1

Analisi ammortizzata

In medio stat virtus.
(Locuzione latina)

In molte situazioni pratiche, ci troviamo a eseguire un algoritmo ripetutamente nel tempo. Ad esempio, i gestori di motori di ricerca per Internet vedono arrivare miliardi di interrogazioni al giorno, ognuna delle quali corrisponde all'esecuzione di un certo algoritmo di ricerca. In contesti di questo tipo, è spesso utile analizzare le prestazioni di una operazione algoritmica *in media* su una sequenza di esecuzioni, piuttosto che su una singola esecuzione nel caso peggiore. Infatti, alcune operazioni potrebbero richiedere episodicamente molto tempo per una configurazione dei dati svantaggiosa, per poi essere più veloci in seguito. Un'analisi puntuale nel caso peggiore darebbe risultati molto elevati, mentre un'analisi su una sequenza di esecuzioni rivelerebbe invece un buon comportamento in media. Questo tipo di analisi viene chiamata *analisi ammortizzata* e spesso caratterizza in modo più preciso le prestazioni di una operazione algoritmica che viene effettuata ripetutamente nel tempo. Per illustrare questo concetto, usiamo un semplice esempio.

Esempio 1.1 (Incremento di un contatore binario) Si consideri la funzione C di Figura 1.1 che, dato un array `v` di `n` interi che rappresentano i bit di un contatore binario, ne incrementa di 1 il valore. La funzione assume che `v[0]` contenga il bit meno significativo del contatore. E' facile convincersi che il tempo richiesto nel caso peggiore dalla funzione `incrementa` è $O(n)$. Supponiamo ora di partire da un array con tutti gli elementi a zero e di effettuare una sequenza di k operazioni. Quanto sarà il tempo richiesto dall'esecuzione dell'intera sequenza? Moltiplicando per k la stima sul tempo per operazione nel caso peggiore precedentemente calcolata, otteniamo $O(n \cdot k)$ per l'intera sequenza. Il tempo medio stimato per operazione `incrementa` sarà pertanto $O(n)$. Certamente, l'analisi è corretta, nel senso che nessuna sequenza lunga k potrà richiedere tempo maggiore di $O(n \cdot k)$. Una domanda che sorge spontanea è però se questa stima non sia troppo pessimistica. Dopo tutto, nell'analisi non si tiene conto del fatto che il `break` potrebbe interrompere il ciclo prima di aver completato le n iterazioni. Per capire

```
  void incrementa(int v[], int n){
1     int i;
2     for (i=0; i<n; i++) {
3         v[i]=!v[i];
4         if (v[i]) break;
5     }
  }
```

Figura 1.1 Operazione di incremento di un contatore binario rappresentato come array di interi.

n	k	T tot. (μsec)	T medio ($nsec$)
20	1048576	5946	5.7
21	2097152	13092	6.2
22	4194304	24081	5.7
23	8388608	46832	5.6
24	16777216	92921	5.5
25	33554432	189888	5.7
26	67108864	379983	5.7
27	134217728	758461	5.7
28	268435456	1514401	5.6
29	536870912	3020634	5.6
30	1073741824	6040832	5.6

Tabella 1.1 Analisi sperimentale del tempo totale e del tempo medio per operazione su una sequenza di $k = 2^n$ incrementi di un contatore binario a n bit.

quanto una stima di questo tipo possa essere accurata, effettuiamo un'analisi sperimentale dei tempi medi per operazione `incrementa`.

Obiettivi

L'obiettivo dell'esperimento è quello di misurare il tempo di esecuzione medio della funzione `incrementa` di Figura 1.1 su array di dimensioni diverse e su sequenze di invocazioni della funzione che generano tutti i possibili valori rappresentabili da ciascun array.

Impianto sperimentale

Piattaforma. Gli esperimenti sono stati condotti su un MacBook Pro Intel Core 2 Duo a 2.8 GHz con 4 GB RAM, Mac OS X 10.6.2 e `gcc` 4.2.1.

Scenari di test. Come scenario di test, sono stati considerati array di dimensione n compresa tra 16 e 28, con tutte le celle inizialmente a zero. Per ciascun array, è stata effettuata una sequenza di $k = 2^n$ operazioni, tale quindi da generare tutti i possibili numeri binari con n bit.

Misurazione dei tempi. I tempi sono stati misurati mediante la funzione `clock()`. Il tempo medio è stato calcolato misurando il tempo totale richiesto da ciascuna sequenza di invocazioni della funzione `incrementa`, e dividendolo per la lunghezza della sequenza stessa. Per ridurre gli errori di misurazione, gli stessi esperimenti sono stati ripetuti tre volte, riportando la media dei tempi misurati in ciascun esperimento.

