UNIVERSITÀ DEGLI STUDI DI CATANIA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Triennale in Informatica

ESERCIZI SVOLTI DI ALGORITMI E COMPLESSITÀ

Autore:

GABRIELE BASILE

ANNO ACCADEMICO 2010-2011

1 Analisi Ammortizzata

In questa sezione verranno presentati degli esercizi svolti sull'analisi ammortizzata. Tutti gli esercizi, anche se non esplicitamente richiesto, saranno svolti usando tutti e tre i metodi dell'analisi ammortizzata, ovvero il metodo dell'aggregazione, degli accantonamenti e del potenziale.

Esercizio 1

Si supponga di avere una struttura dati sulla quale viene eseguita una sequenza S di $n > k$ operazioni $op1$ e $op2$. Si supponga inoltre che la complessità di $op1$ sia 2 e quella di $op2$ sia minore o uguale a k, e che ogni blocco di k operazioni consecutive contenga al più due sole operazioni di tipo $op2$. Si stimi la complessità della sequenza S utilizzando il metodo dell'aggregazione e quello degli accantonamenti.

Soluzione

`Metodo dell'aggregazione.`

In una sequenza di lunghezza $n > k$ possiamo avere al massimo:

$$n_{op2} = 2\lceil\frac{n}{k}\rceil = 2\lfloor\frac{n}{k}\rfloor + 2$$

$$n_{op1} = n - n_{op2} \leq n$$

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n_{op1}} 2 + \sum_{i=1}^{n_{op2}} k \leq$$

$$2\sum_{i=1}^{n} 1 + \sum_{i=1}^{2\lfloor\frac{n}{k}\rfloor + 2} k \leq$$

$$2n + k \sum_{i=1}^{2\frac{n}{k} + 2} 1 =$$

$$2n + k\,(2\frac{n}{k} + 2) = 4n + 2k = O(n)$$

`Metodo degli accantonamenti.`

La seguente tabella mostra qual'è il costo della i-esima operazione.

i	c_i
1	2
...	...
$k-2$	2
$k-1$	k
k	k
$k+1$	2
...	...
$2k-2$	2
$2k-1$	2
$2k$	k
...	...

Ogni k operazioni vi sono due $op2$ il cui costo è al massimo k. Adesso distribuiamo il costo 2k nelle k con in modo da ammortizzare il costo delle $op2$. Quindi:

$$\frac{2k}{k} \;=\; 2$$

da cui:

$$\hat{c_{op1}} = 2 \;+\; 2 = 4$$

$$\hat{c_{op2}} = 2$$

$$\sum_{i=1}^{n} \hat{c_i} \;\leq\; 4n \;=\; O(n)$$

`Metodo del potenziale.`

Senza perdere di generalità supponiamo che venga fatta un operazione $op2$ ogni $\frac{k}{2}$ operazioni e consideriamo la seguente funzione potenziale:

$$\Phi(i) = 2(i \mod \frac{k}{2})$$

la quale è scelta in quanto supposto che ogni $\frac{k}{2}$ operazioni viene fatta un operazione $op2$ e quindi il potenziale accumulato dopo la i-esima operazioni è esattamente $d(i \mod \frac{k}{2})$ dove d è la costante che permette di distribuire il costo k tra le $\frac{k}{2}$ operazioni. Per $i = 0$ la funzione vale 0 mentre per $i > 0$ è ≥ 0 e quindi è verificato il lemma del potenziale da cui

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Calcoliamo adesso i costi ammortizzati:

caso in cui $\frac{k}{2} \nmid i$

$$\hat{c_i} \;=\; c_i + \Phi(i) - \Phi(i-1) \;=$$

$$2 \;+\; 2(i \mod \frac{k}{2}) \;-\; 2((i-1) \mod \frac{k}{2}) \;=\; 2+2 \;=\; 4$$

caso in cui $\frac{k}{2} \mid i$

$$\hat{c_i} \;=\; c_i + \Phi(i) - \Phi(i-1) \;=$$

$$k \;+\; 2(i \mod \frac{k}{2}) \;-\; 2((i-1) \mod \frac{k}{2}) \;=\; k \;-\; 2(\frac{k}{2} \;-\; 1) \;=\; 2$$

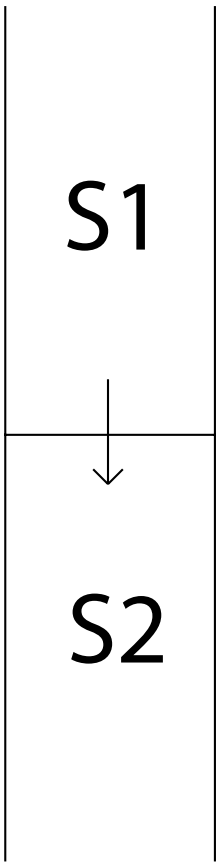
Infine troviamo che

$$\sum_{i=1}^{n} \hat{c_i} \;\leq\; 4n \;=\; O(n)$$

Esercizio 2

Si illustri come simulare in maniera efficiente una coda mediante due stack e si analizzi la simulazione mediante almeno due dei metodi di analisi ammortizzata studiati.

Soluzione



Come possiamo vedere nella figura accanto, per implementare una coda con 2 stack basta utilizzare lo stack S1 per le enqueue mentre lo stack S2 per le dequeue. Quando verrà chiamata una enqueue con lo stack S2 vuoto basterà travasare tutti gli elementi da S1 a S2 ed estrarre l'elemento. Eccovi lo pseudo-codice della enqueue e della dequeue:

function $enqueue(Q, x)$
  $push(S1, x)$

function $dequeue(Q)$
  if $|S2| = 0$ **then**
    while $|S1| > 0$ **do**
      $x \leftarrow pop(S1)$
      $push(S2, x)$
    end while
  end if
  $pop(S2)$

`Metodo dell'aggregazione.`

Supponiamo che vi siano $n_1 \leq n$ enqueue e $n_2 \leq n$ dequeue, quindi

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n_1} 1 + \sum_{i=1}^{n_2} 3 \;\leq\; 3n \;=\; O(n)$$

`Metodo degli accantonamenti.`

$$\hat{c_{enq}} = 1 + 1 + 1 = 3$$

1 unità per il costo reale dell'inserimento nello stack S1

1 unità per il costo dell' estrazione dallo stack S1

1 unità per il costo dell'inserimento nello stack S2

$$c_{\hat{deq}} = 1$$

1 unità per il costo reale dell' estrazione dallo stack S2

In conclusione, essendo che $\sum_{i=1}^{n} \hat{c_i} \ \geq \ \sum_{i=1}^{n} c_i$ allora

$$\sum_{i=1}^{n} c_i \ \leq \ 3n = O(n)$$

`Metodo del potenziale.`

Consideriamo la seguente funzione potenziale:

$$\Phi(S1, S2) \ = \ 2|S1|$$

la quale è scelta perchè 2 è il costo dell'operazione di travaso di ogni elemento e $|S1|$ è il numero di elementi coinvolti nel travaso. Quando gli stack sono vuoti il potenziale vale 0 e durante gli inserimenti non scende mai sotto lo zero e quindi è verificato il lemma del potenziale per cui

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Calcoliamo adesso i costi ammortizzati:

enqueue

$$c_{\hat{enq}} \ = \ c_i \ + \ \Phi(i) \ - \ \Phi(i-1) \ = \ 1 \ + \ 2 \ = \ 3$$

dequeue quando vi sono elmenti in S2

$$c_{\hat{deq}} \ = \ c_i \ + \ \Phi(i) \ - \ \Phi(i-1) \ = \ 1 \ + \ 0 \ = \ 1$$

dequeue quando S2 è vuoto

$$\hat{c_{deq}} \;=\; c_i \;+\; \Phi(i) \;-\; \Phi(i-1) \;=\; 2|S1| \;+\; 1 \;-\; 2|S1| \;=\; 1$$

In conclusione, essendo che $\sum_{i=1}^{n} \hat{c_i} \;\geq\; \sum_{i=1}^{n} c_i$ allora

$$\sum_{i=1}^{n} c_i \;\leq\; 3n = O(n)$$

Esercizio 3

Una sequenza di operazioni $PUSH$ e POP viene eseguita su uno stack la cui dimensione non supera mai il valore k. Dopo ogni k operazioni, viene fatta automaticamente una copia di backup dell'intero stack, per motivi di sicurezza. Si dimostri che il costo di n operazioni (con l'ovvia inclusione anche del costo dovuto alle copie di backup) è O(n).

Soluzione

`Metodo dell'aggregazione.`

Considerando la seguente tabella espletiamo la somma reale delle operazioni.

i	$c_{push/pop}$	c_{backup}
1	1	0
2	1	0
...	...	...
k-1	1	0
k	1	k
k+1	1	0
...	...	...

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n} 1 + \sum_{i=1}^{\lfloor \frac{n}{k} \rfloor} k \;\leq$$

$$n + k \sum_{i=1}^{\frac{n}{k}} 1 \;=\; n + k\frac{n}{k} \;=\; 2n \;=\; O(n)$$

`Metodo degli accantonamenti.`

$$\hat{c_i} = 1 + 1 = 2$$

1 unità per il costo reale dell'elemento

1 unità per il costo della copia di backup dell'elemento

In conclusione, essendo che $\sum_{i=1}^{n} \hat{c_i} \ \geq \ \sum_{i=1}^{n} c_i$ allora

$$\sum_{i=1}^{n} c_i \ \leq \ 2n = O(n)$$

`Metodo del potenziale.`

Consideriamo la seguente funzione potenziale:

$$\Phi(i) = i \mod k$$

la quale è scelta in quanto ogni k operazioni viene fatta un operazione costosa e quindi il potenziale accumulato dopo la i-esima operazione è esattamente $c(i \mod k)$ dove c è la costante che permette di distribuire il costo k tra le k operazioni. Per $i = 0$ vale 0 mentre per $i > 0$ è ≥ 0 e quindi è verificato il lemma del potenziale per cui

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Calcoliamo adesso i costi ammortizzati:

caso in cui $k \not| i$

$$\hat{c_i} \ = \ c_i \ + \ \Phi(i) \ - \ \Phi(i-1) \ = \ 1 \ + \ i \mod k \ - \ (i-1) \mod k \ = \ 1 \ + \ 1 \ = \ 2$$

caso in cui $k|i$

$$\hat{c_i} \ = \ c_i \ + \ \Phi(i) \ - \ \Phi(i-1) \ =$$

$$1 \ + \ k \ + \ i \mod k \ - \ (i-1) \mod k \ = \ 1 \ + \ k \ - \ (k \ - \ 1) \ = \ 2$$

Infine troviamo che

$$\sum_{i=1}^{n} \hat{c}_i \;=\; 2n \;=\; O(n)$$

Esercizio 4

Utilizzando il metodo dell'aggregazione e quello del potenziale, si determini il costo ammortizzato per operazione di una sequenza di n operazioni, ove il costo c_i dell'i-esima operazione sia dato da

$$c_i = \begin{cases} i & \text{se i è potenza esatta di 3} \\ 2 & \text{altrimenti} \end{cases} \tag{1}$$

Soluzione

`Metodo dell'aggregazione.`

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1, i \notin P^3}^{n} 2 + \sum_{i=1, i \in P^3}^{n} i \leq 2\sum_{i=1}^{n} 1 + \sum_{j=0}^{\lfloor \log_3 n \rfloor} 3^j \leq$$

$$2n + \sum_{j=0}^{\log_3 n} 3^j = 2n + \frac{3^{\log_3 n + 1} - 1}{3 - 1} =$$

$$2n + \frac{3}{2}n - \frac{1}{2} = O(n)$$

`Metodo degli accantonamenti.`

i	c_i
1	1
2	2
3	3
4	2
...	...
8	2
9	9
10	2
...	...
26	2
27	27
...	...

Quello che adesso si deve fare è distribuire il costo delle operazioni costose tra tutte quelle costanti in modo da "accantonarne" il costo.

$$\frac{3^i}{3^i - 3^{i-1}} = \frac{3^i}{3^{i-1}(3-1)} = \frac{3}{2}$$

da cui:

$$\hat{c_i} = \begin{cases} \frac{3}{2} & \text{se i è potenza esatta di 3} \\ \frac{7}{2} & \text{altrimenti} \end{cases}$$

$$\sum_{i=1}^{n} \hat{c_i} \leq \sum_{i=1}^{n} \frac{7}{2} = \frac{7}{2}n = O(n)$$

`Metodo del potenziale.`

Consideriamo la seguente funzione potenziale:

$$\Phi(i) = \begin{cases} 0 & \text{se } i = 0 \\ \frac{3}{2}(i - 3^{\lfloor \log_3 i \rfloor}) & \text{se } i > 0 \end{cases}$$

Per $i = 0$ vale 0 mentre per $i > 0$ è ≥ 0 e quindi è verificato il lemma del potenziale per cui

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Calcoliamo adesso i costi ammortizzati:

$i \notin P^3, i > 1$

$$\hat{c_i} = c_i + \Phi(i) - \Phi(i-1) = 2 + \frac{3}{2}(i - 3^{\lfloor \log_3 i \rfloor}) - \frac{3}{2}(i - 1 - 3^{\lfloor \log_3 i - 1 \rfloor}) = 2 + \frac{3}{2} = \frac{7}{2}$$

$i \in P^3, i = 1$

$$\hat{c_i} \;=\; c_i \;+\; \Phi(i) \;-\; \Phi(i-1) \;=\; 1 \;+\; \frac{3}{2}(1 - 3^{\lfloor \log_3 1 \rfloor})) \;=\; 1$$

$i \in P^3, i > 1$

$$\hat{c_i} \;=\; c_i \;+\; \Phi(i) \;-\; \Phi(i-1) \;=\; i \;+\; \frac{3}{2}(i - 3^{\lfloor \log_3 i \rfloor}) \;-\; \frac{3}{2}(i - 1 - 3^{\lfloor \log_3 i-1 \rfloor}) \;=$$

$$i \;-\; \frac{3}{2}(i - 1 - \frac{i}{3}) \;=\; i \;-\; \frac{3}{2}i \;+\; \frac{1}{2}i \;+\; \frac{3}{2} \;=\; \frac{3}{2}$$

Esercizio 5

Si illustri come gestire l'inserimento di elementi in una tabella dinamica in modo tale che

(a) Il fattore di carico non scenda mai al di sotto di $\frac{1}{3}$ (ma possa scendere al di sotto di $\frac{1}{2}$);

(b) n inserimenti in una tabella inizialmente vuota siano effettuati in tempo O(n);

Si verifichi la validità di (b) per la soluzione proposta utilizzando i tre metodi dell'analisi ammortizzata.

