Algoritmi e strutture di dati

Corso di Laurea in Informatica

Dispense

a.a. 2012-2013

Giorgio Gambosi

UNIVERSITÀ DI ROMA ``TOR VERGATA''

Indice

Capitolo 1

Problemi di flusso su reti

Definizioni

Una rete N è una quadrupla $\langle G, s, t, c \rangle$, dove

- $G = (V, E)$ è un grafo orientato
- $s \in V$ è un nodo di V, detto sorgente (source), avente soltanto archi uscenti
- $t \in V$ è un nodo di V, detto destinazione (target), avente soltanto archi entranti
- $c : E \mapsto \mathbb{R}^+$ associa ad ogni arco $e = (u, v) \in E$ una capacità $c(e)$, o $c(u, v)$

Un flusso su una rete N è una funzione $f : E \mapsto \mathbb{R}^+$. In flusso è ammissibile se sono verificate le condizioni seguenti:

1. Limite per la capacità: il flusso su un arco è al più pari alla sua capacità

$$\forall e \in E, 0 \leq f(e) \leq c(e)$$

2. Conservazione del flusso: ad eccezione di s e t, il flusso entrante in un nodo è pari a quello uscente

$$\forall v \in V - \{s, t\}, \sum_{(v,x) \in E} f(v, x) = \sum_{(x,v) \in E} f(x, v)$$

Il valore di un flusso f è definito come il flusso uscente dalla sorgente

$$v(f) = \sum_{(s,x) \in E} f(s, x)$$

Per comodità di notazione, nel seguito, per ogni sottoinsieme $U \subseteq V$ di nodi, indicheremo con $\mathcal{I}(U)$ ed $\mathcal{O}(U)$ gli insiemi di archi entranti in U e uscenti da esso. Più formalmente, $\mathcal{I}(U) = \{(u, v) \in E : u \in V - U, v \in U\}$ e $\mathcal{O}(U) = \{(u, v) \in E : u \in U, v \in V - U\}$. Inoltre, nel caso particolare in cui $U = \{u\}$, scriveremo $\mathcal{I}(u), \mathcal{O}(u)$ per indicare l'insieme degli archi entranti ed uscenti da u. Possiamo allora scrivere

$$v(f) = \sum_{e \in \mathcal{O}(s)} f(e)$$

Dato che tutti i nodi interni (diversi da s, t) conservano il flusso, dovrà necessariamente aversi che il flusso uscente dalla sorgente deve essere pari al flusso entrante nella destinazione.

Teorema 1.1. Per ogni flusso ammissibile f,

$$v(f) = \sum_{e \in \mathcal{O}(s)} f(e) = \sum_{e \in \mathcal{I}(t)} f(e)$$

Dimostrazione. Per il vincolo di conservazione del flusso,

$$\begin{aligned} v(f) &= \sum_{e \in \mathcal{O}(s)} f(e) \\ &= \sum_{e \in \mathcal{O}(s)} f(e) - \sum_{v \in V - \{s,t\}} \left(\sum_{e \in \mathcal{I}(v)} f(e) - \sum_{e \in \mathcal{O}(v)} f(e) \right) \end{aligned}$$

(per ogni nodo diverso da s e t flusso in entrata e in uscita sono uguali)

$$= \sum_{e \in E - \mathcal{I}(t)} (f(e) - f(e)) + \sum_{e \in \mathcal{I}(t)} f(e)$$

(per ogni arco (u, v) non entrante in t il suo flusso compare in uscita per u e in entrata per v)

$$= \sum_{e \in E - \mathcal{I}(t)} f(e)$$

□

Un taglio (cut) tra s e t è un sottoinsieme $A \subset V$ dei nodi della rete tale che $s \in A$ e $t \in V - A$. La capacità $c(A)$ del taglio A è definito come la somma delle capacità di tutti gli archi $e \in \mathcal{O}(A)$ uscenti da A (e quindi entranti in $V - A$, tali cioè che $e \in \mathcal{I}(V - A)$)

$$c(A) = \sum_{e \in \mathcal{O}(A)} c(e)$$

Generalizzando il teorema precedente, possiamo mostrare che un flusso ammissibile deve attraversare qualunque taglio della rete.

Teorema 1.2. Per ogni flusso ammissibile f e per ogni taglio A,

$$v(f) = \sum_{e \in \mathcal{O}(A)} f(e) - \sum_{e \in \mathcal{I}(A)} f(e)$$

Dimostrazione. La dimostrazione è del tutto simile a quella del Teorema 1.1: infatti, ancora per il vincolo di conservazione del flusso,

$$\begin{aligned} v(f) &= \sum_{e \in \mathcal{O}(s)} f(e) \\ &= \sum_{e \in \mathcal{O}(s)} f(e) - \sum_{v \in A - \{s\}} \left(\sum_{e \in \mathcal{I}(v)} f(e) - \sum_{e \in \mathcal{O}(v)} f(e) \right) \\ &= \sum_{e \in A^2} (f(e) - f(e)) + \sum_{e \in \mathcal{O}(A)} f(e) - \sum_{e \in \mathcal{I}(A)} f(e) \end{aligned}$$

(il flusso di un arco tra due nodi (v, v') in A compare due volte: sommato, quando uscente da v e sottratto, quando entrante in v'. Il flusso di un arco uscente da A compare soltanto sommato, quello di un arco entrante in A soltanto sottratto)

$$= \sum_{e \in \mathcal{O}(A)} f(e) - \sum_{e \in \mathcal{I}(A)} f(e)$$

□

Dato che un flusso ammissibile deve attraversare qualunque taglio, e per attraversarlo deve avere valore inferiore alla capacità del taglio stesso, ne deriva che un flusso ammissibile deve avere valore inferiore alla capacità di qualunque taglio nella rete.

Teorema 1.3. Per ogni flusso ammissibile f e per ogni taglio A,

$$v(f) \leq c(A)$$

Dimostrazione. L'enunciato deriva immediatamente osservando che

$$v(f) = \sum_{e \in \mathcal{O}(A)} f(e) - \sum_{e \in \mathcal{I}(A)} f(e) \leq \sum_{e \in \mathcal{O}(A)} f(e) \leq \sum_{e \in \mathcal{O}(A)} c(e) = c(A)$$

□

Quindi, per un qualunque flusso ammissibile f, la capacità di un qualunque taglio $s - t$ fornisce una delimitazione superiore di $v(f)$.

Data una rete N, il problema del massimo flusso (max-flow) chiede di trovare il flusso ammissibile f di valore massimo. Chiaramente, per il Teorema 1.3, abbiamo che

$$\max_{f \text{ ammissibile}} v(f) \leq \min_{A \subset V} c(A) \tag{1.1}$$

Una prima questione è allora se in generale, in questa relazione, vale l'uguaglianza.

Algoritmo greedy per max-flow

Consideriamo un semplice algoritmo ``goloso" (greedy) per trovare il massimo flusso. L'algoritmo, iterativamente, verifica se esiste un cammino (orientato) da s a t i cui archi sono tutti non saturi (diciamo che una arco e è saturo se $f(e) = c(e)$, e quindi se il flusso su e non può aumentare): in questo caso, diciamo che il cammino è non saturo. Se tale cammino esiste, il flusso lungo esso viene incrementato per quanto possibile, in modo da renderlo saturo, altrimenti l'algoritmo termina.

Algorithm 1.1: Algoritmo greedy per maxflow

Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$
Output: Flusso $f : E \mapsto \mathbb{R}^+$ da s a t

```
1 foreach e ∈ E do f(e) ← 0;
2 while esiste un cammino P da s a t con μ_P = min_{e∈P}(c(e) − f(e)) > 0 do
3     foreach e ∈ P do f(e) ← f(e) + μ_p;
4 return f
```

Possiamo facilmente verificare che l'Algoritmo 1.1 non restituisce necessariamente il flusso massimo. Infatti come vediamo, la sua applicazione alla rete in Figura 1.1 può portare a selezionare, come cammino da saturare, quello evidenziato in Figura 1.2, giungendo ad una situazione in cui non esistono altri cammini non saturi e si ha $v(f) = 1$.

Come si può osservare in Figura 1.3, esiste però una soluzione migliore (e ottima) con $v(f) = 2$.

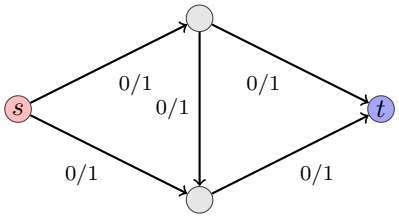


Figura 1.1: Esempio di rete

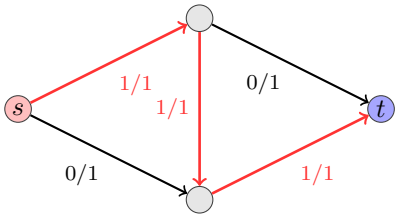


Figura 1.2: Possibile cammino saturato dall'algoritmo: $v(f) = 1$

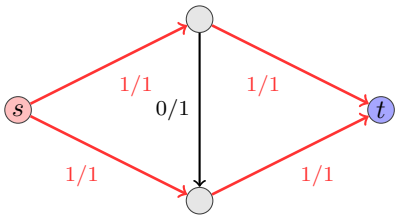


Figura 1.3: Soluzione con flusso massimo: $v(f) = 2$

Teorema di Ford e Fulkerson

Al fine di individuare un algoritmo che effettivamente restituisca sempre il flusso massimo, torniamo alla questione se la relazione tra valori del massimo flusso e del minimo taglio dell'Equazione 1.1 sia effettivamente un'uguaglianza. Possiamo mostrare, nel seguente classico teorema, che la risposta a tale domanda è affermativa.

Teorema 1.4 (Ford, Fulkerson). Per ogni rete N, il valore del massimo flusso è uguale a quello del taglio minimo.

$$\max_{f \text{ ammissibile}} v(f) = \min_{A \subset V} c(A)$$

Al fine di dimostrare il Teorema 1.4, introduciamo il concetto di rete residua.

Definizione 1.1. Data una rete N e un flusso ammissibile f su N, la rete residua N_f è la quadrupla $\langle G_f, s, t, c_f \rangle$, dove

- $G_f = (V, E_f)$, con $E_f = \{(u,v) \in E : f(u,v) < c(u,v)\} \cup \{(u,v) : (v,u) \in E, f(v,u) > 0\}$
- per ogni $(u,v) \in E_f$, $c_f(u,v) = c(u,v) - f(u,v)$ e $c(v,u) = f(u,v)$

Chiamiamo gli archi in N_f archi residui e capacità residue le relative capacità, definite da c_f. N_f rappresenta, con la sua struttura, come sia possibile modificare la distribuzione di flusso su N. Essa comprende, come definito sopra, due tipi di archi:

- archi di N su cui è ancora possibile inviare flusso: un arco di questo tipo ha una capacità residua pari alla differenza tra la sua capacità e il flusso che lo attraversa

- archi opposti agli archi di N che trasportano flusso: un arco di questo tipo ha capacità residua pari al flusso dell'arco di N a cui è opposto

Quindi, ogni arco (u, v) di N è rappresentato in N_f nel modo seguente

- se $f(u, v) = 0$, e quindi l'arco non trasporta flusso, mediante un arco (u, v) di capacità $c_f(u, v) = c(u, v)$
- se $f(u, v) = c(u, v)$, e quindi l'arco è saturo, mediante un arco (v, u) di capacità $c_f(v, u) = f(u, v) = c(u, v)$
- altrimenti, se l'arco trasporta flusso ma non è saturo, mediante un arco (u, v) di capacità $c_f(u, v) = c(u, v) - f(u, v)$ e un arco (v, u) di capacità $c_f(v, u) = f(u, v)$

Si noti che la somma delle capacità residue degli archi (al più due) di N_f corrispondenti ad un arco di N è invariabilmente pari alla sua capacità: la presenza di un flusso che lo attraversa determina soltanto la ripartizione della capacità tra i due archi opposti di N_f.

Per ogni arco $(u, v) \in E$ non saturo (per cui quindi $f(u, v) < c(u, v)$) la capacità residua $c_f(u, v)$ dell'arco è la quantità di flusso che è ancora possibile inviare su (u, v), mentre la capacità residua dell'arco opposto $c_f(v, u)$ è la quantità di flusso che è possibile sottrarre da (u, v).

Definizione 1.2. Data una rete N e un flusso f, un cammino aumentante è un cammino da s a t in N_f.

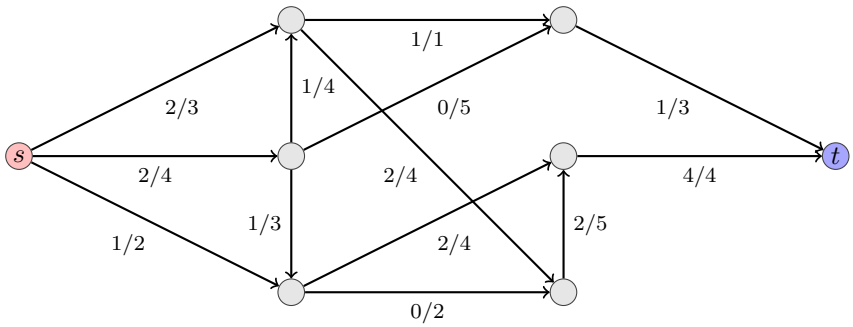


Figura 1.4: Esempio di flusso su rete

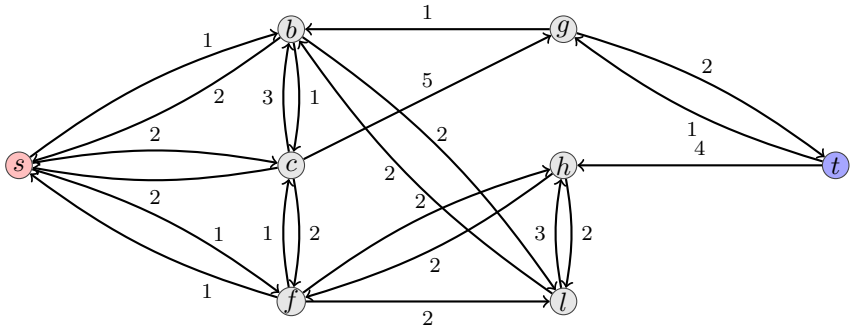


Figura 1.5: Rete residua

Data una rete N e un flusso f, assumiamo che esista un cammino aumentante in N_f e sia $\mu_P = \min_{e \in P} c_f(e)$ la minima capacità residua sugli archi di P.

Definiamo un nuovo flusso f' su E nel modo seguente:

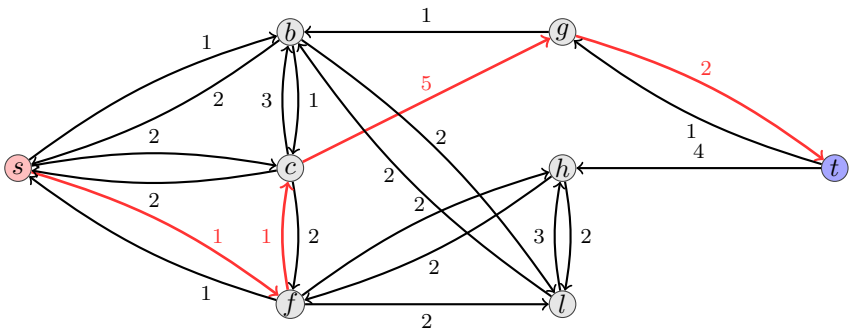


Figura 1.6: Cammino aumentante

1. $f'(u,v) = f(u,v) + \mu_p$ se $(u,v) \in P$
2. $f'(u,v) = f(u,v) - \mu_p$ se $(v,u) \in P$
3. $f'(u,v) = f(u,v)$ altrimenti

Chiaramente, f' è un flusso ammissibile: infatti, la condizione sulla capacità dei nodi è soddisfatta in quanto, per ogni arco $(u,v) \in P$ dove viene incrementato, il flusso non supera, per definizione di μ_P la capacità dell'arco. Inoltre, per ogni arco (u,v) tale che $(v,u) \in P$ dove viene decrementato, il flusso non decresce, ancora per definizione di μ_P, oltre lo 0. Infine, la conservazione del flusso è mantenuta, in quanto in ogni nodo interessato (quelli sul cammino P) si ha il medesimo incremento del flusso in entrata e in uscita.

Inoltre, è evidente che $v(f') = v(f) + \mu_P$.

Possiamo allora dimostrare il seguente teorema.

Teorema 1.5. Data una rete N e un flusso ammissibile f, le seguenti condizioni sono equivalenti:

1. f è un flusso massimale
2. Non esistono cammini aumentanti in N_f
3. Esiste un taglio A tale che $v(f) = c(A)$

Dimostrazione. Mostriamo l'equivalenza dimostrando che $1 \implies 2 \implies 3 \implies 1$.

1. $1 \implies 2$: infatti abbiamo già mostrato poco sopra l'implicazione equivalente $\neg 2 \implies \neg 1$, in cui dalla presenza di cammini aumentanti deriva che il flusso non è massimale
2. $2 \implies 3$: chiamiamo S_f l'insieme dei nodi raggiungibili da s in G_f. Sicuramente $t \notin S_f$ perché altrimenti esisterebbe un cammino aumentante. Possiamo osservare che per ogni $(u,v) \in E$ per cui $u \in S_f$ e $v \notin S_f$, si ha certamente $f(u,v) = c(u,v)$, in quanto altrimenti l'arco (u,v) sarebbe presente in G_f con capacità residua $c_f(u,v) = c(u,v) - f(u,v) > 0$, per cui v sarebbe raggiungibile da s, contraddicendo l'ipotesi che $v \notin S_f$. Inoltre, per ogni $(u,v) \in E$ per cui $u \notin S_f$ e $v \in S_f$ deve essere $f(u,v) = 0$, in quanto altrimenti in G_f sarebbe presente l'arco (v,u) con capacità residua $f(u,v) > 0$, per cui u sarebbe raggiungibile da s, contraddicendo ancora l'ipotesi che $u \notin S_f$.

   Per il Teorema 1.2, abbiamo che

   $$v(f) = \sum_{e \in \mathcal{O}(S_f)} f(e) - \sum_{e \in \mathcal{I}(S_f)} f(e)$$

   ma, dato che $f(e) = c(e)$ per ogni arco e che collega un nodo di S_f ad uno di $V - S_f$, ne deriva che $\sum_{e \in \mathcal{O}(S_f)} f(e) = \sum_{e \in \mathcal{O}(S_f)} c(e)$; inoltre, per ogni arco e' che collega un nodo di $V - S_f$ ad un nodo di

S_f, abbiamo che $f(e') = 0$, per cui $\sum_{e \in \mathcal{I}(S_f)} f(e) = 0$ e quindi, in definitiva,

$$v(f) = \sum_{e \in \mathcal{O}(S_f)} c(e) = c(S_f)$$

3. 3 $\implies$ 1: per il Teorema 1.3, $v(f)$ non può essere maggiore della capacità di un qualunque taglio su V. Dato che, per ipotesi, $v(f) = c(S_f)$, ne deriva che $v(f)$ è il valore massimo possibile per un flusso ammissibile su N.

□

La dimostrazione del Teorema 1.4 deriva immediatamente osservando che S_f deve essere necessariamente un taglio di capacità minima, in quanto altrimenti, detto M il taglio di capacità minima $c(M) < c(S_f)$ il flusso massimo f avrebbe valore $v(f) > c(M)$, contraddicendo il Teorema 1.3.

Algoritmo di Ford e Fulkerson

Le osservazioni precedenti permettono di definire un algoritmo per il calcolo del massimo flusso in una rete.

Algorithm 1.2: Algoritmo di Ford e Fulkerson per il massimo flusso

Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$
Output: Flusso $f : E \mapsto \mathbb{R}^+$ da s a t

```
1 foreach e ∈ E do f(e) ← 0;
2 Deriva N_f;
3 while esiste un cammino aumentante da s a t in N_f do
4     Prendi un qualunque cammino aumentante P;
5     foreach (u, v) ∈ P do
6         if (u, v) ∈ E then f(e) ← f(e) + min_{e∈P} c_f(e);
7         else f(e) ← f(e) − min_{e∈P} c_f(e)
8     Aggiorna N_f
9 return f
```

In Figura 1.7 viene mostrata la rete residua derivante da quella in Figura 1.5 aumentando il flusso lungo il cammino aumentante in Figura 1.6. In Figura 1.8 viene rappresentato il flusso corrispondente: si può osservare come il valore del flusso da s a t sia aumentato da 5 a 6.

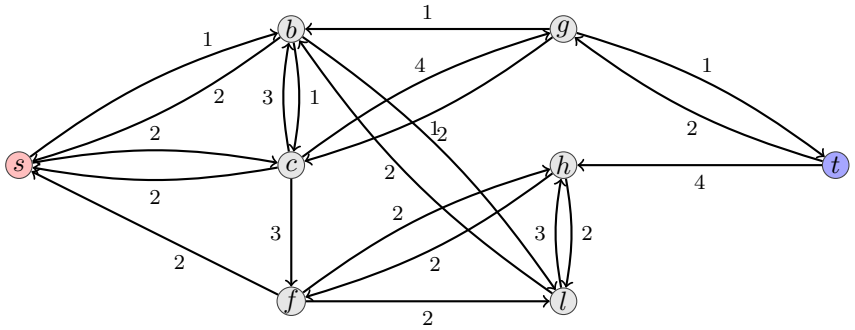


Figura 1.7: Rete residua successiva

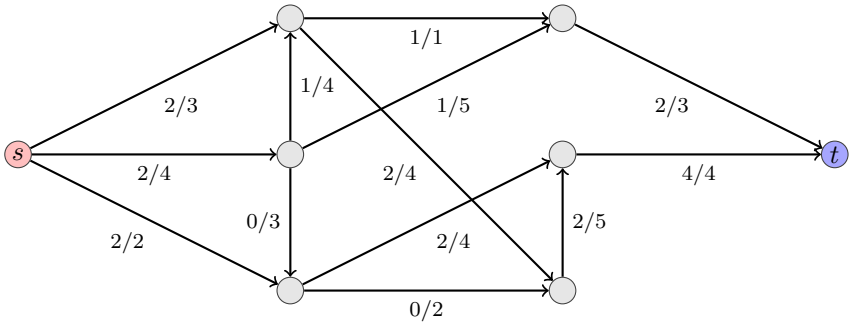


Figura 1.8: Flusso corrispondente alla rete residua di Figura 1.7

Per il Teorema 1.5, se l'algoritmo si ferma, non trovando cammini aumentanti, il flusso ottenuto è massimale. Si osservi però che, in presenza di flussi e capacità definite sui reali, l'incremento di flusso ad ogni singolo passo, pur necessariamente positivo, può tendere a 0 e l'algoritmo può non terminare, approssimando asintoticamente, all'infinito, il valore del flusso massimale.

Tale situazione può verificarsi, ad esempio, per la rete in Figura 1.9, in cui con r abbiamo indicato la soluzione positiva dell'equazione $x^2 + x - 1 = 0$, vale a dire

$$r = \frac{\sqrt{5} - 1}{2} \simeq 0.618$$

(incidentalmente, tale valore è l'inverso della sezione aurea $\phi = \frac{1+\sqrt{5}}{2}$). Si noti che, per definizione, $r^2 = 1 - r$ e che, in generale, $r^{n+2} = r^n - r^{n+1}$ per $n \geq 1$. Si noti che il flusso massimo ha valore $v(f^*) = 1 + r + r^2 = 2$.

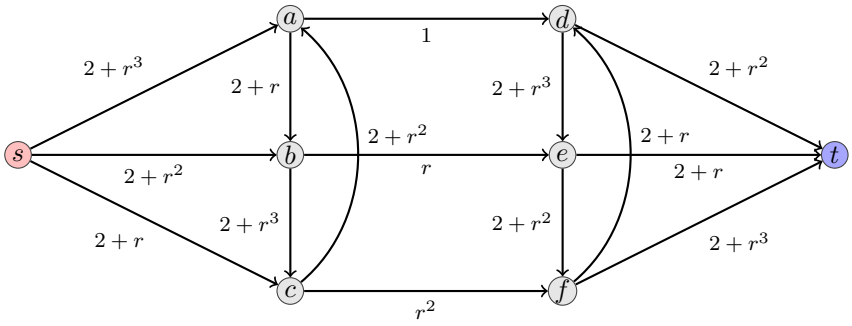


Figura 1.9: Esempio di rete su cui l'algoritmo di Ford e Fulkerson converge all'ottimo in infiniti passi

Assumiamo che il primo cammino aumentante selezionato dall'algoritmo corrisponda a $s - a - d - t$: assegnare agli archi su tale cammino la capacità residua minima su di essi (pari a 1), porta alla situazione in Figura 1.10, in cui ad ogni arco è associata la capacità residua, e ad un flusso di valore $v(f) = 1$.

Assumiamo ora che il nuovo cammino aumentante selezionato dall'algoritmo sia $s - c - f - d - a - b - e - t$: con capacità residua minima pari a r^2: ciò porta alla situazione in Figura 1.11, in cui ad ogni arco è associata la capacità residua (ricordando che $r - r^2 = r^3$), e ad un flusso di valore $v(f) = 1 + r^2$.

Assumiamo ora che il nuovo cammino aumentante selezionato dall'algoritmo sia $s - b - e - f - c - a - d - t$: con capacità residua minima pari a r^3: ciò porta alla situazione in Figura 1.12, in cui ad ogni arco è associata la capacità residua (ricordando che $r^2 - r^3 = r^4$), e ad un flusso di valore $v(f) = 1 + r^2 + r^3$.