Risultati sperimentali

La Tabella 1.1 mostra i tempi totali richiesti da ciascuna sequenza di k operazioni di incremento e i tempi medi per operazione. Si noti che i tempi medi si mantengono praticamente costanti indipendentemente dal numero di operazioni e dal numero n di bit del contatore. Appare evidente che la stima teorica $O(n)$ per operazione nel caso peggiore descrive in modo troppo pessimistico i tempi medi effettivi. □

1.1 Costo di una sequenza di operazioni

Per stimare il tempo medio di un'operazione, faremo vedere come ottenere una buona stima del tempo totale di una sequenza di operazioni nel caso peggiore. In questo capitolo useremo i termini *tempo* e

costo (temporale) come sinonimi, intendendo il numero di istruzioni elementari mandate in esecuzione. Inoltre, assumeremo sempre di avere a cha fare con *sequenze di operazioni di caso peggiore*, cioè tali da forzare il massimo lavoro possibile su una struttura dati.

Definizione 1.1 (**Costo totale di una sequenza**) *Sia* $\langle op_1, \ldots, op_k \rangle$ *una sequenza di caso peggiore di* k *operazioni, e sia* $T^i_{ef}(n)$ *il costo* effettivo *richiesto dall'operazione* i*-esima in funzione della dimensione* n *dell'input. Denotiamo il* costo totale effettivo *nel caso peggiore richiesto dalla sequenza di operazioni con:*

$$T(n,k) = \sum_{i=1}^{k} T^i_{ef}(n).$$

Si noti che nella definizione di costo totale assumiamo che l'input abbia la stessa dimensione ad ogni operazione. Ove questo non sia vero, si può prendere come n la massima dimensione dei dati durante la sequenza, ottenendo una stima per eccesso del costo totale.

Definizione 1.2 (**Costo ammortizzato**) *Definiamo il* costo ammortizzato $T_{am}(n)$ *di una operazione su una sequenza di* k *operazioni che richiedono tempo totale* $T(n,k)$ *nel caso peggiore come:*

$$T_{am}(n) = \frac{T(n,k)}{k}$$

Nel paragrafo seguente descriveremo un metodo generale che consente di stimare in modo accurato il costo totale di una sequenza di operazioni nel caso peggiore. Si noti che il costo ammortizzato di un'operazione descrive il tempo in media su una sequenza, ma non ha nulla a che fare con l'*analisi di caso medio* di un algoritmo dove si assume di avere una distribuzione di probabilità sui dati di input e il tempo richiesto dall'algoritmo è una variabile aleatoria di cui si vuole calcolare il valore atteso.

1.2 Il metodo dei crediti

Il metodo dei crediti consiste nell'assegnare a ciascuna operazione che compare in una sequenza di operazioni un *costo artificiale* tale che la somma dei costi artificiali sia una stima mai per difetto del tempo totale effettivo richiesto dalla sequenza. Sotto opportune ipotesi, il costo artificiale definito con il metodo dei crediti fornisce un limite superiore al costo ammortizzato delle operazioni di una sequenza. Vedremo quindi che il metodo dei crediti fornisce un modo di calcolare una stima del costo ammortizzato delle operazioni. Il metodo dei crediti consiste nell'associare agli elementi di una struttura dati dei *crediti*, che possono essere pensati come monete immaginarie[1] in grado di pagare il costo computazionale di alcune operazioni sulla struttura dati stessa. Assumiamo quindi che:

$$\textit{1 credito (o moneta)} = O(1) \textit{ passi di calcolo}.$$

L'idea generale è che alcune operazioni depositano crediti sulla struttura dati. Questi crediti "pagheranno" il costo effettivo di operazioni future che preleveranno crediti dalla struttura dati. Nella sua forma più generale, il *costo artificiale* associato all'operazione i-esima di una sequenza di operazioni $\langle op_1, op_2, \ldots, op_k \rangle$ mediante il metodo dei crediti è definito come:

$$T^i_{am}(n) = T^i_{ef}(n) + deposito(n) - prelievo(n)$$

dove:

[1] Si tenga presente che le monete sono solo una tecnica di analisi che consente di ottenere stime più accurate rispetto al modello di costo tradizionale e non compaiono in alcun modo nella definizione degli algoritmi.

- n è un limite superiore alla dimensione della struttura dati su cui operano le operazioni della sequenza;
- $T^i_{ef}(n)$ è il costo effettivo dell'operazione i-esima;
- $deposito(n)$ sono i crediti depositati dall'operazione i-esima;
- $prelievo(n)$ sono i crediti prelevati dall'operazione i-esima.

