

Algoritmi e Strutture Dati

Capitolo 12

Minimo albero ricoprente:
Algoritmi di Prim e di Borůvka

Domanda di approfondimento: soluzione

Confrontare le complessità computazionali delle implementazioni di Kruskal con alberi **QuickFind** ed alberi **QuickUnion** (senza euristiche di bilanciamento).

Soluzione: Denotiamo con $T(\text{UF}(n,m))$ il costo per eseguire le $2m$ operazioni di **find** e $n-1$ operazioni di **union**. Si osservi innanzitutto che con alberi **QF** si ha che $T_{\text{QF}}(n,m) = O(m \cdot \log n + n^2)$, mentre con alberi **QU** si ha $T_{\text{QU}}(n,m) = O(m \cdot n)$. Quindi, poiché $m \cdot n = \omega(m \cdot \log n)$ e $m \cdot n = \Omega(n^2)$ [in quanto $m = \Omega(n)$], ne consegue che $T_{\text{QU}}(n,m) = \Omega(T_{\text{QF}}(n,m))$. Ci si domanda ora se **per ogni** valore di m si ha $T_{\text{QU}}(n,m) = \omega(T_{\text{QF}}(n,m))$. La risposta è **NO**. Si osservi infatti che per $m = \Theta(n)$, si ha $T_{\text{QF}}(n,m) = O(n \cdot \log n + n^2) = O(n^2)$, mentre $T_{\text{QU}}(n,m) = O(n^2)$, e quindi $T_{\text{QF}}(n,n) = T_{\text{QU}}(n,n)$. Esistono altri valori di m per cui i due approcci si equivalgono? No!

Riepilogo: regole del **taglio** e del **ciclo**

Scegli un taglio del grafo, e tra tutti gli archi che attraversano il taglio, scegline uno di costo minimo e coloralo di blu (cioè, **aggiungilo alla soluzione**).

Scegli un ciclo nel grafo, e tra tutti gli archi che appartengono al ciclo, scegline uno di costo massimo e coloralo di rosso (cioè, **scartalo per sempre**).

Algoritmo di Prim (1957) (in realtà scoperto nel 1930 da Jarník)

Strategia

- Mantiene un unico **albero blu** T , che all'inizio consiste di un vertice arbitrario
- Applica per $n-1$ volte il seguente **passo goloso**: scegli un **arco di peso minimo incidente su T** (ovvero con un estremo in $V(T)$ e l'altro estremo in $V \setminus V(T)$) e coloralo di **blu** (in altre parole, applica ripetutamente la **regola del taglio**, da cui ne consegue la correttezza!)
- La regola rossa non viene mai invocata, se non implicitamente: tutti gli **archi non blu** che uniscono vertici nell'albero T sono da considerare **rossi**
- **Complessità computazionale di un approccio brutale**: In ognuno degli $n-1$ passi, guardo tutti gli $O(m)$ archi che attraversano il taglio $(V(T), V \setminus V(T))$ corrente, e scelgo quello di peso minimo \Rightarrow costo $O(m \cdot n)$

Un approccio più efficiente (simil-Dijkstra)

- Per ogni $v \notin T$, definiamo **arco azzurro** associato a v un arco (u,v) tale che $u \in T$, ed (u,v) ha peso minimo tra tutti gli archi che connettono v ad un vertice in T
- L'algoritmo mantiene in una **coda di priorità** i nodi non ancora aggiunti alla soluzione ma connessi ad essa, e assegna a ciascuno di essi una **chiave** pari al **peso del rispettivo arco azzurro associato**; l'insieme delle chiavi viene memorizzato anche in un vettore ausiliario $d[1..n]$ ($+\infty$ nel caso in cui il nodo non sia ancora entrato nella coda); intuitivamente, d svolge il ruolo di “vettore stime-di-distanza” che svolgeva in Dijkstra
- Ad ogni passo, viene estratto il **minimo** dalla coda, aggiungendo il nodo associato alla soluzione, e si procede quindi all'eventuale aggiornamento delle chiavi nella coda di priorità e/o inserimento di nuovi nodi appesi, controllando tutti gli archi incidenti il nodo appena aggiunto alla soluzione