Assumiamo ora che il nuovo cammino aumentante selezionato dall'algoritmo sia $s - a - d - e - b - c - f - t$: con capacità residua minima pari a r^4: ciò porta alla situazione in Figura 1.13, in cui ad ogni arco è associata la capacità residua (ricordando che $r^3 - r^4 = r^5$), e ad un flusso di valore $v(f) = 1 + r^2 + r^3 + r^4$.

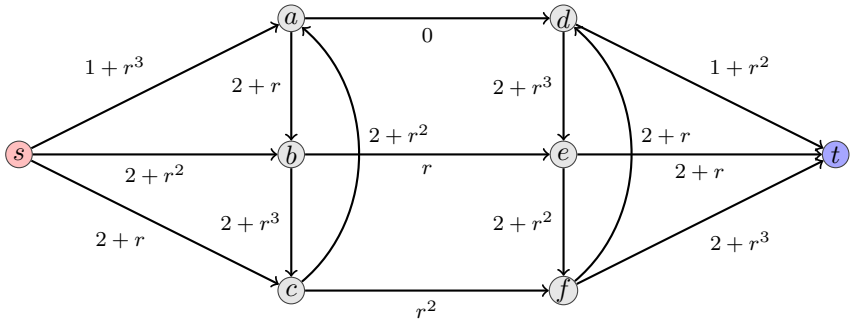


Figura 1.10: Rete precedente con flusso assegnato dopo un passo dell'algoritmo di Ford e Fulkerson

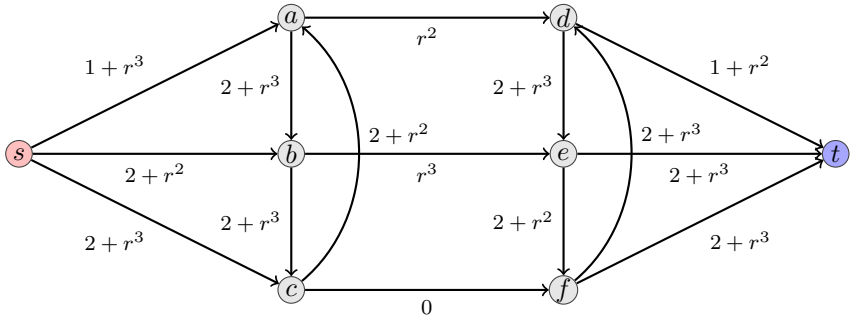


Figura 1.11: Rete precedente con flusso assegnato dopo due passi dell'algoritmo di Ford e Fulkerson

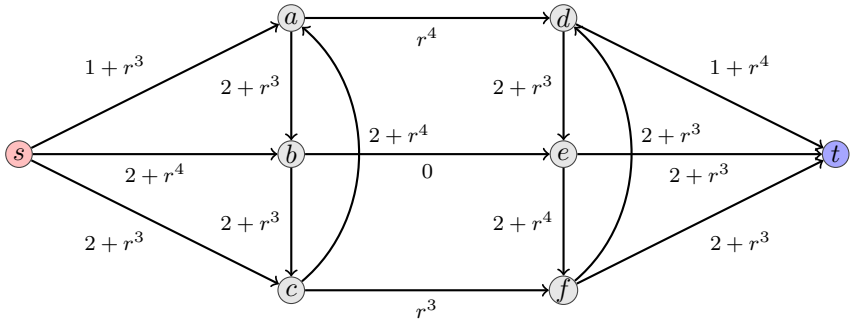


Figura 1.12: Rete precedente con flusso assegnato dopo tre passi dell'algoritmo di Ford e Fulkerson

Proseguendo allo stesso modo, il flusso aggiunto dal k-esimo cammino aumentante ha valore r^k. Quindi, dopo k iterazioni, il flusso ha valore

$$v(f_k) = 1 + \sum_{i=2}^{k} r^i = -r + \sum_{i=0}^{k} r^i = \frac{1 - r^{k+1}}{1 - r} - r$$

Per k che tende ad infinito, abbiamo allora che

$$\lim_{k \to \infty} v(f_k) = \frac{1}{1 - r} - r = \frac{1 - r + r^2}{1 - r} = \frac{2(1 - r)}{1 - r} = 2$$

per cui il valore del flusso tende, in un numero infinito di passi, al valore del flusso massimo.

La possibilità di una convergenza all'ottimo soltanto in un numero infinito di passi viene esclusa se ci si limita a considerare capacità definite su insiemi discreti, come ad esempio gli interi.

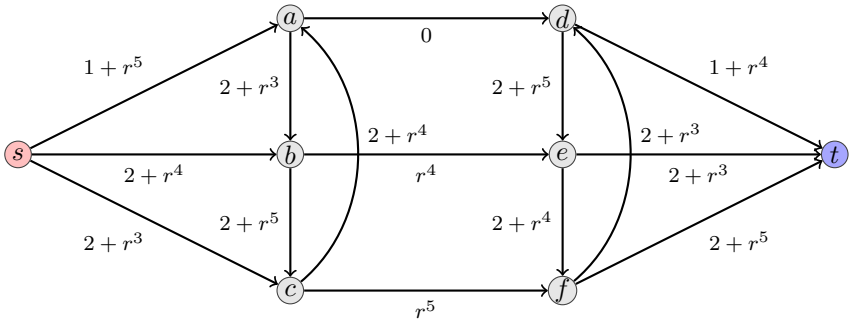


Figura 1.13: Rete precedente con flusso assegnato dopo quattro passi dell'algoritmo di Ford e Fulkerson

Teorema 1.6. Se $c(e) \in \mathbb{N}$ per tutti gli archi $e \in E$, allora l'Algoritmo 1.2 termina dopo un numero finito di passi.

Dimostrazione. Dato che le capacità degli archi di G sono intere, sono interi anche i valori dei flussi e le capacità degli archi di N_f. Quindi, se esiste un cammino aumentante, il suo arco di capacità minima avrà capacità intera, e quindi il flusso di tutti gli archi o rimane immutato (se gli archi non appartengono al cammino aumentante) o viene incrementato di un valore intero (se sono sul cammino).

Dato che $v(f) \leq \sum_{e \in \mathcal{O}(s)} c(e)$, ne consegue che $v(f)$ può essere incrementato al più $\sum_{e \in \mathcal{O}(s)} c(e) < \infty$ volte. □

Trovare un cammino aumentante su N_f richiede una visita in ampiezza del grafo, e quindi tempo $O(|E_f|) = O(|E|)$. Di conseguenza, l'Algoritmo 1.2 ha complessità $O(v(f^*) \cdot |E|)$, dove f^* è il flusso massimo. Si noti che l'algoritmo non è quindi polinomiale nella dimensione dell'input (che ha dimensione $\Theta(|E|(\log |V| + \log c_{\text{max}}))$), dove c_{max} è il valore della massima capacità di un arco in N.

Un esempio di rete con capacità intere per la quale l'Algoritmo 1.2 si comporta in modo inefficiente è mostrata in Figura 1.14.

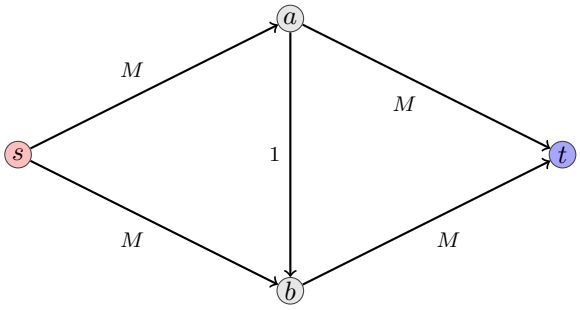


Figura 1.14: Esempio di rete su cui l'algoritmo di Ford e Fulkerson richiede tempo esponenziale

Supponiamo che il primo cammino aumentante selezionato sia $s - a - b - t$, attraverso il quale è possibile inviare un flusso di valore unitario. La rete residua derivante è mostrata in Figura 1.15 , in cui ad ogni arco è associata la capacità residua. Il valore del flusso è $v(f) = 1$.

Se il successivo cammino aumentante selezionato è $s - b - a - t$, attraverso il quale è possibile inviare un flusso di valore unitario, ne deriva la rete rete residua in Figura 1.16 , in cui ad ogni arco è associata la capacità residua. Il valore del flusso è $v(f) = 2$. Iterando i passi precedenti, abbiamo che ad ogni iterazione il flusso aumenta di 1 unità. Dato che il flusso massimo è chiaramente pari a $2M$ (il valore del taglio minimo), ne deriva che l'algoritmo di Ford e Fulkerson esegue $2M$ iterazioni.

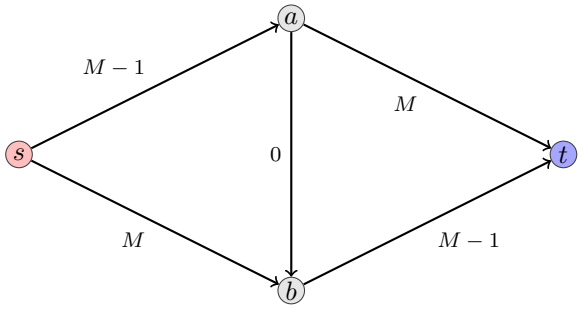


Figura 1.15: Rete precedente con flusso assegnato dopo un passo dell'algoritmo di Ford e Fulkerson

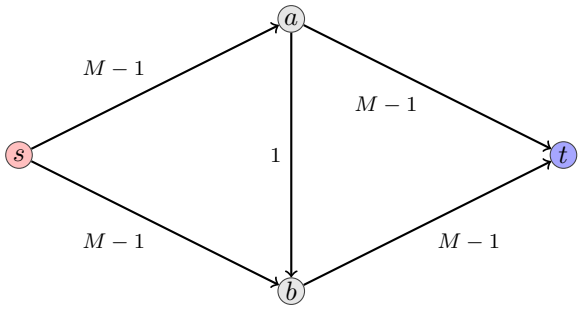


Figura 1.16: Rete precedente con flusso assegnato dopo due passi dell'algoritmo di Ford e Fulkerson

Decomposizione del flusso

Al fine di definire algoritmi polinomiali per il problema del massimo flusso, è necessario selezionare il cammino aumentante in modo oculato e al tempo stesso efficiente. Prima di considerare vari modi di far questo, mostriamo che una scelta sufficientemente oculata può consentire di avere ad ogni iterazione un aumento significativo del flusso, nel senso che esiste sempre un cammino aumentante la cui selezione porterebbe a tale incremento.

Per mostrare ciò, introduciamo per prima cosa il teorema seguente, che ci dice che un flusso ammissibile può essere decomposto su un insieme opportuno di cicli e cammini $s - t$.

Teorema 1.7. (Flow decomposition) Dato un flusso ammissibile f, esiste un insieme di cammini $\mathcal{P}$, un insieme di cicli $\mathcal{C}$ e una assegnazione di pesi $w : \mathcal{P} \cup \mathcal{C} \mapsto \mathbb{R}^+$, che interpretiamo come flussi lungo cammini e cicli, tali che:

1. $\forall e \in E, f(e) = \sum_{p \in \mathcal{PC}(e)} w(p)$, dove $\mathcal{PC}(e) = \{p \in \mathcal{P} \cup \mathcal{C} : e \in p\}$: in altri termini, il flusso su ogni arco può essere decomposto nella somma dei flussi associati a tutti i cammini e i cicli che includono l'arco stesso
2. $v(f) = \sum_{p \in \mathcal{P}} w(p)$: il flusso totale da s a t è decomposto nella somma dei flussi lungo tutti i cammini
3. $|\mathcal{P}| + |\mathcal{C}| \leq |E_f|$, dove $E_f = \{e \in E : f(e) > 0\}$ è l'insieme degli archi aventi flusso positivo: vale a dire, il numero di cicli e cammini non supera quello degli archi che trasportano flusso

Dimostrazione. La dimostrazione è effettuata per induzione sul numero di archi $e \in E_f$.

Caso base. $f(e) = 0$ per ogni $e \in E$, per cui $E_f = \emptyset$. Allora il lemma è banalmente verificato per $\mathcal{P}, \mathcal{C} = \emptyset$.

Passo induttivo. Assumiamo che le condizioni del teorema siano verificate per ogni flusso avente valore positivo su k archi. Consideriamo un qualunque flusso f avente valore positivo su $k + 1$ archi. Sia $(u, v) \in E$ un arco con $f(u, v) > 0$. Se $v \neq t$, per il vincolo di conservazione del flusso al nodo v esiste un nodo x tale che $f(v, x) > 0$. Simmetricamente, se $u \neq s$ esiste $y \in V$ tale che $f(y, u) > 0$. Iterando la stessa

considerazione per x e y, otteniamo o un cammino da s a t o un ciclo, che in entrambi i casi indichiamo come $\overline{p}$. Posto $w(\overline{p}) = \min_{e \in \overline{p}} f(e)$, consideriamo il flusso f' definito nel modo seguente:

$$f'(i,j) = \begin{cases} f(i,j) - w(\overline{p}) & (i,j) \in \overline{p} \\ f(i,j) & \text{altrimenti} \end{cases}$$

$f'(i,j)$ è quindi il flusso ottenuto riducendo al massimo quanto trasportato lungo $\overline{p}$. Questo insieme esiste e soddisfa le tre proprietà dell'enunciato, in quanto f' ha almeno un arco con flusso positivo in meno di f (quello di flusso minimo in $\overline{p}$). Indichiamo con $\mathcal{PC}'$ l'insieme di cicli e cammini associato a f' e con w' la relativa funzione di peso.

Definendo $\mathcal{PC} = \mathcal{PC}' \cup \{\overline{p}\}$ e $w(p) = w'(p)$ per ogni $p \in \mathcal{PC}'$, possiamo osservare che:

- la proprietà 1 vale anche per f: infatti, per ogni $e \notin \overline{p}$ si ha che $f(e) = f'(e)$ per definizione di f' e $\mathcal{PC}(e) = \mathcal{PC}'(e)$, mentre se $e \in \overline{p}$ allora $f(e) = f'(e) + w(\overline{p})$ e $\mathcal{PC}(e) = \mathcal{PC}'(e) \cup \{\overline{p}\}$.
- la proprietà 2 valeper f in quanto se $\overline{p} \in \mathcal{P}$ allora $\mathcal{P}' = \mathcal{P} \cup \{\overline{p}\}$ e

  $$v(f) = v(f') + w(\overline{p}) = \sum_{p \in \mathcal{P}} w(p) + w(\overline{p}) = \sum_{p \in \mathcal{P}'} w(p)$$

  mentre se se $\overline{p} \in \mathcal{C}$ allora $\mathcal{P}' = \mathcal{P}$ e $v(f) = v(f')$.
- infine, per quanto riguarda la proprietà 3, basta osservare che, nel passare da f' a f, il numero complessivo di cammini e cicli è aumentato di 1 ($\overline{p}$), mentre il numero di archi con flusso positivo è aumentato di almeno 1 (l'arco di flusso minimo in $\overline{p}$).

□

Possiamo allora derivare immediatamente il teorema seguente, che ci dice che esiste sempre un cammino da s a t sul quale viene trasportata una frazione significativa del flusso massimo.

Teorema 1.8. Data una rete N di flusso massimo f^* esiste un cammino da s a t i cui archi hanno assegnato ciascuno un flusso di valore almeno $\dfrac{v(f^*)}{|E|}$.

Dimostrazione. Per il Teorema 1.7, esiste in N un insieme $\mathcal{P}$ di cammini da s a t che si ripartiscono l'intero f^*.

Quindi, esiste un cammino $p_i \in \mathcal{P}$ con assegnato un flusso f_i tale che

$$v(f_i) \geq \frac{v(f^*)}{|\mathcal{P}|} \geq \frac{v(f^*)}{|E|} \qquad \text{dato che } |E| \geq |E_{f^*}| \geq |\mathcal{P}| + |\mathcal{C}| \geq |\mathcal{P}|$$

Dato che ogni arco su un cammino che porta flusso f deve necessariamente avere un flusso di valore almeno pari a $v(f)$, ne deriva l'enunciato. □

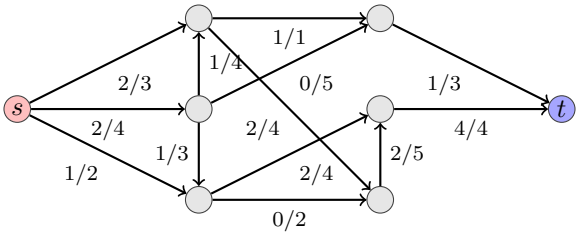
Il Teorema 1.7 ci dice che qualunque flusso può essere decomposto in un insieme di cammini da s a t e un insieme di cicli (al più tanti quanti sono gli archi del grafo), ognuno con flusso associato w, in modo tale che il flusso su un arco è dato dalla somma dei flussi sui cammini e sui cicli cui l'arco appartiene.

Il teorema seguente ci dice che il problema del massimo flusso si riduce a sé stesso quando da una rete si sottrae un flusso.

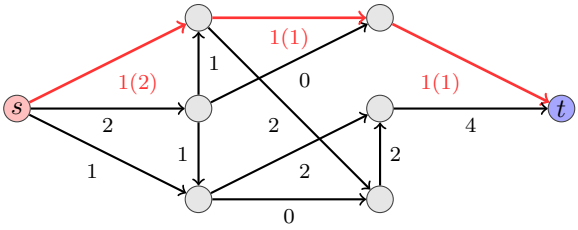
Teorema 1.9. Sia data una rete N di flusso massimo f^* e sia f un flusso su essa. Allora, il flusso massimo sulla rete N' ottenuta ponendo per ogni arco e la relativa capacità come $c'(e) = c(e) - f(e)$ ha valore $v(f^*) - v(f)$.

Dimostrazione. Come prima cosa, mostriamo che ogni flusso in N' ha valore al più pari a $v(f^*) - v(f)$. A tal fine, consideriamo un qualunque flusso f' in N' e definiamo il flusso $\overline{f}$ come:

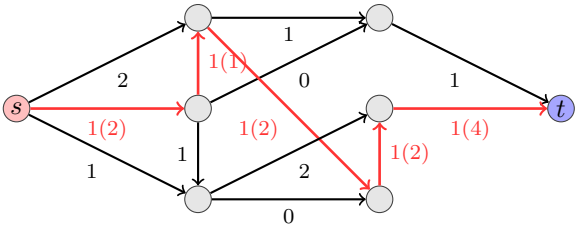
$$\overline{f}(e) = f(e) + f'(e) \qquad \forall e \in E$$



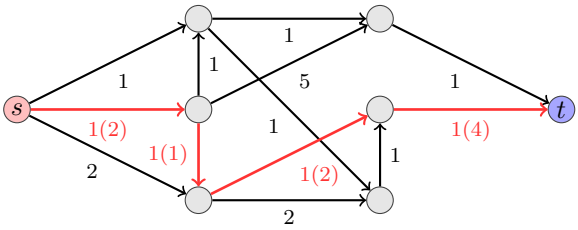
(a) Esempio di flusso



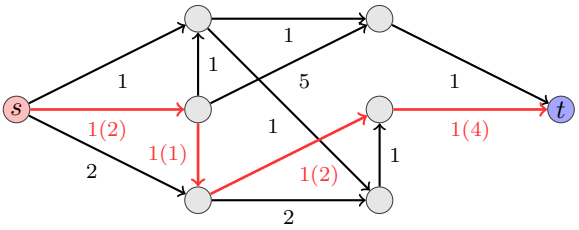
(b) Cammino 1



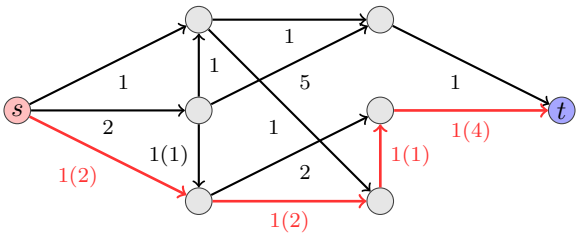
(c) Cammino 2



(d) Cammino 3



(e) Cammino 4



(f) Cammino 5

Figura 1.17: Esempio di decomposizione di flusso in cammini

Il flusso $\overline{f}$ è quindi ottenuto a partire da f', ripristinando il flusso f originariamente sottratto da N: chiaramente, $\overline{f}$ è un flusso su N, in quanto le condizioni di continuità sui nodi e di rispetto delle capacità degli archi continuano a valere. Inoltre, si ha che $v(\overline{f}) = v(f) + v(f')$ per definizione e che $v(\overline{f}) \leq v(f^*)$ per la massimalità di f^*: da ciò deriva che $v(f) + v(f') \leq v(f^*)$ e quindi $v(f') \leq v(f^*) - v(f)$

Definiamo ora il flusso $\hat{f}$ come:

$$\hat{f}(e) = f^*(e) - f(e) \qquad \forall e \in E$$

Osserviamo che $\hat{f}$ è un flusso in N', in quanto $\hat{f}(e) = f^*(e) - f(e) \leq c(e) - f(e)$. Inoltre, dato che $v(\hat{f}) = v(f^*) - v(f)$ per definizione, $\hat{f}$ è un flusso massimo in N'. □

Chiaramente, per definizione di rete residua, $v(f^*) - v(f)$ è anche il massimo flusso nella rete residua N_f corrispondente.

Algoritmi polinomiali per max-flow

Al fine di rendere più efficiente l'Algoritmo 1.2, dobbiamo effettuare, ad ogni iterazione, una scelta oculata del cammino aumentante, selezionando cammini che consentano un incremento significativo del flusso.

Cammino di capacità massima

Una scelta ragionevole appare quella di selezionare il cammino aumentante di capacità massima, ottenendo l'Algoritmo 1.3

Algorithm 1.3: Algoritmo di cammino aumentante di massima capacità

Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$
Output: Flusso $f : E \mapsto \mathbb{R}^+$ da s a t

```
1 foreach e ∈ E do f(e) ← 0;
2 Deriva N_f;
3 while esiste un cammino aumentante da s a t in N_f do
4     Prendi il cammino aumentante P tale che min_{e∈P}(c_f(e)) è massimo;
5     foreach (u, v) ∈ P do
6         if (u, v) ∈ E then f(e) ← f(e) + min_{e∈P} c_f(e);
7         else f(e) ← f(e) − min_{e∈P} c_f(e)
8     Aggiorna N_f
9 return f
```

Al fine di mostrare che l'algoritmo opera in tempo polinomiale, dimostriamo il seguente teorema.

Teorema 1.10. L'Algoritmo 1.3 calcola il flusso massimo f^* effettuando $O(|E| \log v(f^*))$ iterazioni.

Dimostrazione. Indichiamo con f_i il flusso totale trovato dopo i iterazioni, e quindi dopo aver trovato i cammini aumentanti (e aver aumentato il flusso per i volte). Mostriamo che nella rete residua ottenuta N_{f_i} il flusso massimo ha valore al più pari a

$$v(f^*)\left(1 - \frac{1}{2|E|}\right)^i$$

Notiamo che in generale, per il Teorema 1.9, il flusso complessivo ancora disponibile nella rete N_{f_i} è $v(f^*) - v(f_i)$ e, per il Teorema 1.8, esiste uno di tali cammini (che chiamiamo p_i^*) di capacità almeno pari al rapporto tra $v(f^*) - v(f_i)$ e il numero di archi nella rete, che in una rete residua è al più $2|E|$.

Quindi, il cammino aumentante in N_{f_i} corrispondente a p_i^* ha capacità almeno $\dfrac{v(f^*) - v(f_i)}{2|E|}$. L'Algoritmo 1.3, selezionando proprio tale cammino, fa sì che

$$v(f_{i+1}) \geq v(f_i) + \frac{v(f^*) - v(f_i)}{2|E|}$$

Il flusso ancora disponibile dopo l'$(i+1)$-esima iterazione è allora

$$\begin{aligned} v(f^*) - v(f_{i+1}) &\leq v(f^*) - \left(v(f_i) - \frac{v(f^*) + v(f_i)}{2|E|} \right) = (v(f^*) - v(f_i)) \left(1 - \frac{1}{2|E|} \right) \\ &\leq (v(f^*) - v(f_{i-1})) \left(1 - \frac{1}{2|E|} \right)^2 \leq \dots \\ &\leq (v(f^*) - v(f_{i-k})) \left(1 - \frac{1}{2|E|} \right)^{k+1} \end{aligned}$$

Dato che inizialmente $v(f_0) = 0$, abbiamo che

$$v(f^*) - v(f_{i+1}) \leq v(f^*) \left(1 - \frac{1}{2|E|} \right)^{i+1}$$

In generale, si ha che $1 - x \leq e^{-x}$ se $x \geq 0$,per cui

$$v(f^*) - v(f_{i+1}) \leq v(f^*) \left(1 - \frac{1}{2|E|} \right)^{i+1} \leq v(f^*) e^{-\frac{i+1}{2|E|}} = e^{\log v(f^*) - \frac{i+1}{2|E|}}$$

se $i > 2|E| \log v(f^*) - 1$ ne consegue che $e^{\log v(f^*) - \frac{i+1}{2|E|}} < 1$, quindi $v(f^*) - v(f_{i+1}) < 1$ e di conseguenza $v(f^*) - v(f_{i+1}) = 0$ necessariamente, in quanto le capacità e i flussi sono interi per ipotesi.

In definitiva, abbiamo mostrato che l'Algoritmo 1.3 in al più $1+2|E| \log f(v^*)$ iterazioni raggiunge una situazione in cui la rete residua non ha più cammini aumentanti, e quindi il flusso trovato ha valore massimo $v(f^*)$. □

La complessità totale dell'Algoritmo 1.3 dipende inoltre dal costo della ricerca di un cammino di capacità massima, operazione effettuata ad ogni iterazione. Mostriamo ora che questa ricerca può essere effettuata mediante una semplice modifica dell'algoritmo di Dijkstra per la ricerca del cammino minimo tra s e t, riportato come Algoritmo 1.4.