Diremo che uno schema di crediti è *consistente* se il bilancio di crediti depositati e prelevati non è mai in "rosso" e quindi ci sono sempre sufficienti crediti per ogni prelievo.

Il seguente teorema mostra che, se lo schema di crediti è consistente, allora la somma dei costi artificiali fornisce un limite superiore al costo totale di una sequenza di operazioni.

Teorema 1.1 *Sia $\langle op_1, op_2, \ldots, op_k \rangle$ una sequenza di operazioni su una struttura dati di dimensione n e siano $T^1_{ef}, \cdots, T^k_{ef}$ i tempi effettivi per operazione. Per ogni schema di crediti consistente si ha:*

$$\sum_{i=1}^{k} T^i_{am}(n) \geq T(n,k).$$

Dimostrazione. In base alla definizione di costo mediante il metodo dei crediti, si ha:

$$\sum_{i=1}^{k} T^i_{am}(n) = \sum_{i=1}^{k} \left(T^i_{ef}(n) + deposito^i(n) - prelievo^i(n) \right) \geq T(n,k) =$$

$$\sum_{i=1}^{k} T^i_{ef}(n) + \sum_{i=1}^{k} deposito^i(n) - \sum_{i=1}^{k} prelievo^i(n).$$

Assumendo che per ogni prelievo ci siano sufficienti monete, allora:

$$\sum_{i=1}^{k} deposito^i(n) - \sum_{i=1}^{k} prelievo^i(n) \geq 0$$

e quindi:

$$\sum_{i=1}^{k} T^i_{am}(n) \geq \sum_{i=1}^{k} T^i_{ef}(n) = T(n,k).$$

□

Il seguente corollario del Teorema 1.1 stabilisce che il costo artificiale definito con il metodo dei crediti permette di ottenere un limite superiore al costo ammortizzato delle operazioni di una sequenza.

Corollario 1.1 *Se $T^i_{am}(n) = O(f(n)) \ \ \forall i = 1, \ldots, k$, allora:*

$$T_{am}(n) = \frac{T(n,k)}{k} = O(f(n)).$$

Dimostrazione. In base al Teorema 1.1, si ha:

$$T(n,k) \leq \sum_{i=1}^{k} T^i_{am}(n) = \sum_{i=1}^{k} O(f(n)) = k \cdot O(f(n)).$$

La dimostrazione segue immediatamente dividendo il primo e l'ultimo membro per k. □

Esempio 1.2 (Incremento di un contatore binario) Vediamo ora di applicare il metodo dei crediti per analizzare il costo ammortizzato dell'algoritmo su cui si basa la funzione `incrementa` di Figura 1.1. Usiamo il seguente semplice schema di crediti:

- ogni istruzione che mette un bit da 0 a 1 deposita un credito sulla cella messa a 1;
- ogni istruzione che mette un bit da 1 a 0 preleva un credito dalla cella messa a 0, con cui viene pagato il costo dell'istruzione stessa;

E' immediato convincersi che questo schema di crediti è consistente e mantiene la seguente invariante:

su ogni cella dell'array `v` *che contiene il valore 1 vi è depositato un credito*.

Denotiamo con $T^i_{ef}(n) = q_i^{1\to 0} + q_i^{0\to 1}$ il costo effettivo di un'operazione di incremento del contatore binario, dove $q_i^{1\to 0}$ è il numero di bit che passano da 1 a 0 e $q_i^{0\to 1} = 1$ è il bit che passa da 0 a 1. Definiamo il costo artificiale dell'i-esima operazione `incrementa` secondo il metodo dei crediti come:

$$T^i_{am}(n) = \underbrace{\cancel{q_i^{1\to 0}} + q_i^{0\to 1}}_{T^i_{ef}(n)} + \underbrace{q_i^{0\to 1}}_{deposito^i(n)} - \underbrace{\cancel{q_i^{1\to 0}}}_{prelievo^i(n)} = 2 = O(1).$$

In base al Corollario 1.1, il costo ammortizzato dell'operazione `incrementa` è pertanto $O(1)$, come avevamo già osservato sperimentalmente. □

Esempio 1.3 (Cammini minimi dinamici) Mostriamo ora un secondo esempio più articolato di analisi ammortizzata per un classico problema su grafi dinamici, che consiste nel mantenere le distanze di ciascun nodo da una sorgente prefissata in un grafo orientato soggetto a operazioni di rimozione di archi.