Soluzione

La procedura di inserimento in una deve essere modificata in modo da triplicare la dimensione della tabella quando piena invece che raddoppiarla. Ecco il codice:

---

function $insert(T, x)$
  if $size = 0$ **then**
    $i \leftarrow 0$
    $-$ *si allochi* $table[T]$ *di dimensione* 1
    $size[T] \leftarrow 1$
  end if
  if $num[T] = size[T]$ **then**
    $-$ *si allochi newtable di dimensione* $3 * size[T]$
    $-$ *si copi* $table[T]$ *in newtable*
    $-$ *si deallochi* $table[T]$
    $table[T] \leftarrow newtable$
    $size[T] \leftarrow 3size[T]$
  end if
  $-$ *si inserisca* x
  $num[T] \leftarrow num[T] + 1$

`Metodo dell'aggregazione.`

I costo del i-esimo inserimento vale

$$c_i = \begin{cases} i & \text{se i-1} \in P^3 \\ 1 & \text{altrimenti} \end{cases}$$

quindi:

$$\sum_{i=1}^{n} c_i = \sum_{(i-1)\notin P^3}^{n} c_i + \sum_{(i-1)\in P^3}^{n} c_i =$$

$$\sum_{(i-1)\notin P^3}^{n} 1 + \sum_{(i-1)\in P^3}^{n} i \;\leq\; \sum_{i=1}^{n} 1 + \sum_{j=0}^{\lfloor \log_3(n-1)\rfloor} 3^j \;=$$

$$n + \frac{3^{\lfloor \log_3(n-1)\rfloor+1} - 1}{2} \;\leq\; n + \frac{1}{2}(3^{\log_3(n-1)+1} - 1) \;=$$

$$n + \frac{1}{2}(3(n-1) - 1) \;= n + \frac{3}{2}n - 2 \;=\; \frac{5}{2}n - 2 \;=\; O(n)$$

`Metodo degli accantonamenti.`

$$\hat{c_i} = 1 + 1 + \frac{1}{2} = \frac{5}{2}$$

1 unità per il costo di inserimento dell'elemento

1 unità per il costo della copia dell'elemento

$\frac{1}{2}$ unità per il costo della copia degli elementi inseriti in precedenza (sono la metà)

In conclusione, essendo che $\sum_{i=1}^{n} \hat{c_i} \;\geq\; \sum_{i=1}^{n} c_i$ allora

$$\sum_{i=1}^{n} c_i \;\leq\; 3n = O(n)$$

`Metodo del potenziale.`

Consideriamo la seguente funzione potenziale:

$$\Phi(T) = \frac{3}{2}(num[T] - \frac{1}{3}size[T]) = \frac{3}{2}num[T] - \frac{1}{2}size[T]$$

Per T vuota vale 0 mentre per una generica configurazione di T è ≥ 0 in quanto $num[T] \geq \frac{1}{3}size[T]$ e quindi è verificato il lemma del potenziale per cui

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Calcoliamo adesso i costi ammortizzati:

Inserimento senza espansione: In questa risoluzione abbrevieremo $num[T]$ con n e $size[T]$

con s.

$$\hat{c_i} = c_i + \Phi(T) - \Phi(i-1) \; =$$

$$1 + \frac{3}{2}n_i - \frac{1}{2}s_i - \frac{3}{2}n_{i-1} + \frac{1}{2}s_{i-1} \; =$$

$$1 + \frac{3}{2}(n_{i-1} + 1) - \frac{3}{2}n_{i-1} \; =$$

$$1 + \frac{3}{2} \; = \; \frac{5}{2}$$

Inserimento con espansione:

$$\hat{c_i} = c_i + \Phi(T) - \Phi(i-1) =$$

$$n_i + \frac{3}{2}n_i - \frac{1}{2}s_i - \frac{3}{2}n_{i-1} + \frac{1}{2}s_{i-1} \; =$$

$$n_i + \frac{3}{2}n_i - \frac{3}{2}(n_i - 1) - \frac{3}{2}(n_i - 1) + \frac{1}{2}(n_i - 1) \; =$$

$$n_i + \frac{3}{2}n_i - \frac{3}{2}n_i + \frac{3}{2} - \frac{3}{2}n_i + \frac{3}{2} + \frac{1}{2}n_i - \frac{1}{2} \; =$$

$$\frac{3}{2} + \frac{3}{2} - \frac{1}{2} \; = \; \frac{5}{2}$$

2 B-Tree

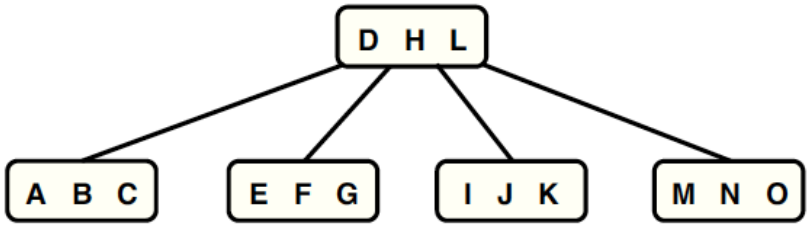
Esercizio 1

Svolgere i seguenti punti:

(a) Si definisca in maniera precisa la nozione di B-albero.

(b) Si dia e si dimostri una maggiorazione dell'altezza di un B-albero in funzione del suo grado minimo e del numero di chiavi in esso contenute.

(c) Si consideri il seguente albero di ricerca:



Si stabilisca se esso possa essere considerato un B-albero e, in caso affermativo, si dica quali sono i valori di grado minimo compatibili con esso. (Si giustifichino adeguatamente le risposte.)

Soluzione

Il B-tree è un albero di ricerca bilanciato il cui fattore di ramificazione è più alto rispetto agli altri alberi bilanciati. Il generico B-tree T gode delle seguenti proprietà:

I. Ogni nodo x ha i seguenti campi:

- $n[x]$: intero che indica il numero delle chiavi nel nodo x;
- $leaf[x]$: booleano che indica se il nodo x è una foglia.

II. Ciascun nodo interno contiene $n[x] + 1$ puntatori ai figli:

$$c_1[x],\ c_2[x],\ \ldots,\ c_{n[x]+1}[x]$$

III. Se k_i denota una qualsiasi chiave nell'albero con radice $c_i[x]$ allora vale il seguente ordinamento (Proprietà di ricerca):

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq ... \leq k_{n[x]} \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$

IV. Tutti le foglie sono alla stessa profondità.

V. I B-tree sono caratterizzati da una quantità $t \geq 2$, detta grado minimo, tale che:

- tutti i nodi, a parte la radice, devono contenere almeno $t - 1$ chiavi. Se il B-tree non è vuoto, la radice deve contenere almeno una chiave;
- tutti i nodi devono contenere al massimo $2t - 1$ chiavi.

da

$depth$	n'
0	1
1	$2(t-1)$
2	$2t(t-1)$
3	$2t^2(t-1)$
4	$2t^3(t-1)$
...	...
h	$2t^{h-1}(t-1)$

Per rispondere al punto (b) se T è un generico B-tree di grado minimo t avente n chiavi, indicheremo con T' il B-tree di grado minimo t avente il minimo numero di chiavi. Sia n' il numero di chiavi di T', dalla tabella a sinistra otteniamo la seguente equazione

$$n' \;=\; 1 + 2(t-1)\sum_{i=0}^{h-1} 2t^i \;=\; 1 + 2(t-1)\frac{t^h - 1}{t-1} \;=\; 2t^h - 1$$

cui espletando h e sfruttando $n \geq n'$ otteniamo

$$h \;=\; \log_t \frac{n'+1}{2} \;\leq\; \log_t \frac{n+1}{2}$$

che è una maggiorazione dell'altezza.

Per rispondere al punto (c) basta verificare che l'albero in figura soddisfi le porprietà 3,4,5 espresse nel punto (a):

- La proprietà 3 è soddisfatta in quanto valgono le diseguaglianze

$$k_1 \;\leq\; key_1[x] \;\leq\; k_2 \;\leq\; key_2[x] \;\leq\; ... \;\leq\; k_{n[x]} \;\leq\; key_{n[x]}[x] \;\leq\; k_{n[x]+1}$$

- La proprietà 4 è anch'essa soddisfatta in quanto tutte le foglie sono alla stessa profondità

- Per verificare la proprietà 5 bisogna che

$$\exists t,\; t \in \mathbb{N} : \begin{cases} t - 1 \leq min(n_1, ..., n_m) \\ 2t - 1 \geq \max(n_1, ..., n_m) \end{cases}$$

essendo n_i il numero di chiavi dell'i-esimo nodo.

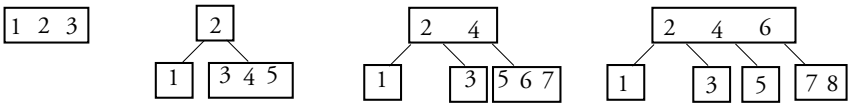
Nel nostro caso il minimo ed il massimo numero di chiavi valgono 3, quindi basta prendete $2 \leq t \leq 4$.

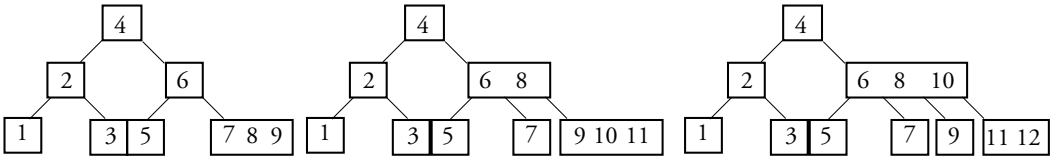
Esercizio 2

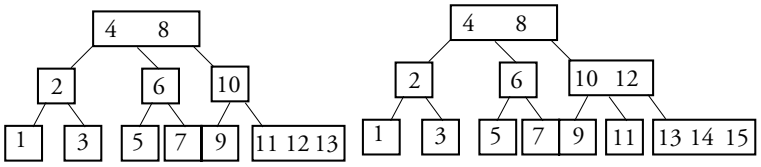
Si costruisca un B-albero di grado minimo 2 contenente almeno 15 chiavi.

Soluzione

Inserimento delle chiavi da 1 a 15 in un B-tree di grado 2







Esercizio 3

Svolgere i seguenti punti:

(a) Si effettui l'inserimento delle chiavi D, G, R, F, M, H, P, Q, I, L, A, B, C (nell'ordine dato) in un B-tree di grado minimo 2, inizialmente vuoto, e quindi si cancelli la chiave F.

(b) Si diano una minorazione ed una maggiorazione dell'altezza h di un B-tree di grado minimo 2 in funzione del numero $n \geq 1$ di chiavi in esso contenute.

Soluzione

$depth$	n'	n''
0	1	3
1	2	4 3
2	2^2	4^2 3
3	2^3	4^3 3
4	2^4	4^4 3
...	...	...
h	$2h$	4^h 3

Per rispondere al punto (b) se T è un generico B-tree di grado minimo 2 avente n chiavi, indicheremo con T' il B-tree di grado minimo 2 avente il minimo numero di chiavi e con con T'' il B-tree di grado minimo 2 avente il massimo numero di chiavi. Siano n' e n'' rispettivamente il numero di chiavi di T' e di T'', dalla tabella a sinistra otteniamo le seguenti equazioni

$$n' = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$n'' = 3\sum_{i=0}^{h} 4^i = 4^{h+1} - 1$$

da cui espletando h e sfruttando $n' \leq n \leq n''$ otteniamo

$$\log_4(n+1) - 1 \leq h \leq \log_2(n+1) - 1$$

Esercizio 4

Svolgere i seguenti punti:

(a) Sia T_k un B-tree in cui ciascun nodo contiene esattamente k chiavi ($k \geq 1$). Si determini per quanti e per quali valori di t (in funzione di k) l'albero T_k possa essere considerato un B-tree di grado minimo t.

(b) Si generalizzi il risultato precedente al caso di un generico B-tree i cui ϕ nodi (con l'esclusione della radice) contengono un numero di chiavi rispettivamente pari a $k_1, k_2, ..., k_\phi$.

Soluzione

Risolviamo il punto (a): per la proprietà 5 dei B-tree deve

$$\exists t,\ t \in \mathbb{N} : \begin{cases} t - 1 \leq k \\ 2t - 1 \geq k \end{cases}$$

ovvero deve $\exists t,\ t \in \mathbb{N}\ :\ \lceil \frac{k+1}{2} \rceil\ \leq\ t\ \leq\ k + 1$

Nel punto (b) i nodi contengono un numero variabile di chiavi. Siano k_{min} e k_{max} rispettivamente il minimo e il massimo numero di chiavi allora per la proprietà 5 deve

$$\exists t,\ t \in \mathbb{N} : \begin{cases} t - 1 \leq k_{min} \\ 2t - 1 \geq k_{max} \end{cases}$$

da cui, risolvendo il sistema otteniamo la seguente condizione

$$\exists t,\ t \in \mathbb{N}\ :\ \lceil \frac{k_{max} + 1}{2} \rceil\ \leq\ t\ \leq\ k_{min} + 1$$

Esercizio 5

Un B-tree di grado minimo t e altezza h si dice minimo [massimo] se ha il minor [maggior] numero di chiavi tra tutti i B-tree di grado minimo t e altezza h. Sia B_1 un B-tree massimo di grado minimo t e altezza h e sia n_1 il numero di chiavi in esso contenute. Inoltre, sia B_2 un B-tree minimo di grado minimo 2t e altezza h+1 e sia n_1 il numero di chiavi in esso contenute. Si calcoli il rapporto $\frac{n_1+1}{n_2+1}$.

Soluzione

$depth$	n_2	n_1
0	1	$2t-1$
1	$2(2t-1)$	$2t(2t-1)$
2	$2(2t)(2t-1)$	$(2t)^2(2t-1)$
3	$2(2t)^2(2t-1)$	$(2t)^3(2t-1)$
4	$2(2t)^3(2t-1)$	$(2t)^4(2t-1)$
...	...	...
h	$2(2t)^{h-1}(2t-1)$	$(2t)^h(2t-1)$
$h+1$	$2(2t)^h(2t-1)$	/

Dalla tabella a sovrastante ricaviamo le seguenti equazioni

$$n_1 \;=\; (2t \;-\; 1)\;\sum_{i=0}^{h}(2t)^i \;=\; (2t)^{h+1}-1$$

$$n_2 \;=\; 1 \;+\; 2\;(2t \;-\; 1)\sum_{i=1}^{h+1}(2t)^i \;=\; 2\;(2t)^{h+1}-1$$

da cui

$$\frac{n_1 \;+\; 1}{n_2 \;+\; 1} \;=\; \frac{(2t)^{h+1}}{2(2t)^{h+1}} \;=\; \frac{1}{2}$$

3 Splay-Tree

Esercizio 1

Svolgere i seguenti punti:

(a) Si eseguano su uno splay tree inizialmente vuoto le seguenti operazioni, nell'ordine dato: Insert 0, 2, 4, 6; Search 2; Insert 5; Search 0; Delete 4.

(b) Si descriva come modificare gli Splay Tree affinchè possa essere gestita in maniera efficiente anche l'operazione $Select(k, T)$ per la ricerca del k-esimo elemento nello splay-tree T. Si discuta inoltre la complessità ammortizza dell'implementazione proposta per l'operazione $Select(k, T)$.

Soluzione

Risoluzione punto (b).