Algorithm 1.4: Algoritmo di Dijkstra per la ricerca di un cammino minimo

Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$;
Output: Cammino minimo da s a t;

```
1  foreach v ∈ V − {s} do d(v) ← ∞;
2  d(s) ← 0;
3  foreach v ∈ V do
4      Inserisci v in una coda di priorità PQ con chiave d(v);
5      π(v) ←nil;
6  while PQ ≠ ∅ do
7      Estrai da PQ il nodo u con d(u)minimo;
8      foreach v tale che (u, v) ∈ E do
9          if d(v) > d(u) + l(u, v) then
10             d(v) ← d(u) + l(u, v);
11             π(v) ← u;
12             Decrementa a d(v) la chiave di v in PQ;
13 Sia P il cammino da t a s definito dai puntatori π;
14 return il cammino inverso di P
```

Come detto, l'algoritmo di Dijkstra può essere modificato in modo semplice per trovare il cammino di capacità massima in un grafo. A tal fine osserviamo che:

1. nella ricerca di un cammino minimo il valore associato ad un cammino comprendente gli archi $E_P = \{e_1, \dots, e_k\}$ è dato da $\sum_{e \in E_P} l(e)$; nel ricerca di un cammino di capacità massima, il valore del cammino è $\min_{e \in E_P} c(e)$

2. nel primo caso cerchiamo un minimo, nel secondo un massimo

L'Algoritmo 1.4 è basato su alcune caratteristiche del problema del cammino minimo, che sono valide ``mutatis mutandis'' anche per il problema del cammino di capacità massima.

- il costo di un cammino è monotono all'estendere del cammino stesso: ad esempio, per il cammino minimo, il costo di un cammino $E_P = \{e_1, \ldots, e_k\}$ è almeno pari al costo del cammino $E_{P'} = \{e_1, \ldots, e_t\}$, con $t < k$ (per il cammino di capacità massima, è al più pari a tale valore)
- è possibile determinare il costo di un cammino $E_P = \{e_1, \ldots, e_k\}$ a partire dal costo del cammino $E_{P''} = \{e_1, \ldots, e_{k-1}\}$ e da $l(e_k)$ (o $c(e_k)$)

Da queste considerazioni, deriva l'Algoritmo 1.5 per la ricerca del cammino di capacità massima.

Algorithm 1.5: Algoritmo di ricerca di un cammino aumentante di massima capacità

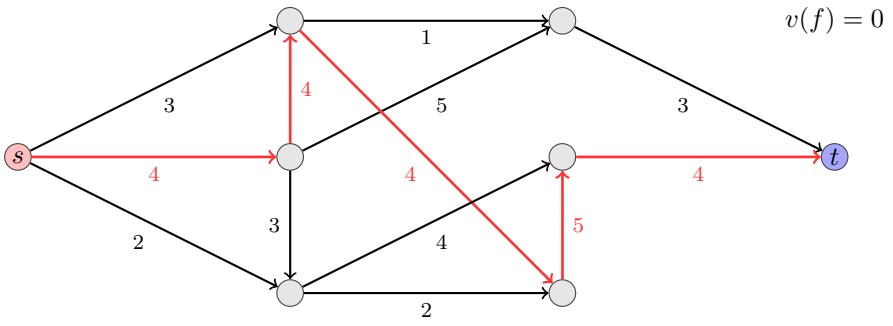
Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$;
Output: Cammino di capacità massima P da s a t;

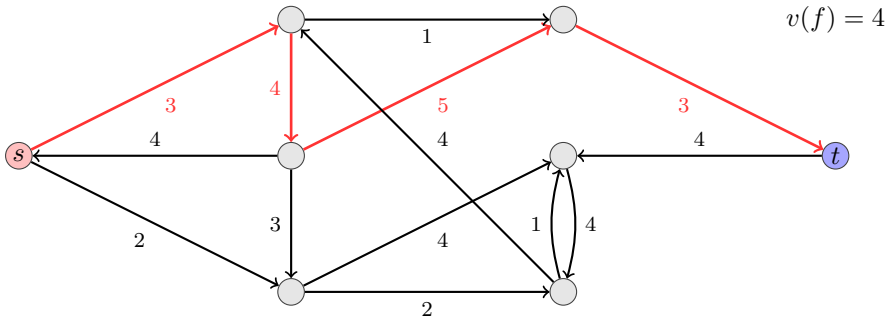
```
1  foreach v ∈ V − {s} do m(v) ← 0;
2  m(s) ← ∞;
3  foreach v ∈ V do
4      Inserisci v in una coda di priorità PQ con chiave m(v);
5      π(v) ←nil;
6  while PQ ≠ ∅ do
7      Estrai da PQ il nodo u con m(u) massimo;
8      foreach v tale che (u, v) ∈ E do
9          if m(v) < min{m(u), c(u, v)} then
10             m(v) ← min{m(u), c(u, v)};
11             π(v) ← u;
12             Incrementa a m(v) la chiave di v in PQ;
13 Sia P il cammino da t a s definito dai puntatori π;
14 return il cammino inverso di P
```

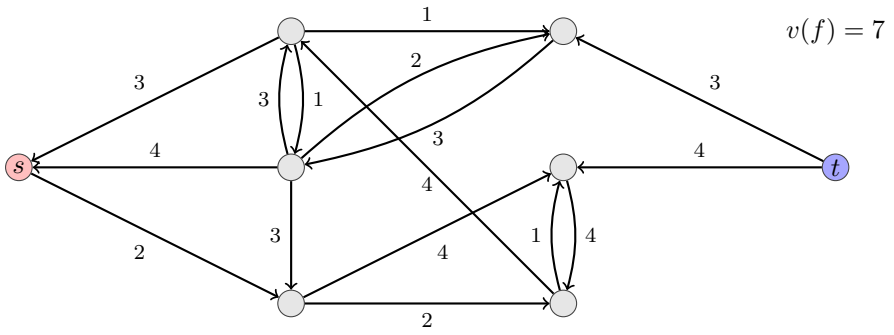
L'Algoritmo 1.5 ha la stessa struttura e la stessa complessità $O((|V| + |E|) \log |V|)$ dell'Algoritmo 1.4, da cui deriva che l'Algoritmo 1.3 ha complessità totale $O((|V| + |E|)|E| \log |V| \log v(f^*))$.

La complessità ottenuta, come si può vedere, è funzione del valore del flusso ottimo, che non è un dato in input del problema. Possiamo però osservare, ad esempio, che $v(f^*) \leq \sum_{e \in \mathcal{O}(s)} c(e)$ e ottenere così una ulteriore delimitazione superiore $O((|V| + |E|)|E| \log |V| \log \sum_{e \in \mathcal{O}(s)} c(e))$ di tale complessità.

Di seguito, viene illustrato l'effetto sulla rete di Figura 1.4 dell'esecuzione dell'algoritmo basato sui cammini di massima capacità. In rosso è mostrato, di volta in volta, il cammino di capacità massima da s a t. Si noti che il massimo viene trovato utilizzando due cammini aumentanti, in quanto dopo due iterazioni non esistono più cammini da s a t.







Cammino δ-aumentante

Una variante della ricerca del cammino aumentante di massima capacità è costituita dal limitarsi a selezionare un cammino di capacità ``sufficientemente grande", in modo tale da incrementare comunque il flusso di una quantità significativa, anche se non la massima possibile. Se un cammino di questo tipo non viene trovato (nel senso che tutti i cammini disponibili hanno capacità limitata), allora possiamo verificare che il valore del flusso attuale è vicino al massimo.

Definizione 1.3. Un cammino P da s a t viene detto δ-aumentante se è aumentante e tutti i suoi archi hanno capacità almeno δ: cioè, se $\forall e \in P, c(e) \geq \delta$.

L' Algoritmo 1.6 di capacity scaling opera cercando iterativamente cammini δ-aumentanti, facendo tendere δ a 0.

Ad ogni iterazione, l'algoritmo verifica la presenza di un cammino aumentante di capacità almeno δ (inizialmente, δ è circa pari alla capacità dell'arco di capacità massima). Se tale cammino esiste, viene selezionato e flusso e rete residua vengono aggiornati, altrimenti, il valore di δ viene dimezzato. L'algoritmo termina quando non esistono più cammini aumentanti.

La ricerca, dato δ, di un cammino aumentante di capacità almeno δ può essere effettuata mediante una semplice modifica dell'algoritmo di visita di un grafo in cui si considerano i soli archi di capacità almeno δ.

Teorema 1.11. L'Algoritmo 1.6 determina il flusso ottimo di una rete N in tempo $O((|V| + |E|)|E| \log C)$, dove $C = \max_{e \in E} c(e)$.

Dimostrazione. Chiaramente l'algoritmo, procedendo fino a che esistono cammini aumentanti, trova il flusso ottimo.

Per quanto riguarda la complessità, assumiamo, per induzione, che, nel momento in cui l'algoritmo cerca cammini di capacità almeno δ, non ne esistano di capacità $\geq 2\delta$. Ciò è banalmente verificato all'inizio, quando $\delta = 2^{\lfloor \log_2 C \rfloor} > 2^{\log_2 C - 1} = \dfrac{C}{2}$.

Dato che, per ogni cammino di capacità pari almeno a δ esiste almeno un arco con tale capacità, ne consegue che il numero di cammini individuati per ogni δ non può essere superiore a $2|E|$ (si ricordi che N_f può avere fino

Algorithm 1.6: Algoritmo di cammino δ-aumentante

Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$
Output: Flusso $f : E \mapsto \mathbb{R}^+$ da s a t

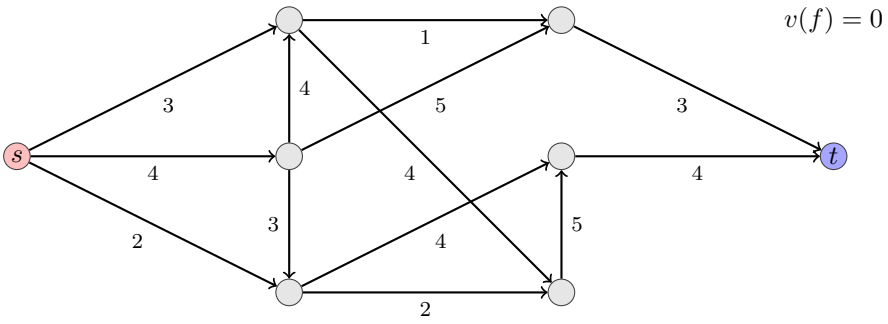
```
1  C ← max_{e∈E} c(e);
2  δ ← 2^⌊log₂ C⌋;
3  foreach e ∈ E do f(e) ← 0;
4  Deriva N_f;
5  while esiste un cammino aumentante da s a t in N_f do
6      if esiste un cammino δ-aumentante P then
7          foreach (u, v) ∈ P do
8              if (u, v) ∈ E then f(e) ← f(e) + δ;
9              else f(e) ← f(e) − δ
10     else δ ← δ/2;
11     Aggiorna N_f
12 return f
```

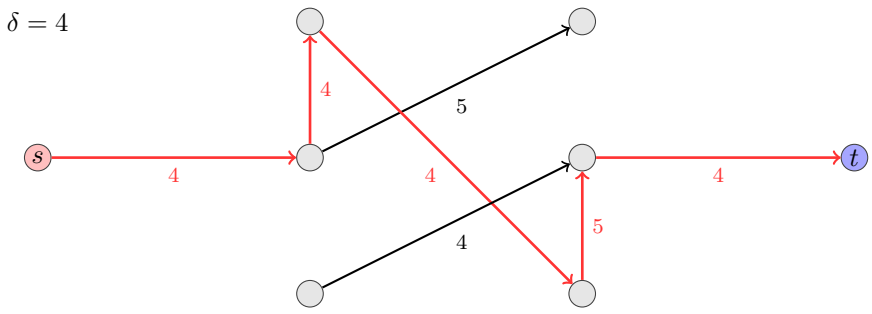
a $2|E|$ archi). Le capacità di tutti i cammini sono almeno dimezzate, dato che le loro capacità erano, per l'ipotesi induttiva, minori di 2δ.

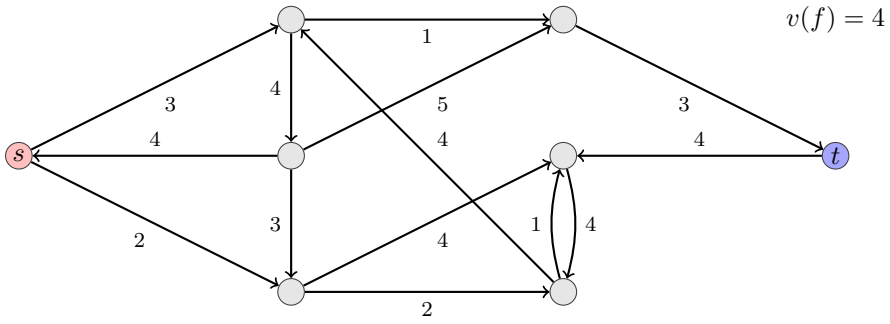
Per ogni δ, possono essere quindi effettuate al più $2|E|$ ricerche di cammini di capacità almeno δ. Ognuna di tali ricerche può essere effettuata per mezzo di una visita in ampiezza di N_f, e quindi in tempo $O(|V| + |E|)$.

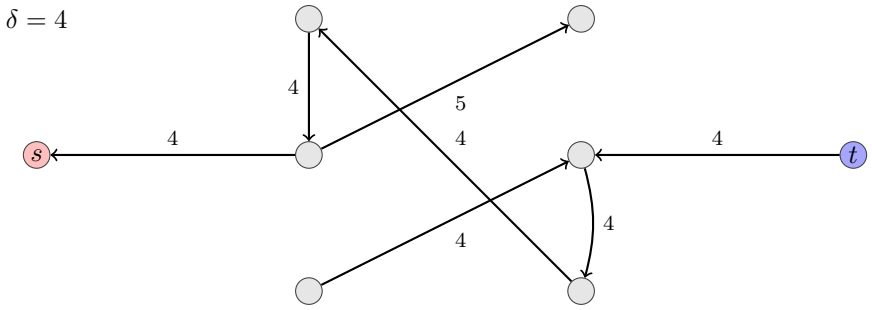
Il numero di valori δ considerati è al massimo $O(\log C)$, per cui la complessità totale risulta in effetti $O((|V| + |E|)|E| \log C)$ e, assumendo che la rete sia connessa, $O(|E|^2 \log C)$. □

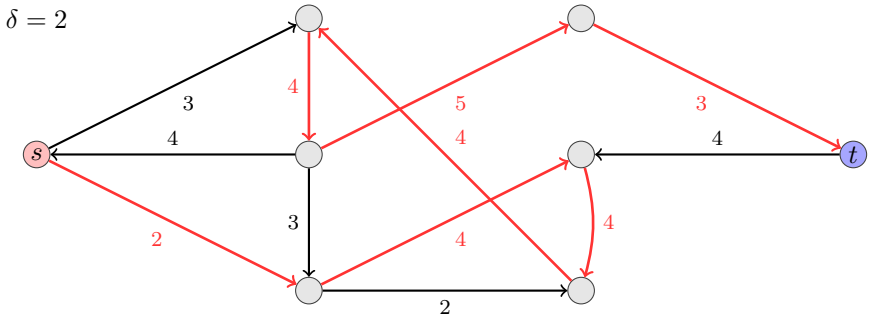
Di seguito, viene illustrato l'effetto sulla rete di Figura 1.4 dell'esecuzione dell'algoritmo basato sui cammini δ-aumentanti. In rosso è mostrato, di volta in volta, il cammino δ-aumentante selezionato, sulla rete ottenuta eliminando tutti gli archi di capacità inferiore a δ.

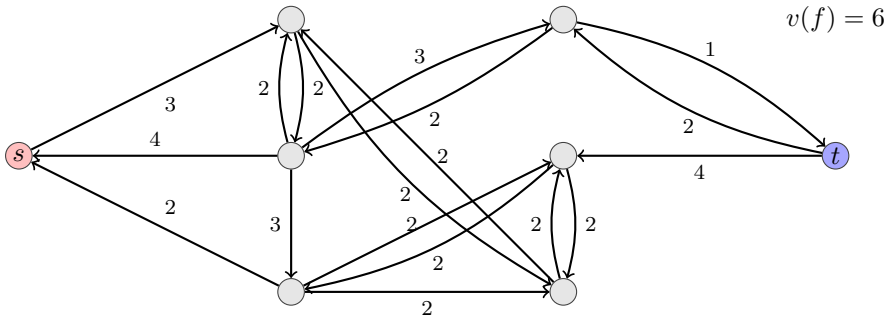


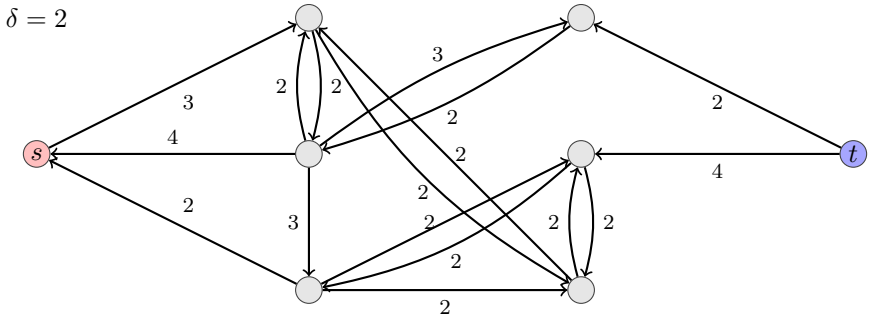


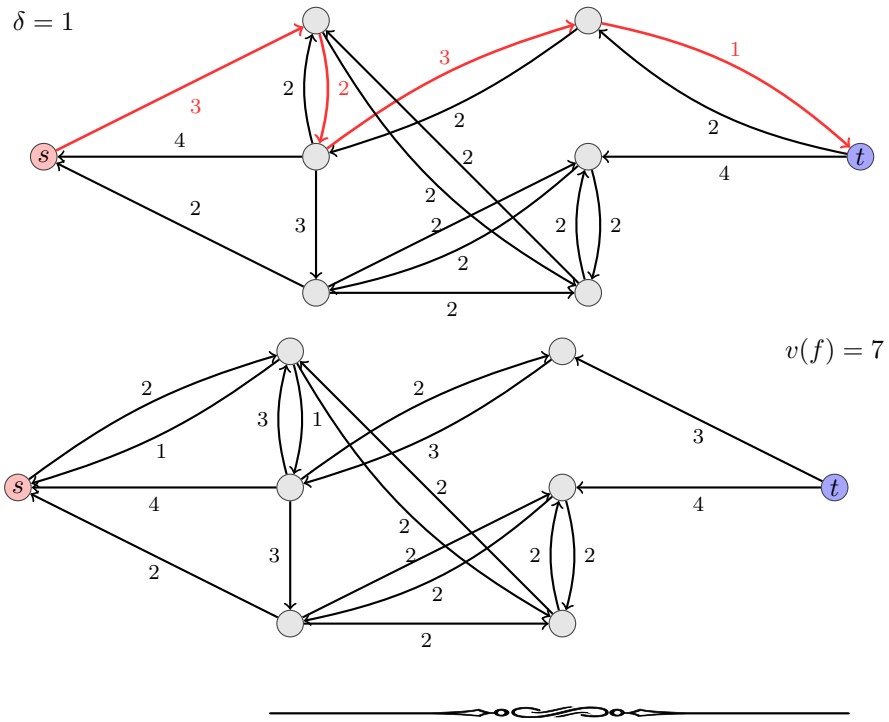

$v(f) = 4$
s
t
1
3
4
5
3
4
4
4
3
4
2
4
1
4
2


$\delta = 4$
s
t
4
5
4
4
4
4
4


$\delta = 2$
s
t
4
3
5
3
4
4
4
3
2
4
4
2


$v(f) = 6$
s
t
3
2
2
1
3
4
2
2
2
4
2
3
2
2
2
2
2
2


$\delta = 2$
s
t
3
2
2
2
3
4
2
2
4
2
3
2
2
2
2
2



Cammino di lunghezza minima

Si può osservare che sia la complessità sia dell'Algoritmo 1.3 che quella dell'Algoritmo 1.6 dipendono dai valori numerici (le capacità degli archi) in input.

Possiamo però eliminare questa dipendenza, ottenendo un algoritmo fortemente polinomiale (strongly polynomial)[1] attraverso una semplice variante dell'algoritmo di Ford e Fulkerson (1.2), detta Algoritmo di Edmonds e Karp, in cui ad ogni iterazione, il cammino aumentante in N_f è ricercato mediante una visita in ampiezza. Quindi, l'algoritmo, riportato come Algoritmo 1.7, seleziona, ad ogni iterazione, il cammino aumentante che attraversa il minimo numero di archi.

Come si può vedere, l'algoritmo ha la struttura dell'Algoritmo 1.2, così come dell'Algoritmo 1.3: l'unica differenza consiste nelle modalità di scelta del prossimo cammino aumentante da considerare.

Algorithm 1.7: Algoritmo di cammino aumentante di lunghezza minima

Input: $G = (V, E)$, $s, t \in V$, $c : E \mapsto \mathbb{R}^+$
Output: Flusso $f : E \mapsto \mathbb{R}^+$ da s a t

```
1 foreach e ∈ E do f(e) ← 0;
2 Deriva N_f;
3 while esiste un cammino aumentante da s a t in N_f do
4     Prendi il cammino aumentante P tale che |P| è minima;
5     foreach (u, v) ∈ P do
6         if (u, v) ∈ E then f(e) ← f(e) + min_{e∈P} c_f(e);
7         else f(e) ← f(e) − min_{e∈P} c_f(e)
8     Aggiorna N_f
9 return f
```

[1] Un algoritmo strongly polynomial ha complessità che non dipende dai valori numerici in input, assumendo che una operazione aritmetica richieda tempo costante.

Essendo una specifica implementazione dell'algoritmo di Ford e Fulkerson, sappiamo che l'Algoritmo 1.7 fornisce il flusso massimo in N. Per quanto riguarda la sua complessità computazionale, vale il seguente teorema.

Teorema 1.12. La lunghezza del cammino minimo da s a t in N_f ha un andamento monotono non decrescente al susseguirsi delle iterazioni. Inoltre, non si possono avere più di $|E|$ iterazioni (necessariamente consecutive) in cui il valore di tale grandezza rimane immutato.

Dimostrazione. Assumiamo che, dopo T iterazioni, il cammino minimo nella rete residua N_f abbia lunghezza l. Effettuando una visita in ampiezza a partire da s, otteniamo una partizione di $V - \{s\}$ in l sottoinsiemi $V_1, V_2, \ldots, V_l$, dove V_i è l'insieme dei nodi a distanza i da s.

Si può osservare che, per le proprietà della visita in ampiezza, per ogni arco (u, v), se $u \in V_i$ e $V \in V_j$, allora necessariamente $j \leq i + 1$: chiamiamo arco forward un arco tale che $j = i + 1$, un arco quindi che fa incrementare la distanza da s.

Un cammino minimo da s a t deve evidentemente essere composto da soli archi forward (Figura 1.18) o, in modo equivalente, un cammino in cui compare un arco non forward non è minimo.

Nel corso del $T + 1$-esima iterazione un cammino di lunghezza l viene selezionato in N_f, e il flusso su tutti i suoi archi viene incrementato il massimo possibile, vale a dire della capacità residua minima lungo il cammino. Di conseguenza almeno un arco nel cammino viene ad avere capacità residua nulla e quindi non compare nella rete residua risultante. Inoltre, per definizione di N_f, per ogni arco (u, v) del cammino, che ha capacità residua $c(u, v) - f(u, v)$ esiste l'arco opposto (v, u), con capacità residua $f(u, v)$, se $f(u, v) > 0$.

Per ogni arco nella nuova rete residua continua ad essere verificata la proprietà di incrementare la distanza da s di al più 1, in quanto i nuovi archi che sono eventualmente stati introdotti sono necessariamente archi opposti a quelli del cammino: di conseguenza, la distanza da s a t non è certamente diminuita.

Osserviamo ora che, nel caso in cui la distanza sia immutata, il nuovo cammino minimo avrà la stessa lunghezza del cammino minimo precedente, per cui userà soltanto archi forward già presenti nella rete residua precedente (Figura 1.19).

Quindi, per tutte le iterazioni successive alla T-esima per le quali la distanza da s a t rimane immutata e pari ad l, esiste un cammino aumentante di lunghezza minima che era già presente al momento dell'iterazione T.

Dato che ad ogni iterazione esiste almeno un arco che ha assegnato un flusso che satura la sua capacità, e che quindi non può più far parte di cammini di lunghezza l, ne deriva che il numero di iterazioni possibili su reti residue di distanza l da s a t è al più pari a $|E|$. □

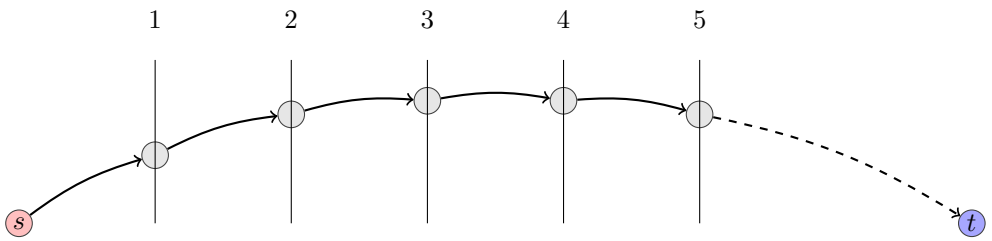


Figura 1.18: Cammino di lunghezza minima

La distanza da s a t è al più $|V| - 1$ quindi, per il Teorema 1.12, l'Algoritmo 1.7 esegue al più $|E|(|V| - 1)$ iterazioni, nell'ambito di ognuna delle quali viene eseguita una visita in ampiezza del grafo (in tempo $O(|V| + |E|)$). Di conseguenza, la complessità totale è $O(|E||V|(|E| + |V|))$ e quindi $O(|E|^2|V|)$ assumendo nuovamente che la rete sia connessa.

