Lezione 16

La lezione è centrata su un esercizio per impratichirsi sui seguenti algoritmi:

- algoritmo greedy per il problema dello zaino
- algoritmo greedy per il problema dell'albero ricoprente minimo

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Affrontiamo lo stesso problema della scorsa lezione: scrivere un programma `knapsack.c` che risolva il problema dello zaino. Questa volta, la soluzione non sarà esatta, ma euristica, cioè priva di garanzia di ottimalità. Gli algoritmi che realizzeremo sono *greedy*, cioè basati sull'idea di partire da un sottoinsieme vuoto e aumentarlo di un elemento alla volta, scelto come il più "promettente" fra quelli che non violano il vincolo di capacità. I formati di ingresso e uscita rimangono quelli già definiti.

Traccia della risoluzione

Per cominciare, si possono scorrere i lucidi che ricordano la definizione di Ottimizzazione Combinatoria e di algoritmo esaustivo, e poi tratteggiano lo schema generale di algoritmo *greedy* e introducono il caso specifico del problema dello zaino unitario (con oggetti di volume uniforme), osservando che in esso l'algoritmo *greedy* sembra funzionare, mentre chiaramente non funziona per il problema dello zaino generico.

L'esercizio parte dal file `knapsack0.c` da cui già comcinciava la scorsa lezione. Siamo quindi in grado di caricare istanze del problema dello zaino, stampare soluzioni, ecc...

Prima fase (`knapsack-greedy1.c` Questa fase dell'esercizio dovrebbe realizzare l'algoritmo *greedy* base, che

1. parte da un sottoinsieme vuoto
2. valuta se esistono elementi aggiungibili al sottoinsieme corrente nel rispetto del vincolo di capacità;
3. determina l'oggetto di valore massimo fra quelli aggiungibili;
4. lo aggiunge alla soluzione corrente e torna al punto 2.

In realtà, gestire l'insieme degli oggetti aggiungibili non è la soluzione più efficiente, perché comporta di scorrere ad ogni passo l'insieme $B \setminus x$ degli oggetti esterni per sfrondarlo degli elementi che sono diventati inammissibili dopo l'ultima estensione di x. È in fondo più semplice gestire direttamente $B \setminus x$ e valutare di volta in volta se l'oggetto di valore massimo è aggiungibile o no: nel primo caso, si procede come sopra, nel secondo, ci si limita ad eliminarlo. Quindi l'algoritmo che realizzeremo:

1. parte da un sottoinsieme vuoto
2. determina l'oggetto di valore massimo fra quelli esterni;

3. se tale elemento è aggiungibile, procede ad aggiungerlo alla soluzione corrente; altrimenti, lo elimina; in entrambi i casi, torna al punto 2.

Il problema fondamentale di questo algoritmo è determinare in modo efficiente l'oggetto di valore massimo. Per farlo, si possono mettere i premi in un *max-heap*. Ma questa struttura restituisce solo il valore massimo, mentre a noi serve conoscere l'indice dell'oggetto corrispondente, per aggiungerlo al sottoinsieme corrente. Una prima soluzione è costruire un vettore di "oggetti", cioè di strutture contenenti indici, valori e volumi, e ordinare tale vettore per valori crescenti. Si noti che gli indici sono necessari perché, una volta scambiati gli oggetti rispetto all'ordinamento iniziale, non si ha più modo di indicarli per nome. Alternativamente, si possono gestire tre vettori di interi (uno per gli indici, uno per i valori e uno per i volumi), e ordinare il secondo, mantenento però gli altri due ordinati in parallelo, attraverso l'esecuzione di scambi coordinati con quelli eseguiti sul secondo.

Per esempio, supponendo di partire dai vettori:

```
ind = [ 1 2 3 4 5 6 ]
phi = [ 7 2 4 5 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

la costruzione del *max-heap* comporta di scorrere le posizioni 3, 2 e 1 confrontando ciascuna con i due figli:

- la posizione 3 è già corretta, dato che `phi[3] > phi[6]` e non esiste `phi[7]`;
- la posizione 2 non è corretta, dato che `phi[2] < phi[5] < phi[4]`, per cui bisogna scambiare 2 e 5 (e operare ricorsivamente su 5, dove non si fa nulla);
- la posizione 1 è corretta, dato che `phi[1] > phi[2] = phi[3]`.

ottenendo i vettori

```
ind = [ 1 4 3 2 5 6 ]
phi = [ 7 5 4 2 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

Scambiare gli elementi di tutti i vettori (o gli elementi dell'unico vettore di strutture) può essere pesante, e in più distrugge l'ordinamento iniziale dei dati, che potrebbe essere utile conservare per altri motivi.