L'idea di base è quella di aggiungere alla struttura dati il campo $size[x]$ il quale indicherà il numero di nodi totali del sotto-albero radicato nel nodo x (x incluso). Ecco allora lo psudocodice della procedura $Select(k, T)$ il quale sfrutta la procedura ricorsiva $R - Select(x, i)$ per la ricerca del k-esimo elemento: L'algoritmo per prima cosa calcola $r \leftarrow size[left[x]]+1$

---

function $Select(k, T)$
  $R - Select(root[T], k)$

---

function $R - Select(x, k)$
  if $left[x] = nil$ **then**
    $r \leftarrow 1$
  else
    $r \leftarrow size[left[x]] + 1$
  end if
  if $k = r$ **then**
    $returnx$
  else if $k < r$ **then**
    return $R - Select(left[x], k)$
  else
    return $R - Select(right[x], k - r)$
  end if

---

che rappresenta il rango di x all'interno del sotto-albero con radice in x. Se $k = r$, allora

il nodo x è il k-esimo elemento più piccolo, quindi abbiamo finito. Se $k < r$, allora il k-esimo elemento più piccolo è nel sotto-albero sinistro di x, quindi si effettua una ricorsione su $left[x]$. Se $k > r$, allora il k-esimo elemento più piccolo è nel sotto-albero destro di x. Poichè ci sono r elementi nel sotto-albero con radice in x che precedono il sotto-albero destro di x in un attraversamento simmetrico, il k-esimo elemento più piccolo nel sotto-albero con radice in x è il $(k - r)$-esimo elemento più piccolo nel sotto-albero con radice in $right[x]$. Dato che le rotazioni di tipo zig, zig-zig e zig-zag modificano il rango dei soli elementi x,p,g basterà aggiungere le seguenti righe alla fine delle relative procedure:

zig :

$$size[x] \leftarrow size[p]$$
$$size[p] \leftarrow size[left[p]] + size[right[p]] + 1$$

zig-zig :

$$size[x] \leftarrow size[g]$$
$$size[g] \leftarrow size[left[g]] + size[right[g]] + 1$$
$$size[p] \leftarrow size[left[p]] + size[right[p]] + 1$$

zig-zag :

$$size[x] \leftarrow size[g]$$
$$size[p] \leftarrow size[left[p]] + size[right[p]] + 1$$
$$size[g] \leftarrow size[left[g]] + size[right[g]] + 1$$

Naturalmente le modifiche delle rotazioni simmetriche sono simmetriche. La complessità della procedura $Select(k, T)$ dipende dall'altezza dell'albero ma essendo che si dimostra che dopo n operazioni gli Splay-Tree hanno altezza $\lfloor \log N \rfloor$ allora, essendo che le modifiche alle procedure per le rotazioni hanno tempo $\Theta(1)$ la complessità ammortizzata di $Select(k, T)$ sarà $O(\log N)$

Esercizio 2

Svolgere i seguenti punti:

(a) Si eseguano su uno splay tree inizialmente vuoto le seguenti operazioni, nell'ordine dato: Insert 0, 1, 2, 3, 4, 5, 6, 7; Search 3; Insert 8; Search 5; Delete 3; Search 7.

(b) Si descriva come modificare gli Splay Tree affinchè possa essere gestita in maniera efficiente anche l'operazione Splay-L-D(T) per la ricerca della minima chiave residente in un nodo di profondità massima. In particolare, si descrivano un'implementazione della procedura Splay-L-D(T) e le modifiche da apportare alle operazioni zig-zag, zig-zig e zig. Qualè il costo ammortizzato dell'operazione Splay-L-D(T)? Perchè?

Soluzione

Risoluzione punto (b).

L'idea di base è quella di aggiungere alla struttura dati il campo $height[x]$ il quale indicherà l'altezza del sotto-albero radicato nel nodo x (x incluso). Ecco allora lo psudo-codice della procedura $Splay - L - D(T)$ il quale sfrutta la procedura ricorsiva $R - Splay - L - D(x)$ per la ricerca del k-esimo elemento: L'algoritmo per prima cosa controlla se siamo arrivati

---

function $Splay - L - D(T)(T)$
  $R - Splay - L - D(T)(root[T])$

---

function $R - Splay - L - D(x)$
  if $height[x] = 0$ **then**
    return x
  end if
  $l \leftarrow height[left[x]]$
  $r \leftarrow height[right[x]]$
  if $l \geq r$ **then**
    return $R - Splay - L - D(left[x])$
  else
    return $R - Splay - L - D(right[x])$
  end if

---

ad una foglia; se si abbiamo finito. In seguito calcola confronta le altezze dei sotto-alberi. Se $l \geq r$, allora l'elemento minimo alla massima profondità è nel sotto-albero sinistro di x, quindi si effettua una ricorsione su $left[x]$. Se $l < r$, allora l'elemento minimo alla massima

profondità è nel sotto-albero destro di x. Dato che le rotazioni di tipo zig, zig-zig e zig-zag modificano l'altezza dei soli elementi x,p,g basterà aggiungere le seguenti righe alla fine delle relative procedure:

zig :

$$height[p] \leftarrow \max(height[left[p]], height[right[p]]) + 1$$
$$height[x] \leftarrow \max(height[left[x]], height[right[x]]) + 1$$

zig-zig, zig-zag :

$$height[g] \leftarrow \max(height[left[g]], height[right[g]]) + 1$$
$$height[p] \leftarrow \max(height[left[p]], height[right[p]]) + 1$$
$$height[x] \leftarrow \max(height[left[x]], height[right[x]]) + 1$$

Naturalmente le modifiche delle rotazioni simmetriche sono simmetriche inoltre si osserva che $\max(nil, a) = \max(a, nil) = a$ con $a \in \mathbb{N}$ e $\max(nil, nil) = -1$. Un ultima cosa da notare è che se per qualche motivo non si dovesse risalire fino alla radice una volta applicata una rotazione il campo $height[x]$ dei nodi tra la radice e il punto di arresto non risulterebbe più coerente, fortunatamente l'unica procedura che modifica la struttura dell'albero applicando le rotazioni è la procedura $Splay$ che risale fino alla radice. La complessità della procedura $R - Splay - L - D(x)$ dipende dall'altezza dell'albero ma essendo che si dimostra che dopo n operazioni gli Splay-Tree hanno altezza $\lfloor \log N \rfloor$, essendo che le modifiche alle procedure per le rotazioni hanno tempo $\Theta(1)$ la complessità ammortizzata di $R - Splay - L - D(x)$ sarà $O(\log N)$

4 Heap Binomiali

Esercizio 1

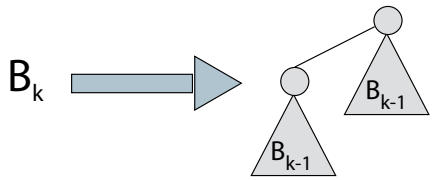
Si definiscano gli alberi binomiali e si enuncino le loro principali proprietà, dimostrandole adeguatamente. Si definiscano gli heap binomiali e si dia una maggiorazione al grado massimo di un un nodo in uno heap binomiale contenente n nodi.

Soluzione

Definizione Alberi Binomiali

Per ogni $k \in \mathbb{N}$ esiste un albero binomiale B_k di grado k, definito come segue:

- B_0 è l'albero formato da un solo nodo.
- Dato B_{k-1} definiamo B_k combinando due copie B_{k-1} come in figura



Proprietà Alberi Binomiali

Per ogni $k \in \mathbb{N}$ valgono le seguenti proprietà:

1. B_k ha 2^k nodi
2. L'altezza di B_k è k
3. B_k ha $\binom{k}{i}$ nodi a profondità i per $i = 0, 1, ..., k$
4. La radice di un B_k ha grado k ed ogni altro nodo del B_k ha grado $< k$.
   Inoltre i figli della radice di un B_k sono radici di $B_{k-1}, B_{k-2}, ..., B_1, B_0$ nell'ordine dato.

Dimostrazione. Per la dimostrazione delle quattro proprietà procediamo per induzione.

caso base $k = 0$

1. B_0 ha $2^0 = 1$ nodi

2. B_0 ha altezza 0
3. B_0 ha $\binom{0}{0} = 1$ nodi a profondità 0
4. La radice ha grado 0

passo induttivo Supponiamo che il lemma sia vero per $k-1$ con $k \geq 1$

1. B_k ha $\#(B_{k-1}) + \#(B_{k-1}) = 2\,2^{k-1} = 2^k$ nodi
2. B_k ha altezza $height(B_{k-1}) + 1 = k - 1 + 1 = k$
3. Sia $1 \leq i \leq k-1$. Il numero di nodi a profondità i sarà

$$\binom{k-1}{i-1} + \binom{k-1}{i} = \binom{k}{i}$$

   Inoltre B_k ha

   $\binom{k-1}{0} = 1$ nodi a profondità 0

   $\binom{k-1}{k-1} = \binom{k}{k} = 1$ nodi a profondità k

4. - La radice ha grado $degree(B_{k-1}) + 1 = k - 1 + 1 = k$.
   - Inoltre ciascun altro nodo appartiene ad un B_{k-1} e per l'ipotesi induttiva ha grado $\leq k-1$ che è $< k$.
   - Il primo figlio di un B_k è radice di un B_{k-1}. Inoltre per ipotesi induttiva i successivi $k-1$ figli della radice di un B_k sono radici di $B_{k-2}, B_{k-3}, ..., B_1, B_0$ nell'ordine.

□

Definizione Heap Binomiali

Un Heap Binomiale H è un insieme di alberi binomiali tale che

- ciascun albero binomiale in H gode della proprietà $min - heap$
- per ogni $k \in \mathbb{N}$, H contiene al più $un\ solo$ albero binomiale di grado k

Maggiorazione Grado Massimo

Sia H un Heap Binomiale con n nodi e dalla definizione e dalla proprità 4 degl'alberi binomiali si evince che la radice del B_k, l'albero binomiale di grado massimo in H, ha grado

maggiore di tutti gli altri nodi e quindi tutti i nodi in un Heap binomiale hanno grado $\leq k$. In definitiva sia B_k l'albero binomiale di grado massimo in H, allora si ha la relazione $2^k \leq n$ da cui $k \leq \lfloor \log n \rfloor$ e quindi tutti i nodi in hanno grado $\leq \lfloor \log n \rfloor$.

Esercizio 2

Si illustrino gli heap di Binomiali e le operazioni da essi supportate.

Soluzione

Dato che nell'esercizio 1 sono già stati ampiamente presentati gli Heap Binomiali adesso illustreremo le operazioni da essi supportate:

Make-Heap(): Crea e restituisce un heap senza elementi. Per far ciò esegue l'inizializzazione della struttura dati che comporta l'esecuzione di un numero costante di operazioni(Complessità $O(1)$);

Union(H_1, H_2): Crea e restituisce un nuovo heap H contenete tutti i nodi di H_1 e H_2 (H_1 e H_2 vengono distrutti). Per far ciò per prima cosa fonde le liste delle radici degli heap H_1 e H_2 in una unica la quale è ordinata monotonicamente in base ai gradi degli alberi. Poichè potrebbe esserci al massimo due alberi con lo stesso grado allora si procede con unione delle radici di pari grado finchè non rimarrà una singola radice per ciascun grado. La complessità della procedura è $O(\log n)$. Per provarlo supponiamo che H_1 contenga n_1 nodi e che H_2 contenga n_2 nodi e che sia $n = n_1 + n_2$. Allora H_1 conterrà al massimo $\lfloor \log n_1 \rfloor + 1$ radici mentre H_2 ne conterrà al massimo $\lfloor \log n_2 \rfloor + 1$, ma

$$\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2 \leq 2\lfloor \log n \rfloor + 2 = O(\log n).$$

Quindi dopo aver unito le liste delle radici di H_1 e H_2 saranno processati al massimo $O(\log n)$ alberi radici il quale rappresenta il costo della procedura.

Insert(H, x): Inserisce nell'heap H il nodo x. Per far ciò prima inserisce x in un heap vuoto H' e successivamente chiama la procedura *Union(H, H')*. La complessità è $O(\log n)$ dato che vi è la chiamata *Union(H, H')*.

ExtractMin(H): Estrae dall'heap H il nodo con chiave minima restituendone il puntatore. Per far ciò scandisce tutta la lista delle radici alla ricerca del nodo x con chiave minima, e lo elimina dalla lista delle radici. Crea un nuovo heap H' contenete la lista

dei figli di x in ordine invertito e chiama la procedura *Union(H, H')* restituendo x. La complessità è $O(\log n)$ dato che il processamento delle lista delle radici e l'unione di due heap prendono tempo logaritmico.

Minimum(H): restituisce un puntatore al nodo con chiave minima nell'heap H. Per far ciò scandisce tutta la lista delle radici alla ricerca del nodo x con chiave minima, e lo restituisce.

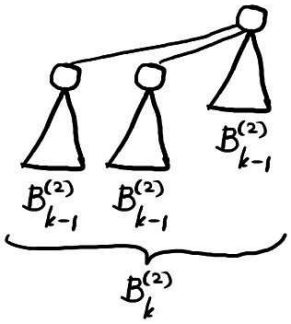
Decrease-Key(H, x, k): assegna al nodo x nell'heap H una chiave con valore non maggiore del precendente. Per far ciò fa risalire il nodo x nell'albero finchè non incontra una chiave minore di quella in possesso del nodo x. Dato che l'altezza massima di un albero è $O(\log n)$ la sua complessità sarà logaritmica.

Delete(H, x): Cancella il nodo x dall'heap H. Per far ciò sfrutta la procedura *Decrease-Key* per decrementare a $-\infty$ il valore della chiave in x e farla risalire fino alla radice. Dopo di che, dato il valore della sua chiave il nodo verrà rimosso con la chiamata alla procedura *ExtractMin()*. Dato entrambe le procedure sfruttate hanno complessità $O(\log n)$ anche questa avrà complessità logaritmica.

Esercizio 3

Un *albero binomiale* $B_k^{(2)}$ *di ordine* 2 (in breve, albero 2-binomiale) è un albero ordinato definito ricorsivamente come segue:

- l'albero 2-binomiale $B_0^{(2)}$ è formato da un solo nodo;
- l'albero 2-binomiale $B_k^{(2)}$ per $k \geq 1$, è formato da tre alberi 2-binomiali $B_{k-1}^{(2)}$ collegati insieme in modo tale che le radici di due alberi 2-binomiali risultino i due figli più a sinistra della radice del terzo albero (si veda la figura).



(a) Si enuncino e si dimostrino le principali proprietà degli alberi 2-binomiali in analogia a quanto visto per gli alberi binomiali. (In particolare, per quanto riguarda il conteggio del numero D(k,i) di nodi a profondità i nell'albero 2-binomiale $B_k^{(2)}$, si dimostri che D(k,i) è uguale al coefficiente del monomio $x^i y^{k-i}$ nell'espansione del polinomio $(2x + y)^k$.)

(b) Si proponga una definizione di heap 2-binomiali.

Soluzione

Prima di procedere con le proprietà calcoliamo il coefficiente del monomio $x^i y^{k-i}$ nell'espansione del polinomio $(2x + y)^k$:

$$(2x + y)^k = \sum_{i=0}^{k} \binom{k}{i} (2x)^i y^{k-i} = \sum_{i=0}^{k} 2^i \binom{k}{i} x^i y^{k-i}$$

quindi il coefficiente del monomio $x^i y^{k-i}$ sarà $2^i \binom{k}{i}$.