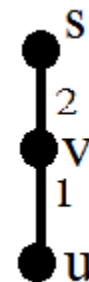
Pseudocodice

```

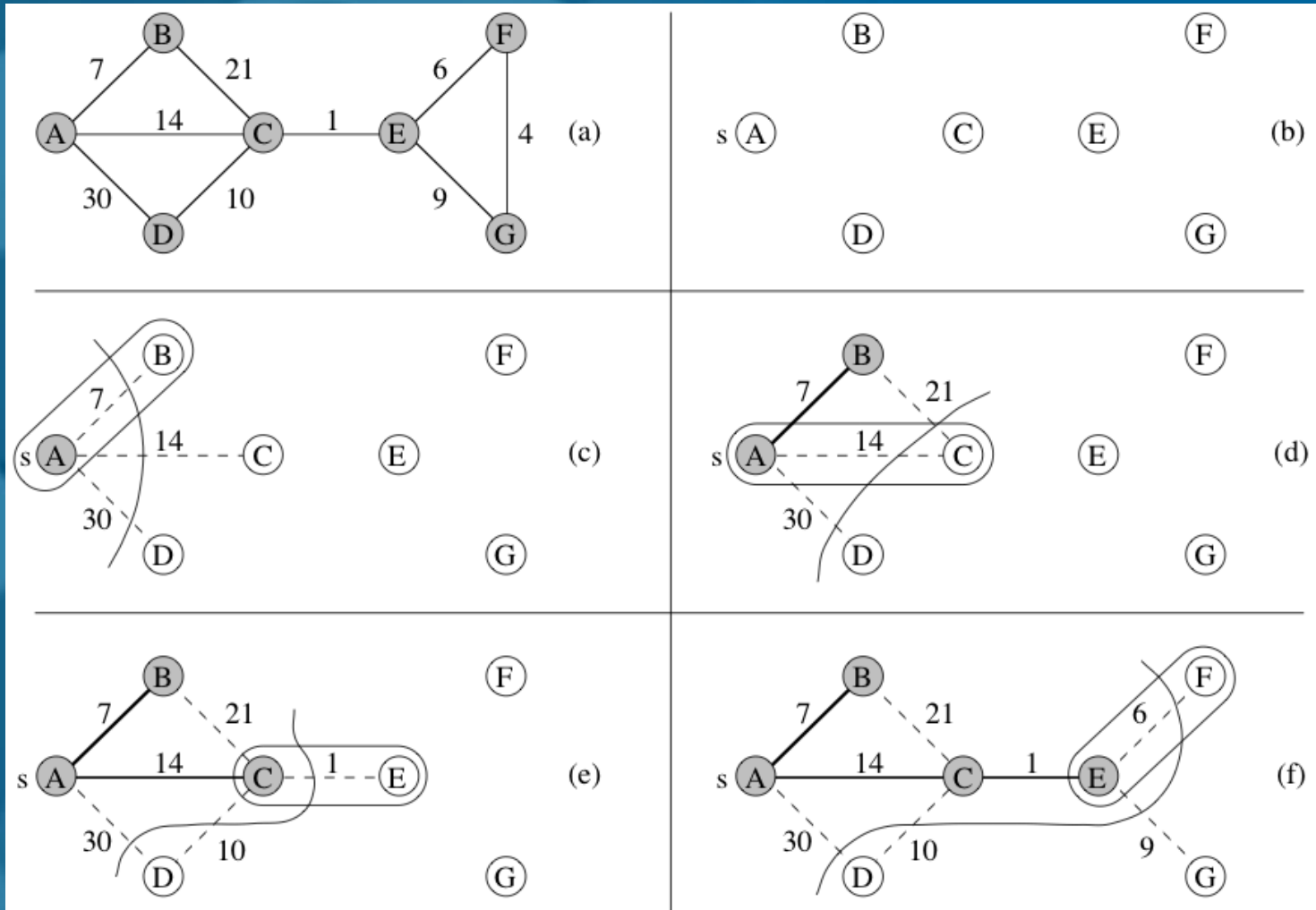
algoritmo Prim (grafo  $G$ )  $\rightarrow$  albero
1.   for each ( vertice  $v$  in  $G$  ) do  $d(v) \leftarrow +\infty$ 
2.    $T \leftarrow$  albero formato da un solo nodo  $s$ 
3.   CodaPriorita  $S$ 
4.    $d(s) \leftarrow 0$ 
5.    $S.insert(s, 0)$ 
6.   while ( not  $S.isEmpty()$  ) do
7.      $u \leftarrow S.deleteMin()$ 
8.     for each ( arco  $(u, v)$  in  $G$  ) do
9.       if ( $d(v) = +\infty$ ) then
10.         $S.insert(v, w(u, v))$ 
11.         $d(v) \leftarrow w(u, v)$ 
12.        rendi  $u$  padre di  $v$  in  $T$ 
13.       else if ( $w(u, v) < d(v)$  and  $v \in S$ ) then
14.         $S.decreaseKey(v, d(v) - w(u, v))$ 
15.         $d(v) \leftarrow w(u, v)$ 
16.        rendi  $u$  nuovo padre di  $v$  in  $T$ 
17.   return  $T$ 

