo di Prim specializzata per questo tipo di grafi:

```
SICUR_PRIM(G: grafo non diretto, pesato (con pesi 0,1) e connesso)
    P: vettore dei padri, inizializzato a 0
    A: array dei costi e delle posizioni nella lista L1, A[u].c costo di u e
       A[u].p posizione in L1 (o NULL se non in L1)
    Inizializza l'array A con i costi a 2 e le posizioni a NULL
    A[s].c <- 0       /* s è un nodo qualsiasi di G */
    P[s] <- s
    L0 <- lista inizializzata con s
    L1 <- lista vuota
    WHILE L0 e L1 non sono entrambe vuote DO
        IF L0 non è vuota THEN
            u <- estrai il primo elemento di L0
        ELSE
            u <- estrai il primo elemento di L1
        FOR ogni adiacente v di u DO
            IF P[v] = 0 AND p{u, v} < A[v].c THEN
                P[v] <- u
                IF p{u, v} = 0 THEN
                    L0.append(v)
                    IF A[v].c = 1 THEN
                        Rimuovi v da L1 conoscendo la posizione A[v].p
                        A[v].p <- NULL
                    A[v].c <- 0
                ELSE
                    A[v].p <- L1.append(v)
                    A[v].c <- 1
    RETURN P
```

A differenza dell'implementazione generale dell'algoritmo di Prim, l'elaborazione di un arco del grafo prende tempo O(1), anziché O(log$n$). L'inizializzazione richiede tempo O(n), quindi la complessità è O($n + m$).

Soluzione esercizio [univoco] L'algoritmo greedy più semplice che viene in mente consiste nello scegliere ad ogni passo l'arco uscente di peso massimo tra quelli che possono essere scelti (cioè che non escono da nodi da cui escono archi già scelti).

```
UNIVOCO(G: grafo diretto e pesato)
    A <- insieme vuoto
    WHILE ci sono archi che possono essere aggiunti ad A DO
        Trova un arco e di peso massimo tra quelli che possono essere
aggiunti a A
        A <- A ∪ {e}
    RETURN A
```

Chiaramente l'algoritmo UNIVOCO ritorna sempre un insieme univoco di archi. Vogliamo dimostrare che tra tutti gli insiemi univoci di archi, l'algoritmo ne ritorna uno di peso totale massimo. Cerchiamo di usare lo schema generale per dimostrare la correttezza degli algoritmi greedy. Sia A_h l'insieme `A` al termine dell'h-esima iterazione del `WHILE`. A_0 è l'insieme iniziale. Sia k il numero di passi eseguiti.

Per ogni h = 0,1,..., k, esiste una soluzione ottima A^* che contiene A_h.

Dimostrazione Per induzione su h. Per $h = 0$ è banalmente vero. Supponiamolo vero per $h \geq 0$ e proviamolo per $h + 1$. Per ipotesi induttiva esiste una soluzione ottima A^* che contiene A_h. Se A^* contiene anche A_{h+1}, abbiamo fatto. Altrimenti, sia e l'arco scelto nell'(h+1)-esima iterazione. Chiaramente, e non è in A^*, vogliamo quindi modificare A^* trasformandola in un'altra soluzione ottima che contenga A_{h+1}. Se tentiamo di aggiungere e ad A^*, c'è sicuramente un'altro arco e' in A^* che esce dallo stesso nodo u da cui esce l'arco e. Definiamo $A^\# = (A^* - \{e'\}) \cup \{e\}$. Ovviamente $A^\#$ è un insieme univoco. L'arco e' non può essere in A_h perchè altrimenti l'arco e non poteva essere scelto. Quindi e' era tra gli archi che potevamo essere scelti nell'(h+1)-iterazione. Siccome è stato scelto e, deve essere che $p(e) \geq p(e')$. Perciò il peso di $A^\#$ è almeno pari a quello di A^* e quindi $A^\#$ è una soluzione ottima che contiene A_{h+1}.

Per quanto riguarda un'implementazione efficiente, osserviamo che l'algoritmo è equivalente a scegliere per ogni nodo *u* del grafo un arco uscente da *u* di peso massimo.

```
UNIVOCO(G: grafo diretto pesato)
    A <- insieme vuoto
    FOR ogni nodo u DO
        IF u ha archi uscenti THEN
            max <- primo arco uscente da u
            FOR ogni arco uscente x da u DO
```

```
                IF p(x) > p(max) THEN
                    max <- x
            A <- A ∪ {max}
    RETURN A
```

Ogni nodo viene considerato e poi l'elaborazione di ogni arco richiede tempo O(1), quindi la complessità è O(n + m).