Proprietà Alberi Binomiali

Per ogni $k \in \mathbb{N}$ valgono le seguenti proprietà:

1. $B_k^{(2)}$ ha 3^k nodi
2. L'altezza di $B_k^{(2)}$ è k
3. $B_k^{(2)}$ ha $2^i \binom{k}{i}$ nodi a profondità i per $i = 0, 1, ..., k$
4. La radice di un $B_k^{(2)}$ ha grado $2k$ ed ogni altro nodo del $B_k^{(2)}$ ha grado $< 2k$. Inoltre i figli della radice di un $B_k^{(2)}$ sono radici di $B_{k-1}^{(2)}, B_{k-1}^{(2)}, B_{k-2}^{(2)}, B_{k-2}^{(2)}, ..., B_1^{(2)}, B_1^{(2)}, B_0^{(2)}, B_0^{(2)}$ nell'ordine dato.

Dimostrazione. Per la dimostrazione delle quattro proprietà procediamo per induzione.

caso base $k = 0$

1. $B_0^{(2)}$ ha $3^0 = 1$ nodi
2. $B_0^{(2)}$ ha altezza 0
3. $B_0^{(2)}$ ha $\binom{0}{0} = 1$ nodi a profondità 0
4. La radice ha grado 0

passo induttivo Supponiamo che il lemma sia vero per $k-1$ con $k \geq 1$

1. $B_k^{(2)}$ ha $\#(B_{k-1}^{(2)}) + \#(B_{k-1}^{(2)}) + \#(B_{k-1}^{(2)}) = 3\,3^{k-1} = 3^k$ nodi
2. $B_k^{(2)}$ ha altezza $height(B_{k-1}^{(2)}) + 1 = k - 1 + 1 = k$
3. Sia $1 \leq i \leq k-1$. Il numero di nodi a profondità i sarà

$$2^{i-1}\binom{k-1}{i-1} + 2^{i-1}\binom{k-1}{i-1} + 2^i\binom{k-1}{i} =$$

$$2^i\binom{k-1}{i-1} + 2^i\binom{k-1}{i} =$$

$$2^i[\binom{k-1}{i-1} + \binom{k-1}{i}] = 2^i\binom{k}{i}$$

Inoltre B_k ha

$2^0\binom{k-1}{0} = 1$ nodi a profondità 0

$2^{k-1}\binom{k-1}{k-1} + 2^{k-1}\binom{k-1}{k-1} = 2^k\binom{k-1}{k-1} = 2^k\binom{k}{k} = 2^k$ nodi a profondità k

4.
   - La radice ha grado $degree(B^{(2)}_{k-1}) + 2 = 2(k-1) + 2 = 2k$.
   - Inoltre ciascun altro nodo appartiene ad un $B^{(2)}_{k-1}$ e per l'ipotesi induttiva ha grado $\leq 2k-2$ che è $< 2k$.
   - Il primo figlio di un $B^{(2)}_k$ è radice di un $B^{(2)}_{k-1}$ così come il suo secondo figlio. Inoltre per ipotesi induttiva i successivi $2k-2$ figli della radice di un $B^{(2)}_k$ sono radici di

     $B_{k-2}, B_{k-2}, ..., B_0, B_0$ nell'ordine.

□

Esercizio 4

Si consideri la seguente procedura per incrementare il valore di una chiave in un heap binomiale:

---

function $My - Increase - Key(x, k)$
Se $key[x] = k$, errore.
Se x è una foglia, si ponga $key[x] \leftarrow k$
altrimenti
- si determini il figlio y di x avente chiave minima
- si ponga $key[x] \leftarrow key[y]$
- si chiami ricorsivamente $My - Increase - Key(y, k)$

(a) Si determini la complessità della procedura $My - Increase - Key$ nel caso pessimo e se ne dimostri la correttezza nel caso degli heap binomiali.

(b) Si proponga un algoritmo più efficiente della procedura $My - Increase - Key$ per incrementare il valore di una chiave in un heap binomiale indicandone la complessità nel caso pessimo.

(c) Si eseguano le operazioni indicate su un heap binomiale inizialmente vuoto:

- Insert 27, 17, 19, 20, 24, 12, 11, 10, 14, 18
- Extract-Min
- Decrease-Key(19,7)
- Delete(17)
- Decrease-Key(24,5)
- Extract-Min

Soluzione

`Risoluzione punto (a).`

Sia H un Heap Binomiale con n nodi allora la complessitá della procedura nel caso peggiore è $\log^2 n$ in quanto la complessità è data dal massimo grado dei nodi dei nodi ($\leq \log n$) moltiplicato la massima altezza dell'Heap ($\leq \log n$). Per dimostrare tale asserzione bisognerà

prima dimostrare alcune proprietà degli alberi binomiali nonchè enunciare la definizione degli Heap binomiali.

Definizione Heap Binomiali

Un Heap Binomiale H è un insieme di alberi binomiali tale che

- ciascun albero binomiale in H gode della proprietà $min - heap$
- per ogni $k \in \mathbb{N}$, H contiene al più $un\ solo$ albero binomiale di grado k

Proprietà Alberi Binomiali

Per ogni $k \in \mathbb{N}$ valgono le seguenti proprietà:

1. B_k ha 2^k nodi
2. L'altezza di B_k è k
3. La radice di un B_k ha grado k ed ogni altro nodo del B_k ha grado $< k$.
   Inoltre i figli della radice di un B_k sono radici di $B_{k-1}, B_{k-2}, ..., B_1, B_0$ nell'ordine dato.

Dimostrazione. Per la dimostrazione delle quattro proprietà procediamo per induzione.

caso base $k = 0$

1. B_0 ha $2^0 = 1$ nodi
2. B_0 ha altezza 0
3. La radice ha grado 0

passo induttivo Supponiamo che il lemma sia vero per $k-1$ con $k \geq 1$

1. B_k ha $\#(B_{k-1}) + \#(B_{k-1}) = 2\,2^{k-1} = 2^k$ nodi
2. B_k ha altezza $height(B_{k-1}) + 1 = k - 1 + 1 = k$
3. - La radice ha grado $degree(B_{k-1}) + 1 = k - 1 + 1 = k$.
   - Inoltre ciascun altro nodo appartiene ad un B_{k-1} e per l'ipotesi induttiva ha grado $\leq k - 1$ che è $< k$.

- Il primo figlio di un B_k è radice di un B_{k-1}. Inoltre per ipotesi induttiva i successivi $k-1$ figli della radice di un B_k sono radici di $B_{k-2}, B_{k-3}, ..., B_1, B_0$ nell'ordine.

□

Dalla definizione degli Heap Binomiale e dalla proprità 3 degl'alberi binomiali si evince che la radice dell'albero binomiale di grado massimo in H, ha grado maggiore di tutti gli altri nodi mentre dalla definizione di Heap Binomiale e dalla proprità 2 degl'alberi binomiali si evince che l'albero binomiale di grado massimo in H, ha altezza maggiore di tutti gli altri alberi. In definitiva sia B_k l'albero binomiale di grado massimo in H, allora la sua altezza sarà k e la sua radice avrà grado k inoltre si ha la relazione $2^k \leq n$ da cui $k \leq \lfloor \log n \rfloor$.

`Risoluzione punto (b).`

Una procedura migliore avente complessità $O(\log n)$ è quella mostrata sotto

---

function $My - Increase - Key(H, x, k)$
$Delete(H, x)$
$y \leftarrow Alloc - Heap - Node(k)$
$Insert(H, y)$

La complessità è $O(\log n)$ in quanto sia $Insert(y)$ che $Delete(x)$ hanno complessità $O(\log n)$ mentre $Alloc - Heap - Node(k)$ ha complessità costante.

5 Heap di Fibonacci

Esercizio 1

Si illustrino gli heap di Fibonacci e le operazioni da essi supportate.

Soluzione

Definizione Heap di Fibonacci

Un Heap di Fibonacci è una collezione di alberi con la proprietà di Heap. Gli alberi *non devono* essere necessariamente alberi binomiali non ordinati in quanto ogni nodo può perdere al massimo un figlio.

Funzione Potenziale degli Heap di Fibonacci

Durante l'analisi ammortizzata delle operazioni si farà uso della seguente funzione potenziale

$$\Phi(H) = t(H) + 2m(H)$$

dove $t(H)$ è il numero di nodi nella lista delle radici mentre $m(H)$ il numero di nodi marcati (quelli che hanno perso un figlio). Inoltre si farà uso di un limite superiore $D(n)$ per il grado massimo di un nodo qualsiasi in un heap di fibonacci che si dimostra essere $\leq O(\log n)$

Operazioni sugli Heap di Fibonacci

Make-Heap(): Crea e restituisce un heap senza elementi. Per far ciò esegue l'inizializzazione della struttura dati che comporta l'esecuzione di un numero costante di operazioni(Complessità $O(1)$);

Union(H_1, H_2): Crea e restituisce un nuovo heap H contenete tutti i nodi di H_1 e H_2 (H_1 e H_2 vengono distrutti). Per far ciò concatena semplicemente le liste delle radici degli heap H_1 e H_2 e aggiorna il puntatore al minimo. Il costo ammortizzato di questa procedura è

$$\hat{c_{uni}} \;=\; c_{uni} + \Delta t + \Delta m \;=\; c_{uni} + 0 + 0 = O(1)$$

Insert(H, x): Inserisce nell'heap H il nodo x. Per far ciò la procedura inserisce x nella lista delle radici e aggiorna il puntatore al minimo. Il costo ammortizzato di questa

procedura è

$$\hat{c_{uni}} \;=\; c_{uni} + \Delta t + \Delta m \;=\; c_{uni} + 1 + 0 = O(1)$$

ExtractMin(H): Estrae dall'heap H il nodo con chiave minima restituendone il puntatore. Per far ciò la procedura inizia staccando il puntatore al nodo minimo dalla lista delle radici e riversando i suoi figli nella stessa lista. A questo punto confronta le varie radici aventi lo stesso grado a due a due e innesta l'albero con la radice avente chiave maggiore in quello con chiave minore finchè non rimane al più una radice avente grado k. Il costo reale della procedura è $O(D(n))$ per il processamento dei figli del minimo e $O(D(n) + t(h))$ per il processamento della lista delle radici nella fase di consolidamento. In definitiva il costo reale è $cext = O(D(n) + t(h))$ mentre il costo ammortizzato di questa procedura è

$$\hat{c_{ext}} \;=\; c_{ext} + \Delta t + \Delta m \;=\; O(D(n) + t(h)) + D(n) - t(H) \;=\; O(D(n))$$

a patto di scalare il potenziale per dominare la costante nascosta in $O(t(h))$.

Minimum(H): restituisce un puntatore al nodo con chiave minima nell'heap H.

Decrease-Key(H, x, k): assegna al nodo x nell'heap H una chiave con valore non maggiore del precendente. Per far ciò si assegna il valore k alla chiave di x e si staccano ricorsivamente tutti i nodi marcati consecutivi che vi sono tra il nodo x e la radice. Tutti i nodi staccati vengono inseriti nella radice. Il costo ammortizzato di questa procedura supponendo che vengano staccati d nodi è $\Delta t = d\ \Delta m \leq 2(-(d-1)+1) = 4 - 2d$

$$\hat{c_{key}} \;=\; c_{key} + \Delta t + \Delta m \;=\; O(d) + d - 2d + 4 \;=\; O(d) - d + 4 \;=\; O(1)$$

a patto di scalare il potenziale per dominare la costante nascosta in $O(d)$.

Delete(H, x): Cancella il nodo x dall'heap H. Per far ciò sfrutta la procedura *Decrease-Key* per decrementare a $-\infty$ il valore della chiave in x e farla risalire fino alla radice. Dopo di che, dato il valore della sua chiave il nodo verrà rimosso con la chiamata alla procedura *ExtractMin()*. Il costo ammortizzato di questa procedura è $O(D(n))$ dato

che *Decrease-Key* ha costo ammortizzato $O(1)$ e *ExtractMin()* ha costo ammortizzato $O(D(n))$.

Esercizio 2

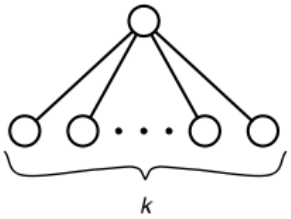
Si eseguano nell'ordine dato le seguenti operazioni su un heap di Fibonacci inizialmente vuoto:

- Insert 27, 17, 19, 20, 24, 12, 11, 10, 14, 18
- Extract-Min
- Decrease-Key(19,7)
- Delete(17)
- Decrease-Key(24,5)
- Extract-Min

Dopo ciascuna operazione, si disegni l'heap di Fibonacci risultante, tenendo conto che nella lista delle radici i nuovi elementi vanno inseriti a destra del minimo corrente e che il consolidamento della lista delle radici ha sempre inizio nel nodo a destra del minimo appena cancellato.

Esercizio 3

Per ogni $k \geq 5$, trovare una sequenza di operazioni sugli heap di Fibonacci che, a partire da una famiglia vuota di heap, ne costruisca uno formato da un solo albero avente la forma riportata nella figura a lato, oppure stabilire che una siffatta sequenza non esiste.



Soluzione

Un Heap di Fibonacci siffatto non potrebbe esistere per il seguente lemma: Sia x un nodo in un Heap di Fibonacci e sia $degree[x] = k$.
Siano $y_1, y_2, ..., y_k$ i figli di x nell'ordine in cui sono stati innestati in x allora

$$degree[y_i] \geq \max(i-2, 0), \text{ per } i = 1, 2, ..., k$$

Dimostrazione. Procediamo per casi:

Caso $i = 1$

si ha $degree[y_1] \geq 0 = \max(-1, 0)$

Caso $i \geq 2$

notiamo che quando innestiamo y_i in x, il nodo x ha già $y_1, ..., y_{i-1}$ tra i suoi figli, per cui $degree[y_i] = degree[y_{i-1}] \geq i - 1$. Ma da quando innestiamo y_i esso può perdere al massimo un nodo e quindi

$$degree[y_i] \geq i - 2$$

□

Infatti per il lemma il terzo figlio dovrebbe avere grado ≥ 1 invece come si può vedere dalla figura esso ha grado 0 e quindi non può esistere tale heap di fibonacci.

Esercizio 4

Dopo aver descritto gli Heap di Fibonacci, si indichi come implementare anche le operazioni

- SecondMinimum()
- ExtractSecondMinimum()

valutandone la complessità ammortizzata.

Soluzione

Ecco lo pseudo-codice delle due operazioni La complessità ammortizzata della SecondMi-

```
function SecondMinimum()
  x ← ExtractMin()
  y ← Minimum()
  Insert(x)
  return y
```

```
function ExtractSecondMinimum()
  x ← ExtractMin()
  y ← ExtractMin()
  Insert(x)
  return y
```

nimum() è uguale a

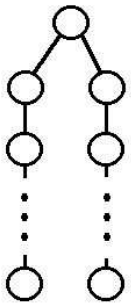
$$O(\log n) + O(1) + O(1) = O(\log n)$$

mentre la complessità ammortizzata della ExtractSecondMinimum() è uguale a

$$O(\log n) + O(\log n) + O(1) = O(\log n)$$

Esercizio 5

Trovare una sequenza di operazioni sugli heap di Fibonacci che a partire da una famiglia vuota di heap costruisca un heap formato da un solo albero avente la seguente forma

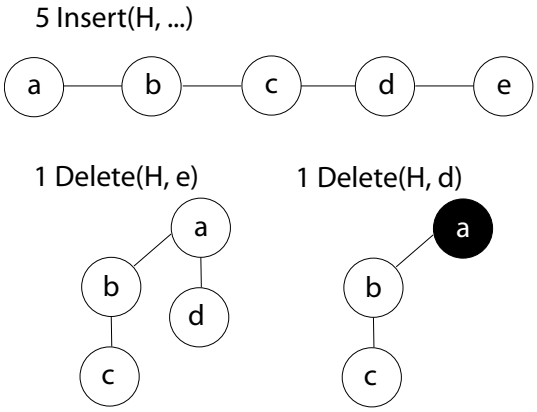


oppure stabilire che una siffatta sequenza di operazioni non esiste.