Ricordiamo che un *grafo orientato* (o *diretto*) è una coppia $G = (V, E)$ avente come insieme di nodi V e come insieme di archi $E \subseteq V \times V$. Denoteremo con $n = |V|$ il numero di nodi e con $m = |E|$ il numero di archi del grafo. Inoltre, diremo che un *cammino* π_{xy} fra due nodi x e y nel grafo è una sequenza di archi $\pi_{xy} = \langle (x, v_1), (v_1, v_2), \ldots, (v_{h-1}, y)\rangle$ che permette di raggiungere y a partire da x. La *lunghezza* $\ell(\pi_{xy})$ di un cammino è pari al numero di archi che attraversa. Infine, la *distanza* $dist_{xy}$ fra due nodi x, y connessi nel grafo è pari alla lunghezza del cammino più corto fra di essi, cioè:

$$dist_{xy} = \min_{\pi_{xy}}\{\ell(\pi_{xy})\}.$$

Se y non è raggiungibile da x allora possiamo assumere che la distanza fra di essi è infinita, cioè $dist_{xy} = \infty$.

In Figura 1.2 formalizziamo il problema dei cammini minimi dinamici definendo un tipo di dato astratto `DistGrafoDinamico` che consente di mantenere le distanze a partire da una sorgente prefissata in un grafo soggetto a operazioni di inserimento e rimozione di archi.

Implementazione 1. Una prima implementazione inefficiente del tipo di dato astratto può essere ottenuta semplicemente aggiornando il grafo e ricalcolando da zero le distanze di ogni nodo dalla sorgente dopo ciascuna operazione `init`, `insert` e `delete`. Così facendo, l'operazione `dist` richiede tempo costante. Le distanze dei nodi in un grafo da una sorgente prefissata possono essere calcolate applicando l'algoritmo di visita in ampiezza, che richiede tempo $O(m + n)$ nel caso peggiore. Sotto la ragionevole assunzione che $n = O(m)$, questa prima realizzazione fornisce le seguenti prestazioni per operazione nel caso peggiore:

- `init`, `insert`, `delete`: $O(m)$
- `dist`: $O(1)$

Pertanto, una qualsiasi sequenza di k operazioni richiede tempo $O(k \cdot m)$ nel caso peggiore, e quindi un tempo ammortizzato per operazione pari a $O(m)$.

tipo `DistGrafoDinamico`:
dati:
grafo $G = (V, E)$, sorgente $s \in V$, distanza $d(v) = dist_{sv}$ per ogni nodo $v \in V$.

operazioni:

`init`(G, s) — *costruttore*
inizializza la struttura dati con grafo iniziale G e sorgente s.

`insert`(u, v) — *operazione modifica*
aggiunge l'arco (u, v) a G.

`delete`(u, v) — *operazione modifica*
rimuove l'arco (u, v) da G.

`dist`$(v) \to \mathbb{N}$ — *oper. interrogazione*
restituisce $d(v)$, la distanza corrente di v da s nel grafo G.

Figura 1.2 Il tipo di dato `DistGrafoDinamico`.

Implementazione 2. Vediamo ora una implementazione più raffinata che evita di ricalcolare da zero le distanze dopo ogni modifica del grafo. In particolare, ci concentreremo sulle sole operazioni `init`, `delete` e `dist` e mostreremo come ottenere un tempo ammortizzato per operazione pari a $O(n)$ su qualunque sequenza di $k = \Omega(m)$ operazioni `delete` e `query` che inizia con una `init`. Osserviamo che questa seconda realizzazione ottiene un miglioramento prestazionale pari a un fattore $O(m/n)$ per operazione rispetto alla prima realizzazione[2]. Per grafi densi con $m \approx n^2$, si ha un guadagno di un fattore circa n.

▷ **Dati –** Oltre al grafo G, la sorgente s e la distanza $d(v)$ di ogni nodo v dalla sorgente, manteniamo le seguenti informazioni per ogni nodo v (si veda la Figura 1.3) usando in totale spazio $S(n, m) = \Theta(m + n)$:

- $In(v) = \langle u_1, u_2, \ldots \rangle$: lista dei nodi da cui escono gli archi entranti in v;
- $Out(v) = \{z \mid (v, z) \in E\}$: insieme dei nodi in cui entrano gli archi uscenti da v;
- $curr(v)$: puntatore nella lista $In(v)$ che specifica la "posizione corrente" nella lista (vedi oltre);

In ogni istante, la nostra realizzazione manterrà le seguenti invarianti per ogni nodo $v \in V$:

- per ogni predecessore p di $curr(v)$ in $In(v)$ si ha $d(u_p) > d(u_{curr(v)})$, dove u_p denota il nodo di $In(v)$ puntato da p. Questa proprietà stabilisce che tutti i nodi che precedono quello corrente nella lista $In(v)$ hanno una distanza dalla sorgente strettamente maggiore.
- $d(v) = \begin{cases} 0 & \text{se } v = s \\ d(u_{curr(v)}) + 1 & \text{se } v \neq s \text{ è raggiungibile da } s \\ \infty & \text{se } v \text{ non è raggiungibile da } s \end{cases}$
  Questa proprietà stabilisce che, se $v \neq s$ è raggiungibile da s, $u_{curr(v)}$ è il predecessore di v sul cammino mimimo che porta da s a v.

