Algoritmi

Mattia Ceron

27 maggio 2022

Indice

1 Introduzione

1.1 Notazioni

Informalmente, un **algoritmo** è una procedura di calcolo ben definita che prende un certo valore, o un insieme di valori, come input e genera un valore, o un insieme di valori, come output. Un algoritmo è quindi una sequenza di passi computazionali che trasforma l'input in output.Un algoritmo si dice **corretto** se, per ogni istanza di input, termina con l'output corretto. Diciamo che un algoritmo corretto risolve il problema computazionale dato. Un algoritmo errato potrebbe non terminare affatto con qualche istanza di input o potrebbe terminare fornendo una soluzione diversa da quella desiderata.

Per rappresentare un algoritmo si utilizza uno pseudolinguaggio ovvero una descrizione in Inglese/Italiano che non deve rappresentare ambiguità. Prima di scrivere l'algoritmo ci si accorda sulle nomenclature.

Esempio Pseudocodice

```
Scrivi: "Inserisci la tua età: "
Se l'età è uguale o maggiore di 18:
    Scrivi "Sei maggiorenne"
Altrimenti:
    Scrivi "Non sei maggiorenne"
```

Questo è uno pseudo codice del blocco decisionale che verifica se un utente è maggiorenne

1.2 Efficienza di un Algoritmo

Per calcolare l'efficienza di un algoritmo ci serve una rappresentazione costante che al variare dei dati in input non cambi. Utilizzare come riferimento il tempo che impiega, le risorse che utilizza oppure il numero di istruzioni utilizzate sono tutte variabili influenzabili da calcolatore a calcolatore perciò possono variare. Un modo per calcolare l'efficienza è a complessità computazionale che è una funzione

$$Dimensione \rightarrow \mathbb{R}^{>0}$$

che dato una dimensione di un problema ti riesce ad esprimere in numeri quanto impiega tale algoritmo.

1.2.1 Esempio moltiplicazione tra matrici

Prima di iniziare con la stesura dello pseudocodice è necessario imporre delle convenzioni in modo da evitare incomprensioni, in questo esempio sfruttiamo il fatto che un'array parta dall'indice 1.

```
1  mult(A,B)
2  m <- cols(A)
3  l <- cols(b)
4  for i <- 1 to m
5    for j <- 1 to l
6        C[i][j] <- 0
7        for k <- 1 to n
8            C[i][j]=C[i][j] + A[i][k] * B[k][j]
```

Dopo aver scritto l'algoritmo è opportuno domandarsi quanto è efficiente, per farlo leggiamo le righe dall'ultima alla prima e determiniamone il costo di ognuna. **Il costo dell'ultima riga è c**.

linea	costo
1	0
2	0
3	0
4	0
5	(((c+1)n+3)l+2)m+1
6	((c+1)n+3)l+1
7	(c+1)n+3
8	(c+1)n+1
9	c

Tabella 1.1: Rappresentazione tabulare del costo associato ad una linea

Il risultato totale è:

$$cnlm + 3nlm + 3lm + 2m + 4$$

L'espressione ricavata è difficile da capire poiché iniziano ad esserci numerosi polinomi, però essendo che il termine più grande che "comanda" è **cnlm**, prenderemo quello come valore da confrontare.

$$3lm \leq 3nlm \leq cnlm$$

basta prendere una costante c e $cnlm$ è **approssimativamente** il valore da prendere in considerazione per constatare l'efficienza dell'algoritmo.

> Le costanti di solito non vengono impiegate nei confronti tra algoritmi di diverse grandezze, ma soltanto fra algoritmi dello stesso ordine.

Una differenza importante tra gli algoritmi è il modo in cui essi si comportano in base alla dimensione del problema

- lineare: se raddoppio la dimensione raddoppia la complessità;
- logaritmico: se raddoppio la dimensione aumenta di 1 passo la complessità;
- esponenziale: se raddoppio la dimensione, aumenta esponenzialmente di 1 la complessità;

Un algoritmo è preferibile con una rappresentazione logaritmica.

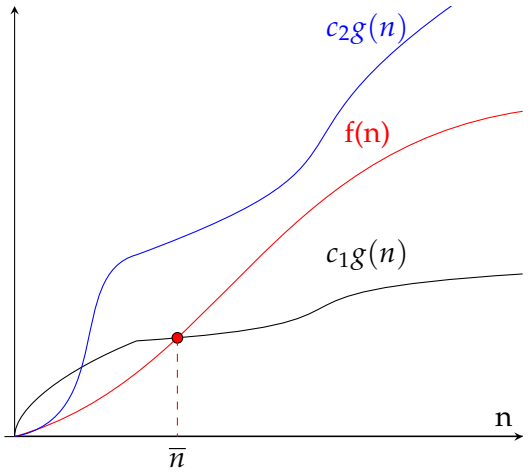
2 Notazione Asintotica

Un algoritmo in base al numero degli elementi da gestire può impiegare più tempo come nel caso della ricerca sequenziale di un valore all'interno di una lista che dipende dal numero di elementi che la compongono. Quando operiamo con dimensioni dell'input abbastanza grandi da rendere rilevante soltanto il tasso di crescita del tempo di esecuzione, si studia **l'efficienza asintotica** degli algoritmi. Le notazioni che utilizziamo per descrivere il tempo di esecuzione asintotico di un algoritmo sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbb{N} = 1, 2, 3, \ldots$.