Soluzione

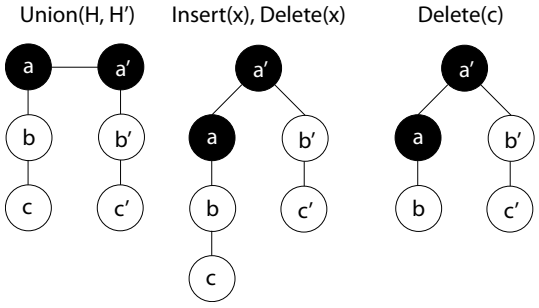
Un siffatto heap di fibonacci potrebbe risultare da una sequenza di iterazioni come quella nelle figure sottostanti:

Passo 1: Si eseguano 5 inserimenti e in seguito 2 cancellazioni



così da ottenere una lista di nodi verticale di lunghezza 3.

Passo 2: Si compiano le stesse operazioni del punto uno su un heap inizialmente vuoto diverso dal precedente. Dopo di che si eseguano le operazioni in figura sui 2 heap ottenuti...



così da ottenere l'heap richiesto nell'esercizio con $k = 2$.

Per estendere ad heap con $k > 2$ basterà costruire due heap con un albero binomiale ordinato di altezza k, privare gli alberi di tutti i nodi che non appartengono al "ramo" più lungo, e infine procedere col passo 2.

6 Cammini Minimi

Esercizio 1

Sia $G = (V, E)$ un grafo orientato con funzione peso $w : E \to \mathbb{R}^+$ e sorgente $s \in V$.

(a) Si definisca il grafo G'_s *dei cammini minimi* da s in G nonchè la nozione di *albero dei cammini minimi* da s in G (rispetto alla funzione peso w).

(b) Si dimostri che un arco $(u, v) \in E$ è presente in G'_s se e solo se

$$\delta(s, u) \ + \ w(u, v) \ = \ \delta(s, v)$$

(c) Si proponga un algoritmo efficiente per la costruzione del grafo dei cammini minimi.

(d) Siano $w_1 : E \to \mathbb{R}^+$ e $w_2 : E \to \mathbb{R}^+$ due funzioni peso su G e siano δ_1 e δ_2 le relative funzioni-distanza indotte. Si illustri un algoritmo efficiente per la costruzione di un albero dei cammini da s in G che risultino simultaneamente minimi rispetto a δ_1 e δ_2.

Soluzione

(a) **Grafo dei cammini minimi**: Sia $G = (V, E)$ un grafo pesato e sia $s \in V$ una sorgente assegnata, allora definiamo *Grafo dei cammini minimi* il sottografo $G_s = (V, E'_s)$ tale che E'_s contenga tutti gli archi che fanno parte di un qualsiasi cammino minimo.
Albero dei cammini minimi: Sia $G = (V, E)$ un grafo pesato e sia $s \in V$ una sorgente assegnata, allora definiamo *Albero dei cammini minimi* un albero radicato in s contenete esattamente un cammino minimo in G, da s, per ciascun nodo $v \in V$ raggiungibile da s in G.

(b)

$$(u, v) \in G_s \Leftrightarrow \delta(s, u) + w(u, v) = \delta(s, v)$$

Dimostrazione. ($\Rightarrow$) Supponiamo che $(u, v) \in G_s$ allora $\exists \pi = s \longmapsto u \to v$ con $w(\pi) = \delta(s, v)$. Per la sottostruttura ottima del problema si ha che se $\pi_{su} = s \longmapsto u$ allora $w(\pi_{su}) = \delta(s, u)$ e quindi

$$\delta(s, u) + w(u, v) = w(\pi_{su}) + w(u, v) = w(\pi) = \delta(s, v)$$

($\Leftarrow$) Supponiamo che $\pi_{su} = s \longmapsto u$ sia un cammino minimo da s a u, allora $w(\pi_{su}) = \delta(s, u)$. Sia $\pi_{sv} = s \longmapsto u \to v$ allora per ipotesi π_{sv} è minimo infatti

$$w(\pi_{sv}) = w(\pi_{su}) + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$$

□

(c) Per calcolare il grafo dei cammini minimi eseguo i seguenti passi:

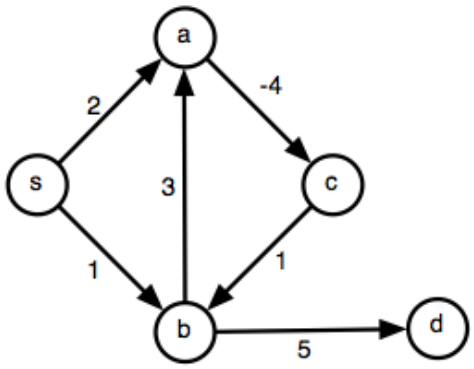
- Calcolo le distanze minime di ogni nodo da s applicando l'algoritmo di Bellman-Ford perchè non sappiamo che tipo di funzione peso ha il grafo. $[O(VE)]$.
- Per ogni arco controllo che valga la condizione $\delta(s, u) + w(u, v) = \delta(s, v)$. $[O(V + E)]$.

(d) I passi da eseguire per la costruzione di un albero dei cammini da s in G che risultino simultaneamente minimi rispetto a δ_1 e δ_2 sono:

- Calcolo le distanze minime δ_1 e δ_2 di ogni nodo da s applicando l'algoritmo di Dijkstra prima al grafo con funzione peso w_1 e poi a quello con funzione peso w_2. Dijkstra è applicabile in quanto entrambe le funzioni peso sono a valori positivi. $[O(E + V \log V)]$.
- Per ogni arco controllo che valgano simultaneamente le condizioni $\delta_1(s, u) + w_1(u, v) = \delta_1(s, v)$ e $\delta_2(s, u) + w_2(u, v) = \delta_2(s, v)$. $[O(V + E)]$.
- Effettuo una visita (BFS, DFS) del grafo per ottenere un albero. $[O(V + E)]$.

Esercizio 2

Si dia un algoritmo in grado di calcolare i cammini minimi nel seguente grafo dalla sorgente s a tutti i rimanenti nodi e si esibisca la traccia della relativa computazione. Inoltre si dimostri la correttezza dell'algoritmo dato e se ne valuti la complessità computazionale.



Soluzione

La figura mostra un grafo con pesi negativi e avente cicli, quindi applico l'algoritmo di Bellman-Ford.

Correttezza di Bellman-Ford. Si deve dimostrare che se $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}$ ammette cammini minimi (non ci sono cicli di peso) allora al termine dell'algoritmo di Bellman-Ford deve valere che $d_s[v] = \delta(s, v)$ per ogni $v \in V$ raggiungibile da s.

Dimostrazione. Dimostrazione. Sia v un vertice raggiungibile da s, e sia $\pi = (v_0, v_1, ..., v_k)$ un cammino minimo da $s = v_0$ a $v = v_k$. Il cammino π è semplice (perchè per ipotesi non ci sono cicli di peso negativo) e quindi $k \leq |V| - 1$. Vogliamo dimostrare, per induzione, che per $i = 0, 1, ..., k$ si ha $d_s[v_i] = \delta(s, v_i)$ dopo l'i-esima passata sugli archi di G, e che questa uguaglianza viene mantenuta nel seguito: infatti

- $d_s[v] \geq \delta(s, v) \quad \forall v \in V$
- i valori di $d_s[v]$ formano una successione monotona non crescente.

caso base: $d_s[v_0] = 0 = \delta(s, v_0)$ e questa stima viene mantenuta anche in seguito.

passo induttivo: si assuma che $d_s[v_{i-1}] = \delta(s, v_{i-1})$ dopo la (i-1)-esima passata. Poichè

l'arco (v_{i-1}, v_i) viene rilassato nella i-esima passata $d_s[v_i]$ vale:

$$\begin{aligned} d_s[v_i] &\leq d_s[v_{i-1}] + w(v_{i-1}, v_i) = && \text{(perchè l'arco viene rilassato)} \\ &= \delta(s, v_{i-1}) + w(v_{i-1}, v_i) = && \text{(per ipotesi induttiva)} \\ &= w(\pi_{s,v_{i-1}}) + w(v_{i-1}, v_i) = \\ &= w(\pi_{s,v_i}) = \\ &= \delta(s, v_i) \end{aligned}$$

Quindi $d_s[v_i] = \delta(s, v_i)$ dopo l'i-esima passata e ad ogni istante successivo, il che completa la dimostrazione. □

□

Per tracciare la computazione fornisco una tabella in cui le colonne contengono tutti i nodi e le righe l'evoluzione della stima della distanza. L'ordine di rilassamento dei nodi è s,a,b,c,d e, (si ricorda che l'algoritmo di Bellman-Ford rilassa tutti gli archi ad ogni passata facendo una scan(v) per ogni $v \in V$):

Azioni	s	a	b	c	d
inizio	0	∞	∞	∞	∞
scan(s)	0	2	1	∞	∞
scan(a)	0	2	1	-2	∞
scan(b)	0	2	1	-2	6
scan(c)	0	2	-1	-2	6
...	...	...	...	...	...
scan(a)	0	2	-1	-2	6
scan(b)	0	2	-1	-2	4
...	...	...	...	...	...
scan(d)	0	2	-1	-2	4

Esercizio 3

Sia (G, w) un grafo orientato $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}$.

(a) Che cosa si intende dicendo che (G, w) ammette cammini minimi da s?

(b) Si fornisca una caratterizzazione di tale proprietà (solo enunciato).

(c) Si descriva un algoritmo per stabilire se (G, w) ammette cammini minimi da s. Se ne dimostri la correttezza e se ne valuti la complessità computazionale.

Soluzione

(a) Che non vi siano cicli di peso negativo raggiungibili da s.

(b) Un grafo pesato (G, w) ammette cammini minimi sa una sorgente s se e solo non ci sono cicli di peso negativo raggiungibili da s in G.

(c) Sia $G = (V, E)$ un grafo orientato con funzione peso $w : E \to \mathbb{R}$. Si inserisca un nodo s in V e lo si colleghi a tutti gli altri nodi con archi di peso nullo. Allora l'esecuzione dell'algoritmo di Bellman-Ford dalla sorgente s sul nuovo grafo restituisce FALSE se nel grafo vi sono cicli di peso negativo. La Correttezza di Bellman-Ford è stata dimostrata nell'esercizio 2, mentre la complessità di questo algoritmo è $(V+1)(E+V) = O(VE)$. Da notare che la prima passata di Bellman-Ford sul nuovo grafo non fa altro che mettere la stima $d[v] = 0$ per ogni $v \in V$ diverso da s. Allora per diminuire la costante nascosta dalla notazione asintotica si può eliminare sia il nodo che tutti gli archi aggiunti e modificare l'inizializzazione dell'algoritmo ponendo stima $d[v] = 0$ per ogni $v \in V$. Anche in questo caso l'esecuzione dell'algoritmo ha complessità $O(VE)$ ma con una costante di proporzionalità molto più bassa.

Esercizio 4

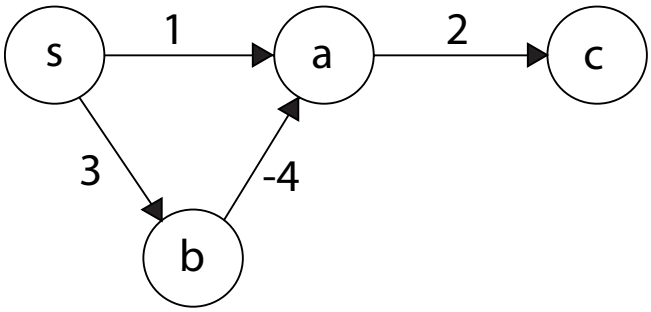
(a) Si descriva l'algoritmo di Dijkstra per i cammini minimi da singola sorgente specificando con chiarezza il dominio di applicabilità ed esibendo anche un esempio al di fuori di detto dominio in cui l'algoritmo di Dijkstra non dà risultati corretti.

(b) Si dimostri la correttezza dell'algoritmo di Dijkstra e se ne valuti la complessità computazionale.

Soluzione

(a) L'algoritmo di Dijkstra risolve il problema di cammini minimi con sorgente singola su un grafo orientato e pesato $G = (V, E)$ nel caso in cui tutti i pesi degli archi siano non negativi. L'algoritmo mantiene un insieme S contenente i vertici che sono andati a convergenza, ovvero, per tutti i vertici $v \in S$, vale $d[v] = \delta(s, v)$, e una coda di priorità contenete tutti i nodi in $V - S$ ordinati secondo la stima della propria distanza. L'algoritmo seleziona ripetutamente il nodo $u \in V - S$ con la minima stima di distanza, inserisce u in S, e rilassa tutti gli archi uscenti da u. Un esempio di grafo su cui Dijkstra fallisce è riportato di seguito insieme alla traccia della sua computazione espressa su una tabella le cui righe nere rappresentano l'evoluzione della funzione distanza durante le varie passate mentre la riga rossa è la distanza corretta dei nodi

s	a	b	c
0	∞	∞	∞
0	1	3	∞
0	1	3	3
0	-1	3	3
0	-1	3	1



(b) **Correttezza di Dijkstra**. Sia $G = (V, E)$ un grafo con funzione peso $w : E \to \mathbb{R}^+$, allora al termine dell'algoritmo di Dijkstra deve valere che $d_s[v] = \delta(s, v)$ per ogni $v \in V$ raggiungibile da s.

Dimostrazione. Bisogna dimostrare che, per ogni nodo $v \in V$, si ha $d_s[v] = \delta(s, v)$ al momento in cui v viene inserito nell'insieme S, e che questa uguaglianza viene mantenuta nel seguito: infatti

- $d_s[v] \geq \delta(s, v) \quad \forall v \in V$
- i valori di $d_s[v]$ formano una successione monotona non crescente.