Di seguito, viene illustrato l'effetto sulla rete di Figura 1.4 dell'esecuzione dell'algoritmo basato sulla ricerca del cammino di lunghezza minima. In rosso è mostrato, di volta in volta, l'albero BFS individuato.

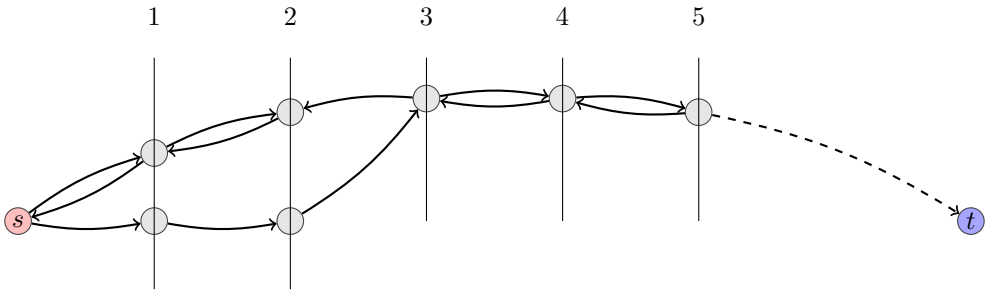
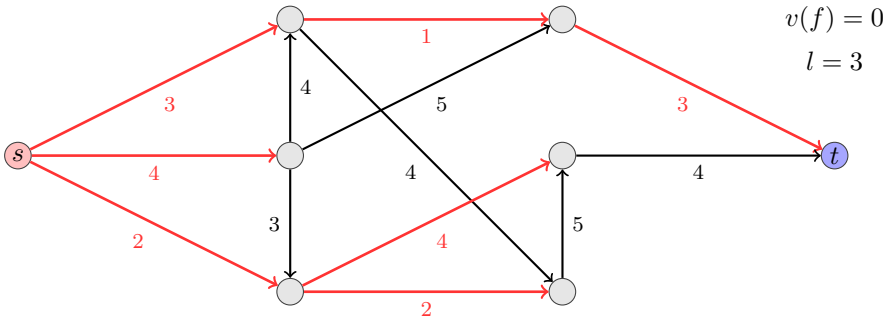
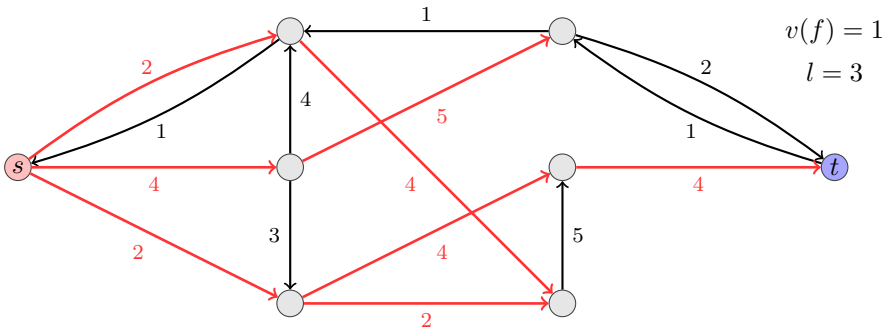
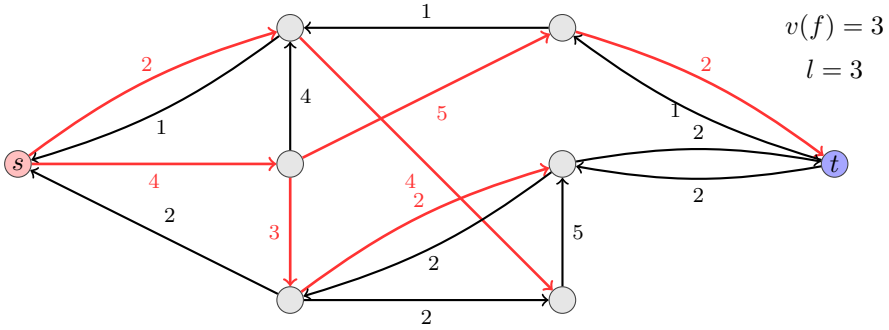
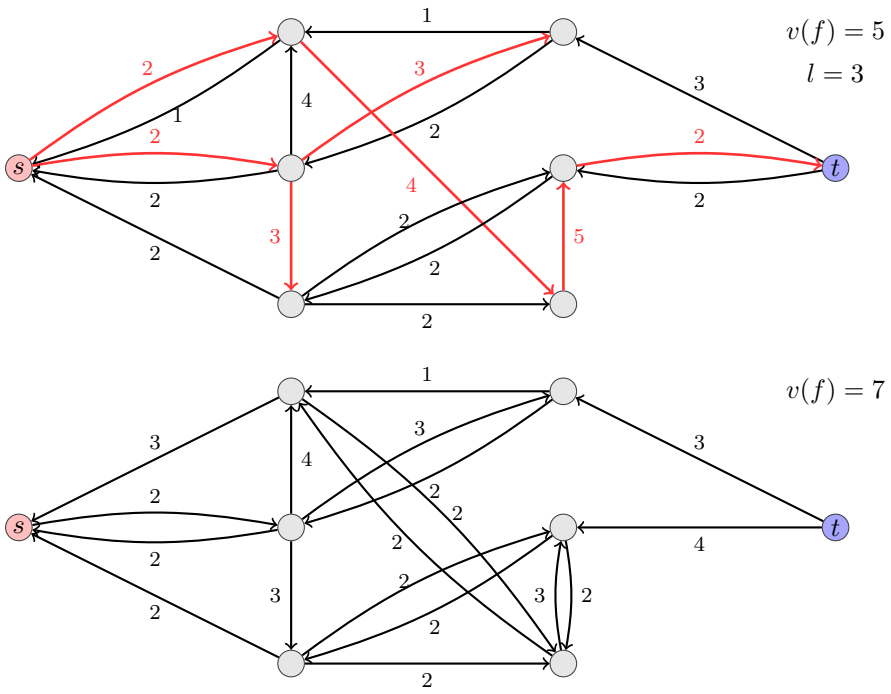


Figura 1.19: Caso in cui l'eliminazione del cammino minimo non aumenta la distanza









Applicazioni del massimo flusso

Car sharing

Supponiamo che n studenti decidano, per un periodo di m giorni, di utilizzare a turno le proprie automobili per andare a lezione. Supponiamo anche ogni auto sia sufficientemente capiente da poter trasportare tutti gli studenti.

Non tutti gli studenti devono andare a lezione negli stessi giorni: in altri termini, esiste una funzione booleana che associa ad una coppia studente-giorno il valore true se lo studente deve andare a lezione quel giorno e il valore false altrimenti.

Un esempio molto ridotto di tale problema, con $n = 4$ e $m = 5$ è riportato nella tabella sottostante, in cui per ogni giorno da Lunedì a Venerdì viene riportato se lo studente i-esimo (assumiamo $i = 1, \ldots, 4$) deve andare a lezione.

	Lu	Ma	Me	Gi	Ve
1	X	X	X		
2		X		X	X
3	X				X
4	X		X	X	

Vogliamo trovare una assegnazione ``equa'' dell'utilizzo delle automobili, nel senso intuitivo che chi deve andare più spesso a lezione dovrà anche mettere più spesso a disposizione la propria automobile.

Possiamo assegnare ad ogni studente un valore che misura la sua quota di utilizzo del car sharing nel modo seguente:

1. per ogni giorno, se k sono i viaggiatori, ognuno di essi riceve un valore pari a $1/k$
2. il valore totale T_i assegnato ad uno studente è pari alla somma dei valori ricevuti ogni gionro

La tabella seguente mostra i valori nell'esempio considerato, l'ultima colonna specifica la quota di utilizzo per ogni viaggiatore.

	Lu	Ma	Me	Gi	Ve	T
1	$1/3$	$1/2$	$1/2$			$4/3$
2		$1/2$		$1/2$	$1/2$	$3/2$
3	$1/3$				$1/2$	$5/6$
4	$1/3$		$1/2$	$1/2$		$4/3$

L'assegnazione che cerchiamo dovrà far s√¨ che il viaggiatore i non usi la propria automobile per più di $\lceil T_i \rceil$ giorni, sugli m complessivi.

Modelliamo il problema sotto forma di flusso su rete nel modo seguente:

- per l'insieme dei nodi si ha $V = \{s, r, t\} \cup \{v_i, i = 1, \ldots, n\} \cup \{d_j, j = 1, \ldots, m\}$
- per l'insieme degli archi si ha che E include
  1. (s, r) di capacità m
  2. per $i = 1, \ldots, m$, (r, t_i) di capacità $\lceil T_i \rceil$
  3. per $i = 1, \ldots, m$, $j = 1, \ldots, n$ (t_i, d_j) di capacità 1
  4. per $j = 1, \ldots, n$, (d_j, t) di capacità 1

La rete relativa all'esempio considerato è riportata in Figura 1.20. Gli archi senza valori associati hanno capacità 1.

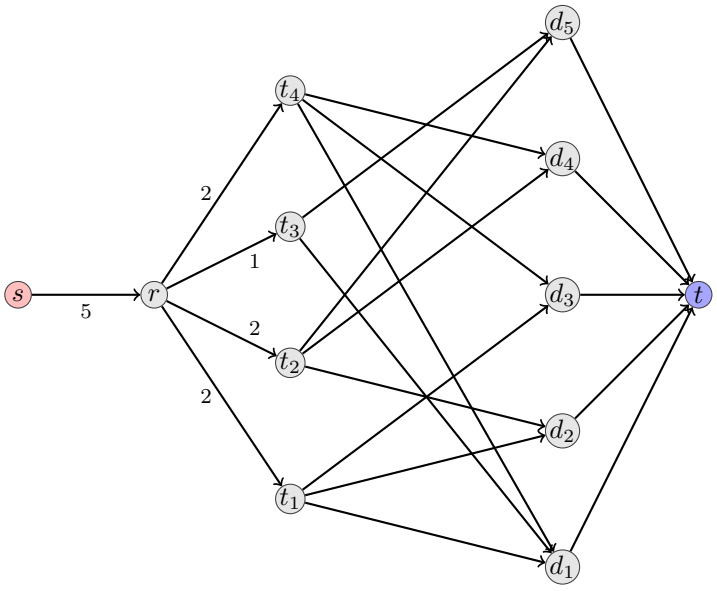


Figura 1.20: Rete per l'esempio di car sharing

Dato che le capacità degli archi della rete sono intere, il flusso massimo sarà intero. Inoltre, per ogni flusso ammissibile:

- per ogni giorno, dato che in uscita da d_i c'è capacità totale 1, al più un arco da un qualche viaggiatore t_j avrà flusso non negativo; tale flusso sarà al più 1 (la capacità di (t_j, d_i)) e quindi, per il flusso massimo, esattamente 1. Quindi, nella assegnazione corrispondente al flusso massimo, per ogni giorno c'è al più un viaggiatore associato (quello che utilizza la sua auto)
- per ogni viaggiatore t_i, si potranno avere, nel flusso massimo, al più tanti archi uscenti con flusso 1 quanta è la capacità dell'arco (d, t_i). Quindi, nella assegnazione corrispondente al flusso massimo, ogni viaggiatore utilizza la sua auto per un numero di giorni non superiore al sua quota di utilizzo del car sharing.

In definitiva, il flusso ottimo della rete fornisce una assegnazione di guidatori a giorni che soddisfa in parte le condizioni poste (un viaggiatore non guida per più della sua quota, per ogni giorno non ci sono più di un guidatore): dobbiamo però ancora mostrare che per ogni giorno c'è esattamente un guidatore assegnato. Ciò è equivalente a verificare che $v(f^*) = m$: dal Teorema 1.4, $v(f^*)$ è pari alla capacità del taglio minimo che separa s da t ed è facile rendersi conto che i tagli minimi nella rete, con capacità m, sono quello che taglia l'arco (s, r) e quello che taglia tutti gli archi (d_i, t).

Flusso ammissibile

In questo problema, consideriamo:

1. un grafo $G = (N, E)$ con archi di capacità $c : E \mapsto \mathbb{Z}$
2. una funzione $b : N \mapsto \mathbb{Z}$, che possiamo interpretare come offerta ($b(i) > 0$) o domanda ($b(i) < 0$) di una determinata merce, e per la quale assumiamo $\sum_{v \in N} b(v) = 0$, l'equilibrio complessivo quindi tra domanda e offerta.

Ci chiediamo se esiste una modalità di trasferimento della merce attraverso gli archi del grafo che soddisfi per ogni nodo il limite posto dalla sua capacità e inoltre faccia s√¨ che avvenga un trasferimento perfetto della merce, che soddisfi tutta la domanda. In termini più precisi, ci chiediamo se esiste un flusso f tale che

- per ogni nodo $v \in N$,

$$\sum_{e \in \mathcal{O}(v)} f(e) - \sum_{e \in \mathcal{I}(v)} f(e) = b(v)$$

- per ogni $e \in E$, $0 \leq f(e) \leq c(e)$

In Figura 1.4 viene mostrata una semplice istanza di esempio di questo problema: i valori $b(v)$ sono mostrati all'interno dei nodi corrispondenti. Si noti che domanda e offerta complessive sono uguali in modulo e pari a 9,

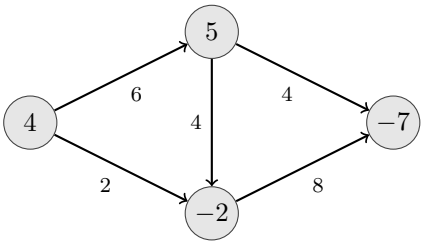


Figura 1.21: Esempio di istanza di feasible flow

Il problema può essere risolto in modo semplice attraverso la ricerca di un flusso massimo su una rete ottenuta da G introducendo i due nodi s e t e, per ogni nodo v, se $b(v) > 0$ inserendo un arco (s, v) con $c(s, v) = b(v)$, se $b(v) < 0$ inserendo un arco (v, t) con $c(v, t) = -b(v)$. La rete derivata da quella in Figura 1.4 è mostrata in Figura 1.22.

Possiamo osservare che:

- se esiste un flusso ammissibile f sul grafo, allora lo stesso flusso esteso ponendo $f(s, v) = b(v)$ per gli archi da s e $f(v, t) = -b(v)$ per gli archi verso t, è un flusso massimo, in quanto satura il taglio composto dal solo nodo s (e anche quello comprendente tutti i nodi eccetto t).
- se il flusso massimo nella rete satura tutti i nodi nel taglio comprendente il solo nodo s (e necessariamente anche quello comprendente tutti i nodi eccetto t), lo stesso flusso, applicato ai soli nodi in N soddisfa le condizioni di flusso ammissibile.

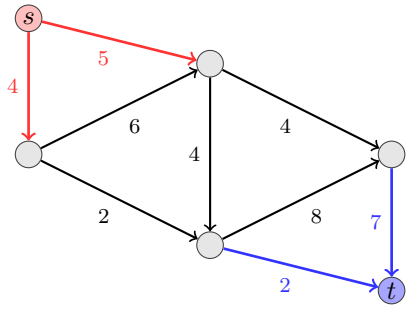


Figura 1.22: Istanza di max flow derivata

Quindi, per determinare se esiste un flusso ammissibile nel grafo G, è sufficiente verificare se il flusso massimo nella rete derivata satura il taglio composto dal solo nodo s: equivalentemente, potremmo dire che esiste un flusso ammissibile in G se e solo se il taglio composto dal solo nodo s è un taglio minimo che separa s da t. È facile verificare che il massimo flusso in Figura 1.22 ha valore pari a 9, e quindi satura il taglio, dal che consegue che nell'esempio in Figura 1.21 esiste un flusso ammissibile.

Al contrario, l'istanza in Figura 1.23 non ammette un flusso ammissibile in quanto la rete derivata, mostrata in Figura 1.21, ha valore del massimo flusso minore di 9, mentre esiste un taglio, mostrato in figura, che separa s da t e ha capacità pari a 8.

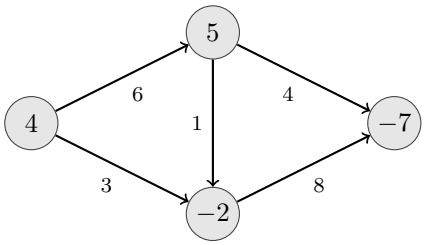


Figura 1.23: Esempio di istanza negativa di feasible flow

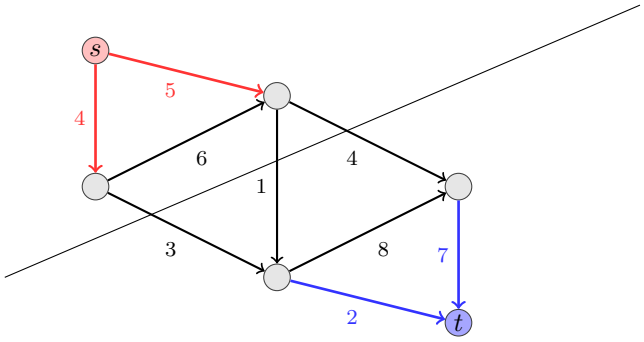


Figura 1.24: Istanza di max flow derivata da istanza negativa

Il problema dei rappresentanti

Una città ha r residenti $R_1, \ldots, R_r$, q associazioni $C_1, \ldots, C_q$ e p partiti politici $P_1, \ldots, P_p$. Ogni residente è membro di almeno un'associazione e di esattamente un partito. Ogni associazione deve nominare uno dei suoi membri come proprio rappresentante nel consiglio della città, in modo che, per ogni partito P_k, il numero di rappresentanti non sia superiore a un valore predefinito $u_k > 0$. Esiste una scelta dei rappresentanti possibile?

Data una istanza del problema, è possibile costruire una rete in cui $N = \{R_1, \ldots, R_r, C_1, \ldots, C_q, P_1, \ldots, P_p, s, t\}$ e l'insieme degli archi è definito come segue

- $(s, C_k) \in E$ per $k = 1, \ldots, q$, con $c = 1$
- $(C_i, R_j) \in E$ se R_j appartiene a C_i, con $c = 1$
- $(R_i, P_j) \in E$ se R_i appartiene a P_j, con $c = 1$
- $(P_k, t) \in E$ per $k = 1, \ldots, p$, con $c = u_k$

Ad esempio, consideriamo il caso in cui $r = 7$, $q = 4$, $p = 3$ e valgono le seguenti appartenenze, ad associazioni e partiti:

- $C_1 = \{R_1, R_2\}$, $C_2 = \{R_2, R_3, R_4\}$, $C_3 = \{R_4, R_5\}$, $C_4 = \{R_4, R_5, R_6, R_7\}$
- $P_1 = \{R_1, R_2\}$, $P_2 = \{R_3, R_4\}$, $P_3 = \{R_5, R_6, R_7\}$

e assumiamo $u_1 = 1$, $u_2 = 2$, $u_3 = 2$. Ne deriva la rete in Figura 1.25, in cui archi non etichettati vanno considerati di capacità unitaria.

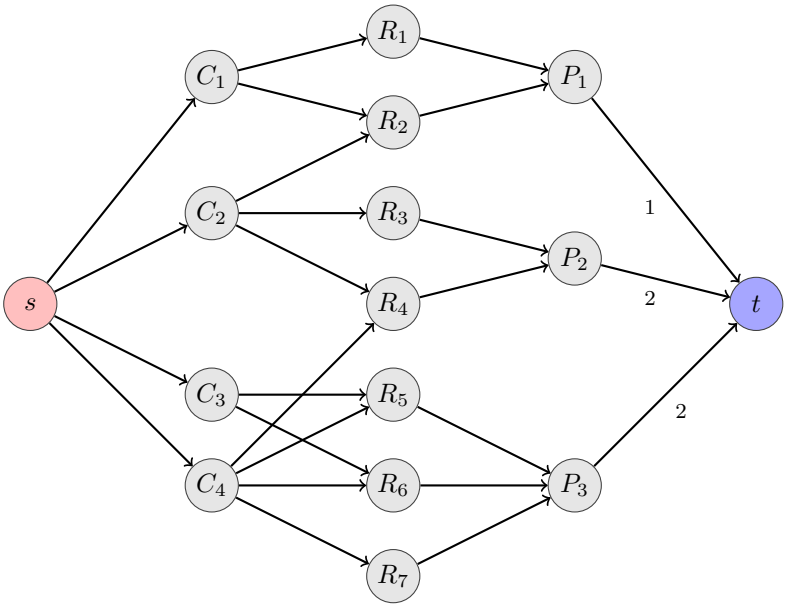


Figura 1.25: Rete derivata da istanza del problema dei rappresentanti

Possiamo mostrare che una scelta dei rappresentanti che soddisfi i vincoli è possibile se e solo se nella rete esiste un flusso di valore pari a q (si osservi che non può esistere un flusso di valore maggiore, per il taglio $\{s\}$, avente capacità q).

- se esiste una scelta, allora per ogni C_i esiste esattamente un residente R_j appartenente ad essa scelto: poniamo $f = 1$ per gli archi (s, C_i), (C_i, R_j), (R_j, P_k), dove P_k è il partito di appartenenza di R_j. Si noti che, in tal modo, gli archi in uscita da s hanno tutti flusso pari a 1, per ogni C_i esiste uno ed un solo arco uscente di flusso pari a 1, un R_j ha flusso entrante e uscente pari a 1 se e solo se è un residente scelto, il flusso entrante in P_k è pari al numero di residenti scelti appartenenti a quel partito, che per costruzione è inferiore alla capacità dell'arco (P_k, t). Ne deriva che il flusso è ammissibile e di valore pari a q.
- se esiste un flusso di valore pari a q, allora per ogni C_i c'è uno e un solo arco uscente (C_i, R_j) di flusso pari a 1. R_j è allora un residente scelto. Per il bilanciamento del flusso a R_j, l'unico arco uscente (R_j, P_k) ha

flusso pari a 1: quindi, per ogni P_k, il flusso entrante e uscente è pari al numero di residenti scelti appartenenti al k-esimo partito. Dato che il flusso sull'arco (P_k, t) deve essere non superiore alla sua capacità, ne deriva che il vincolo sul massimo numero di scelti da ogni partito è verificato. Quindi, in definitiva, esiste al meno un residente scelto per associazione (per l'ipotesi che il flusso sia pari a q), esiste al più un residente scelto per associazione (per il vincolo sulla capacità degli archi uscenti da s), ogni partito P_k è rappresentato da al più u_k appartenenti (dal vincolo sulla capacità degli archi entranti in t).

Problemi di matching

Un accoppiamento (o matching) in un grafo è un insieme di coppie di nodi tale che ogni nodo compare in al più una coppia. Più formalmente, dato un grafo non orientato $G = (V, E)$ un accoppiamento è un sottoinsieme degli archi $M \subseteq E$ tale che ogni nodo appare in al più un elemento di M. Se ogni nodo compare in esattamente un arco di M (e quindi tutti i nodi sono accoppiati) il matching è detto perfetto.

Il problema del massimo accoppiamento chiede, dato un grafo $G = (V, E)$, di trovare un matching di dimensione massima. In questa sede, consideriamo questo problema per una classe di grafi particolari: i grafi bipartiti.

Un grafo non orientato $G = (V, E)$ è detto bipartito se esiste una partizione (U, W) di V, per cui quindi $V = U \cup W$ e $U \cap W2 = \emptyset$, tale che per ogni $(u, v) \in E$ si ha $u \in U$ e $v \in W$ (o viceversa). Un esempio di grafo di questo tipo è dato in Figura 1.26.

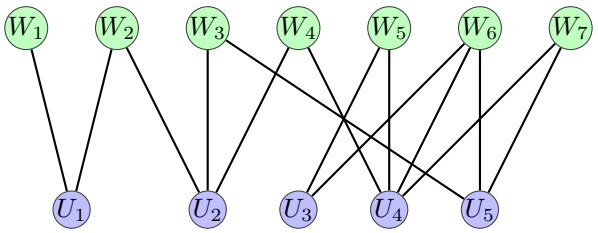


Figura 1.26: Esempio di grafo bipartito

Nel calcolo del matching massimo su un grafo bipartito vogliamo quindi selezionare il più grande insieme di coppie (u_i, w_i) tali che $u_i \in U, w_i \in W$ e $u_i \neq u_j$, $w_i \neq w_j$, per $i \neq j$. Un esempio di matching massimo sul grafo bipartito precedente è mostrato in Figura 1.27.

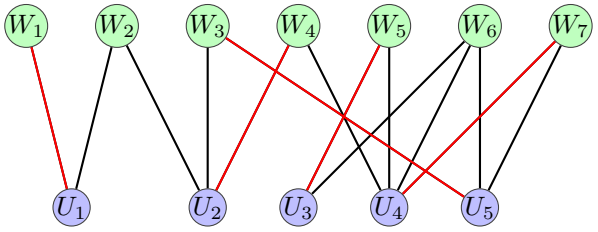


Figura 1.27: Esempio di matching massimo

Si noti che il problema del matching perfetto su un grafo bipartito può essere facilmente risolto se si sa risolvere il problema del matching massimo, verificando che $|U| = |W|$ e che il matching massimo includa $|U|$ elementi.