[2] Si noti che ad oggi non è nota nessuna realizzazione migliore dell'implementazione 1 se si vuole supportare contemporaneamente sia la `insert` che la `delete`.

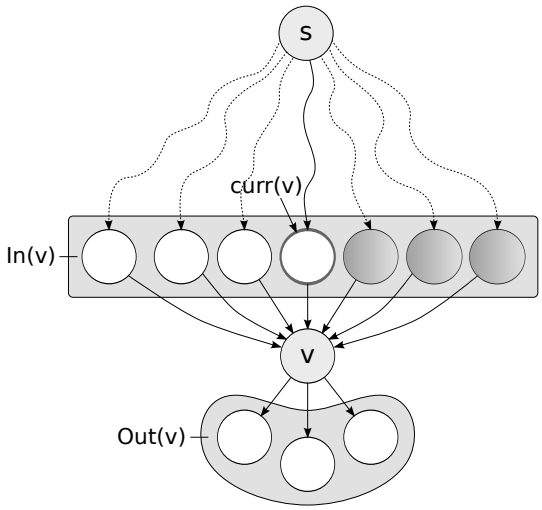

Figura 1.3 Dati mantenuti dall'implementazione 2 del tipo di dato `DistGrafoDinamico`: lista $In(v)$ degli entranti in v, puntatore $curr(v)$ alla posizione corrente in $In(v)$, e insieme $Out(v)$ degli uscenti da v. I predecessori del nodo corrente nella lista $In(v)$ hanno una distanza dalla sorgente strettamente maggiore di quella del nodo corrente stesso, e pertanto non sono utili se si cerca un cammino alternativo che preservi la distanza di v a fronte di un aumento di distanza del nodo puntato da $curr(v)$.

▷ **Operazioni –** La realizzazione del tipo di dato `DistGrafoDinamico` è mostrata in Figura 1.4. Discutiamo ora il funzionamento di ciascuna operazione:

- `init`(G, s): l'operazione funge da costruttore per la struttura dati calcolando le distanze di tutti i nodi dalla sorgente s nel grafo iniziale G.
- `dist`(v): l'operazione restituisce la distanza corrente del nodo v dalla sorgente (riga 4).
- `delete`(u, v): l'operazione rimuove l'arco (u, v) e aggiorna le distanze esplorando il grafo a partire dal nodo v. L'esplorazione viene effettuata inserendo in una coda Q i nodi la cui distanza potrebbe aumentare a fronte della rimozione di (u, v). Inizialmente, la coda contiene il solo nodo v (riga 6). I nodi da esplorare vengono via via estratti da Q (riga 8) finché questa non si svuota (riga 7). Per verificare se un nodo mantiene la stessa distanza dalla sorgente, aggiornandola eventualmente in caso di variazione e aggiungendo a Q i suoi vicini, viene richiamata una procedura ausiliaria `update` (riga 9).
- `update`(x): la procedura verifica inizialmente se il nodo x non ha più archi entranti (questo può avvenire solo per $x = v$), nel qual caso la sua distanza è posta a infinito (riga 10). Se invece $In(x)$ non è vuota, avanza il puntatore $curr(x)$ alla ricerca di un cammino alternativo che permetta al nodo x di mantenere invariata la propria distanza (righe 11–12). Se questo avviene ($curr(x)$ non ha raggiunto la fine della lista $In(x)$), la procedura `update` termina senza modificare $d(x)$ e $curr(x)$ punta al nuovo predecessore di x sul cammino minimo che porta da s a x (riga 13). In caso contrario ($curr(x)$ è diventato **nil** avendo finito di scorrere la lista $In(x)$), è possibile concludere che $d(x)$ aumenta strettamente. Infatti, l'implementazione mantiene l'invariante che tutti i predecessori di $curr(x)$ in $In(x)$ hanno una distanza da s maggiore di $d(x) - 1$, pertanto nessuno di essi può essere utile per mantenere invariata $d(x)$. Per aggiornare $d(x)$, l'intera lista $In(x)$ viene esplorata per intero (riga 14) e $curr(x)$ viene indietreggiato a puntare al primo nodo di $In(x)$ per cui passa il nuovo cammino minimo che porta a x (riga 15). Infine, i nodi di $Out(x)$ vengono inseriti in Q, poiché la variazione di distanza di x potrebbe ripercuotersi anche sui suoi vicini (riga 16).