Supponiamo per assurdo che in un momento della nostra iterazione stiamo per inserire in S un nodo v tale che $v \in V - S$, $d_s[v] > \delta(s, v)$ e $d_s[v] = min\{\, d_s[v] : u \in V - S \,\}$. Sia $\pi = (v_0, v_1, ..., v_k)$ con $v_0 = s$ e $v_0 = v$ un cammino minimo in G da s a v. Sia $i = min_j\{\, v_j : v_j \in V - S \,\}$ il più piccolo indice dei nodi che stanno in $V - S$, allora $d_s[v_i] = \delta(s, v_i)$. Ma

$$d_s[v] > \delta(s, v) = w(\pi) = w(v_0, ..., v_i) + w(v_i, ..., v_k) = \delta(s, v_i) + w(v_i, ..., v_k) \geq \delta(s, v_i) = d_s[v_i]$$

che contraddice la minimalità di $d_s[v]$. Quindi quando selezioniamo un nodo e lo mettiamo in S vale che $d_s[v] = \delta(s, v)$ e dato che durante le iterazioni questa uguaglianza non cambia la nostra dimostrazione si può dire conclusa. □

Esercizio 5

Sia $G = (V, E)$ un grafo orientato con funzione peso $w : E \to \mathbb{R}^+$, sorgente $s \in V$ e tale inoltre che contenga esattamente due nodi in V contrassegnati come "speciali". Diciamo che un cammino da u a v in G è ammissibile se esso contiene al più un solo nodo speciale (compresi gli estremi). Si descriva un algoritmo per il calcolo dei cammini ammissibili minimi dalla sorgente s a tutti i nodi del grafo.

Soluzione

Siano $t_1, t_2 \in V$ i due nodi speciali, allora l'algoritmo proposto si sviluppa nei seguenti passi che hanno complessità $O(E + V \log V)$ (la complessità dell'algoritmo di Dijkstra):

- Si consideri l'insieme dei nodi $V_1 = \{v : v \in V, v \neq t_1\}$ e l'insieme degli archi $E_1 = \{(u, v) : u \neq t_1, v \neq t_1\}$ e si esegua l'algoritmo di Dijkstra (funzione dei pesi a valori positivi) sul grafo $G_1 = (V_1, E_1)$ in modo da calcolare il cammino minimo e la relativa distanza per ogni nodo raggiungibile da s.

- Si consideri l'insieme dei nodi $V_2 = \{v : v \in V, v \neq t_2\}$ e l'insieme degli archi $E_2 = \{(u, v) : u \neq t_2, v \neq t_2\}$ e si esegua l'algoritmo di Dijkstra (funzione dei pesi a valori positivi) sul grafo $G_2 = (V_2, E_2)$ in modo da calcolare il cammino minimo e la relativa distanza per ogni nodo raggiungibile da s.

- Infine si esegua il seguente codice

  for $v \in V$ **do**
    if $d_1[v] \leq d_2[v]$ **then**
      $d[v] = d_1[v]$
      $prev[v] = prev_1[v]$
    else
      $d[v] = d_2[v]$
      $prev[v] = prev_2[v]$
    end if
  end for

Esercizio 6

Dopo aver definito la nozione di chiusura transitiva di un grafo, si descriva un algoritmo efficiente per il suo calcolo e se ne valuti la complessità computazionale.

Soluzione

Chiusura Transitiva. Dato un grafo $G = (V, E)$, si definisce chiusura transitiva di G il grafo $G' = (V, E')$ dove $E' = \{(u, v) \in E \ : \ v$ è raggiungibile da $u\}$. Un algoritmo efficiente con complessità $O(V^3)$ è dato dal seguente pseudo-codice:

```
for i ← 1 to |V| do
  for j ← 1 to |V| do
    if i = j ∨ (i, j) ∈ E then
      t[i, j] ← 1
    else
      t[i, j] ← 0
    end if
  end for
end for
for k ← 1 to |V| do
  for i ← 1 to |V| do
    for j ← 1 to |V| do
      t[i, j] ← t[i, j] ∨ (t[i, k] ∧ t[k, j])
    end for
  end for
end for
```

Esercizio 7

In analogia con la nozione di distanza (minima) tra due nodi, si proponga una definizione della nozione di distanza massima tra due nodi in un grafo orientato pesato. Quindi si descriva un algoritmo per determinare le distanze massime di tutti i nodi da una data sorgente $s \in V$ in un grafo orientato aciclico $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}^+$ e se ne valuti la complessità computazionale.

Soluzione

Distanza Massima. $\mu(s, v) = \sup\{w(\pi) : \pi \in Paths(G, s, v)\}$. **Cammini Massimi**. Un grafo ammette cammini massimi da s se e solo se non vi sono cicli di peso positivo raggiungibili da s. **Algoritmo Calcolo Cammini Massimi**. Basta eseguire i seguenti passi:

- Assegno a ogni arco il proprio peso cambiato di segno. [$O(V + E)$]
- Eseguo l'algoritmo per DAG (perchè il grafo è aciclico) e calcolo i cammini minimi. [$O(V + E)$]
- Assegno ad ogni nodo la distanza calcolata cambiata di segno. [$O(V)$]

Complessità finale $O(V + E)$

Esercizio 8

Sia $G = (V, E)$ un grafo orientato con funzione peso $w : E \to \mathbb{R}^+$ e sorgente $s \in V$.

(a) Si definisca il grafo G'_s dei cammini minimi da s in G.

(b) Si dimostri che ogni possibile cammino da s in G'_s è un cammino minimo in G.

Soluzione

(a) Sia $G = (V, E)$ un grafo qualsiasi e sia $s \in V$ una sorgente assegnata, allora definiamo *Grafo dei cammini minimi* il sottografo $G_s = (V, E'_s)$ tale che E'_s contenga tutti gli archi che fanno parte di un qualsiasi cammino minimo.

(b)

$$MinPaths(G, s) \neq 0 \;\Rightarrow\; Paths(G'_s, s) = MinPaths(G, s)$$

Dimostrazione. Dalla definizione di *Grafo dei cammini minimi* segue che

$$Paths(G'_s, s) \supseteq MinPaths(G, s)$$

Supponiamo per assurdo che

$$Paths(G'_s, s) \not\subseteq MinPaths(G, s)$$

Allora supponiamo che esista $\pi \in Paths(G'_s, s)$ di **lunghezza minima** tale che $\pi \notin MinPaths(G, s)$. Si ponga $\pi = (v_0, v_1, ..., v_k)$ con $v_0 = s$, allora $k \geq 1$ perchè il cammino di lunghezza 0 è banalmente contenuto in entrambi gli insiemi, inoltre per la minimalità di π si ha che $\pi' = (v_0, v_1, ..., v_{k-1}) \in MinPaths(G, s)$ e quindi $w(\pi') = \delta(s, v_{k-1})$. Ma

$$w(\pi) \;=\; w(\pi') + w(v_{k-1}, v_k) \;=\; \delta(s, v_{k-1}) + w(v_{k-1}, v_k) \;=\; \delta(s, v_k)$$

ovvero $\pi \in MinPaths(G, s)$, assurdo.

Quindi $Paths(G'_s, s) \subseteq MinPaths(G, s)$ da cui

$$Paths(G'_s, s) = MinPaths(G, s)$$

□

Esercizio 9

Sia $G = (V, E)$ un grafo orientato con funzione peso $w : E \to \mathbb{R}^+$ e siano s_1 ed s_2 due nodi distinti di G. Si descriva un algoritmo efficiente per calcolare una partizione (V_1, V_2) di V tale che:

- $\delta(s_1, u) \leq \delta(s_2, u)$, per ogni $u \in V_1$, e
- $\delta(s_2, v) \leq \delta(s_1, v)$, per ogni $v \in V_2$,

dove δ è la funzione distanza in (G, w).

Soluzione

L'algoritmo segue i seguenti passi:

- Modifico l'inizializzazione in modo che ubbidisca alla seguente relazione

$$d[v] = \begin{cases} 0 & \text{se } v = s_1 \text{ o } v = s_2 \\ \infty & \text{altrimenti} \end{cases}$$

- Eseguo l'algoritmo di Djikstra (funzione peso a valori positivi). [$O(E + V \log V)$].

Esercizio 10

Sia $G = (V, E)$ un grafo orientato, $q \in V$ un nodo assegnato di G e $w : E \to \mathbb{R}^+$ una funzione peso in G a valori positivi. Un q-cammino in G da u a v è un cammino in G da u a v passante per q. Un q-cammino in G da u a v si dice minimo se il suo peso è minimo (relativamente alla funzione peso w) rispetto a tutti i q-cammini in G da u a v.
Si descriva un algoritmo efficiente che calcoli per ciascuna coppia di nodi $(u, v) \in V \times V$ un q-cammino minimo da u a v e se ne valuti la complessità computazionale.

Soluzione

L'algoritmo da me pensato svolge i seguenti passi:

- Eseguo l'algoritmo di Dijkstra con sorgente q sul grafo di partenza ottenendo le minime distanze da q a tutti gli altri nodi. [$O(E + V \log V)$].
- Inverto tutti gli archi del grafo di partenza. [$O(V + E)$].
- Eseguo nuovamente l'algoritmo di Dijkstra con sorgente q sul grafo trasposto ottenendo le minime distanze da tutti i nodi a q. [$O(E + V \log V)$].
- Pongo
  $d[u, v] = d_2[u] + d_1[v]$
  $prev[u, v] = prev_1[v]$.
  $[O(V^2)]$.

La complessità finale dovrebbe essere $O(E + V^2)$.

Esercizio 11

Sia dato un grafo orientato $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}^+$. Assegnati inoltre un insieme di nodi $V_0 \subset V$ ed un nodo $t \in V - V_0$ si ponga

$$\delta(V_0, t) = \min_{v \in V_0} \delta_G(v, t)$$

dove δ_G è la funzione distanza in (G, w). Si progetti un algoritmo per calcolare $\delta(V_0, t)$ e se ne valuti la complessità computazionale in funzione di $|V|$, $|V_0|$ e $|E|$.

Soluzione

L'algoritmo esegue i seguenti passi:

- Modifico l'inizializzazione in modo che ubbidisca alla seguente relazione

$$d[v] = \begin{cases} 0 & \text{se } v \in V_0 \\ \infty & \text{se } v \in V - V_0 \end{cases}$$

- Eseguo l'algoritmo di Dijkstra sul grafo $G(V, E)$. [$O(E + V \log V)$].
- Estraggo $d[t]$ che corrisponderà al $\min_{v \in V_0} \delta_G(v, t)$. [$O(1)$].

Un altro algoritmo che risolve il problema è il seguente:

- Inverto tutti gli archi del grafo di partenza calcolando il grafo trasposto di G. $[O(V + E)]$.
- Eseguo l'algoritmo di Dijkstra con sorgente t sul grafo trasposto ottenendo le minime distanze da tutti i nodi a t. [$O(E + V \log V)$].
- Estraggo $\min_{v \in V_0} d[v]$ che corrisponderà al $\min_{v \in V_0} \delta_G(v, t)$. [$O(V)$]

7 Alberi di Copertura Minimi

Esercizio 1

Si descriva l'algoritmo di Prim e se ne illustri un'implementazione efficiente.

Soluzione

L'algoritmo di Prim costruisce l'albero di copertura minimo per un grafo pesato $G = (V, E)$. L'albero A parte da un arbitrario vertice radice r e cresce finchè non copre tutti i vertici in V. Ad ogni passo viene aggiunto all'albero l'arco con il peso minimo che collega un vertice in A ad un vertice in $V - A$. Un implementazione efficiente viene sviluppata sfruttando una coda di priorità per ordinare gli i vertici in base al minimo arco che li collega all'albero, in modo da estrarre velocemente l'arco di peso minimo da aggiungere all'albero. Di seguito vi è lo pseudo-codice della procedura: Usando un heap binario o uno binomiale la com-

```
for v ∈ V do
   d[v] ← ∞
   prev[v] ← nil
end for
d[r] ← 0
Q ← make-queue(V,d)
while Q ≠ ∅ do
   u ← extract-min(Q)
   for v ∈ Adj[u] do
      if v ∈ Q and d[v] > w(u, v) then
         d[v] ← w(u, v)
         prev[v] ← u
      end if
   end for
end while
```

plessità della procedura è $O(E \log V)$ mentre usando un heap di fibonacci la complessità della procedura scende a $O(E + V \log V)$.

Esercizio 2

Siano T_1 e T_2 due MST distinti di un dato grafo non-orientato connesso e pesato (G, w). Si verifichi che:

(a) $\min_{e \in T_1} w(e) = \min_{e' \in T_2} w(e')$

(b) $\max_{e \in T_1} w(e) = \max_{e' \in T_2} w(e')$

Soluzione

(a) Supponiamo per assurdo che $\min_{e \in T_1} w(e) \neq \min_{e' \in T_2} w(e')$. Poichè T_1 e T_2 sono diversi, senza perdita di generalità possiamo supporre che $\min_{e \in T_1} w(e) < \min_{e' \in T_2} w(e')$. Sia $e = (u, v)$ l'arco con peso strettamente minore di tutti gli archi in T_1 e T_2. Dato che T_2 connette tutti i nodi di G allora ci sarà un arco (u', v), con $u' \neq u$, che connette v in T_2. A questo punto se sostituiamo l'arco (u', v) con l'arco (u, v) in T_2 otteniamo un albero T_2' che è un $Spanning\ Tree$ e ha una somma degli archi strettamente minore di T_2 il che è assurdo in quanto supposto che T_2 sia un MST. Quindi deve essere

$$\min_{e \in T_1} w(e) = \min_{e' \in T_2} w(e')$$

(b) Supponiamo per assurdo che $\max_{e \in T_1} w(e) \neq \max_{e' \in T_2} w(e')$. Poichè T_1 e T_2 sono diversi, senza perdita di generalità possiamo supporre che $\max_{e \in T_1} w(e) < \max_{e' \in T_2} w(e')$. Sia $e' = (u', v')$ l'arco con peso strettamente maggiore di tutti gli archi in T_1 e T_2. Dato che T_1 connette tutti i nodi di G allora ci sarà un arco (u, v'), con $u \neq u'$, che connette v' in T_1 con peso strettamente minore di (u', v'). A questo punto se sostituiamo l'arco (u, v') con l'arco (u', v') in T_2 otteniamo un albero T_2' che è un $Spanning\ Tree$ e ha una somma degli archi strettamente minore di T_2 il che è assurdo in quanto supposto che T_2 sia un MST. Quindi deve essere

$$\max_{e \in T_1} w(e) = \max_{e' \in T_2} w(e')$$

Esercizio 3

Si descrivano i passi "blu" e quelli "rossi" negli algoritmi per il calcolo del minimum spanning tree. Quindi si enunci e si dimostri il cosiddetto lemma della "invariante del colore".

Soluzione

(a) **Passo Blu:** si selezioni un taglio non attraversato da nessun arco blu, e si colori di blu uno degli archi che attraversa il taglio e ha peso minimo.
Passo Rosso: si selezioni un ciclo semplice privo di archi rossi, e si colori di rosso uno degli archi di peso massimo facenti parti del ciclo.

(b) **Lemma.** Ad ogni passo di colorazione esiste sempre un MST che contiene tutti gli archi blu e nessuno rosso.

Dimostrazione. Il lemma sarà dimostrato per induzione.