Il problema del matching massimo in un grafo bipartito può essere trovato mediante riduzione al problema del massimo flusso. Il concetto di riduzione è molto importante nell'ambito della teoria degli algoritmi e della teoria della complessità: sostanzialmente, esso ci descrive la possibilità di risolvere un problema P_1 se si sa risolvere un diverso problema P_2.

Una riduzione da P_1 a P_2 opera nel modo seguente. Assumiamo di avere una istanza x di P_1, allora:

1. x è trasformata in una opportuna istanza y di P_2

2. y è risolta per mezzo di un algoritmo per P_2, fornendo una soluzione $s(y)$
3. la soluzione $s(x)$ di x è derivata da $s(y)$

Nel caso che si sta considerando, un grafo bipartito, istanza del problema del matching massimo in un grafo bipartito, viene trasformato in una rete, istanza del problema del massimo flusso nel modo seguente: il grafo della rete N ha insieme dei nodi $V = U \cup W \cup \{s, t\}$ e insieme degli archi (orientati) $E = \{(u, w), u \in U, w \in W\} \cup \{(s, u), u \in U\} \cup \{(w, t), w \in W\}$. La capacità degli archi di N è posta uniformemente pari a 1.

La rete derivata dal grafo di esempio è data in Figura 1.28

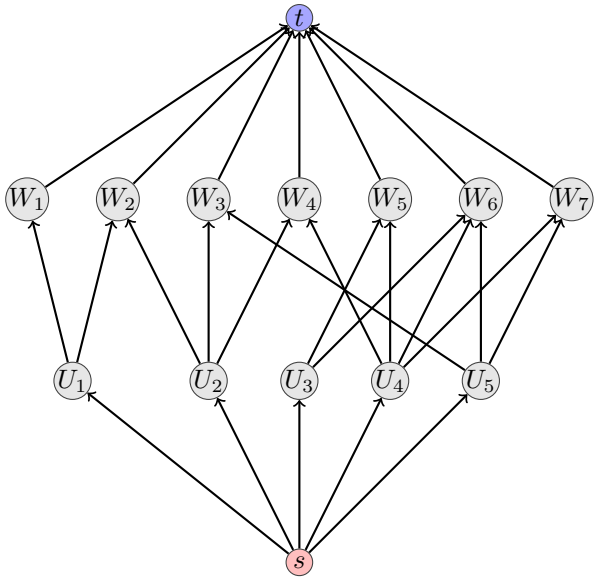


Figura 1.28: Istanza di maxflow derivata

L'applicazione di un algoritmo di maxflow sulla rete N fornirà un flusso massimo nel quale, necessariamente dato che le capacità sono tutte unitarie, il flusso su ogni arco è 0 o 1.

Mostriamo che il matching massimo è dato dall'insieme delle coppie $(u, w), u \in U, w \in W$ tali che $f(u, v) = 1$. A tal fine, mostriamo (1) che l'algoritmo fornisce un matching e (2) che il matching fornito è ottimo.

1. L'algoritmo fornisce un matching di dimensione pari al flusso massimo in N. Dato il flusso massimo f trovato, infatti, si ha che per ogni nodo $u \in U$

$$\sum_{(u,w)\in E} f(u, w) = f(s, u) \leq 1$$

quindi, al più uno degli archi tra u e i nodi di W ha flusso pari a 1 (ed è quindi incluso nel matching). Allo stesso modo, per ogni nodo $w \in W$

$$\sum_{(u,w)\in E} f(u, w) = f(w, t) \leq 1$$

quindi, al più uno degli archi tra w e i nodi di U ha flusso pari a 1 (ed è quindi incluso nel matching).

2. Non esistono matching più grandi di quello corrispondente al massimo flusso in N. Consideriamo infatti il matching massimo M^* e definiamo a partire da esso un flusso in N ponendo, per ogni $(u, w) \in M^*$, $f(s, u) = f(u, w) = f(w, t) = 1$ e ponendo a zero il flusso su tutti gli archi rimanenti. Il flusso è chiaramente ammissibile, in quanto soddisfa i vincoli sui nodi e sulle capacità, e ha valore pari al numero di archi nel matching, e quindi a M^*. Quindi, esiste un flusso ammissibile corrispondente al matching massimo, per cui non esistono matching più grandi del valore di tale flusso e, a maggior ragione, del valore del flusso massimo.

Per quanto riguarda la complessità dell'algoritmo, la costruzione di N richiede tempo $O(|E|)$, così come la derivazione del matching massimo dal massimo flusso. A tali costi va aggiunta la complessità di calcolo del maxflow su N: dato che il flusso massimo su N è certamente limitato superiormente da $\max(|U|, |W|)$, l'algoritmo di Ford e Fulkerson ha complessità $O(|E| \cdot \max(|U|, |W|)) = O(|E||V|)$.

Arrotondamento di valori in una matrice

Sia data una matrice $D^{p \times q}$ di reali d_{ij}, e siano specificati $p + q$ valori, corrispondenti a somme α_i sulle righe e β_j sulle colonne.

Vogliamo arrotondare ognuno dei d_{ij}, degli α_i e dei β_j ad un valore intero (quello immediatamente minore o quello immediatamente maggiore) in modo consistente, facendo sì che le somme sulle righe e sulle colonne rimangano verificate anche con i nuovi valori.

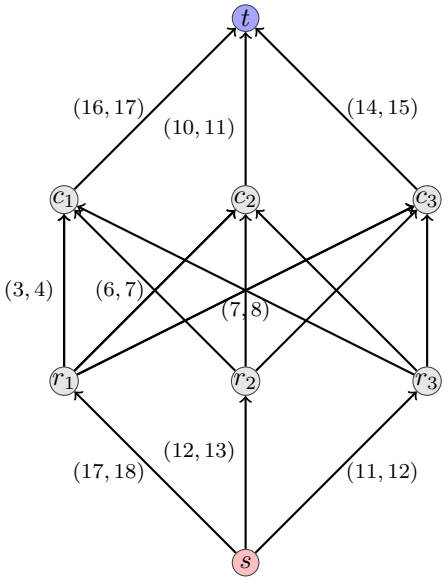
Possiamo rappresentare questo problema come un problema di massimo flusso in cui sono presenti dei vincoli sia superiori (capacità degli archi) che inferiori (minimo flusso) sui flussi negli archi. È possibile in generale (anche se non lo verifichiamo in questa sede) risolvere un problema con tali vincoli aggiuntivi riducendolo ad un problema di massimo flusso con soli vincoli superiori su una rete derivata in modo opportuno.

Data la matrice D, la rete corrispondente ha $p + q$ nodi $r_i, i = 1, \ldots, p$ e $c_j, j = 1, \ldots, q$, corrispondenti alle righe e alle colonne. Per quanto riguarda gli archi, per ogni coppia r_i, c_j esiste l'arco corrispondente, con capacità $c(r_i, c_j) = \lceil d_{ij} \rceil$ e flusso minimo $m(r_i, c_j) = \lfloor d_{ij} \rfloor$, inoltre per ogni r_i esiste l'arco (s, r_i) con capacità $c(s, r_i) = \lceil \alpha_i \rceil$ e flusso minimo $m(r_i, c_j) = \lfloor \alpha_i \rfloor$ e per ogni colonna c_j esiste l'arco (c_i, t) con capacità $c(c_j, t) = \lceil \beta_j \rceil$ e flusso minimo $m(c_j, t) = \lfloor \beta_j \rfloor$.

Data ad esempio la matrice

3.1	6.8	7.3	17.2
9.6	2.4	0.7	12.7
3.6	1.2	6.5	11.3
16.3	10.4	14.5	

La rete corrispondente è la seguente, in cui, per semplicità, sono indicati i valori associati ai soli archi da s e r_1 e verso t.



Non è difficile verificare che un flusso ammissibile nella rete (che necessariamente è intero) corrisponde a un arrotondamento consistente per D.

Selezione di progetti

Consideriamo un insieme P di progetti da intraprendere. Il progetto i, se attivato garantisce un profitto (differenza tra ritorno e investimento) associato p_i che può essere positivo, nullo o negativo. Inoltre, su P è definita una relazione di dipendenza, che specifica, per ogni progetto i, l'insieme π_i di altri progetti che devono essere necessariamente attivati per attivare i. Questa relazione può essere modellata per mezzo di un grafo orientato aciclico $G = (P, E)$.

Un insieme di progetti $A \subseteq P$ è ammissibile se per ogni progetto $i \in A$ si ha che ogni progetto in π_i è anch'esso incluso in A. Il profitto di A è dato da $\sum_{i \in A} p_i$.

Il problema che consideriamo è quello di determinare un insieme ammissibile di progetti di profitto massimo.

Possiamo costruire una istanza di maxflow nel modo seguente:

- l'insieme dei nodi è $P \cup \{s, t\}$
- l'insieme degli archi corrisponde alle relazioni di dipendenza tra progetti: in particolare, esiste l'arco (i, j) se i dipende da j. Inoltre, per ogni progetto i con $p_i > 0$ aggiungiamo un arco (s, i) e, per ogni progetto j con $p_j < 0$ aggiungiamo un arco (j, t)
- le capacità degli archi sono definite pari a p_i per gli archi (s, i) e pari a $-p_j$ per gli archi (j, t); tutti gli altri archi (quelli di E) hanno capacità $C + 1$, dove

  $$C = \sum_{i \in P; p_i > 0} p_i$$

  è la capacità totale degli archi da s che, necessariamente, è non superiore al taglio minimo e quindi al flusso massimo nella rete. Di conseguenza ogni arco in E ha capacità maggiore del massimo flusso, per cui non pone un vincolo effettivo sul flusso che lo attraversa.

Mostriamo ora che il minimo taglio nella rete corrisponde all'insieme ottimo di progetti.

1. Sia A un insieme ammissibile di progetti e consideriamo il taglio tra $A' = A \cup \{s\}$ e $B' = (P - A) \cup \{t\}$. Dato che A è ammissibile, non esistono archi da A a $P - A$. Tutti gli archi da A' a B' sono o uscenti da s o entranti in t. Per i primi abbiamo capacità complessiva

   $$\sum_{i \notin A; p_i > 0} p_i = C - \sum_{i \in A; p_i > 0}$$

   mentre per i secondi abbiamo

   $$\sum_{i \in A; p_i < 0} -p_i$$

   La capacità del taglio è data dalla somma di questi due termini

   $$C - \sum_{i \in A; p_i > 0} + \sum_{i \in A; p_i < 0} -p_i = C - \sum_{i \in A} p_i$$

2. Inoltre, dato un taglio $A' = A \cup \{s\}$ di capacità non superiore a C, tale taglio non può essere attraversato da archi corrispondenti a dipendenze, in quanto ognuno di tali archi a capacità maggiore di C. Di conseguenza, l'insieme di progetti A è ammissibile.

3. Quindi, esiste una corrispondenza biunivoca i tagli di capacità al più C e gli insiemi ammissibili di progetti. Inoltre, il profitto di un insieme ammissibile di progetti A è pari a $C - c(A')$, dove $c(A')$ è la capacità del taglio $A' = A \cup \{s\}$. Quindi, l'insieme ottimo di progetti corrisponde al taglio di capacità minima.

Cammini disgiunti sugli archi

Dato un grafo orientato $G = (V, E)$ e due nodi $s, t \in V$, ci chiediamo quale sia il massimo numero di cammini da s a t che siano disgiunti sugli archi, intendendo con ciò che non esiste alcun arco che compare in più di un cammino. Si noti che due cammini disgiunti sugli archi possono però attraversare uno stesso nodo.

Questo problema può essere risolto facilmente considerando una istanza di maxflow, derivata da G, in cui la capacità degli archi è unitaria. Infatti il flusso massimo nella rete risultante sarà necessariamente decomponibile nell'unione di un insieme di cammini da s a t e in un insieme di cicli. Per l'ipotesi sulla capacità, il flusso su tali cicli e cammini è necessariamente unitario, dal che deriva che non esistono archi appartenenti a più di un cammino (o ciclo), per cui i cammini e i cicli in questione sono disgiunti sugli archi. A maggior ragione lo sono quindi i soli cammini: infine, dato che il flusso da s a t è massimo e ogni cammino contribuisce per 1, ne consegue che è massimo il numero di cammini.

Al contrario, ogni insieme massimo di cammini disgiunti sugli da s a t fornisce chiaramente un flusso ottimo sulla rete, costituito assegnando flusso unitario agli archi dei cammini.

Cammini disgiunti sui nodi

Dato un grafo orientato $G = (V, E)$ e due nodi $s, t \in V$, ci chiediamo quale sia il massimo numero di cammini da s a t che siano disgiunti sui nodi, intendendo con ciò che non esiste alcun nodo che compare in più di un cammino.

Questo problema può essere risolto applicando lo stesso approccio precedente, considerando però il grafo $G' = (V', E')$ derivato da G introducendo, per ogni nodo $v \in V$, due nodi $v_{in}, v_{out} \in V'$. Ogni arco $(w, u) \in E$ è sostituito da un arco $(w, v_{in}) \in E'$ e ogni arco $(v, w) \in E$ è sostituito da un arco $(v_{out}, w) \in E'$. Si noti che un cammino è disgiunto sui nodi in G se e solo se è lo è, sugli archi, in G'.

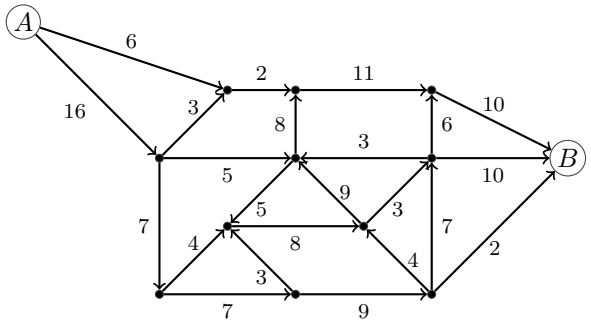
Problemi

1. Tra la città A e la città B esistono un certo numero di percorsi stradali possibili, mostrati in figura. I numeri associati indicano il massimo numero di veicoli per ora che il tratto di strada può sopportare.

   a) Determinare il numero massimo di veicoli che possono viaggiare in un'ora da A a B

   b) Proporre un tratto di strada il cui ampliamento aumenterebbe il valore precedente, precisando l'incremento ottenuto

   c) Come cambia il traffico sulle altre tratte, in corrispondenza all'ampliamento proposto?



2. Nella tabella sottostante è mostrato il massimo numero di passeggeri, in centinaia, trasportabili via treno tra varie città in un giorno (la matrice si assume simmetrica). Non si considerano i tempi di spostamento.

	BA	NA	RM	FI	PI	GE	BO	AN	VE	MI	TO
BA		10						5			
NA			20								
RM				25	10			5			
FI					15		30				
PI						25					
GE										20	15
BO								10	25	35	15
AN											
VE									30		
MI											35
TO											

Quante persone possono raggiungere in un giorno Milano da Bari? Quante persone in totale possono raggiungere Milano in un giorno, da qualunque altra città?

3. Siano (S, T) e (S', T') due tagli $s-t$ minimi in una rete. Mostrare che sia $(S \cap S', T \cup T')$ che $(S \cup S', T \cap T')$ sono anch'essi tagli $s - t$ minimi.

4. Un flusso è aciclico se il sottografo comprendente i soli archi con flusso positivo è aciclico.

   - Mostrare che per ogni flusso f esiste un flusso aciclico f' con $v(f') = v(f)$
   - Un path flow è una assegnazione di valori di flusso ai soli archi di un cammino $s - t$ (semplice, vale a dire senza cicli). Mostrare che ogni flusso aciclico può essere decomposto in $O(|E|)$ path flow.
   - Un cycle flow è una assegnazione di valori di flusso ai soli archi di un cammino. Mostrare che ogni flusso f con $v(f) = 0$ può essere decomposto in $O(|E|)$ cycle flow.

5. Consideriamo il caso in cui ogni arco della rete ha una domanda di flusso $d(e)$, invece di una capacità. Un flusso è allora ammissibile se $f(e) \geq d(e)$ per ogni arco. Vogliamo trovare un flusso ammissibile minimo.

   - Descrivere un algoritmo efficiente per calcolare un flusso ammissibile su una rete
   - Descrivere un algoritmo per la ricerca di un flusso minimo in una rete con domanda sugli archi, eventualmente utilizzando un sotto-programma in grado di calcolare il flusso massimo su una rete con capacità sugli archi. Risolvere il problema effettuando una sola invocazione del sotto-programma.
   - Dimostrare l'analogo del teorema maxflow-mincut per reti con domanda sugli archi.

6. Consideriamo una modifica dell'algoritmo di Ford-Fulkerson in cui, una volta individuato un cammino aumentante e il relativo flusso f, la capacità sugli archi del cammino viene decrementata di f (eliminando gli archi con capacità risultante pari a 0), senza mantenere la rete residua.

   - Mostrare che questo algoritmo non calcola il flusso massimo per tutte le possibili reti.
   - Mostrare che, nell'ipotesi in cui un oracolo indichi ad ogni iterazione il particolare cammino aumentante da considerare, l'algoritmo calcola il flusso massimo.

7. Dato un grafo $G = (V, E)$ e tre nodi $u, v, w \in V$, fornire un algoritmo che determina se esiste un cammino da u a v passante per w.

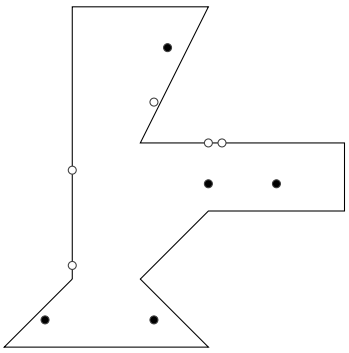
8. Si consideri il seguente problema. Un gruppo di famiglie $\{F_1, \ldots, F_n\}$ si incontra per una cena in un ristorante. Ogni famiglia F_i è composta da f_i persone. Il ristorante mette a disposizione un insieme di tavoli $\{T_1, \ldots, T_m\}$, con il tavolo T_j adatto per t_i commensali. Si vuole assegnare i posti a tavola in modo tale da far familiarizzare le famiglie, evitando quindi che due membri della stessa famiglia siedano allo stesso tavolo: ci chiediamo se esiste una possibile assegnazione che soddisfi questo vincolo. Individuare una modellazione di questo problema mediante maxflow.

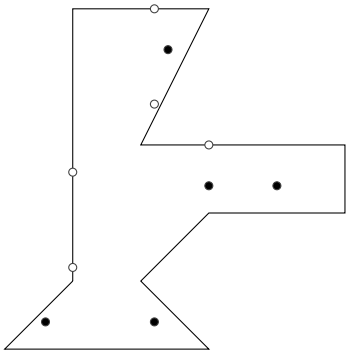
9. Si considerino le seguenti affermazioni, ricordando che un arco è saturato da un flusso se $f(e) = c(e)$:

- ``Dato un flusso f su una rete $G = (V, E)$ con funzione di capacità $c : E \mapsto \mathbb{N}$, se f è un flusso massimo allora satura tutti gli archi del tipo (s, v), uscenti da s"
- ``Dato un flusso f su una rete $G = (V, E)$ con funzione di capacità $c : E \mapsto \mathbb{N}$, se f satura tutti gli archi del tipo (s, v), uscenti da s allora è un flusso massimo"

Per ognuna delle due affermazioni stabilire se è vera (dandone una dimostrazione) o falsa (fornendo un controesempio).

10. Si consideri il problema seguente. Data una pianta di un appartamento con la posizione di n punti luce ed n interruttori, si vuole determinare se essa è ``ergonomica", nel senso che è possibile collegare ciascun interruttore a un punto luce in modo tale che ogni punto luce sia visibile dall'interruttore che lo comanda. Nella pianta a sinistra ciò è possibile, mentre non lo è in quella a destra.





Assumendo di avere un sottoprogramma che, dati due segmenti nel piano, determina in tempo $O(1)$ se si intersecano, descrivere un algoritmo che, data una pianta, determina se è ergonomica in tempo polinomiale in n ed m, numero delle pareti della pianta.

11. Un centro trasfusioni ha a disposizione d_A unità di sangue di gruppo A, d_B di gruppo B, d_0 di gruppo 0, d_{AB} di gruppo AB. Per le proprietà degli antigeni del sangue, persone di gruppo 0 possono ricevere soltanto sangue del medesimo gruppo, persone di gruppo A o B possono ricevere sangue del loro stesso gruppo e sangue di gruppo 0, persone di gruppo AB possono ricevere sangue di qualunque gruppo. Date le previsioni sulle esigenze di sangue per la prossima settimana, rappresentate dal numero n di pazienti e da due funzioni $g : \{1, \ldots, n\} \mapsto \{A, B, 0, AB\}$ (che determina il gruppo sanguigno di ogni paziente) e $u : \{1, \ldots, n\} \mapsto \mathbb{N}$ (che definisce il numero di unità necessarie ad ogni paziente), si vuole determinare se le unità a disposizione consentono di soddisfare le richieste. Si descriva un algoritmo che verifica tale condizione.

Capitolo 2

Randomizzazione e analisi ammortizzata

L'algoritmo di ordinamento per distribuzione quicksort segue il paradigma del divide et impera e opera nel modo seguente.

1. Decomposizione: se la sequenza ha almeno due elementi, scegli un elemento pivot e dividi la sequenza in due sottosequenze, dove la prima contiene elementi minori o uguali al pivot e la seconda contiene elementi maggiori o uguali di esso, di conseguenza, nessun elemento della prima sottosequenza è maggiore di un qualche elemento della seconda.
2. Ricorsione: ordina ricorsivamente le due sottosequenze.
3. Ricombinazione: concatena le due sottosequenze ordinate in un'unica sequenza ordinata.

L'algoritmo di quicksort è illustrato in maggior dettaglio come Algoritmo 2.1, le funzioni utilizzate al suo interno sono presentate come Algoritmi 2.2, 2.3.

Algorithm 2.1: Quicksort

Input: Array $a = [1, \ldots, n]$
Output: Array $a = [1, \ldots, n]$ ordinato in modo non decrescente

```
1 qsort(a,1,n);
2 return a
```

Algorithm 2.2: Funzione qsort

```
1 Function qsort(a,min,max);
2 if min < max then
3     seleziona pivot in [min,max], ad esempio pivot ← max;
4     r = distribute(a,min,max,pivot);
5     qsort(a,min,r − 1);
6     qsort(a,r + 1,max)
7 return
```

La complessità dell'algoritmo può essere espressa mediante relazione di ricorrenza. Sia $T(n)$ il tempo di esecuzione su un array di n elementi. Nel caso base $T(n) = O(1)$ per $n \leq 1$. Nel passo ricorsivo, possiamo osservare che l'elemento pivot ha rango $r + 1$ nell'array ordinato: ci sono $r - 1$ elementi minori del pivot (e quindi a sinistra) e $n - r$ elementi maggiori (a destra). Di conseguenza, $T(n) = T(r - 1) + T(n - r) + O(n)$, in quanto distribute richiede $O(n)$ tempo.

Algorithm 2.3: Funzione distribute

```
1  Function distribute(a,min,max,pivot);
2  if pivot ≠ max then Scambia(pivot,max);
3  i ← min;
4  j ← max − 1;
5  while i < j do
6      while (i ≤ j) and (a[i] ≥ a[max]) do i ← i + 1;
7      while (i ≤ j) and (a[j] ≤ a[max]) do j ← j − 1;
8      if i < j then Scambia(i,j)
9  if i ≠ max then Scambia(i,max);
10 return i
```

Il caso pessimo si presenta se il pivot è il minimo dell'array ($r = 1$) oppure il massimo ($r = n$): in entrambi i casi, la relazione diventa $T(n) = T(n-1) + T(0) + cn$. Se il caso pessimo si presenta ad ogni esecuzione di qsort, ad esempio se viene selezionato come pivot l'ultimo elemento dell'array e l'array stesso è già ordinato ne deriva che $T(n) = O(n^2)$. Non è difficile vedere che in questa situazione, l'analisi del costo risulta simile a quella dell'ordinamento per selezione o per inserimento. In particolare,

$$\begin{aligned} T(n) &= T(n-1) + cn + T(0) \\ &= T(n-2) + c(n-1) + cn + T(0) \\ &= T(n-3) + c(n-2) + c(n-1) + cn + T(0) \\ &= \cdots = c\sum_{k=2}^{n} k + T(1) + T(0) = \Theta(n^2) \end{aligned}$$

Se la distribuzione degli elementi dell'array è bilanciata ($r \simeq n/2$), la ricorsione riduce il problema a due istanze di dimensione metà, $T(n) = 2T(n/2) + cn$. Se questo avviene, possiamo dimostrare che la complessità è $\Theta(n \log n)$ sviluppando la relazione di ricorrenza nel modo seguente:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ &= 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn \\ &= 4\left(2T\left(\frac{n}{8}\right) + c\frac{n}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \\ &= 2^k T\left(\frac{n}{2^k}\right) + kcn \\ &= \cdots = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + cn\log_2 n \leq nT(1) + cn\log_2 n = \Theta(n\log n) \end{aligned}$$

Teorema 2.1. La complessità dell'algoritmo quicksort nel caso medio è $\Theta(n \log n)$, assumendo che le istanze si presentino in modo equiprobabile.