classe `Implementazione2` **implementa** `DistGrafoDinamico`:
dati: $S(n,m) = \Theta(n+m)$
grafo $G = (V, E)$, sorgente $s \in V$, per ogni nodo $v \in V$: distanza $d(v)$, lista $In(v)$, insieme $Out(v)$, posizione corrente $curr(v)$

operazioni:
`init`(G, s) $T_{am}(n,m) = O(mn)$
1. ▷ imposta G come grafo iniziale ed s come sorgente
2. ▷ calcola distanze $d(v)$ per ogni nodo v applicando algoritmo visita in ampiezza
3. ▷ inizializza $In(v)$, $Out(v)$ e $curr(v)$ per ogni nodo v

`dist`$(v) \to \mathbb{N}$ $T_{am}(n,m) = O(1)$
4. **return** $d(v)$

`delete`(u, v) $T_{am}(n,m) = O(n)$
5. rimuovi (u, v) da E, u da $In(v)$ e v da $Out(u)$, e avanza $curr(v)$ di una posizione in avanti se $u_{curr(v)} = u$
6. $Q \leftarrow \{v\}$ *Q è una coda*
7. **while** $(Q \neq \emptyset)$ **do**
8. estrai x da Q
9. **if** $(x \neq s)$ **then** *update*(x)

update(x) $T_{am}(n,m) = 0$
10. **if** $(In(x) = \emptyset)$ **then** $d(x) \leftarrow \infty$; **return**
11. **while** ($curr(x) \neq$ **nil and** $d(u_{curr(x)}) + 1 > d(x)$) **do**
12. avanza $curr(x)$ di una posizione in avanti
13. **if** $(curr(x) \neq$ **nil**$)$ **then return** $d(x)$ *non aumenta*
14. $d(x) \leftarrow \min_{z \in In(x)}\{d(z) + 1\}$ $d(x)$ *aumenta*
15. sia $curr(x)$ il puntatore al primo nodo di $In(x)$ tale che $d(x) = d(u_{curr(x)}) + 1$
16. $Q \leftarrow Q \cup Out(x)$

Figura 1.4 Implementazione 2 del tipo di dato `DistGrafoDinamico`. Dopo aver pagato inizialmente $O(mn)$ per la `init`, il costo ammortizzato delle successive operazioni `delete` è $O(n)$, mentre quelle delle operazioni `dist` è costante su qualsiasi sequenza di operazioni.

▷ **Analisi 1 –** Analizziamo dapprima in modo diretto il costo totale di una sequenza di operazioni `dist` e `delete` dopo una `init` iniziale. Vedremo poi come ottenere lo stesso risultato usando il metodo dei crediti. Effettuiamo due tipi di analisi: per la `init`, per la `dist` e per le righe 5–8 della `delete`, stimiamo il numero di istruzioni mandate in esecuzione nel caso peggiore da ciascuna operazione. Per per la `update` (e quindi la riga 9 della `delete`), studiamo invece il tempo totale richiesto nel caso peggiore su un'intera sequenza di operazioni per ciascun nodo x, e sommiamo poi su tutti i nodi.

- `init`: (analisi per singola operazione)
  - righe 1–3: richiedono tempo $O(m + n)$ per operazione;
- `dist`: (analisi per singola operazione)
  - riga 4: richiede tempo costante per operazione;
- `delete`: (analisi per singola operazione)
  - righe 5–6: richiedono tempo costante per operazione;

  - righe 7–8: richiedono tempo $O(n)$ per operazione, essendo ogni nodo estratto da Q al più una volta in ciascuna `delete`.

- `update`: (analisi sull'intera sequenza) Un'osservazione cruciale è che le distanze nel grafo non possono mai diminuire a fronte della rimozione di un arco. Inoltre, ogni nodo potrà vedere aumentare la propria distanza da s al più $n-1$ volte, essendo le distanze degli interi non negativi ed essendo $n-1$ la lunghezza massima di un qualunque cammino minimo. Possiamo pertanto concludere che:
  - righe 10–13: per ogni x, il loro costo è dominato dalla riga 12, che viene eseguita al più $O(|In(x)|)$ volte per ciascuno degli $O(n)$ possibili valori che $d(x)$ può assumere su un'intera sequenza di cancellazioni di archi. Infatti, la lista $In(x)$ viene scandita dal ciclo delle righe 11–12 solo una volta per ciascuna possibile distanza;
  - righe 14–16: per ogni x, vengono eseguite solo quando $d(x)$ aumenta, e quindi al più $O(n)$ volte in totale su qualunque sequenza di cancellazioni di archi, e scorrono tutti i nodi adiacenti di x in tempo $O(|In(x)| + |Out(x)|)$.