- **Passo 0:** inizialmente non essendo nessun arco colorato l'invariante è banalmente vera (una proprietà su un insieme vuoto è sempre vera).
- **Passo k+1:** supponiamo che al **passo k** si ha un MST che indichiamo con T che contiene tutti gli archi blu e nessun arco rosso e vediamo cosa accade al **passo k+1**:
  1. **il passo k+1 seleziona il taglio** (V_1, V_2) **non attraversato da nessun arco blu e colora di blu l'arco** $e = (u, v)$**.**
     - Se $e \in T$ l'invariante del colore è banalmente verificata.
     - Se $e \notin T$ allora si consideri il cammino da u a v in T. Sia (u_1, v_1) l'arco che attraversa il taglio (V_1, V_2) nel suddetto cammino, il quale non sarà colorato: non può essere rosso perchè T non contiene archi rossi e non può essere blu percheè tutti gli archi che attraversano (V_1, V_2) non possono essere blu. Consideriamo T' tale che $T' = (T - \{(u_1, v_1)\}) \cup \{(u, v)\}$. Poichè $w(u, v) \leq w(u_1, v_1)$ allora si ha la relazione $w(T') \leq w(T) \leq w(T')$

da cui $w(T') = w(T)$, cioè T' è un MST che contiene tutti gli archi blu e nessun arco rosso al passo $k+1$.

2. **il passo k+1 seleziona un ciclo semplice $\pi = (u_1, u_2, ..., u_k)$ privo di archi rossi e colora di rosso l'arco $e = (u_1, u_2)$.**
   - Se $e \notin T$ l'invariante del colore è banalmente verificata.
   - Se $e \in T$ allora $T - e$ ha due componenti connesse che determinano una partizione (taglio) (V_1, V_2) di V. Sia e' un arco sul ciclo π che attraversa il taglio (V_1, V_2) (c'è ne sarà almeno uno perchè π è un ciclo), il quale non sarà colorato: non è blu perchè $e' \notin T$ (se $e \in T \Rightarrow e' \notin T$, per $e' \neq e, \quad e', e \in \pi$) e non è rosso perchè π non contiene archi rossi. Consideriamo T' tale che $T' = (T - \{e\}) \cup \{e'\}$. Poichè $w(e') \leq w(e)$ allora si ha la relazione $w(T') \leq w(T) \leq w(T')$ da cui $w(T') = w(T)$, cioè T' è un MST che contiene tutti gli archi blu e nessun arco rosso al passo k+1.

□

Esercizio 4

Si etichettino i seguenti punti del piano $(1,3)$, $(2,1)$, $(2,6)$, $(1,6)$, $(6,6)$, $(3,4)$, $(3,6)$, $(5,2)$, $(5,7)$ con le lettere da A ad I, rispettivamente. Prendendo come pesi le distanze tra gli estremi degli archi (distanza euclidea), si consideri il grafo pesato (G,w) avente i seguenti archi (A,B), (A,C), (C,G), (F,G), (B,F), (B,E), (B,H), (B,D), (D,H), (E,I), (G,I), (E,F), (E,H).

(a) Si illustri l'esecuzione degli algoritmi di Kruskal e di Prim sul grafo pesato (G,w), imponendo tra gli archi l'ordinamento lessicografico nei casi di parità tra i pesi (quindi, ad esempio, l'arco (A,C) precede (B,F) che, a sua volta, precede (B,H)). In particolare, per quanto riguarda l'algoritmo di Prim, lo si esegua a partire dal nodo A.
Nota: per semplificare i calcoli, si utilizzino i quadrati delle distanze al posto delle distanze stesse.

(b) Si spieghi brevemente, ma in maniera rigorosa, perchè pur utilizzando i quadrati delle distanze si ottengono MST validi per il grafo originale (G,w), con le distanze euclidee.

Soluzione

Ecco la costruzione del grafo con i relativi pesi degl'archi in ordine lessicografico:

$(A, B) = 1 + 4 = 5$

$(A, C) = 1 + 9 = 10$

$(B, D) = 1 + 25 = 26$

$(B, E) = 16 + 25 = 41$

$(B, F) = 1 + 9 = 10$

$(B, H) = 9 + 1 = 10$

$(C, G) = 1 + 0 = 1$

$(D, H) = 16 + 16 = 32$

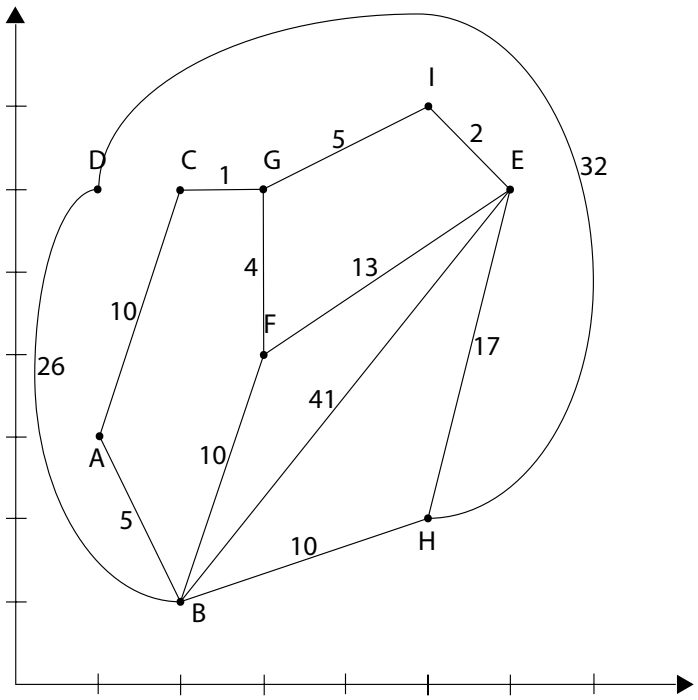
$(E, F) = 9 + 4 = 13$

$(E, H) = 1 + 16 = 17$

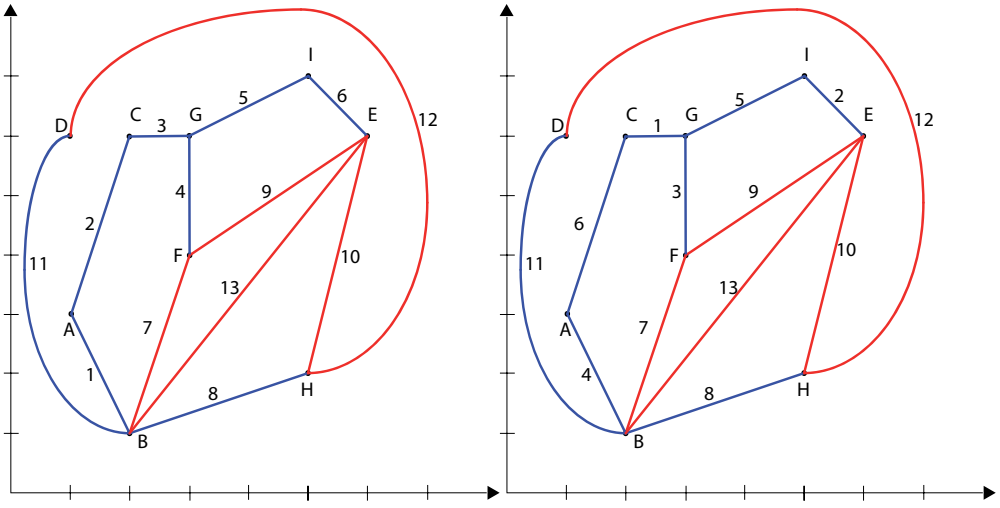
$(E, I) = 1 + 1 = 2$

$(F, G) = 0 + 4 = 4$

$(G, I) = 4 + 1 = 5$



(a) Ecco i grafi colorati secondo l'algoritmo di Kruskal a destra e di Prim a sinistra; nelle figure indico con dei numeri crescenti l'ordine di colorazione degl'archi:



(b) La prima cosa che bisogna notare è che gli algoritmi di Kruskal e Prim non usano il peso degli archi per calcolare il peso degli MST che forniscono in output, ma si servono di tale misura solo come criterio di ordinamento degli archi. Quindi, una volta fissato tale ordinamento il peso degli archi lo si può "buttare via"; dato che il quadrato delle distanze non influenza tale ordinamento, come si può vedere sotto,

$$\forall a, b \in \mathbb{N} \quad a = b \;\Rightarrow\; a^2 = b^2, \quad a < b \;\Rightarrow\; a^2 < b^2$$

allora gli MST calcolati sono validi.

Esercizio 5

Definizione Dato un grafo non orientato $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}$ e dato un sottoinsieme U di V, il sottografo di (G, w) indotto da U è il grafo pesato ottenuto rimuovendo da G tutti i nodi non appartenenti a U e tutti gli archi che toccano qualche nodo non appartenente a U.
Sia quindi $G = (V, E)$ un grafo connesso non orientato con funzione peso $w : E \to \mathbb{R}$.
Per ciascuna delle seguenti asserzioni, stabilire se è necessariamente vera oppure no, motivando adeguatamente le risposte.

(a) Sia $T = (V, T)$ un albero ricoprente minimo di (G, w) e sia $e \in T$ un suo arco. Rimuovendo e da T si formano due alberi T_1 e T_2 insistenti rispettivamente sugli insiemi di nodi V_1 e V_2. Allora,

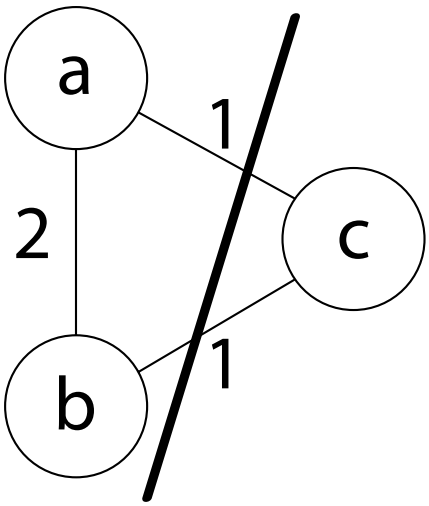
- e è un arco di peso minimo che attraversa il taglio (V_1, V_2) di G;
- T_i è un albero ricoprente minimo del sottografo di (G, w) indotto da V_i, per $i = 1, 2$.

(b) Sia (V_1, V_2) un taglio di G, sia e un arco di peso minimo che attraversa il taglio (V_1, V_2), e sia $T_i = (V_i, T_i)$ un albero ricoprente minimo del sottografo di (G, w) indotto da V_i, per $i = 1, 2$. Allora, il grafo $(V, T_1 \cup T_2 \cup e)$ è un albero ricoprente minimo di (G, w).

Soluzione

(a)
- Supponiamo che e non fosse un arco di peso minimo che attraversa il taglio (V_1, V_2) e consideriamo l'arco minimo che lo attraversa, sia esso e'. Sia T' lo *spanning tree* tale che $\xi[T'] = (\xi[T] - e) \cup e'$. Ma $w(T') < w(T)$ il che è assurdo in quanto supposto che T fosse un MST. Quindi e ha peso minimo.
- Se T_i non fosse minimo allora troveremmo un altro albero ricoprente T_i' di peso strettamente minore. Consideriamo T' tale che $\xi[T'] = \xi[T_i'] \cup e \cup \xi[T_j]$ con $i, j = 1, 2$ e $j \neq i$. Allora avremo $w(T') < w(T)$ il che è assurdo in quanto supposto che T fosse un MST. Quindi T_i è un MST per V_i, per $i = 1, 2$.

(b) Se questa proprietà fosse vera allora avremmo criterio molto forte per costruire i nostri MST perchè basterebbe partire dai singoli nodi per arrivare velocemente ad un MST completo. Quindi per invalidare questa affermazione mi basta mostrare un controesempio:



Come si può vedere dalla figura, se prendiamo il minimo tra (a, c) e (b, c) e lo uniamo ai due MST indotti $T_1 = (\{a, b\}, \{(a, b)\}$ e $T_2 = (\{c\}, \varnothing)$ otteniamo un albero ricoprente non di peso minimo. Questo completa l'invalidazione della tesi.

Esercizio 6

Sia $G = (V, E)$ un grafo non orientato connesso e sia $w : E \to \mathbb{R}$ una funzione peso su G. Sia inoltre $e \in E$ un arco di G. Si descriva un algoritmo per stabilire se l'arco e è contenuto in qualche MST di (G, w) e se ne valuti la complessità computazionale.

Soluzione

Supponendo che $e = (u, v)$ il mio algoritmo segue i seguenti passi:

- Eseguo l'algoritmo di Prim sul grafo G in modo da avere in T un MST. [$O(E + V \log V)$]
- Se $(u, v) \in T$ allora restituisco **true**.
- Se $(u, v) \notin T$ allora consideriamo il cammino π da u a v in T. Se esiste un arco in π con peso uguale ad $w(u, v)$ allora restituisco **true**, altrimenti restituisco **false**.

La complessità di questo algoritmo è $O(E + V \log V)$. Per confermare questo algoritmo dimostriamo il seguente lemma:
L'arco (u, v) appartiene ad un MST di G se e solo se non esistono cammini da u a v in G (diversi da (u, v)) tali che tutti gli archi hanno peso strettamente minore di $w(u, v)$.

Dimostrazione. ($\Rightarrow$) Sia T un MST di G e (u, v) un arco di T. Supponiamo per assurdo che esistano cammini da u a v in G (diversi da (u, v)) tali che tutti gli archi hanno peso minore di $w(u, v)$. Sia π uno di questi cammini. Allora $\pi \cup (u, v)$ è un ciclo e dato che l'arco e ha peso massimo nel ciclo esso sarà colorato di rosso. Ma ciò è assurdo per *l'invariante del colore* in quanto supposto che e fosse un arco di T e T può contenere solo archi di colore blu. ($\Leftarrow$) Sia G un grafo in cui non esistano cammini da u a v (diversi da (u, v)) tali che tutti gli archi hanno peso strettamente minore di $w(u, v)$. Supponiamo per assurdo che non esista un MST di G che contiene l'arco (u, v). Sia T uno di questi MST e sia π il cammino da u a v in T. Per ipotesi, π deve avere un arco di peso maggiore o uguale a $w(u, v)$, sia esso $(u'v')$. Consideriamo T' tale che $\xi[T'] = (\xi[T] - (u', v')) \cup (u, v)$. Si ha $w(T') \leq w(T)$, il che è assurdo in quanto se $w(T') = w(T)$ si contraddice l'ipotesi che non esistono MST che contengono l'arco (u, v) mentre se $w(T') < w(T)$ si contraddice l'ipotesi che T sia un MST. Quindi esiste almeno un MST che contiene l'arco (u, v). □

Considerando il lemma dimostrato sopra possiamo pensare ad un algoritmo per cui non serva calcolare un preventivamente un MST per controllare se un arco $e = (u, v)$ appartiene ad un qualsiasi MST di G, ma basta controllare se esiste un cammino da u a v con tutti archi aventi peso strettamente minore di $w(e)$. L'algoritmo pensato è il seguente:

- Si faccia una visita (DFS, BFS) partendo dal nodo u nel grafo G, con la condizione che ogni arco visitato abbia peso strettamente minore di $w(e)$. [$O(V + E)$].
- Se esiste un cammino da u a v nell'albero calcolato dalla visita allora restituisco **false**, altrimenti restituisco **true**.

Complessità $O(V + E)$.

8 Reti di Flusso

Esercizio 1

Si descriva il metodo di Ford-Fulkerson e se ne illustri un'esecuzione su una rete di flusso a piacere.