```

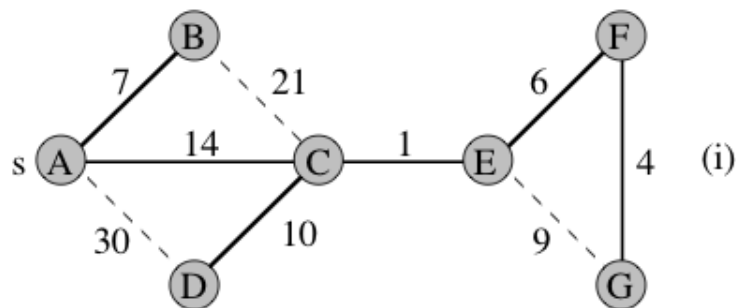
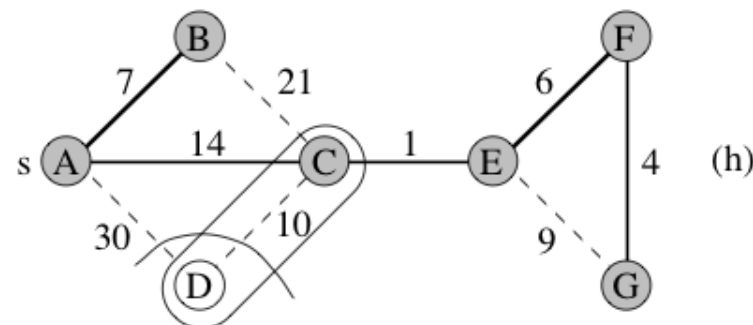
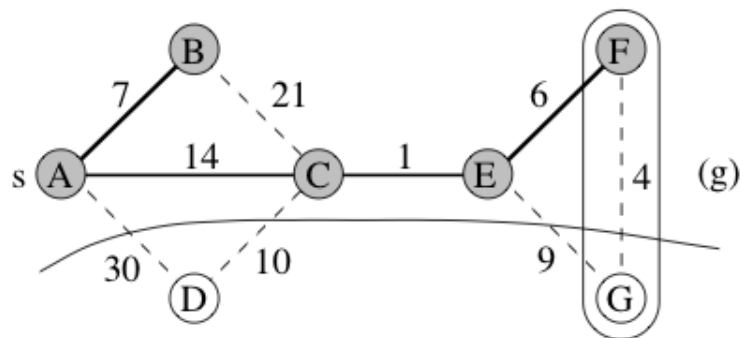
Dobbiamo tenere traccia (in un array) dei nodi aggiunti alla soluzione, per evitare di chiamare delle *decreaseKey* su nodi che sono usciti da S , cosa che per esempio potrebbe accadere in un grafo di questo tipo quando inserisco u nella soluzione:



Esempio (1/2)



Esempio (2/2)



Tempo di esecuzione: implementazioni elementari

Supponendo che il grafo G sia connesso e rappresentato tramite liste di adiacenza, avremo n `insert`, n `deleteMin` e al più m `decreaseKey` nella coda di priorità, al costo di:

	Insert	DelMin	DecKey
Array non ord.	$O(1)$	$O(n)$	$O(1)$
Array ordinato	$O(n)$	$O(1)$	$O(n)$
Lista non ord.	$O(1)$	$O(n)$	$O(1)$
Lista ordinata	$O(n)$	$O(1)$	$O(n)$

- $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$ con array non ordinati
- $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(m \cdot n)$ con array ordinati
- $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$ con liste non ordinate
- $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(m \cdot n)$ con liste ordinate

Tempo di esecuzione utilizzando **heap**

Supponendo che il grafo G sia **connesso** e rappresentato tramite liste di adiacenza, avremo n `insert`, n `deleteMin` e al più m `decreaseKey`