  La `update` (righe 10–16) costa pertanto $O(n \cdot (|In(x)| + |Out(x)|))$ per ogni nodo x in totale su qualsiasi sequenza di `delete`. Sommando quindi su tutti i nodi, si ottiene un costo totale per la `update` pari a:

$$\sum_{x \in V} O(n \cdot (|In(x)| + |Out(x)|)) = O(mn).$$

Su qualsiasi sequenza di $k_{\texttt{delete}} \leq m$ operazioni `delete`[3], $k_{\texttt{dist}}$ operazioni `dist` e una `init` iniziale, otteniamo pertanto un costo totale pari a:

$$T(n,k) = \overbrace{O(m+n)}^{\substack{\texttt{init} \\ \text{(righe 1–3)}}} + \overbrace{O(k_{\texttt{dist}})}^{\substack{\texttt{dist} \\ \text{(riga 4)}}} + \overbrace{O(n \cdot k_{\texttt{delete}})}^{\substack{\texttt{delete} \\ \text{(righe 5–8)}}} + \overbrace{\sum_{x \in V} O(n \cdot (|In(x)| + |Out(x)|))}^{\substack{\texttt{update} \\ \text{(righe 10–16)}}} =$$

$$= O(mn + k_{\texttt{dist}}),$$

con $k = 1 + k_{\texttt{dist}} + k_{\texttt{delete}}$. Per qualunque sequenza mista di $k = \Omega(m)$ operazioni `dist` e `delete` che seguono una `init` iniziale, il costo ammortizzato per operazione è pertanto $T(n,k)/k = O(n)$. Si noti che, diversamente dal caso del contatore binario dell'Esempio 1.2, in cui si otteneva un costo $O(1)$ ammortizzato per sequenze di qualunque lunghezza, in questo caso il costo $O(n)$ ammortizzato si ha per sequenze "sufficientemente" lunghe.

▷ **Analisi 2 –** Forniamo ora un'analisi alternativa del costo ammortizzato delle operazioni usando il metodo dei crediti. Usiamo uno schema di monete che mantiene le seguenti invarianti per ciascun nodo in $v \in V$ raggiungibile da s:

- sul nodo v sono depositate $c(v) \geq (n - d(v)) \cdot (2|In(v)| + |Out(v)|)$ monete;
- su ciascun elemento della lista che segue $u_{curr(v)}$ in $In(v)$ è depositata almeno una moneta (nodi ombreggiati in Figura 1.3).

Discutiamo ora i tempi ammortizzati per ciascuna operazione:

[3] Non possono essere di più, se abbiamo inizialmente m archi.

- `init`: osserviamo innanzitutto che il costo effettivo della `init` è $T_{ef}(n,m) = O(m+n)$, essendo dominato dal costo della visita in ampiezza per calcolare le distanze nel grafo iniziale. Per garantire la prima invariante è sufficiente depositare $(n-0)\cdot(2\cdot|In(v)|+|Out(v)|)$ monete su ogni nodo v del grafo. Per garantire invece la seconda invariante, è sufficiente depositare per ogni nodo v una moneta su ciascun elemento di $In(v)$. E' quindi sufficiente assumere:

$$deposito(n,m) = \sum_{v\in V} n\cdot(2\cdot|In(v)|+|Out(v)|) + \sum_{v\in V}|In(v)| = O(mn).$$

  Il costo artificiale di una `init` usando il metodo dei crediti è pertanto:

$$T_{am}^{\texttt{init}}(n,m) = \underbrace{O(m+n)}_{T_{ef}(n,m)} + \underbrace{O(mn)}_{deposito(n,m)} - \underbrace{0}_{prelievo(n,m)} = O(mn).$$

- `update`: distinguiamo due casi.
  - $d(x)$ non aumenta. In questo caso, il costo effettivo $O(\Delta)$ della scansione della lista $In(x)$ alle righe 10–13 può essere pagato in termini delle monete prelevate dagli elementi di $In(v)$, che per la seconda invariante sono sempre sufficienti:

$$T_{am}^{\texttt{update}_1}(n,m) = \underbrace{O(\Delta)}_{T_{ef}(n,m)} + \underbrace{0}_{deposito(n,m)} - \underbrace{\Delta}_{prelievo(n,m)} = 0.^4$$