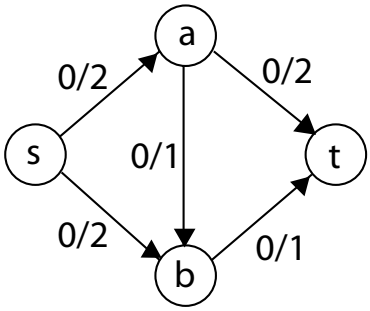
Soluzione

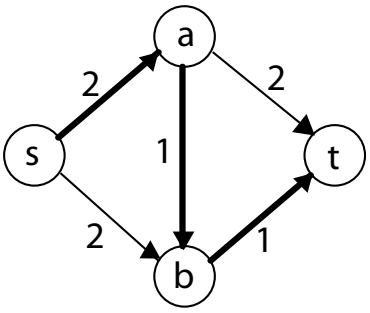
Il metodo di Ford-Fulkerson è un algoritmo molto semplice e ovvio per determinare un flusso massimo su una rete di flusso e segue i seguenti passi:

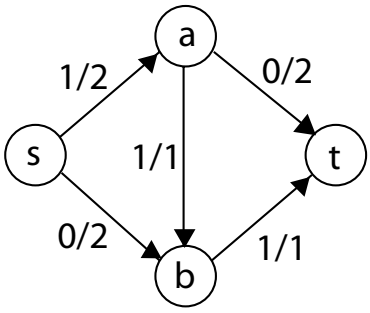
- per ogni coppia $(u, v) \in E$ pongo il flusso da u a v e da v a u uguale a 0: $f(u, v) = f(v, u) = 0$.

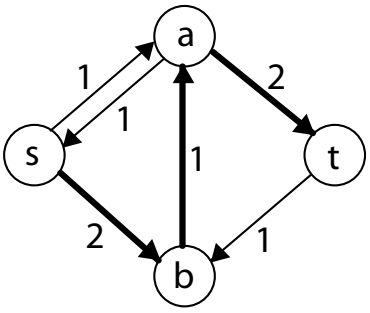
- finchè esiste un cammino p da s a t (aumentate) in G_f allora consideriamo la minima capacità residua $c_f(p)$ tra tutti gli archi del cammino trovato e poniamo $f(u, v) = f(u, v) + c_f(p)$ e $f(v, u) = -f(u, v)$ per ogni $(u, v) \in p$.

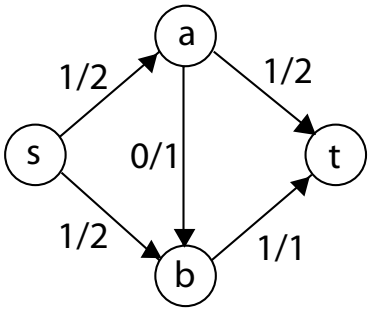
Questo algoritmo dipende molto da come viene scelto il cammino aumentate, infatti se esso non viene scelto bene, l'algoritmo potrebbe non terminare: il flusso potrebbe crescere senza però mai convergere ad una soluzione nel caso in cui il valore sia un numero irrazionale. Se però le capacità degli archi sono numeri interi (o razionali che sono sempre scalabili ad interi) allora l'algoritmo verrà eseguito in tempo $O(E|f^*|)$ dove $|f^*|$ è il valore del flusso massimo: infatti per controllare se esiste un cammino aumentate impieghiamo tempo $O(E)$ (da che il grafo è connesso il fattore V è trascurabile) e dato che nel caso peggiore esso incrementerà il flusso di un unità allora la ricerca del cammino sarà eseguita $|f^*|$ da cui $O(E|f^*|)$.
Di seguito ecco un esempio di esecuzione di questo algoritmo su una rete di flusso molto semplice di 4 nodi:

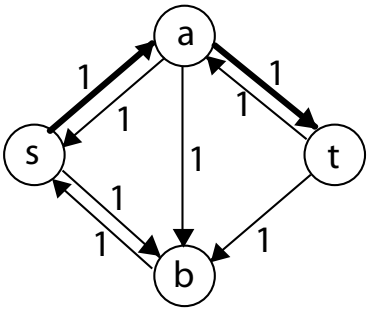

a
0/2
0/2
s
0/1
t
0/2
0/1
b

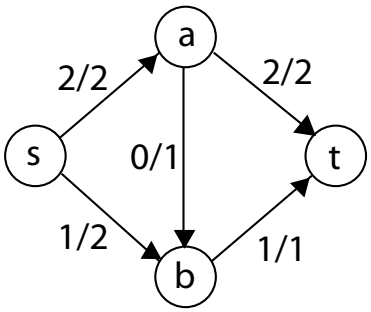

a
2
2
s
1
t
2
1
b


a
1/2
0/2
s
1/1
t
0/2
1/1
b


a
1
2
1
s
1
t
2
1
b


a
1/2
1/2
s
0/1
t
1/2
1/1
b


a
1
1
1
1
s
1
t
1
1
b


a
2/2
2/2
s
0/1
t
1/2
1/1
b

Esercizio 2

Dopo aver definito le nozioni di rete di flusso, flusso, taglio, flusso attraverso un taglio, capacità di un taglio, si enunci e si dimostri il teorema del "massimo flusso/minimo taglio".

Soluzione

(a) **Rete di flusso.** La rete di flusso è definita come un grafo orientato $G = (V, E)$ senza cappi tale che:

- Ad ogni arco $(u, v) \in E$ è associato un valore non negativo, $c(u, v) \geq 0$, detto capacità dell'arco. Se l'arco (u, v) non è presente in E allora assumiamo che esso abbia capacità nulla, cioè $c(u, v) = 0$.
- Nella rete di flusso ci sono due vertici speciali: il vertice **sorgente**, indicato con il simbolo s, ed il vertice **pozzo** (o **destinazione**), indicato con il simbolo t. I rimanenti vertici sono detti **vertici di transizione**.
- Ogni vertice giace su qualche cammino dalla sorgente al pozzo. Il grafo è quindi connesso e pertanto vale $|E| \geq |V| - 1$.

(b) **Flusso Reale.** Data una rete di flusso $G = (V, E)$ con capacità $c : V \times V \to \mathbb{R}_0^+$, il flusso reale in G è una funzione $\varphi : V \times V \to \mathbb{R}_0^+$ tale che:

- **Vincolo di capacità:**

$$\varphi(u, v) \leq c(u, v)$$

- **Conservazione del flusso:**

$$\sum_{v \in V} \varphi(u, v) = \sum_{v \in V} \varphi(v, u)$$

Flusso Netto. Data un rete di flusso $G = (V, E)$ con capacità $c : V \times V \to \mathbb{R}_0^+$ e flusso reale $\varphi : V \times V \to \mathbb{R}_0^+$ definiamo il flusso netto come segue:

$$f_\varphi(u, v) = \varphi(u, v) - \varphi(v, u)$$

(c) **Taglio.** Un taglio (S, T) in una rete di flusso $G = (V, E)$ è una partizione di V in S e $T = V - S$ tale che $s \in S$ e $t \in T$ (s=Sorgente, t=Pozzo).

(d) **Capacità di un Taglio.** La capacità di un taglio (S, T) è data dalla seguente relazione:

$$c(S, T) = \sum_{u \in S,\ v \in T} c(u, v)$$

(e) **Flusso attraverso un Taglio.** Il flusso attraverso un taglio (S, T) è data dalla seguente relazione:

$$f(S, T) = \sum_{u \in S,\ v \in T} f(u, v)$$

(f) **Massimo Flusso/Minimo Taglio.** Sia f^* un flusso in una rete di flusso (G, s, t, c) allora le seguenti condizioni sono equivalenti:

1. f^* è un flusso massimo in G.
2. Nella rete residua G_{f^*} non vi sono cammini aumentati.
3. $|f^*| = c(S, T)$ per qualche taglio (S, T) in G.

Dimostrazione.

1. $(1. \rightarrow 2.)$ Supponiamo per assurdo che f^* sia un flusso massimo in G e nella rete residua sia presente un G_{f^*} sia presente un cammino aumentate di flusso f'. Ma $|f^* + f'| > |f^*|$ sarà un flusso in G, il che contraddice la massimalità di f^*. Quindi non ci possono essere cammini aumentati nella rete residua G_{f^*}.
2. $(2. \rightarrow 3.)$ Supponiamo che la rete residua G_{f^*} non vi siano cammini aumentanti. Quindi G_{f^*} non contiene cammini da s a t. Siano $S = v \in V : v$ è raggiungibile da s e $T = V - S$, allora in G_{f^*} non vi saranno archi (u, v) per $u \in S$ e $v \in T$ in quanto supposto che in G_{f^*} non vi siano cammini aumentanti, e quindi $f(u, v) = c(u, v)$ in G, da cui $|f| = f(S, T) = c(S, T)$ per il seguente lemma: Sia f un flusso su una rete (G, s, t, c), e sia (S, T) un taglio di G. Allora il flusso netto attraverso (S, T) è $f(S, T) = |f|$.

3. $(3. \rightarrow 1.)$ Si ha

$$c(S^*, T^*) = |f^*| \ \leq \sup_{f \text{ flusso in } G} |f| \ \leq \min_{(S,T) \text{ taglio in } G} c(S, T) \ \leq \ c(S^*, T^*)$$

da cui

$$|f^*| \ = \sup_{f \text{ flusso in } G} |f|$$

che implica $|f| \leq |f^*|$ per ogni flusso f in G.

Pertanto f^* è un flusso massimo in G.

□

Esercizio 3

Sia G una rete di flusso e sia V l'insieme dei suoi vertici. Siano inoltre $f_1, f_2 : V \times V \to \mathbb{R}$ due flussi in G. Si consideri la funzione $f_1 + f_2$ definita da:

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \quad \forall (u, v) \in V \times V$$

- Si stabilisca quali proprietà dei flussi sono necessariamente vere per $f_1 + f_2$ e quali no.
- Si risponda ai medesimi quesiti per la funzione $\lambda f_1 + \mu f_2$ definita da

$$(\lambda f_1 + \mu f_2)(u, v) = \lambda f_1(u, v) + \mu f_2(u, v) \quad \forall (u, v) \in V \times V$$

dove $0 \leq \lambda, \mu \leq 1$ e $\lambda + \mu = 1$.

Soluzione

- Verifichiamo le proprietà di un flusso per $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$
  (a) **Antisimmetrica,** $f(u, v) = -f(v, u)$**.**
  $(f_1 + f_2)(u, v) \;=\; f_1(u, v) + f_2(u, v) \;=\; -f_1(v, u) - f_2(v, u) \;=$
  $-\left[f_1(u, v) + f_2(u, v)\right] \;=\; -(f_1 + f_2)(v, u)$
  (b) **Conservazione del flusso,** $\sum_{v \in V} f(u, v) = 0$**.**
  $\sum_{v \in V}(f_1 + f_2)(u, v) \;=\; \sum_{v \in V} f_1(u, v) + \sum_{v \in V} f_2(u, v) \;= 0 + 0 = 0$
  (c) **Vincolo di capacità,** $f(u, v) \leq c(u, v)$**.**
  $(f_1 + f_2)(u, v) \;=\; f_1(u, v) + f_2(u, v) \;\leq\; c(u, v) + c(u, v) \;=\; 2 \cdot c(u, v) \;\neq\; c(u, v)$

  Come dimostrato sopra valgono, le proprietà **Antisimmetrica** e della **Conservazione del flusso**, ma non vale il **Vincolo di capacità**.
- Verifichiamo le proprietà di un flusso per $(\lambda f_1 + \mu f_2)(u, v) = \lambda f_1(u, v) + \mu f_2(u, v)$, dove $0 \leq \lambda, \mu \leq 1$ e $\lambda + \mu = 1$:

(a) **Antisimmetrica,** $f(u,v) = -f(v,u)$**.**

$\lambda f_1 + \mu f_2)(u,v) \;=\; \lambda f_1(u,v) + \mu f_2(u,v) \;=\; -\lambda f_1(v,u) - \mu f_2(v,u) =$

$-\left[\lambda f_1(u,v) + \mu f_2(u,v)\right] \;=\; -(\lambda f_1 + \mu f_2)(v,u)$

(b) **Conservazione del flusso,** $\sum_{v \in V} f(u,v) = 0$**.**

$\sum_{v \in V}(\lambda f_1 + \mu f_2)(u,v) \;=\; \lambda \sum_{v \in V} f_1(u,v) + \mu \sum_{v \in V} f_2(u,v) \;= 0 + 0 = 0$

(c) **Vincolo di capacità,** $f(u,v) \leq c(u,v)$**.**

$(\lambda f_1 + \mu f_2)(u,v) \;=\; \lambda f_1(u,v) + \mu f_2(u,v) \;\leq\; \lambda c(u,v) + \mu c(u,v) \;=\; (\lambda + \mu) c(u,v) \;=$

$c(u,v)$

In questo caso tutte le proprietà di flusso sono valide.

Esercizio 4

$f : V \times V \to \mathbb{R}$ una funzione assegnata sulle coppie ordinate dei vertici di una rete di flusso $G = (V, E)$ con funzione capacità $c : E \to \mathbb{R}_0^+$, sorgente s e pozzo t. Si proponga un algoritmo efficiente per stabilire se

(a) f è un flusso in (G, c, s, t);

(b) f è un flusso massimo in (G, c, s, t)

e se ne valuti la complessità computazionale.

Soluzione

(a) Supponiamo che $|V| = n$ e rappresentiamo la funzione flusso f con una matrice

$$F = \begin{bmatrix} f_{1,1} & f_{1,2} & \dots & f_{1,n} \\ f_{2,1} & f_{2,2} & \dots & f_{2,n} \\ \dots & \dots & \dots & \dots \\ f_{n,1} & f_{n,2} & \dots & f_{n,n} \end{bmatrix}$$

dove $f_{i,j} = f(i, j)$ e $i, j \in V$. Per verificare che f sia un flusso dobbiamo verificare le proprietà: antisimmetria, conservazione del flusso e vincolo di capacità:

- **Antisimmetria**

  Verifichiamo che $f(u, v) = -f(v, u)$ per $u, v \in V$:

```
for i = 1 to n do
  for j = i to n do
    if F[i, j] ≠ −F[j, i] then
      return false
    end if
  end for
end for
return true
```

- **Conservazione del flusso.**

  Verifichiamo che $\sum_{v \in V} f(u,v) = 0$ per $u, v \in V - \{s,t\}$:

```
sum ← 0
for i = 2 to n − 1 do
  for j = 2 to n − 1 do
    sum ← sum + F[i, j]
  end for
  if sum ≠ 0 then
    return false
  end if
end for
return true
```

- **Vincolo di capacità**

  Verifichiamo che $f(u,v) \leq c(u,v)$ per $u, v \in V$:

```
for i = 1 to n do
  for j = 1 to n do
    if F[i, j] > C[i, j] then
      return false
    end if
  end for
end for
return true
```

Dato che ogni funzione per il controllo delle proprietà ha complessità $O(V^2)$ allora l'algoritmo che le richiama tutte avrà complessità $O(V^2)$.

(b) Per verificare che f sia un flusso massimo procediamo così:

- Verifichiamo che f soddisfi le proprietà per un flusso. [$O(V^2)$]
- Calcoliamo la rete residua. [$O(E)$].
- Effettuiamo una visita della rete residua a partire da s. Se t è raggiungibile da s allora f non è massimo altrimenti lo è.