Dimostrazione. Per valutare la complessità nel caso medio, assumiamo per semplicità che gli elementi siano tutti diversi (la presenza di elementi uguali può essere facilmente gestita a parte), e che gli $n!$ possibili ordinamenti abbiano la stessa probabilità di manifestarsi. In conseguenza di ciò, ad ogni esecuzione di qsort la posizione del pivot all'interno dell'array ordinato è equiprobabile: si ha probabilità $\frac{1}{n}$ che il pivot sia il massimo, $\frac{1}{n}$ il secondo più grande e così via. Nel primo caso, la ricorrenza sarà $T(n) = T(n-1) + T(0) + cn$, nel secondo $T(n) = T(n-2) + T(1) + cn$, etc.

Il costo medio sarà dato dalla media di questi costi, ciascuno pesato dalla probabilità che la relativa configurazione si manifesti ($\frac{1}{n}$ in quanto assumiamo l'equiprobabilità). Quindi, indicando con $T(n)$ il costo medio, avremo

$$\begin{aligned} T(n) &= cn + \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)) \\ &= cn + \frac{2}{n}\sum_{i=0}^{n-1}T(i) \end{aligned}$$

in quanto $\sum_{i=1}^{n}T(i-1) = \sum_{i=1}^{n}T(n-i) = \sum_{i=0}^{n-1}T(i)$.

Moltiplicando per n entrambi i lati dell'equazione precedente, abbiamo che

$$nT(n) = cn^2 + 2\sum_{i=0}^{n-1}T(i) \tag{2.1}$$

allo stesso modo, varrà che

$$(n-1)T(n-1) = c(n-1)^2 + 2\sum_{i=0}^{n-2}T(i) \tag{2.2}$$

Sottraendo l'equazione 2.2 dall'equazione 2.1, otteniamo

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= c(n^2-(n-1)^2) + 2T(n-1) \\ &= c(2n-1) + 2T(n-1) \end{aligned}$$

da cui

$$nT(n) = c(2n-1) + (n+1)T(n-1)$$

e, dividendo per $n(n+1)$, otteniamo

$$\frac{T(n)}{n+1} = c\frac{2n-1}{n(n+1)} + \frac{T(n-1)}{n}$$

Osservando ora che in generale, per ogni $k < n$

$$\frac{T(n-k)}{n-k+1} = c\frac{2(n-k)-1}{(n-k)(n-k+1)} + \frac{T(n-k-1)}{n-k} = c\frac{2n-(2k+1)}{(n-k)(n-k+1)} + \frac{T(n-k-1)}{n-k}$$

ne deriva

$$\begin{aligned} \frac{T(n)}{n+1} &= c\frac{2n-1}{n(n+1)} + \frac{T(n-1)}{n} \\ &= \cdots \\ &= c\sum_{k=0}^{n-1}\frac{2n-(2k+1)}{(n-k)(n-k+1)} + T(0) \\ &= c\sum_{k=1}^{n}\frac{2k-1}{k(k+1)} + T(0) = c\sum_{k=1}^{n}\frac{2k-1}{k(k+1)} + c' \qquad \text{con } c' = T(0) \\ &= c\sum_{k=1}^{n}\left(\frac{3}{k+1} - \frac{1}{k}\right) + c' = 3c\sum_{k=1}^{n}\frac{1}{k+1} - c\sum_{k=1}^{n}\frac{1}{k} + c' \\ &= 3c\left(\sum_{k=2}^{n+1}\frac{1}{k}\right) - c\sum_{k=1}^{n}\frac{1}{k} + c' = 3c\left(\sum_{k=1}^{n+1}\frac{1}{k} - 1\right) - c\sum_{k=1}^{n}\frac{1}{k} + c' \end{aligned}$$

Ricordiamo che i numeri armonici sono definiti come

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \frac{1}{2n} + \gamma + O\left(\frac{1}{n^2}\right) \qquad \text{con } \gamma = 0.577216\ldots$$

Abbiamo allora che

$$\begin{aligned}\frac{T(n)}{n+1} &= 3c(H_{n+1} - 1) - cH_n + c' \\ &= 3c\ln(n+1) - c\ln n + \frac{3c}{2(n+1)} - \frac{c}{2n} + 2c\gamma - 3c + c' + O\left(\frac{1}{n^2}\right)\end{aligned}$$

e quindi

$$\begin{aligned}T(n) &= 3c(n+1)\ln(n+1) - c(n+1)\ln n + \frac{3}{2}c - \frac{c}{2}\frac{n+1}{n} + c(2\gamma - 3)(n+1) + c'(n+1) + O\left(\frac{1}{n}\right) \\ &= 3c(n+1)\ln(n+1) - c(n+1)\ln n + o(n\log n)\end{aligned}$$

Si osservi infine che

$$T(n) \geq 3c(n+1)\ln(n+1) - c(n+1)\ln(n+1) + o(n\log n)$$

e che, per n sufficientemente grande,

$$T(n) \leq 3c(n+1)\ln(n+1) + o(n\log n)$$

Ne deriva che $T(n) = \Theta(n\log n)$ □

Quickselect

Possiamo modificare lo schema ricorsivo dell'Algoritmo 2.1 per risolvere il classico problema della selezione dell'elemento con rango r in un array a di n elementi distinti, senza bisogno di ordinarli (ricordiamo un elemento ha rango r se a contiene r elementi minori o uguali di esso). Chiaramente, se $r = 1$ o $r = n$ il problema diventa quello di trovare il minimo o il massimo in a.

Per risolvere il problema per un qualunque valore di r con $1 \leq r \leq n$, ricordiamo che la funzione distribute del Algoritmo 2.3 permette di trovare il rango del pivot, posizionando tutti gli elementi di rango inferiore alla sua sinistra e tutti quelli di rango superiore alla sua destra. In base a tale osservazione, possiamo modificare il codice di ordinamento per quicksort considerando che, per risolvere il problema della selezione, è sufficiente proseguire ricorsivamente nella sola parte dell'array contenente l'elemento da selezionare: otteniamo così l'algoritmo Quickselect, mostrato come Algoritmo 2.4, che determina tale parte sulla base del confronto tra $r-1$ e r.

Algorithm 2.4: Selezione elemento di rango k

Input: Array $a = [1, \ldots, n]$, intero k $(1 \leq k \leq n)$
Output: Elemento di a di rango k nell'array

```
1 select(a,1,n,k);
2 return a
```

Nel caso peggiore, la ricorsione ha termine quando il segmento è composto da un solo elemento, che a quel punto è necessariamente quello voluto e che viene restituito. Per la valutazione del costo medio, vale il teorema seguente.

Algorithm 2.5: Funzione select

```
1  Function select(a,min,max,k);
2  seleziona pivot in [min,max], ad esempio pivot ← max;
3  r = distribute(a,min,max,pivot);
4  if r < k then
5      select(a,r + 1,max,k)
6  else if r > k then
7      select(a,min,r − 1,k)
8  else
9      return r
10 return
```

Teorema 2.2. L'Algoritmo 2.4 ha complessità, nel caso medio, $O(n)$.

Dimostrazione. Osserviamo che, se l'array ha dimensione n, e se assumiamo che il pivot corrisponda all'elemento di rango i, allora con probabilità $\frac{i-1}{n}$ l'elemento che stiamo cercando ha rango minore di i, per cui dovremo ricercare in un array di dimensione $i-1$, con probabilità $\frac{n-i}{n}$ ha rango maggiore di i, per cui dovremo ricercare in un array di dimensione $n-1$, e con probabilità $\frac{1}{n}$ il pivot è l'elemento di rango i, per cui non dobbiamo fare più nulla.

Da ciò deriva la relazione di ricorrenza seguente:

$$\begin{aligned} T(n) &= cn + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{i-1}{n}T(i-1) + \frac{n-i}{n}T(n-i)\right) \\ &= cn + \frac{2}{n}\sum_{i=1}^{n}\frac{i-1}{n}T(i-1) \\ &= cn + \frac{2}{n}\sum_{i=1}^{n-1}\frac{i}{n}T(i) \end{aligned}$$

In modo simile all'analisi per il quicksort, osserviamo che

$$n^2T(n) = cn^3 + 2\sum_{i=1}^{n-1} iT(i)$$

e anche che

$$(n-1)^2T(n-1) = c(n-1)^3 + 2\sum_{i=1}^{n-2} iT(i)$$

Sottraendo, abbiamo

$$n^2T(n) - (n-1)^2T(n-1) = c(n^3 - (n-1)^3) + 2(n-1)T(n-1)$$

e quindi

$$n^2T(n) - (n^2-1)T(n-1) = c(3n^2 - 3n + 1)$$

Dividendo per $n(n+1)$, risulta

$$\frac{n}{n+1}T(n) - \frac{n-1}{n}T(n-1) = c\left(\frac{3n}{n+1} - \frac{3}{n+1} - \frac{1}{n(n+1)}\right) = c\left(3\frac{n-1}{n+1} + \frac{1}{n} - \frac{1}{n+1}\right)$$

dal che deriva

$$\begin{aligned}
\frac{n}{n+1}T(n) &= \frac{n-1}{n}T(n-1) + c\left(3\frac{n-1}{n+1} + \frac{1}{n} - \frac{1}{n+1}\right) \\
&= c\sum_{k=1}^{n}\left(3\frac{k-1}{k+1} + \frac{1}{k} - \frac{1}{k+1}\right) + T(0) \\
&= c\sum_{k=1}^{n}\left(3\left(1 - \frac{2}{k+1}\right) + \frac{1}{k} - \frac{1}{k+1}\right) + c' \qquad \text{ponendo } c' = T(0) \\
&= c\left(\sum_{k=1}^{n} 3 - \sum_{k=1}^{n}\frac{6}{k+1}\right) + c\left(\sum_{k=1}^{n}\frac{1}{k} - \sum_{k=1}^{n}\frac{1}{k+1}\right) + c' \\
&= 3cn - 6c\sum_{k=1}^{n}\frac{1}{k+1} + c\left(\sum_{k=1}^{n}\frac{1}{k} - \sum_{k=2}^{n+1}\frac{1}{k}\right) + c' \\
&= 3cn - 6c\sum_{k=2}^{n+1}\frac{1}{k} + c\left(1 - \frac{1}{n+1}\right) + c' \\
&= 3cn - 6c\sum_{k=1}^{n+1}\frac{1}{k} + 7c - \frac{c}{n+1} + c' \\
&= 3cn + 7c - \frac{c}{n+1} + c' - 6c\left(\ln(n+1) + \frac{1}{2(n+1)} + \gamma + O\left(\frac{1}{n^2}\right)\right) \\
&= 3cn - 6c\ln(n+1) - \frac{4c}{n+1} + c(7 - 6\gamma) + c' + O\left(\frac{1}{n^2}\right) = \Theta(n)
\end{aligned}$$

□

Randomizzazione

La valutazione della complessità nel caso medio degli algoritmi 2.1 e 2.4 fa riferimento all'ipotesi che le varie istanze del problema (permutazioni dell'insieme degli elementi) siano equiprobabili, cosa che potrebbe non essere realistica. L'utilizzo di scelte casuali nell'esecuzione degli algoritmi (randomizzazione) può, come vedremo, rendere l'efficienza degli algoritmi stessi più robusta rispetto alla distribuzione dei dati in ingresso.

Introduciamo quindi ora un'analisi del costo medio per il Quicksort che risulta essere indipendente dall'ordine iniziale degli elementi nell'array e si basa sull'uso di scelte casuali: ciò al fine di far sì che una distribuzione sbilanciata degli elementi in input possa aversi con una probabilità trascurabile. L'analisi per il Quickselect è, come già visto, molto simile.

Introducendo nell'esecuzione del Quicksort delle scelte casuali, facciamo sì che ogni volta che l'algoritmo viene eseguito su uno stesso array in ingresso, il comportamento non è deterministico, ma influenzato dagli esiti delle scelte casuali effettuate. Ne deriva che nell'analisi di complessità il numero medio di passi non si calcola più su tutti i possibili array di ingresso ma su tutti gli esiti delle scelte casuali per un array d'ingresso: questa è una nozione più forte, in quanto uno stesso array non può essere sempre sfavorevole al QuickSort, perché anche le scelte casuali di quest'ultimo adesso entrano in gioco.

Come si può vedere, l'algoritmo utilizza una primitiva `random()` per generare un valore reale r appartenente all'intervallo $0 \leq r \leq 1$, in modo uniforme ed equiprobabile. La funzione `random` è detta generatore random pseudocasuale, in quanto fornisce valori apparentemente casuali: apparentemente perché, trattandosi di un programma deterministico non può fornire valori completamente casuali. In generale, non è semplice ottenere un generatore casuale statisticamente significativo.

Algorithm 2.6: Funzione qsort

```
1 Function qsort(a,min,max);
2 if min < max then
3     pivot ← min + (max − min) × random();
4     r = distribute(a,min,max,pivot);
5     qsort(a,min,r − 1);
6     qsort(a,r + 1,max)
7 return
```

Un algoritmo che utilizza valori casuali (o pseudocasuali) viene detto randomizzato perché impiega la casualitàa per sfuggire a situazioni sfavorevoli, risultando più robusto rispetto a tali eventi (come nel nostro caso, in presenza di un array già in ordine crescente).

Teorema 2.3. L'algoritmo Quicksort randomizzato impiega tempo ottimo $O(n \log n)$ nel caso medio per ordinare n elementi.

Dimostrazione. Il teorema ha la stessa dimostrazione del Teorema 2.1, osservando che la probabilità che il pivot sia il k-esimo elemento su n è anche in questo caso pari a $\frac{1}{n}$.

□

Le stesse considerazioni si applicano per la versione randomizzata di Quickselect.

Liste randomizzate

Descriviamo ora una struttura di dati che implementa un dizionario effettuando l'inserimento e la ricerca di un elemento utilizzando delle scelte casuali, e mostriamo che nel caso medio i tempi delle operazioni di ricerca e inserimento sono ridotti da $O(n)$ a $O(\log n)$: anche se, nel caso peggiore, tali operazioni possono richiedere tempo $O(n)$.

La struttura di dati in questione, denominata skip list (lista a salti), è formata da un insieme di liste collegate tra loro in modo opportuno. Descriviamo inizialmente la struttura di una skip list ``statica", senza considerare le operazioni di inserimento. Mostreremo poi come tale struttura venga approssimata da un implementazione randomizzata di tali operazioni,che farà sì che il limite di complessità faccia riferimento non più al caso peggiore, ma al caso medio.

Per rappresentare un insieme I di n elementi da un universo $\mathcal{U}$ mediante una skip list, consideriamo un ordinamento non decrescente $e_1, \ldots, e_n$ di I.

Consideriamo quindi la funzione $r : \mathbb{N} \mapsto \mathbb{N} \cup \{0\}$ che, ad ogni $i \in \mathbb{N}$, associa la massima potenza di 2 che divide i: di seguito, sono presentati i valori di r per $i = 1, \ldots, 12$.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r(i)$	0	1	0	2	0	1	0	3	0	1	0	2

Detto $m = \max_{1 \le i \le n} r(i)$ il massimo valore di $r(i)$ per tutti gli elementi e_i, possiamo allora considerare, per ogni $k \in \{0, \ldots, m\}$ il sottoinsieme $I_k \subseteq I = \{e_i | r(i) \ge k\}$ degli elementi e_i tali che $r(i)$ è almeno k: per semplicità di notazione, poniamo $n_k = |I_k|$. Chiaramente, $I_0 = I$ e $n_0 = n$. Si noti che $m = \lfloor \log_2 n \rfloor$.

Una skip list per I è composta da $h = m + 1$ liste L_k ($k = 0, \ldots, m$), tali che L_k contiene gli elementi di I_k ordinati in modo non decrescente, preceduti da un elemento $e_0 = -\infty$ e seguiti da un elemento $e_{n_k+1} = \infty$, tali quindi che tali che $e_0 < e < e_{n_k+1}$, per ogni $e \in \mathcal{E}$.

Per ogni lista L_k con $k > 0$, un elemento con valore e_i (eccetto quelli iniziale e finale, di valore $-\infty$ e ∞, rispettivamente) ha due puntatori, il primo all'elemento successivo in L_k e il secondo all'elemento di L_{k-1} avente medesimo valore e_i. Inoltre, l'elemento iniziale ha un puntatore all'elemento con valore minimo nella lista e un

secondo puntatore all'elemento iniziale di L_{k-1}. Per la lista L_0, gli elementi sono collegati dai soli puntatori del primo tipo.

Il punto di accesso alla struttura è dato da un puntatore L all'elemento $-\infty$ della lista L_m

In Figura 2.1 è mostrato un esempio della struttura risultante per $I = \{5, 10, 16, 18, 30, 41, 80\}$.

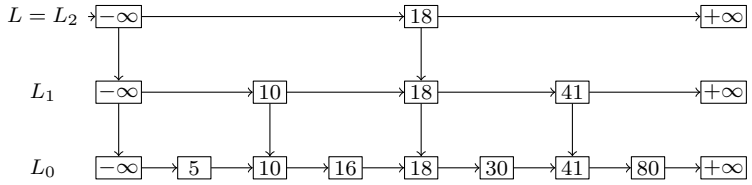


Figura 2.1: Un esempio di skip list di altezza 2.

Osserviamo che se L_0 contiene $n+2$ elementi ordinati, allora L_1 ne contiene al più $2 + \frac{n}{2}$, L_2 ne contiene al più $2 + \frac{n}{4}$ e, in generale, L_k contiene al più $2 + \frac{n}{2^k}$ elementi ordinati. Pertanto, il numero totale di copie presenti nella skip list è al più:

$$\begin{aligned}
(2+n) + (2+\frac{n}{2}) + \cdots + (2+\frac{n}{2^m}) &= 2(m+1) + \sum_{k=0}^{m} \frac{n}{2^k} \\
&= 2(m+1) + n\sum_{k=0}^{m} \frac{1}{2^k} \\
&= 2(m+1) + n\frac{1-(\frac{1}{2})^{m+1}}{1-\frac{1}{2}} \\
&< 2(m+1) + 2n \\
&= 2(\lfloor \log_2 n \rfloor + n + 1) \\
&\leq 2(\log_2 n + n + 1)
\end{aligned}$$

Quindi, lo spazio totale occupato dalla skip list è $O(n)$.

Per descrivere le operazioni di ricerca e inserimento, necessitiamo della nozione di predecessore. Data una lista $L_k = e'_0, e'_1, \ldots, e'_{n_k+1}$ di elementi ordinati e un elemento x, diciamo che $e'_j \in L_k$ (con $0 \leq j \leq n_k$) è il predecessore di x (in L_k) se e'_j è il massimo tra gli elementi in L_k minori di x, ovvero $e'_j \leq x < e'_{j+1}$: osserviamo che il precedessore è sempre ben definito, in quanto il primo elemento di L_k è $-\infty$.

La ricerca di un elemento e è concettualmente semplice. Ad esempio, supponiamo di voler cercare l'elemento 80 nella lista mostrata nella Figura 2.1. Partendo da L_2, troviamo che il predecessore di 80 in L_2 è 18: a questo punto, passiamo alla copia di 18 nella lista L_1 e troviamo che il predecessore di 80 in quest'ultima lista è 41. Passando alla copia di 41 in L_0, troviamo il predecessore di 80 in questa lista, ovvero 80 stesso: pertanto, la chiave è stata trovata.

Tale modo di procedere è implementato nell'Algoritmo 2.7, in cui con `succ` indichiamo il puntatore all'elemento successivo della lista e con `inf` il puntatore alla corrispondente copia dell'elemento nella lista inferiore. Partiamo dalla lista L_m, seguendo il puntatore L, e troviamo il predecessore p_m di e in tale lista. Poiché $L_m \subseteq L_{m-1}$, possiamo raggiungere la copia di p_{m-1} in L_{m-1} e, a partire da questa posizione, scandire quest'ultima lista in avanti per trovare il predecessore p_{m-1} di e in L_{m-1}. Ripetiamo questo procedimento per tutti i livelli k a decrescere: partendo dal predecessore p_k di e in L_k, raggiungiamo la sua copia in L_{k-1}, e percorriamo quest'ultima lista in avanti per trovare il predecessore p_{k-1} di e in L_{k-1}. Quando raggiungiamo L_0 (in cui abbiamo `inf` è uguale a `null`), la variabile `pred` memorizza p_0, che è il predecessore che avremmo trovato se avessimo effettuato una scansione sequenziale di L_0. A questo punto scandiamo L_0 fino a trovare e.

È facile osservare che l'Algoritmo 2.7 è molto simile ad una ricerca binaria in un array: in effetti, ogni movimento seguendo il campo `succ` corrisponde a dimezzare la porzione di sequenza su cui proseguire la ricerca. Per questo motivo, è facile dimostrare che il costo della ricerca effettuata dall'algoritmo è $O(\log n)$, sostanzialmente inferiore quindi al tempo $O(n)$ richiesto da una scansione sequenziale di L_0.

Algorithm 2.7: Ricerca in una skip list

```
1 Function ScansioneSkipList(e);
2 p ← L;
3 while p≠ null do
4     while p.succ.key ≤ e do p ← p.succ;
5     p ← p.inf
6 while p.value < e do p ← p.succ;
7 if p.value = e then return p else return null
```

Il problema sorge con l'operazione di inserimento, la cui realizzazione prevede dapprima la ricerca del punto in L_0 in cui inserire il nuovo elemento e quindi l'inserimento vero e proprio. Una volta trovata la posizione in cui inserire la nuova chiave, però, tale inserimento risulterebbe essere troppo costoso se volessimo continuare a mantenere le proprietà della skip list descritte in precedenza, in quanto questo potrebbe voler dire modificare le copie, nelle liste di livello superiore, di tutti gli elementi che seguono la chiave appena inserita.

Per far fronte a questo problema e limitare la complessità dell'operazione, usiamo la casualità, definendo un algoritmo randomizzato di inserimento nella skip list, in cui non viene garantita la struttura perfettamente bilanciata della lista stessa, ma con alta probabilità l'altezza media risulta logaritmica, e quindi il tempo medio di esecuzione di una ricerca risulta anch'esso logaritmico.

Possiamo definire la casualità in termini di esito di una sequenza di lanci di una moneta equiprobabile, dove ciascun lancio ha una possibilità su due che esca testa (codificata con 1) e una possibilità su due che esca croce (codificata con 0), fornendo un bit casuale equiprobabile. Precisamente, diremo che la probabilità di ottenere 1 è $q = \frac{1}{2}$ e la probabilità di ottenere 0 è $1 - q = \frac{1}{2}$.

Effettuando b lanci successivi, possiamo ottenere una sequenza random di b bit casuali. Ciascuno dei lanci, in pratica, è simulato mediante una chiamata della funzione `random()`: il numero r generato pseudo-casualmente fornisce quindi il bit 0 se $0 \leq r < \frac{1}{2}$ e il bit 1 se $\frac{1}{2} \leq r < 1$. I lanci di moneta sono eseguiti in modo indipendente, per cui ad esempio otteniamo una delle quattro possibili sequenze di $b = 2$ bit (00, 01, 10 oppure 11) con probabilità $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. In generale, le probabilità dei lanci si moltiplicano, se possiamo assumere, come facciamo, che siano indipendenti tra loro: sotto tale ipotesi, una specifica sequenza di b bit casuali ha probabilità $\frac{1}{2^b}$. Osserviamo inoltre che la probabilità di non incontrare un 1 in b lanci è $\frac{1}{2^b}$, e che tale valore tende (esponenzialmente) a 0 al crescere di b: quindi, prima o poi ``dobbiamo" incontrare un 1 nella sequenza, se b è sufficientemente grande.

Algorithm 2.8: Inserimento in una skip list

```
1 Function InserimentoSkipList(e);
2 pred ← [p_0, p_1, ..., p_m];
3 for r ← 1; (r ≤ m + 1) ∧ (random() < 0.5); r ← r + 1 do
4     if r = m + 1 then
5         Crea lista L_{m+1};
6         Inserisci gli elementi −∞, ∞ in L_{m+1};
7         Crea puntatore p_{m+1} al primo elemento di L_{m+1};
8         pred[m + 1] = p_{m+1};
9         m ← m + 1;
10 Inserisci e come successore di pred[0] in L_0;
11 for i ← 1; i ≤ r; i ← i + 1 do
12     Inserisci e come successore di pred[i] in L_i;
13     Fai puntare e all'elemento pred[i − 1] in L_{i−1};
14 return
```

Nell'Algoritmo 2.8 la casualità viene utilizzata per inserire una chiave e in una skip list di altezza $m + 1$. Una volta identificati i predecessori $p_0, p_1, \ldots, p_m$ di e nelle $m + 1$ liste $L_0, \ldots, L_m$, in maniera analoga a quanto

descritto per l'operazione di ricerca, tali valori sono memorizzati in un vettore `pred`. Viene quindi eseguita una sequenza di $r \geq 1$ lanci di moneta, terminata appena viene ottenuto un valore 1, oppure se $r = m + 1$.

Se la condizione raggiunta è $r = m + 1$, l'altezza viene incrementata, creando una nuova lista L_{h+1} composta dalle chiavi $-\infty$ e $+\infty$, ponendo il primo elemento di tale lista come predecessore p_{h+1} dell'elemento e e incrementando il valore di m.

L'algoritmo crea quindi r copie di e, inserendole nelle liste $L_0, L_1, L_2, \ldots, L_r$: ciascun inserimento richiede tempo costante in quanto va creato un nodo immediatamente dopo ciascuno dei predecessori $p_0, p_1, \ldots, p_r$.

Chiaramente, questa implementazione dell'operazione di inserimento non fornisce nessuna garanzia che la struttura di una skip list descritta sopra venga mantenuta: si osservi in particolare che, nel caso in cui in corrispondenza ad ogni inserimento venga immediatamente estratto un valore 1, la struttura ottenuta è composta dalla sola lista L_0, e quindi si riduce ad una lista semplice. Si noti, però, che ottenere per n volte valore 1 (se n è il numero di elementi inseriti) è un evento a probabilità molto bassa.