È però possibile, e preferibile, scambiare solo le posizioni nel vettore `ind`

```
ind = [ 1 4 3 2 5 6 ]
phi = [ 7 2 4 5 4 1 ]
v   = [ 5 3 2 3 1 1 ]
```

osservando che la proprietà *max-heap* vale **indirettamente** sul vettore `phi`: basta che per identificare i nodi figli si usino, al posto degli indici diretti, gli indici indiretti contenuti in `ind`. Per esempio, non diciamo più che l'elemento di indice 2 ha come figli gli elementi di indice 4 e 5 e ha valore non inferiore ad essi, ma che l'elemento di indice `ind[2]` (cioè 4) ha come figli gli elementi di indice `ind[4]` (cioè 2) e `ind[5]` (cioè 5) e ha valore non inferiore ad essi: `phi[ind[2]] >= phi[ind[4]]` (cioè $\phi_4 = 5 \geq 2 = \phi_2$) e `phi[ind[2]] >= phi[ind[5]]` (cioè $\phi_4 = 5 \geq 4 = \phi_5$). Per quanto riguarda la ricerca del massimo, la non diciamo più che la radice corrisponde all'indice 1, così che `phi[1]` è il valore massimo, ma che la radice corrisponde all'indice `ind[1]` (qui casualmente ancora pari ad 1) e quindi `phi[ind[1]]` è il valore massimo. Si tratta cioè di accedere ai vettori con i dati attraverso gli indici del vettore `ind`. È ovvio che sarebbe del tutto equivalente avere invece un vettore di puntatori, anche se questo permetterebbe l'accesso solo a uno dei due vettori con i dati (valori e volumi); sarebbe allora preferibile raccogliere i dati in un solo vettore di strutture, come già proposto più sopra.

Questo dà luogo a un "*heap* indiretto", ma si tratta di un'idea del tutto generale, che può essere facilmente adattata a tutti gli algoritmi di ordinamento visti nel corso, anche se non usano *heap*:

- il vettore che contiene i dati è intatto, e resta disordinato;
- gli scambi avvengono sul vettore degli indici;
- i confronti fra elementi non si fanno direttamente sul vettore dei dati, ma indirettamente sul vettore dei dati passando per il vettore degli indici (`phi[ind[...]]` anziché `phi[...]`).

Dal punto di vista della complessità, l'ordine asintotico non cambia; la lettura è più complicata, e leggermente più lenta, ma i dati non subiscono modifiche e gli scambi sono più veloci.

Inoltre, introducendo più vettori `ind` (di interi o di puntatori) si possono ottenere ordinamenti diversi degli stessi dati. Per esempio, si potrebbero ordinare gli oggetti sia rispetto ai valori sia rispetto ai volumi. Questo consentirebbe di ridiscutere l'algoritmo, consentendo ad ogni aggiunta la cancellazione degli elementi esterni a x che risultano non più aggiungibili: essi infatti sarebbero gli elementi di volume massimo. Si può trattare di un buon esercizio, che comporta la gestione di due *max-heap* sullo stesso insieme e l'individuazione dell'elemento di valore massimo, ma talvolta anche di quello di volume massimo, e la sua cancellazione da entrambi gli *heap*.

Realizzato l'algoritmo, si può facilmente constatare che le sue prestazioni sono piuttosto cattive dal punto di vista della qualità della soluzione (il tempo di calcolo è ovviamente rapidissimo, dato che la complessità nel caso pessimo è facilmente stimabile in $\Theta(n \log n)$).

Seconda fase (`knapsack-greedy2.c`) Il motivo principale del cattivo funzionamento dell'algoritmo *greedy* banale è che non tiene conto in alcun modo del vincolo di capacità, per cui aggiunge al sottoinsieme x corrente oggetti di valore alto, ma anche di volume eccessivo, che esauriscono in fretta la capacità dello zaino. Per ovviare, si può cambiare il criterio di scelta dell'elemento da aggiungere, usando il *valore unitario* ϕ_i/v_i anziché il valore ϕ_i. Questo corrisponde a cercare contemporaneamente di massimizzare il valore e minimizzare il volume. Corrisponde anche a cercare di risolvere il problema di zaino unitario ottenuto rompendo ogni oggetto in "porzioni" di volume unitario, di valore pari al valore dell'oggetto diviso il suo volume (cioè il numero delle porzioni). Questo è ovviamente vietato dai vincoli del problema originario, ma consente di trovare una soluzione inammissibile, che comprende tutte le "porzioni" degli oggetti di valore unitario massimo, più alcune porzioni di un solo oggetto "limite". Scartando queste porzioni, si ottiene una soluzione euristica, che in pratica si dimostra di valore decisamente migliore di quella fornita dall'algoritmo *greedy* banale.