- $n \cdot O(\log n) + n \cdot O(\log n) + O(m) \cdot O(\log n) = O(m \log n)$
utilizzando **heap binari o binomiali** (come Kruskal con le euristiche di bilanciamento)
- $n \cdot O(1) + n \cdot O(\log n) + O(m) \cdot O(1)^* = O(m + n \log n)$
utilizzando **heap di Fibonacci**, ovvero meglio di Kruskal (che costava $O(m \log n)$) se $m = \omega(n)$, mentre i due approcci si equivalgono se $m = \Theta(n)$ (nel qual caso costano $\Theta(n \log n)$)

Algoritmo di Borůvka (1926)

Strategia

- Mantiene una foresta di **alberi blu**, all'inizio coincidente con l'insieme dei nodi del grafo.
- L'algoritmo procede per **fasi successive**; in ogni fase, gli alberi della foresta vengono uniti tra di loro in modo opportuno, e le fasi terminano quando la foresta si riduce ad un solo albero (il MAR)
- In dettaglio, in ogni fase, si eseguono le seguenti operazioni:
 1. Per ogni albero nella foresta, scegli un arco di peso **minimo** uscente da esso, e **coloralo di blu** (applica la **regola del taglio**); tale operazione unisce 2 alberi della foresta;
 2. Dopo aver esaminato tutti gli alberi della foresta, elimina da ogni futura computazione gli archi interni ai nuovi alberi generati durante il passo 1 (**regola del ciclo**).
- **Nota:** Per non rischiare di introdurre cicli durante il passo 1, bisogna assumere che i costi degli archi siano tutti distinti (se così non fosse, basterà perturbare minimamente gli archi aventi stesso peso)

Implementazione

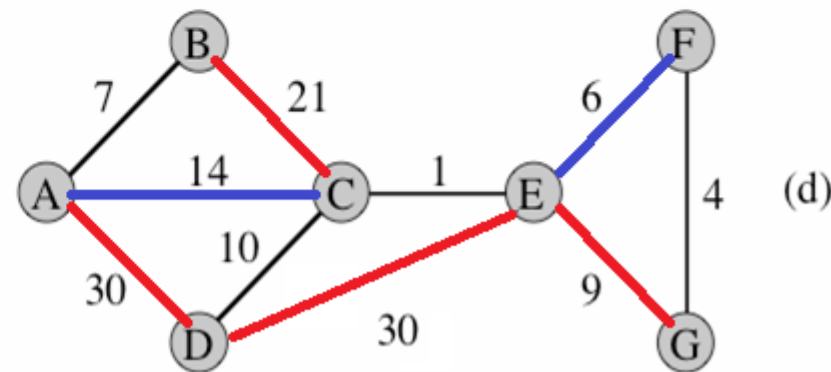
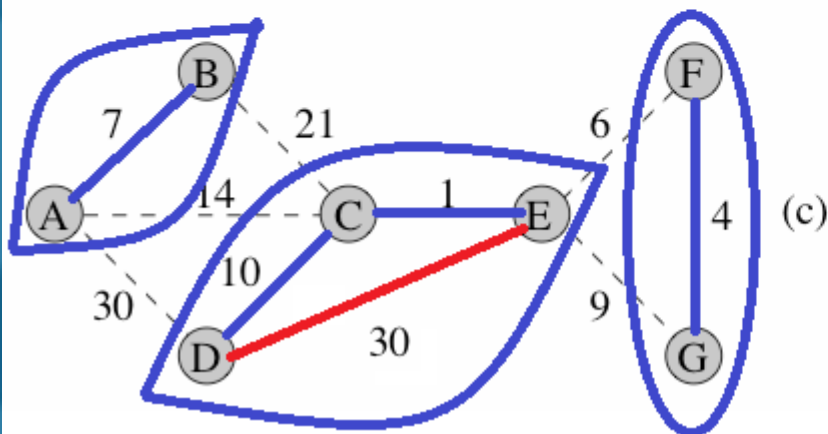
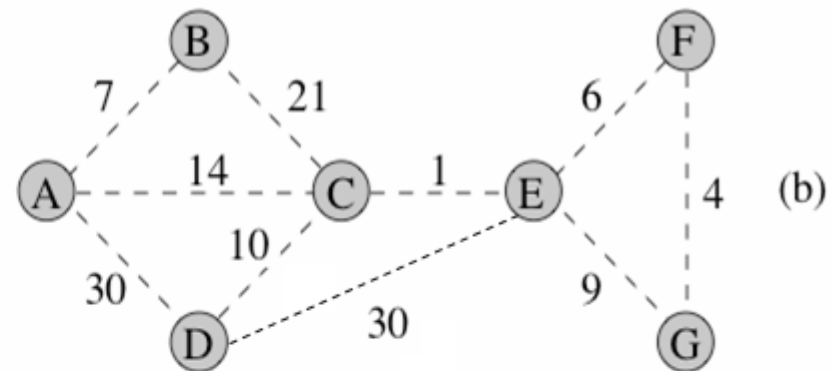
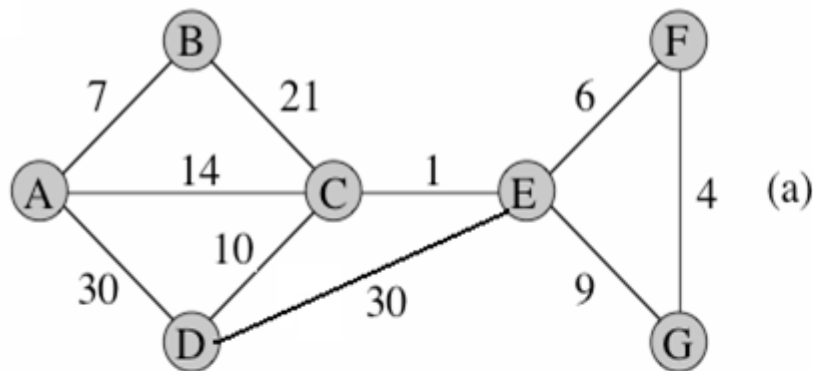
In ogni fase, facciamo uso di una **implementazione elementare UNION-FIND** basata su **array**. Supponiamo che G sia dato in forma di lista di adiacenza; inizialmente ogni nodo di G è un albero. Definiamo quindi un array A di n elementi, in cui l' i -esima cella è associata all' i -esimo nodo di G , e conterrà l'elemento rappresentativo dell'insieme a cui appartiene (inizialmente quindi $A[i]=i$)