La complessità nel caso medio delle operazioni di ricerca e inserimento in una skip list è mostrata nel teorema seguente.

Teorema 2.4. La complessità media delle operazioni di ricerca e inserimento su una skip list è $O(\log n)$.

Dimostrazione. Per determinare il tempo di esecuzione di una ricerca di un elemento e, cominciamo con lo stimare il numero di volte in cui ci si sposta da un elemento al successore in una delle liste, quindi il numero di volte in cui viene seguito il puntatore `succ`.

Consideriamo il momento in cui, nella ricerca, si giunge per la prima volta a un elemento di una lista L_k: per la struttura dell'algoritmo, l'elemento attraversato subito prima o, se $k < m$, è l'elemento corrispondente su L_{k+1} (che è il predecessore di e in tale lista) o, se $k = m$, è il puntatore L. Gli elementi di L_k che saranno considerati da ora in poi non hanno necessariamente un corrispondente in L_{k+1}. Ma, per come sono stati effettuati gli inserimenti, la probabilità che un elemento in L_k non abbia una copia in L_{k+1} è pari a $\frac{1}{2}$. Il valore atteso del numero di elementi attraversati su ogni lista è allora pari al più a

$$\sum_{k=1}^{\infty} k \cdot \frac{1}{2^k} = 2$$

Il numero di livelli attraversati non è evidentemente maggiore dell'altezza della skip list. Mostriamo ora che, con alta probabilità, una skip list di n elementi ha altezza $O(\log n)$. Dicendo ``con alta probabilità" intendiamo dire che la sua probabilità è $1 - O\left(\frac{1}{n^\alpha}\right)$ per ogni $\alpha > 1$, vale a dire che la probabilità che non avvenga tende asintoticamente (e polinomialmente) a 0 al crescere di n.

Osserviamo allora che la probabilità dell'evento E_{iK} che l'elemento e_i si trovi nella lista L_K, con $K = 3\log_2 n$ è pari a

$$P(E_{iK}) = \frac{1}{2^K} = \frac{1}{2^{3\log_2 n}} = \frac{1}{n^3}$$

La probabilità dell'evento E_K che uno qualsiasi degli n elementi si trovi in L_K (e quindi L_K esista, e l'altezza della skip list sia almeno K) è allora pari a

$$P(E_K) = P(E_{1K} \cup E_{2K} \cup \ldots \cup E_{nK}) \leq P(E_{1K}) + P(E_{2K}) + \ldots + P(E_{nK}) = \frac{n}{n^3} = \frac{1}{n^2}$$

e quindi l'altezza della skip list è minore di $3\log_2 n$ (e quindi $O(\log n)$) con probabilità $1 - \frac{1}{n^2}$, quindi con alta probabilità.

Di conseguenza, visto che, con alta probabilità, il numero di liste attraversate durante una ricerca è $O(\log n)$ e, visto che il numero medio di elementi attraversati su ogni lista è 2, ne deriva che durante la ricerca si attraversano in media $O(\log n)$ elementi.

La complessità dell'operazione di inserimento deriva dall'osservazione che tale operazione corrisponde al costo di una ricerca, necessaria per determinare il vettore $pred$ dei puntatori, più il costo relativo alla sequenza di estrazioni casuali e all'inserimento delle copie del nuovo elemento, con relativo aggiornamento dei predecessori: entrambe queste ultime operazioni hanno costo pari al più all'altezza della skip list, e quindi $O(\log n)$ nel caso medio. Dato

che la ricerca richiede anch'essa, come mostrato sopra, tempo medio $O(\log n)$, ne consegue che tale complessità media vale anche per l'inserimento.

□

Per quanto riguarda la cancellazione, sia e l'elemento da eliminare e $p_0, p_1, p_2, \ldots, p_h$ i suoi predecessori, identificati mediante una variante della ricerca di e. Dopo aver memorizzato i predecessori nell'array `pred` come effettuato anche all'inizio dell'Algoritmo 2.7, è sufficiente verificare, per ogni lista L_i, se e compare nella lista: a tal fine è sufficiente verificare se $pred[i]$ punta all'elemento e. In tal caso, e va eliminato da L_i modificando il valore di $pred[i]$. A questo punto,$L[ht]$ potrebbe non contenere più elementi (a parte $-\infty$ e $+\infty$): in tal caso, eliminiamo tale lista e decrementiamo h.

Il costo di tali operazioni, come per l'inserimento, è chiaramente dominato da quello della ricerca, e quindi risulta anch'esso $O(\log n)$.

Analisi ammortizzata

L'analisi ammortizzata è una tecnica utilizzata per analizzare il tempo di esecuzione di un algoritmo nei casi in cui si è interessati a determinare il comportamento asintotico (al crescere della dimensione delle istanze) su sequenze di operazioni. Ad esempio, potremmo essere interessati a valutare il tempo di esecuzione di un'operazione di inserimento in un albero binario di ricerca. In questo caso, l'approccio standard consiste nell'individuare il caso peggiore e valutare il costo dell'operazione in tale caso, ottenendo così un upper bound sul costo per una qualunque istanza. Esistono casi, però, in cui per ogni sequenza di operazioni sufficientemente lunga è impossibile che tutte le operazioni nella sequenza vengano applicate nel caso peggiore, in quanto l'esecuzione di una operazione in una situazione che fa sì che il costo sia elevato determina dei risparmi sulle operazioni successive, ad esempio modificando opportunamente la struttura di dati.

Fondamentalmente, l'analisi ammortizzata si interessa del costo complessivo, nel caso peggiore, di una sequenza di operazioni (e quindi del costo medio, sulla sequenza, di un'operazione, nel caso peggiore). Non ci dice quindi nulla riguardo al costo di una singola operazione, che può essere tipicamente molto più elevato. Si sottolinea la differenza con la valutazione nel caso medio del costo della stessa operazione: in questo caso la media viene intesa rispetto a tutte le istanze, mentre per l'analisi ammortizzata la media è effettuata su tutte le operazioni di una sequenza, per la sequenza peggiore possibile.

Tecniche di analisi ammortizzata

Diamo una descrizione di modalità diverse di analisi ammortizzata utilizzando come motivo conduttore il problema dell'incremento di un contatore. La stessa analisi può essere applicata per valutare il costo dell'operazione di inserimento in una coda binomiale, operazione che, come noto, è sostanzialmente interpretabile in termini di incremento di un contatore binario.

In tale problema, abbiamo un contatore binario di k cifre binarie, memorizzate in un array `contatore` di dimensione k i cui elementi valgono 0 oppure 1. In particolare, il valore del contatore è dato da

$$\sum_{i=0}^{k-1} (\texttt{contatore}[i] \times 2^i)$$

e supponiamo che inizialmente esso contenga tutti 0.

Come mostrato nell'Algoritmo 2.9, l'operazione di incremento richiede un costo in tempo pari al numero di elementi cambiati in contatore e quindi $O(k)$ tempo al caso pessimo: discutiamo tre modi di analisi per dimostrare che il costo ammortizzato di una sequenza di $n = 2^k$ incrementi è soltanto $O(1)$ per incremento.

- Il primo metodo è quello di aggregazione. Nell'applicare questo tipo di analisi conteggiamo il numero totale $T(n)$ di passi elementari eseguiti e lo dividiamo per il numero n di operazioni effettuate. Nel nostro caso,

Algorithm 2.9: Incremento contatore

```
1 Function Incrementa(contatore);
2 i ← 0;
3 while i < k ∧ contatore[i] = 1 do
4     contatore[i] ← 0;
5     i ← i + 1
6 if i = k then contatore[i] ← 1;
7 return
```

conteggiamo il numero di elementi cambiati in `contatore`, supponendo che quest'ultimo assuma il valore iniziale pari a zero. Effettuando n incrementi, osserviamo che l'elemento `contatore`$[0]$ cambia (da 0 a 1 o viceversa) a ogni incremento, quindi n volte; il valore di `contatore`$[1]$ cambia ogni due incrementi, quindi $\frac{n}{2}$ volte; in generale, il valore di `contatore`$[i]$ cambia ogni 2^i incrementi e quindi $\frac{n}{2^i}$ volte. In totale, il numero di passi è

$$T(n) = \sum_{i=0}^{k-1} \frac{n}{2^i} = \left(\sum_{i=0}^{k-1} \frac{1}{2^i} \right) n = \left(\frac{1 - \frac{1}{2}^k}{1 - \frac{1}{2}} \right) n < 2n$$

Quindi, il costo ammortizzato per incremento è $O(1)$ in quanto

$$\frac{T(n)}{n} < 2$$

- Il secondo metodo è basato sul concetto di credito: in questo caso, utilizziamo un fondo comune, in cui depositiamo crediti o li preleviamo, con il vincolo che il fondo deve sempre contenere un numero non negativo di crediti (per cui non è possibile prelevare più crediti di quanti siano effettivamente disponibili). Le operazioni possono sia depositare crediti nel fondo che prelevarne, con il suddetto vincolo di non portare mai il fondo in rosso per coprire il proprio costo computazionale: il costo ammortizzato per ciascuna operazione è il numero di crediti depositati da essa. Osserviamo che tali operazioni di deposito e prelievo di crediti sono introdotte solo ai fini dell'analisi, senza effettivamente essere realizzate nel codice dell'algoritmo così analizzato.

  Nel nostro esempio del contatore, partiamo da un contatore nullo e utilizziamo un fondo iniziale con zero crediti. Con riferimento all'Algoritmo 2.9, per ogni incremento eseguito associamo i seguenti movimenti sui crediti:

  1. preleviamo un credito per ogni valore di `contatore`$[i]$ cambiato da 1 a 0 nella riga 4;
  2. depositiamo un credito quando `contatore`$[i]$ cambia da 0 a 1 nella riga 6.

  Da notare che la situazione al punto 1 può occorrere un numero variabile di volte durante un singolo incremento (dipende da quanti valori pari a 1 sono esaminati dal ciclo); invece, la situazione al punto 2 occorre al più una volta, lasciando un credito per quando quel valore da 1 tornerà a essere 0. Possiamo facilmente renderci conto che ogni volta che serve un credito per cambiare un bit da 1 a 0, possiamo utilizzare il credito depositato quando lo stesso bit è passato da 0 a 1 (cosa che è necessariamente avvenuta, in quanto i bit sono inizialmente tutti 0): quindi, ogni volta che necessitiamo di un credito nel punto 1, possiamo prelevare dal fondo in quanto tale credito è stato sicuramente depositato da un precedente incremento nel punto 2. Questo garantisce che ci sono sempre crediti disponibili nel fondo per eseguire le varie operazioni, da cui deriva che il costo ammortizzato per incremento è $O(1)$, in effetti 1.

  Possiamo applicare il metodo dei crediti per l'analisi ammortizzata del ridimensionamento di un array a lunghezza variabile effettuato raddoppiando la dimensione dell'array quando non ci sono posizioni libere in cui inserire un nuovo elemento. Ogni qualvolta inseriamo un elemento in fondo all'array, depositiamo 2 crediti; ogni volta che raddoppiamo la dimensione dell'array, ricopiando gli elementi memorizzati dal vecchio al

nuovo array, utilizziamo i crediti accumulati dall'ultima volta in cui la dimensione è raddoppiata, che sono $2 \cdot \frac{n}{2} = n$. Si osservi che, depositando c crediti ad ogni inserimento, l'analisi non cambia se si assume di estendere ogni volta l'array di una frazione pari a $\frac{1}{c}$.

- Infine, il terzo metodo è basato sul concetto di potenziale (con relativa metafora fisica). Il potenziale è una opportuna funzione Φ che associa un valore reale non negativo ad ogni possibile stato di una struttura di dati. Alcune operazioni potranno aumentare il potenziale mentre altre lo faranno decrescere (si noti la similitudine con i crediti definiti sopra).

  Indichiamo con D_{i-1} lo stato della struttura di dati dopo $i-1$ operazioni, e con $\Phi(D_{i-1})$ (o Φ_{i-1} per sintesi) il relativo potenziale. Indichiamo con c_i il costo effettivo dell'i-esima operazione: definiamo il costo ammortizzato dell'operazione come

  $$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \tag{2.3}$$

  In questo tipo di analisi si assume quindi che ogni operazione determini una variazione del potenziale, che sostanzialmente viene visto come ``deposito" di tempo di computazione assegnabili, ai fini dell'analisi, alle varie operazioni. Il costo ammortizzato di un'operazione è allora pari al costo effettivo sommato all'incremento di potenziale indotto dall'operazione. Per alcune operazioni si potrà avere un incremento di potenziale, per cui $\hat{c} > c$, accumulando tempo di computazione; altre operazioni comporteranno un decremento, per cui $\hat{c} < c$, o nessuna variazione del potenziale, in quest'ultimo caso avendo $\hat{c} = c$.

  Se indichiamo con D_0 lo stato iniziale della struttura di dati, il costo ammortizzato della sequenza di operazioni sarà

  $$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0 \tag{2.4}$$

  in quanto ogni termine Φ_i con $1 \leq i < n$ compare sia sommato che sottratto.

  Se allora viene rispettata la condizione che $\Phi_n \geq \Phi_0$ (tipicamente ponendo $\Phi_0 = 0$), allora si ha che

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

  e quindi il costo totale ammortizzato è un upper bound sul costo totale delle operazioni. Una delimitazione superiore del costo ammortizzato vale quindi anche per il costo totale.

  Ad esempio, nel caso del contatore binario, poniamo Φ_i uguale al numero di bit di `contatore` uguali a 1 dopo l'i-esimo incremento, con $1 \leq i \leq n$: quindi, $\Phi_0 = 0$ in quanto il contatore inizialmente ha tutti i bit uguali a 0. Per semplicità, assumiamo che il contatore contenga sempre uno 0 in testa. Consideriamo lo i-esimo incremento, indicando con l_i il numero di volte in cui viene eseguita la riga 4 nel ciclo `while` dell'Algoritmo 2.9: il costo, in termini di numero di bit cambiati, è quindi $c_i = l_i + 1$ in quanto l_i valori pari a 1 diventano 0 e un valore pari a 0 diventa 1. Inoltre, la differenza di potenziale $\Phi_i - \Phi_{i-1}$ misura la differenza tra il numero di bit pari a 1 prima e dopo l'incremento: evidentemente, avremo l_i bit che sono diventati 0 da 1 che erano e 1 che è diventato 1 da 0 che era, per cui $\Phi_i - \Phi_{i-1} = -l_i + 1$. Utilizzando la formula (2.3), otteniamo un costo ammortizzato pari a $\hat{c}_i = (l_i + 1) + (-l_i + 1) = 2$. Poiché $\Phi_{n-1} \geq 0$ e $\Phi_{-1} = 0$, in base all'equazione (2.4) abbiamo che

  $$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

Splay trees

Gli alberi tradizionalmente utilizzati per implementare dizionari (AVL trees, red-black trees, B-trees) prevedono, in corrispondenza agli inserimenti e alle eliminazioni, delle operazioni locali di ribilanciamento, che garantiscono che l'altezza dell'albero (e quindi la complessità di tutte le operazioni) sia $O(\log n)$. Questo, oltre a rendere più complesse tali operazioni, richiede la rappresentazione, nei singoli nodi, di una adeguata informazione di struttura (ad esempio, il fattore di bilanciamento in un AVL).

Le skip list sono un primo esempio di struttura di dati che utilizza una implementazione più semplice delle operazioni di inserimento ed eliminazione, ottenendo a fronte di ciò un limite $O(\log n)$ sulla complessità valido nel caso medio, e non nel caso peggiore.

Gli splay tree sono una versione degli alberi binari di ricerca in cui, a fronte di una semplificazione rappresentata dall'assenza di informazione di struttura nei nodi, si ha una complessità $O(\log n)$ per le varie operazioni valida però soltanto in senso ammortizzato: in uno splay tree, quindi, anche se una singola operazione può richiedere tempo di esecuzione superiore (anche $O(n)$), una sequenza di m operazioni viene eseguita in tempo $O(m \log n)$.

Uno splay tree è un albero binario di ricerca, per cui è strutturato in modo tale che, se un nodo contiene un elemento e[1], allora tutti gli elementi nel suo sottoalbero sinistro (se esiste) sono minori di e e tutti gli elementi nel sottoalbero destro (se esiste) sono maggiori di e: la ricerca di un elemento in uno splay tree avviene quindi nel modo usuale, in tempo proporzionale all'altezza dell'albero.

Uno splay tree non è però bilanciato, nel senso che la sua altezza può anche essere $O(n)$ nel caso peggiore. Una (piuttosto semplice) riorganizzazione della struttura dell'albero viene effettuata in corrispondenza ad ogni operazione (ricerca, inserimento, eliminazione) di un elemento e, e prende il nome di `Splay`(e). In particolare:

- se l'operazione è una ricerca di e, il relativo nodo viene ricercato e quindi spostato nella radice dell'albero per mezzo di una `Splay`(e), e quindi di una sequenza di riorganizzazioni locali effettuate iterativamente su tutti i nodi sul cammino da e alla radice dell'albero; se la ricerca ha esito negativo, allora termina necessariamente su una foglia contenente un elemento $e' \neq e$ e possiamo effettuare `Splay`(e')
- se l'operazione è l'inserimento di e, il nodo contenente e viene inserito nel modo usuale nell'albero di ricerca, dopo di che un'operazione `Splay`(e) riorganizza il cammino da tale nodo alla radice
- se l'operazione è l'eliminazione di e, si può procedere nel modo seguente
  1. effettuare uno `Splay`(e), che porta e nella radice
  2. eliminare la radice, contenente e, lasciando due alberi T_1, T_2 corrispondenti ai due sottoalberi (sinistro e destro, rispettivamente) della radice
  3. trovare il nodo contenente l'elemento e' minimo dell'albero T_2; e' è il successore immediato di e nell'insieme
  4. effettuare uno `Splay`(e'), che porta e' nella radice di T_2: necessariamente, la nuova radice, contenendo il minimo, ha un solo figlio, destro
  5. rendere T_1 sottoalbero sinistro di T_2, collegando la radice di T_2 alla radice di T_1

La riorganizzazione da effettuare è determinata sulla base dei collegamenti tra i nodi immediatamente al di sopra del nodo contenente e nel cammino verso la radice. Sono considerati tre possibili casi, i cui costi misuriamo in termini di rotazioni effettuate tra coppie di nodi rispettivamente padre e figlio: dato che, per come sono applicate qui, una rotazione comporta la riduzione di 1 della distanza da e alla radice, il costo è anche misurabile in termini di livelli di cui sale e per effetto della riorganizzazione.

1. Caso ``zig'': se e è figlio della radice, viene applicata la riorganizzazione in Figura 2.2, che in tempo 1 fa diventare radice il nodo contenente e.
2. Caso ``zig-zig'': se e è figlio sinistro del suo nodo padre p, e quest'ultimo a sua volta è figlio sinistro del nodo nonno n di e (o se sono entrambi figli destri), viene applicata la riorganizzazione in Figura 2.3, che in tempo 2 fa diminuire di 2 la distanza tra il nodo contenente e e la radice.

[1] Per semplicità, indicheremo indifferentemente con e sia l'elemento che il nodo in cui è contenuto

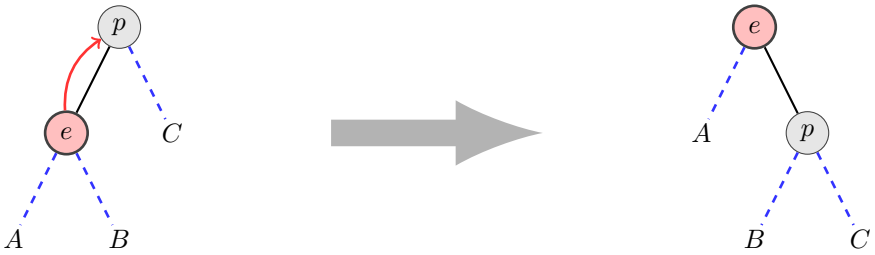


Figura 2.2: Primo caso di riorganizzazione locale in uno splay tree.

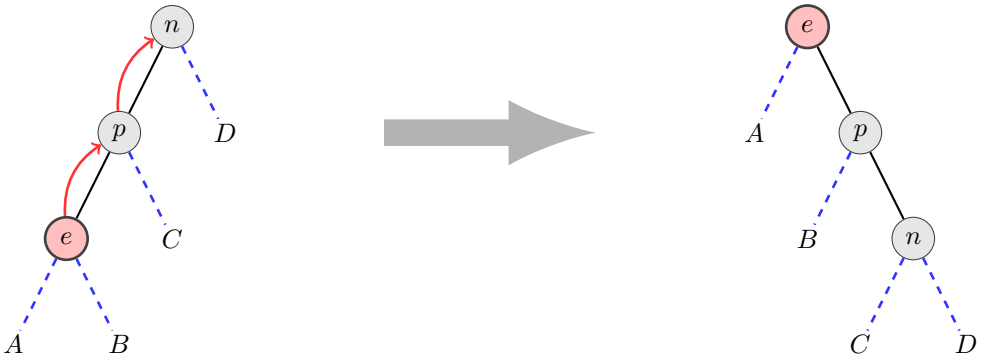


Figura 2.3: Secondo caso di riorganizzazione locale in uno splay tree.

3. Caso ``zig-zag'': infine, se e è figlio sinistro del suo nodo padre p, e quest'ultimo è figlio destro del nodo nonno n di e (o se sono il primo figlio destro e il secondo figlio sinistro), viene applicata la riorganizzazione in Figura 2.4, che in tempo 2 fa diminuire di 2 la distanza tra il nodo contenente e e la radice.

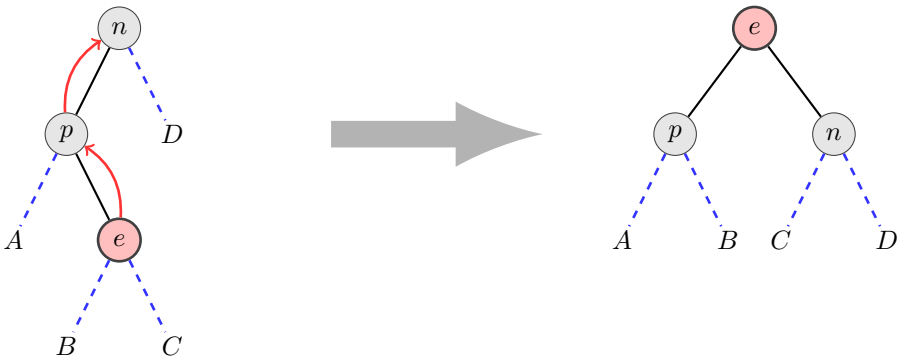


Figura 2.4: Terzo caso di riorganizzazione locale in uno splay tree.

Per come sono definite, le operazioni di ricerca, inserimento ed eliminazione hanno complessità asintotica pari a quella della `Splay`: analizzare il costo di quest'operazione consente quindi di valutare i costi di tutte le operazioni definite su un dizionario implementato mediante uno splay tree.

Per effettuare una analisi della complessità ammortizzata della `Splay`, definiamo una funzione potenziale che associ un valore non negativo ad ogni configurazione dello splay tree. A tal fine, indichiamo, per ogni nodo e, con $w(e)$ il numero di nodi nel sottoalbero di radice e (incluso e stesso) e con $r(e) = \log_2 w(e)$ il rango di e. Il

potenziale di uno splay tree è pari alla somma dei ranghi di tutti i nodi dell'albero

$$\Phi(T) = \sum_{e \in T} r(e)$$

Si noti che, anche se la radice ha il rango più elevato tra tutti i nodi, le foglie contribuiscono al potenziale più di essa, in quanto contribuiscono al rango di ogni nodo predecessore; inoltre, foglie a maggiore distanza dalla radice forniscono un contributo maggiore al potenziale di foglie a minore profondità.

Come misura del costo effettivo di una `Splay`(e), potremo poi utilizzare la profondità nell'albero del nodo e. Possiamo allora mostrare i lemmi seguenti, in cui r' indica il rango successivo all'operazione.