  - $d(x)$ aumenta. In questo caso, oltre al costo $O(\Delta)$ delle righe 11-13, vengono effettuate $O(|In(x)|+|Out(x)|)$ istruzioni alle righe 14–16, per cui $T_{ef}(n,m) = O(\Delta+|In(x)|+|Out(x)|)$. L'operazione preleva Δ monete dagli elementi della lista $In(x)$ e $2|In(x)|+|Out(x)|$ monete dal nodo x. Di queste, Δ pagano per il costo delle righe 10–13, $|In(x)|$ pagano per il costo delle righe 14–15, $|Out(x)|$ pagano per il costo della riga 16, e le rimanenti $|In(x)|$ vengono depositate sugli elementi della lista $In(x)$ per preservare la seconda invariante. Per mostrare che anche la prima invariante è preservata, denotiamo con $d(x)$ la distanza di x da s *prima* della `update` e con $d'(x) > d(x)$ il suo valore *dopo* la `update`. Analogamente, $c(x)$ e $c'(x)$ si riferiscono a prima e dopo l'operazione, rispettivamente. Dimostriamo ora che:

    Lemma 1.1 *Sia* $c(x) \geq (n-d(x))\cdot(2|In(x)|+|Out(x)|)$ *prima di una* `update`. *Allora:* $c'(x) \geq (n-d'(x))\cdot(2|In(x)|+|Out(x)|)$ *dopo l'operazione*.

    Dimostrazione. Si ha:

$$\begin{aligned} c'(x) &= c(x) - (2|In(x)|+|Out(x)|) \geq \\ &\geq (n-d(x))\cdot(2|In(x)|+|Out(x)|) - (2|In(x)|+|Out(x)|) \geq \\ &\geq (n-\underbrace{(d(x)+1)}_{\leq d'(x)})\cdot(2|In(x)|+|Out(x)|) \geq \\ &\geq (n-d'(v))\cdot(2|In(x)|+|Out(x)|). \end{aligned}$$

    Il che completa la dimostrazione. □

    Si ha pertanto:

$$T_{am}^{\texttt{update}_2}(n,m) = \underbrace{\Delta + O(|In(x)|+|Out(x)|)}_{T_{ef}(n,m)} + \underbrace{|In(x)|}_{deposito(n,m)} - \underbrace{(\Delta+2|In(x)|+|Out(x)|)}_{prelievo(n,m)} = 0.$$

[4] Ricordiamo che 1 moneta vale $O(1)$ passi di calcolo, per cui assumiamo con un leggero abuso di notazione che $O(\Delta)-\Delta = 0$.

- `dist`: l'operazione non preleva e non deposita monete, e richiede tempo costante. Pertanto:

$$T_{am}^{\texttt{dist}}(n,m) = \underbrace{O(1)}_{T_{ef}(n,m)} + \underbrace{0}_{deposito(n,m)} - \underbrace{0}_{prelievo(n,m)} = O(1).$$

- `delete`: l'operazione non preleva e non deposita monete, ed effettua al più n iterazioni in cui effettua operazioni di costo costante e richiama `update`, che ha costo ammortizzato zero. Possiamo pertanto concludere:

$$T_{am}^{\texttt{delete}}(n,m) = \underbrace{O(n)}_{T_{ef}(n,m)} + \underbrace{0}_{deposito(n,m)} - \underbrace{0}_{prelievo(n,m)} = O(n).$$

Su qualsiasi sequenza di $k_{\texttt{delete}} \leq m$ operazioni `delete`, $k_{\texttt{dist}}$ operazioni `dist` e una `init` iniziale, applicando il Teorema 1.1 ai costi artificiali discussi sopra, otteniamo un costo totale pari a:

$$T(n,k) = \overbrace{O(mn)}^{\substack{\texttt{init}\\(\text{righe 1–3})}} + \overbrace{O(k_{\texttt{dist}})}^{\substack{\texttt{dist}\\(\text{riga 4})}} + \overbrace{O(n \cdot k_{\texttt{delete}})}^{\substack{\texttt{delete}\\(\text{righe 5–8})}} + \overbrace{0}^{\substack{\texttt{update}\\(\text{righe 10–16})}} =$$
$$= O(mn + k_{\texttt{dist}}).$$

Giungiamo quindi in modo alternativo alle stesse conclusioni che avevamo ottenuto con l'analisi 1: per qualunque sequenza mista di $k = \Omega(m)$ operazioni `dist` e `delete` che seguono una `init` iniziale, il costo ammortizzato per operazione è $T(n,k)/k = O(n)$. □