1. Nella prima fase, consideriamo uno dopo l'altro i nodi di G . Quando esaminiamo un particolare nodo u , scorriamo l'intera lista di archi adiacenti, e **coloriamo di blu** quello di costo minimo. Costo: $\Theta(m+n)=\Theta(m)$
2. Dopo aver esaminato tutti i nodi, avremo selezionato un certo numero di archi blu. Eseguiamo quindi una visita di G ristretta ai soli archi blu, ottenendo in $\Theta(m+n)=\Theta(m)$ le componenti blu (che sono alberi) connesse.

Implementazione (2)

3. Quindi, visitiamo uno dopo l'altro gli alberi blu, scegliamo per ciascuno di essi in modo arbitrario un elemento rappresentativo, che viene memorizzato in tutte le celle di A associate ai nodi del corrispondente albero blu. Costo: $\Theta(n)$ (si noti che questa operazione corrisponde alla **UNION** eseguita sull'array A)
4. La fase prosegue eliminando gli archi interni a tali alberi. Per fare ciò, si scorrono di nuovo le liste di adiacenza, e si rimuove da esse ogni arco (u,v) tale che $\text{FIND}(u)=\text{FIND}(v)$ (ovvero, se $A[u]=A[v]$). Costo: $\Theta(m+n)=\Theta(m)$
5. Infine, per ogni albero blu, si collegano tra di loro le liste di adiacenza così aggiornate di tutti i suoi nodi. In questo modo viene generato un nuovo **multigrafo** come input della fase successiva, in cui gli alberi blu sono stati contratti in **macrovertici**. Costo: $\Theta(m+n)=\Theta(m)$

Esempio



Analisi

- Siano k i macrovertici della fase corrente; allora, i vari passaggi sopra descritti costeranno $O(m+k)=O(m+n)$, in quanto tutte le operazioni possono essere eseguite in tempo lineare nella dimensione del grafo corrente
 - Il numero di fasi è $O(\log n)$: infatti ogni macrovertice creato alla fine della fase $k \geq 1$ contiene almeno 2^k nodi (si dimostra banalmente per induzione)
- ⇒ L'algoritmo di Borůvka, utilizzando solo strutture dati elementari (liste ed array), costa $O(m \log n)$, ovvero come Kruskal implementato con le euristiche di bilanciamento, o come Prim implementato con heap binari/binomiali!

È finita...ora siete **quasi** degli
algoritmisti, vi resta solo l'esame
da fare!

Ci vediamo il 10/1 per il secondo
parziale, e/o il 17/1, 31/1 o 14/2
per lo scritto

Buon Natale!