Lemma 2.1. Durante una `splay`, il costo ammortizzato di una riorganizzazione al nodo e, nel caso "zig" è al più pari a $3(r'(e) - r(e)) + 1$

Dimostrazione. Indichiamo con p il padre di e. Allora è facile osservare, dalla Figura 2.2 che:

- $r'(e) = r(p)$
- $r'(p) \leq r'(e)$
- $r(e) \leq r'(e)$

Il costo ammortizzato è allora

$$\hat{c} = c + \Phi' - \Phi = 1 + (r'(e) + r'(p)) - (r(e) + r(p)) = 1 + r'(p) - r(e) \leq 1 + r'(e) - r(e) \leq 1 + 3(r'(e) - r(e))$$

□

Lemma 2.2. Durante una `splay`, il costo ammortizzato di una riorganizzazione al nodo e, nel caso "zigzig" è al più pari a $3(r'(e) - r(e))$

Dimostrazione. Indichiamo con p il padre e con n il nonno di e. Dalla Figura 2.3 che:

- $r'(e) = r(n)$
- $r'(p) \leq r'(e)$
- $r'(n) \leq r'(e)$
- $r(e) \leq r(p)$

Il costo ammortizzato è per definizione

$$\begin{aligned}
\hat{c} &= c + \Phi' - \Phi \\
&= 2 + (r'(e) + r'(n) + r'(p)) - (r(e) + r(n) + r(p)) \\
&= 2 + r'(n) + r'(p) - r(e) - r(p) \\
&\leq 2 + r'(n) + r'(e) - r(e) - r(e) = 2 + r'(n) + r'(e) - 2r(e)
\end{aligned}$$

Osserviamo poi che $w(e) + w'(n) \leq w'(e)$, per cui

$$\log_2 \frac{w(e) + w'(n)}{2} \leq \log_2 \frac{w'(e)}{2} = r'(e) - 1$$

Per la concavità della funzione logaritmo abbiamo che, in generale, per ogni $0 < c < 1$

$$\log_2(ca + (1-c)b) \geq c \log_2 a + (1-c) \log_2 b$$

e quindi, se $c = \frac{1}{2}$,

$$\log_2\left(\frac{a}{2}+\frac{b}{2}\right) \geq \frac{1}{2}\log_2 a + \frac{1}{2}\log_2 b = \frac{\log_2 a + \log_2 b}{2}$$

per cui

$$\log_2 \frac{w'(e)}{2} = r'(e) - 1 \geq \log_2 \frac{w(e)+w'(n)}{2} \geq \frac{\log_2 w(e) + \log_2 w'(n)}{2} = \frac{r(e)+r'(n)}{2}$$

dal che deriva

$$r(e) + r'(n) \leq 2r'(e) - 2$$

e quindi $r'(n) \leq 2r'(e) - r(e) - 2$. Sostituendo nell'espressione relativa al costo ammortizzato,

$$\hat{c} \leq 2 + r'(n) + r'(e) - 2r(e) \leq 2 + 2r'(e) - r(e) - 2 + r'(e) - 2r(e) = 3(r'(e) - r(e))$$

□

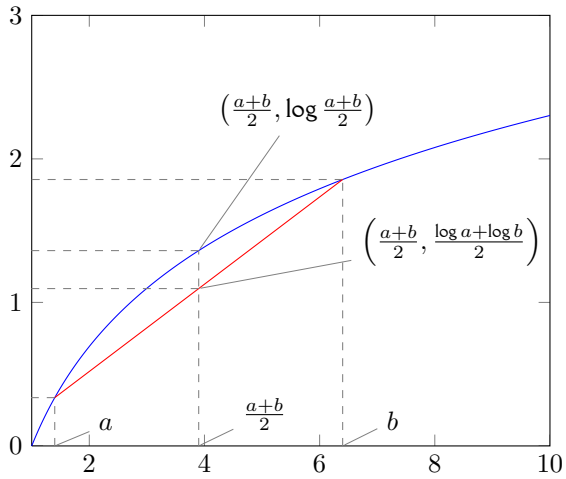


Figura 2.5: Concavità del logaritmo

Lemma 2.3. Durante una `splay`, il costo ammortizzato di una riorganizzazione al nodo e, nel caso "zigzag" è al più pari a $3(r'(e) - r(e))$

Dimostrazione. La dimostrazione è del tutto simile a quella del Lemma 2.2. □

Teorema 2.5. Il costo ammortizzato di un'operazione `Splay` è $O(\log n)$.

Dimostrazione. Assumiamo che l'operazione sia applicata all'elemento e ed effettui k riorganizzazioni. Il suo costo ammortizzato è evidentemente pari alla somma dei costi ammortizzati delle riorganizzazioni, per cui, tenendo conto che il caso "zig" compare esattamente una volta,

$$\hat{c} = \sum_{j=1}^{k} \hat{c}_i \leq \sum_{j=1}^{k} (3(r_j(e) - r_{j-1}(e))) + 1 = 3(r_k(e) - r_0(e)) + 1$$

Si noti che $r_k(e) = \log n$ in quanto al termine della `Splay` e è la radice; inoltre, $r_0(e) \geq 1$ necessariamente, per cui $\hat{c} \leq 3\log n - 2 = O(\log n)$. □

Possiamo infine determinare il costo di una sequenza di operazioni di ricerca, inserimento ed eliminazione.

Teorema 2.6. La complessità di una sequenza di m operazioni di ricerca, inserimento ed eliminazione su uno splay tree è $O((m+n)\log n)$, dove n è la dimensione massima raggiunta dallo splay tree nel corso della sequenza.

Dimostrazione. Dato che ognuna delle tre operazioni esegue un numero costante di `Splay`, nella loro sequenza sono eseguite $t = O(m)$ `Splay`.

Dall'equazione 2.4, abbiamo che

$$\sum_{i=1}^{t} c_i = \sum_{i=1}^{t} \hat{c}_i + \Phi_0 - \Phi_t \leq 3t \log n - 2t + \Phi_0 - \Phi_t$$

si noti che il valore della funzione potenziale su uno splay tree di n nodi non può essere più grande di $n \log n$, per cui la differenza $\Phi_0 - \Phi_t$ è anch'essa minore di $n \log n$. Ne deriva che

$$\sum_{i=1}^{t} c_i \leq 3t \log n + n \log n = O((m+n)\log n).$$

□

Algoritmi on-line e analisi competitiva

In determinate situazioni, è necessario, da parte di un algoritmo, servire una sequenza di richieste, prendendo delle decisioni, venendo a conoscere le richieste una alla volta, man mano che si presentano. Questo tipo di situazione si presenta ad esempio all'interno dei sistemi operativi, in cui l'accesso alle risorse disponibili viene controllato da algoritmi che servono le richieste cercando di massimizzare l'utilizzo delle risorse.

Algoritmi di questo tipo vengono detti on-line, in quanto devono decidere su come servire una richiesta non appena la richiesta stessa viene ricevuta. Ad essi si contrappongono gli algoritmi off-line che operano sulla base della conoscenza preventiva dell'intera sequenza di richieste.

Data una sequenza $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, un algoritmo on-line A serve le richieste considerando prima σ_1 (senza conoscere $\{\sigma_2, \ldots, \sigma_n\}$), fornendo un output a_1, considerando poi σ_2 (senza conoscere $\{\sigma_3, \ldots, \sigma_n\}$), fornendo un output a_2, e così via, fino a fornire l'intera sequenza $a_1, a_2, \ldots, a_n$ di output. Se assumiamo definita una funzione di costo definita sugli output, possiamo indicare con $c_A(\sigma)$ il costo complessivo in cui incorre l'algoritmo on-line per servire la sequenza di richieste $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$.

Un algoritmo off-line potrebbe evidentemente servire le richieste in modo più efficiente potendo contare, nel decidere come servire σ_i della conoscenza di tutte le altre richieste: in particolare, tra tutti gli algoritmi off-line, ce ne sarà uno che serve ogni sequenza $\sigma_1, \sigma_2, \ldots, \sigma_n$ con il costo minimo possibile. Indichiamo con OPT tale algoritmo e con $c_{OPT}(\sigma_1, \sigma_2, \ldots, \sigma_n)$ il costo in cui esso incorre nel servire $\sigma_1, \sigma_2, \ldots, \sigma_n$, e quindi, per definizione, il costo minimo necessario per servire tale sequenza di richieste.

In una situazione di questo tipo, quel che vogliamo è caratterizzare A in termini di quanto, nel caso peggiore, il suo costo supera il costo di OPT, e quindi quanto la non conoscenza del futuro, in cui esso opera, induce in termini di inefficienza della soluzione trovata. La misura utilizzata per questa caratterizzazione è il rapporto di competitività inteso come rapporto tra il costo di A e il costo di OPT per la sequenza per la quale tale rapporto è massimo[2].

Più precisamente, diremo che A è un algoritmo k-competitivo se esiste una costante $a > 0$ tale che, per ogni sequenza σ di richieste, si ha

$$c_A(\sigma) \leq k \times c_{OPT}(\sigma) + a$$

Come esempio elementare di analisi competitiva, consideriamo il seguente problema detto di ski rental: uno sciatore si reca ogni giorno (per un numero di giorni che conosce) sulle piste da sci. Non possedendo sci, per utilizzarli può o pagare un noleggio giornaliero, pari a 1, o decidere, a un certo punto, di acquistarli, spendendo B.

[2] Si noti che in certi casi il problema è definito in termini non di costo ma di profitto: in tal caso, il rapporto è necessariamente minore o uguale a 1 e il caso peggiore corrisponde al minimo.

Ogni giorno, prima di andare a sciare deve decidere se pagare un altro giorno di noleggio o comprare gli sci (non dovendo quindi più pagare il noleggio, nei prossimi giorni).

Si noti che, se lo sciatore sapesse fin dall'inizio per quanti giorni dovrà utilizzare gli sci, il problema sarebbe di facile soluzione: se n è il numero di giorni, la strategia migliore consiste nel noleggiare sempre, se $n < B$ oppure acquistare subito gli sci, se $n > B$; se $n = B$ le due soluzioni sono equivalenti: da questo consegue che

$$c_{OPT} = \min(B, n)$$

Ci chiediamo quale sia la migliore strategia per lo sciatore, per rendere il minimo possibile il rapporto tra la sua spesa e quanto spenderebbe sapendo in anticipo per quanti giorni deve utilizzare gli sci. Ogni possibile strategia non può far altro che noleggiare per un certo numero i di giorni e poi decidersi, al giorno $i+1$, a comprare gli sci: indichiamo tale strategia come A_i. Da ciò consegue che

$$c_{A_i} = \begin{cases} n & \text{se } n \leq i \\ i + B & \text{se } n > i \end{cases}$$

Vogliamo in definitiva individuare il valore di i per il quale il rapporto di competitività tra A_i e OPT è minimo. Per analizzare il rapporto tra c_{OPT} e c_{A_i}, consideriamo i due casi seguenti:

- $i < B$: sotto queste ipotesi, il caso peggiore si ha per $n = i+1$, per cui $c_{OPT} = i+1$ e anche $c_{A_i} = i+B$. Di conseguenza, in questo caso si ha

  $$\frac{c_{A_i}}{c_{OPT}} = \frac{i+B}{i+1} = 1 + \frac{B-1}{i+1}$$

  che è minimizzato per $i = B-1$, nel qual caso si ha

  $$\frac{c_{A_i}}{c_{OPT}} = 1 + \frac{B-1}{B} = 2 - \frac{1}{B}$$

- $i \geq B$: ora il caso peggiore si ha ancora per $n = i+1$, per cui $c_{OPT} = B$ e $c_{A_i} = i+B$, e quindi

  $$\frac{c_{A_i}}{c_{OPT}} = \frac{i+B}{B} = 1 + \frac{i}{B}$$

  che è minimizzato per $i = B$, nel qual caso,

  $$\frac{c_{A_i}}{c_{OPT}} = 2$$

In definitiva, quindi, la strategia migliore consiste nel fissare $i = B-1$.

Liste ad auto-organizzazione

In una lista, l'unica possibilità di effettuare una ricerca è attraverso una scansione sequenziale di tutti gli elementi, eventualmente fino al termine della lista stessa, come implementato dall'Algoritmo 2.10. Come ben noto, il costo nel caso peggiore (e anche nel caso medio) di questo algoritmo è $O(n)$, più precisamente n nel caso peggiore e $\frac{n}{2}$ nel caso medio.

La ricerca sequenziale può trarre beneficio dal principio di località temporale, per il quale se accediamo a un elemento in un dato istante è molto probabile che accederemo a questo stesso elemento in istanti immediatamente (o quasi) successivi. Seguendo tale principio, sembra naturale che possiamo riorganizzare proficuamente gli elementi della lista dopo aver eseguito la loro scansione. Per questo motivo, una lista così gestita viene riferita come struttura di dati ad auto-organizzazione (self-organizing o self-adjusting).

L'operazione fondamentale utilizzata per la riorganizzazione è lo scambio (swap) tra due elementi successivi, per effetto del quale i due elementi scambiano la loro posizione, invertendo le relazioni di predecessore e successore.

Algorithm 2.10: Ricerca sequenziale

```
1 Function RicercaSequenziale(l, k);
2 p ← l;
3 while p ≠ null ∧ p.dato ≠ k do p ← p.succ;
4 return p
```

Supponiamo allora di avere una lista l di n elementi, sulla quale il costo di accesso all'elemento i-esimo è i, e sulla quale assumiamo di poter compiere operazioni di scambio tra due elementi contigui in tempo costante, una volta che si sia ottenuto accesso ad essi.

L'obiettivo che ci poniamo, a partire da una configurazione iniziale della lista, è quello di effettuare una sequenza di m ricerche in tempo minimo, tenendo presente che possono essere effettuate più ricerche ad uno stesso elemento. Si noti che, se la sequenza di operazioni (in effetti gli elementi ricercati) fosse nota in anticipo, si potrebbe definire e utilizzare un algoritmo, che chiamiamo OPT, che utilizzi gli scambi per riordinare la lista in modo tale da minimizzare il tempo complessivo di ricerca. Le prestazioni dei due algoritmi verranno confrontate rispetto al loro costo, definito come la somma dei costi delle singole operazioni.

In assenza di tale informazione preventiva, è necessario utilizzare un metodo di riorganizzazione della lista che operi sulla sola base della conoscenza dell'elemento attualmente ricercato e di quelli precedenti, senza alcuna conoscenza o previsione del futuro. Sarà quindi necessario confrontare il tempo necessario a questo metodo per effettuare tutte le ricerche in una sequenza, confrontandosi con il tempo necessario all'ottimo, conoscendo in anticipo la sequenza stessa. Si noti quindi che, in questo caso, utilizziamo l'analisi ammortizzata non per stimare in assoluto il tempo necessario ad eseguire una sequenza di operazioni, ma per valutare tale costo relativamente al caso in cui la sequenza sia nota a priori. Quel che otteniamo, quindi, è una valutazione di quanto la non conoscenza del futuro ci penalizza nella soluzione di un problema.

Tra le varie strategie di auto-organizzazione, la più diffusa ed efficace è detta move-to-front (MTF): essa consiste nello spostare l'elemento acceduto dalla sua posizione attuale alla cima della lista, senza cambiare l'ordine relativo dei rimanenti elementi, come mostrato nell'Algoritmo 2.11.

Come detto, MTF effettua ogni ricerca senza conoscere le ricerche che dovrà effettuare in seguito: un algoritmo operante in tali condizioni, che deve quindi servire un insieme di richieste man mano che esse pervengono, viene detto in linea (online).

Il costo che considereremo, proporzionale al tempo di esecuzione è dato dal il numero di elementi della lista attraversati prima di raggiungere l'elemento desiderato, a partire dall'inizio della lista: quindi, accedere all'elemento in posizione i ha costo i in quanto dobbiamo attraversare gli i elementi che lo precedono. Lo spostamento in cima alla lista, operato da MTF, non viene conteggiato in quanto richiede sempre un costo costante.

Algorithm 2.11: Move to front

```
1  Function MoveToFront(a, k);
2  p ← a;
3  if p = null ∨ p.dato = k then return p;
4  while p.succ ≠ null ∧ p.succ.dato ≠ k do p ← p.succ;
5  if p.succ = null then return null;
6  tmp ← p.succ;
7  p.succ ← p.succ.succ;
8  tmp.succ ← a;
9  a ← tmp;
10 return a
```

Come osservato sopra, è di fondamentale importanza stabilire le regole di azione di OPT, perché questo può dare luogo a risultati completamente differenti. Nel nostro caso, assumiamo che, esaminate tutte le richieste in anticipo, OPT, in corrispondenza ad ogni richiesta, effettui l'accesso all'elemento ed effettui poi un insieme di scambi tra elementi contigui. Per semplificare la nostra analisi comparativa, assumiamo per ora che OPT e MTF

Lista MTF	=	e_4	e_2	e_6	e_0	e_1	e_3	e_7	e_5
Lista OPT	=	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7

Figura 2.6: Un'istantanea delle liste manipolate da MTF e OPT.

partano con la stessa lista iniziale: quando arriva una richiesta per l'elemento k in posizione i, OPT restituisce l'elemento scandendo i primi i elementi della lista ed effettua poi una riorganizzazione della lista per mezzo di scambi tra elementi contigui, in funzione della sequenza delle richieste future (che sono note per ipotesi ad OPT).

Notiamo che OPT riordina sempre gli elementi in modo tale da minimizzare il suo costo futuro. Chiaramente un tale algoritmo dotato di chiaroveggenza non esiste ma è utile ai fini dell'analisi per valutare le potenzialità di MTF.

Mostriamo ora che, sotto opportune condizioni, il costo di MTF non supera il doppio del costo di OPT.

A tal fine, consideriamo una sequenza arbitraria σ di m operazioni di ricerca su una lista di n elementi, dove le operazioni sono enumerate da 1 a m in base al loro ordine di esecuzione. Per $1 \leq i \leq m$, l'i-esima operazione accede a un elemento $\sigma(i) = k$ nella lista secondo l'Algoritmo 2.11: sia c_i la posizione di k nella lista di MTF e c'_i la posizione di k nella lista di OPT. Poiché vengono scanditi c_i elementi prima di k nella lista di MTF, e c'_i elementi prima di k nella lista di OPT, definiamo il costo delle m operazioni, per entrambi gli algoritmi, come

$$c_{MTF} = \sum_{i=1}^{m} c_i$$
$$c_{OPT} = \sum_{i=1}^{m} c'_i.$$

Possiamo allora dimostrare il teorema seguente, che ci fornisce una delimitazione superiore sull'aumento di costo in cui si incorre nel passare da OPT a MTF.

Teorema 2.7. Per ogni sequenza di ricerche σ, $c_{MTF}(\sigma) \leq 2c_{OPT}(\sigma)$.

Dimostrazione. Indichiamo con L_i e L^*_i le liste ottenute rispettivamente da MTF e da OPT dopo l'i-esima ricerca e, per ogni elemento x, siano $r_i(x)$ e $r^*_i(x)$ le posizioni di e nelle due liste. Inoltre, dati due elementi x, y, scriviamo $x <_i y$ se x precede y in L_i e $x <^*_i y$ se x precede y in L^*_i.

Se e è l'elemento ricercato con l'i-esima operazione, allora abbiamo che

- il costo della MTF è $c_i = r_{i-1}(e)$
- il costo di OPT è $c^*_i = r^*_{i-1}(e) + t_i$, dove t_i è il numero di scambi effettuati da OPT per riorganizzare la lista dopo avere acceduto a e

Supponiamo di aver appena eseguito l'i-esima operazione, che accede all'elemento k, e consideriamo le risultanti liste di MTF e OPT: un esempio di configurazione delle due liste in un certo istante è quello riportato nella Figura 2.6.

Presi due elementi distinti x e y in una delle due liste, questi devono occorrere anche nell'altra: diciamo che l'insieme $\{x, y\}$ è un'inversione quando l'ordine relativo di occorrenza è diverso nelle due liste, ovvero quando x occorre prima di y (non necessariamente in posizioni adiacenti) in una lista mentre y occorre prima di x nell'altra lista. Nel nostro esempio, $\{e_0, e_2\}$ è un'inversione, mentre $\{e_1, e_7\}$ non lo è.

Indichiamo con Φ_i il numero di inversioni tra le due liste dopo che è stata eseguita l'i-esima operazione e con Φ_0 il numero di inversioni all'inizio: evidentemente, dato che assumiamo che i due algoritmi operino a partire da due liste uguali, si ha che $\Phi_0 = 0$. Inoltre, per ogni $1 \leq i \leq m$ si ha che $0 \leq \Phi_i \leq \frac{n(n-1)}{2}$, in quanto $\Phi_i = 0$ se le due liste sono uguali mentre, se sono una in ordine inverso rispetto all'altra, si ha un'inversione per ognuna delle $\binom{n}{2}$ coppie di elementi.

Osserviamo che un singolo scambio tra due elementi contigui o introduce una nuova inversione o elimina una inversione già presente, per cui comporta una variazione del potenziale di ± 1.

Se l'i-esima operazione ricerca l'elemento e, possiamo definire i seguenti insiemi:

- $A = \{x : x <_{i-1} e \wedge x <^*_{i-1} e\}$ degli elementi che precedono e sia in L_{i-1} che in L^*_{i-1}
- $B = \{x : x <_{i-1} e \wedge e <^*_{i-1} x\}$ degli elementi che precedono e in L_{i-1} ma non in L^*_{i-1}
- $C = \{x : e <_{i-1} x \wedge x <^*_{i-1} e\}$ degli elementi che precedono e in L^*_{i-1} ma non in L_{i-1}
- $D = \{x : e <_{i-1} x \wedge e <^*_{i-1} x\}$ degli elementi che non precedono e né in L^*_{i-1} né in L_{i-1}

La struttura di L_{i-1} è quindi quella riportata sotto

L_{i-1}	$A \cup B$	e	$C \cup D$

mentre la struttura di L^*_{i-1} è

L_{i-1}	$A \cup C$	e	$B \cup D$

Ne deriva che $r_{i-1}(e) = |A| + |B| + 1$ e $r^*_{i-1}(e) = |A| + |C| + 1$.
MTF sposta e in cima alla lista, per cui si ha che

L_i	e	$A \cup B \cup C \cup D$

di conseguenza, MTF crea $|A|$ inversioni, eliminandone $|B|$ (si noti che le uniche inversioni modificate sono relative a e).

Le t_i operazioni di scambio effettuate da OPT, inoltre, creano al più t_i nuove inversioni. Ne deriva quindi che

$$\Phi_i - \Phi_{i-1} \leq |A| - |B| + t_i$$

e, per definizione,

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \leq r_{i-1}(e) + |A| - |B| + t_i$$

e, dato che $r_{i-1}(e) = |A| + |B| + 1$,

$$\begin{aligned}
\hat{c}_i &\leq r_{i-1}(e) + |A| - |B| + t_i \\
&= r_{i-1}(e) + |A| - (r_{i-1}(e) - |A| - 1) + t_i \\
&= 2|A| + 1 + t_i \\
&\leq 2r^*_{i-1}(e) + t_i \\
&\leq 2(r^*_{i-1}(e) + t_i) \\
&= 2c^*_i
\end{aligned}$$

in quanto $r^*_{i-1}(e) = |A| + |C| + 1 \geq |A| + 1$ e quindi $2|A| + 1 \leq 2(|A| + 1) \leq 2r^*_{i-1}(e)$.

Quindi,

$$\begin{aligned}
c_{MTF}(\sigma) = \sum_{i=1}^{m} c_i &= \sum_{i=1}^{m} (\hat{c}_i + \Phi_{i-1} - \Phi_i) \\
&= \sum_{i=1}^{m} \hat{c}_i + (\Phi_0 - \Phi_m) \\
&\leq 2 \sum_{i=1}^{m} c^*_i + (\Phi_0 - \Phi_m) \\
&= 2c_{OPT}(\sigma) + \Phi_0 - \Phi_m \\
&\leq 2c_{OPT}(\sigma)
\end{aligned}$$

in quanto, avendo assunto che le liste siano inizialmente uguali, si ha che $\Phi_0 = 0$, e inoltre $\Phi_m \geq 0$ per definizione. □

Se consideriamo ora il caso in cui le liste possono essere inizialmente diverse, allora il potenziale iniziale è $\Phi_0 = \Omega(n^2)$, in quanto nel caso peggiore le due liste sono invertite l'una rispetto all'altra, con $\frac{n(n-1)}{2}$ inversioni. Ne deriva che

$$c_{MTF}(\sigma) \leq 2c_{OPT}(\sigma) + \Omega(n^2)$$

ma $\Omega(n^2)$ è una costante rispetto a m (e diventa irrilevante al crescere di m), per cui MTF rimane 2-competitivo.

Si osservi che algoritmi diversi, simili a MTF, non ottengono un rapporto di competitività costante. Questo è il caso, ad esempio, di:

- TRANS (transpose): che scambia l'elemento acceduto con quello precedente, spostandolo di una posizione verso l'inizio della lista, se non è già il primo elemento della lista stessa
- FC (frequency count): che associa ad ogni elemento un contatore con il numero di volte in cui è stato acceduto, e mantiene la lista ordinata per valori non crescenti di tale contatore

Nel caso di TRANS, una sequenza che accede sempre all'ultimo elemento della lista avrebbe l'effetto di scambiare continuamente i due ultimi elementi tra loro. Ne conseguirebbe che $c_{TRANS}(\sigma) = mn$, mentre OPT potrebbe portare inizialmente i due elementi in cima alla lista e poi accedere ad essi, scambiandoli iterativamente tra loro, con costo $c_{OPT}(\sigma) = 2n + 2m$. Il rapporto di competitività conseguente è

$$\frac{c_{TRANS}(\sigma)}{c_{OPT}(\sigma)} = \frac{mn}{2(m+n)}$$

che tende a $\frac{n}{2}$ al crescere di m.

Nel caso di FC, consideriamo una sequenza che, per un valore $k \geq 2n$, è composta da k accessi al primo elemento, seguiti da $k-1$ accessi al secondo elemento, e così via, fino a $k+1-n$ accessi all'ultimo elemento: si noti che, in questo modo, la lista non è mai modificata da FC. Il costo totale di FC è allora

$$c_{FC}(\sigma) = k+2(k-1)+3(k-2)+\cdots+n(k+1-n) \geq (k-n)(1+2+\cdots+n) = (k-n)\frac{n(n+1)}{2} \geq \frac{kn^2}{4}$$

in quanto $(k-n) \geq \frac{k}{2}$. Sulla stessa sequenza, OPT potrebbe spostare ognuno degli n elementi in cima alla lista la prima volta in cui accede ad esso, con costo al più n , e accedere poi ad esso in tempo costante per tutte le al più k ricerche successive. Ne deriva che il costo complessivo è

$$c_{OPT}(\sigma) \leq n^2 + kn \leq \frac{kn}{2} + kn = \frac{3}{2}kn$$

Il rapporto di competitività risulta allora

$$\frac{c_{FC}(\sigma)}{c_{OPT}(\sigma)} \geq \frac{\frac{kn^2}{4}}{\frac{3}{2}kn} = \frac{1}{6}n = \Theta(n).$$