Per realizzare questo algoritmo, basta sostituire nel precedente il vettore di interi `phi` con un vettore di numeri reali `phi_v` che contiene i dati usati nell'ordinamento. Null'altro cambia. Ovviamente, si potrebbe anche passare alla procedura di ordinamento tutti i dati e modificare le operazioni di confronto, sostituendo i rapporti ϕ_i/v_i al valore ϕ_i.

Problema

La seconda parte della lezione affronta il problema dell'albero ricoprente minimo, per il quale l'algoritmo *greedy* determina la soluzione ottima. Si tratta di scrivere

un programma `kruskal.c` che carichi un grafo non orientato e ne determini un albero ricoprente di costo totale minimo. Il nome del programma viene dal fatto che l'algoritmo *greedy* è noto in in letteratura (e viene indicato nelle dispense) come *algoritmo di Kruskal*, dal nome del suo scopritore.

Formato di ingresso e di uscita Il formato di ingresso è un po' più sofisticato di quello utilizzato nelle lezioni 9 e 10 per il calcolo del sottografo indotto, ma molto simile. Il file si compone di tre righe: la prima riporta il numero dei vertici del grafo, preceduto dalle parole chiave `n =`, la seconda riporta il numero dei lati del grafo, preceduto dalle parole chiave `m =`, la terza riporta l'elenco dei lati stessi, con gli indici dei nodi estremi racchiusi fra parentesi tonde e separati da virgole, seguiti dal costo del lato stesso. Per esempio:

```
n = 9
m = 16
(1,2) 3 (1,4) 1 (1,5) 6 (2,3) 8 (2,5) 5 (3,5) 8 (3,6) 2 (4,5) 2 (4,7)
7 (5,6) 10 (5,7) 8 (5,8) 9 (6,8) 4 (6,9) 5 (7,8) 3 (8,9) 6
```

rappresenta l'esempio riportato nelle dispense.

La soluzione va stampata a video su due righe: la prima riga riporta il costo totale della soluzione, preceduto dalle parole chiave `f* =`; la seconda riporta l'elenco dei lati dell'albero minimo, nello stesso formato usato per il file dei dati, ma preceduto dalle parole chiave `x* =`.

Traccia della risoluzione

La risoluzione ricalca l'algoritmo di Kruskal descritto nelle dispense. L'unici aspette su cui si può discutere è l'implementazione del grafo. È abbastanza ovvio che la matrice di adiacenza è inefficiente dal punto di vista spaziale, ma anche da quello temporale (l'algoritmo non richiede di accedere direttamente a un lato dati gli estremi, mentre richiede di trovare il lato di costo minimo, e questo comporterebbe di scorrere $\Theta\left(n^2\right)$ posizioni). Il vettore delle liste di incidenza (*forward star*) è utile per visitare il grafo, ma non per gestire l'insieme dei lati trovandone il minimo, e non è particolarmente efficiente per determinare se un lato chiude cicli o no con la soluzione corrente. Alla fine, la lista dei lati è la struttura più efficiente, specialmente se i lati sono organizzati in un *min-heap*, cioè in un *heap* dove la proprietà sulle informazioni ai nodi viene invertita rispetto a quella che abbiamo usato per il *KP*.

Per sveltire l'implementazione procederemo come segue:

- useremo la libreria `grafo-la.c`, arricchendo gli archi con un campo intero `c` che riporti il costo dell'arco stesso;
- aggiungeremo un vettore di `posarco` come *min-heap* indiretto;
- gestiremo il test di aciclità con le struttura *foresta con bilanciamento* implementata con un vettore come descritto nelle dispense di teoria.

Ovviamente molte altre soluzioni sono possibili, ed eventualmente preferibili da diversi punti di vista. La lista dei lati potrebbe essere una tabella (risparmiando i puntatori e la sentinella, che in effetti non servono a nulla nell'algoritmo), oppure tre tabelle (due per gli indici dei nodi estremi e una per i costi). Oppure si potrebbe mantenere la lista degli archi e aggiungere un vettore a parte per i costi (con il vantaggio di non modificare la libreria e di tener separate topologia e pesatura del

grafo, ma lo svantaggio che l'*heap* indiretto dovrebbe puntare sia gli archi sia i costi). La foresta con bilanciamento potrebbe essere gestita con una libreria a parte. Le scelte fatte sono ragionevoli, ma essenzialmente volte ad accelerare il più possibile la risoluzione dell'esercizio, usando codice già scritto con modifiche minime.