2.1 Notazione Θ

Per una data funzione g(n), indichiamo con Θ(g(n)) l'insieme delle funzioni

$$\exists(c_1, c_2, \overline{n})\ \forall n \geq \overline{n}\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$



Per tutti i valori di n a destra di $\overline{n}$, il valore di f(n) coincide o sta sopra $c_1 g(n)$ e coincide o sta sotto $c_2 g(n)$.f(n) appartiene all'insieme $\Theta(g(n))$ se esistono delle costanti positive c_1 e c_2 tali che essa può essere "racchiusa" fra $c_1 g(n)$ e $c_2 g(n)$. Si dice che g(n) è un **limite asintoticamente stretto** per f(n).In altre parole per ogni $n \geq \overline{n}$, la funzione f(n) è uguale a g(n) a meno di un fattore costante. Dimostriamo che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Per farlo dobbiamo determinare le costanti positive c_1, c_2 in modo che

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

prendendo una qualsiasi n$\geq \overline{n}$ e dividendo per n^2 si ha

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

La disuguaglianza destra è resa valida per qualsiasi valore di n$\geq$ 1 scendendo $c_2 \geq 1/2$. Analogamente, la disuguaglianza sinistra può essere resa valida per qualsiasi valore di $n \geq 7$, scegliendo $c_1 1/14$. Si possono prendere anche altri costanti, ma la cosa importante è che **esista qualche scelta**.

2.2 Notazione O

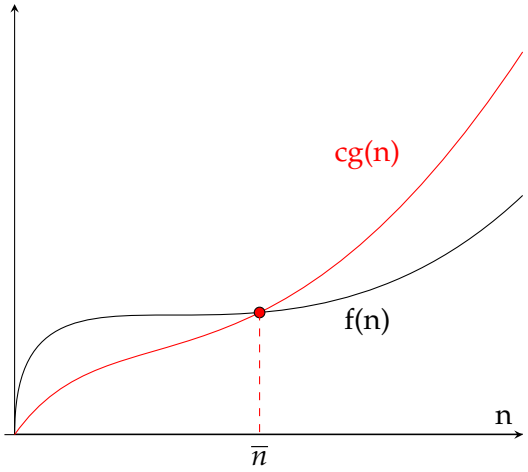
Il primo caso che andiamo a vedere è quando abbiamo un limite asintotico superiore, la convenzione utilizzata è:

$$f(n) \in O(g)$$

e descrive che f asintoticamente non supera g oppure che f non cresce più in fretta di g quando trascuriamo le costanti. La definizione fiscale è la seguente:

$$\exists c\, \exists \overline{n}\, \forall n > \overline{n}\, f(n) \leq cg(n)$$

esiste un c, esiste un $\overline{n}$ che per tutte le n$>$ $\overline{n}$, la funzione f(n) è sempre minore di una costante moltiplicata alla funzione g(n).



come si può vedere f sta sotto a g per una costante c. Per qualsiasi valore n a destra di $\overline{n}$ il valore della funzione $f(n)$ coincide o sta sotto a $g(n)$. Questa notazione definisce un **limite asintotico superiore**. Può sembrare strano che si trovino affermazioni del tipo $n = O(n^2)$ poiché non pare un limite asintotico in senso stretto. La definizione che abbiamo fornito per $f(n) = O(g(n))$ semplicemente afferma che per qualsiasi costante moltiplicata a g(n) è un limite superiore rispetto ad f(n), senza specificare quanto stretto debba essere il limite.

2.2.1 Esempi notazione O(g)

- $4n^3 \in O(2n^2)$: non è valida poiché ricordando le proprietà algebriche dei polinomi non esiste nessuna costante c tale che $a^x < ca^{x-k}$ con k> 0
- $n \in O(n^2)$: è valida poiché il grado di n è minore di quello di n^2;

Questa nozione non è molto precisa, per capirlo utilizziamo sempre la ricerca di un elemento all'interno di un array non ordinato. Il caso peggiore è ovviamente dato da n(*nel caso in cui l'elemento da cercare sia inesistente*). Si scrive $f(n)$=$O(g(n))$ per indicare che una funzione f(n) è un membro dell'insieme O(g(n)). Notare che se $f(n) \in \Theta(g(n))$ allora $f(n)$ = O(g(n)).

2.3 Notazione Ω

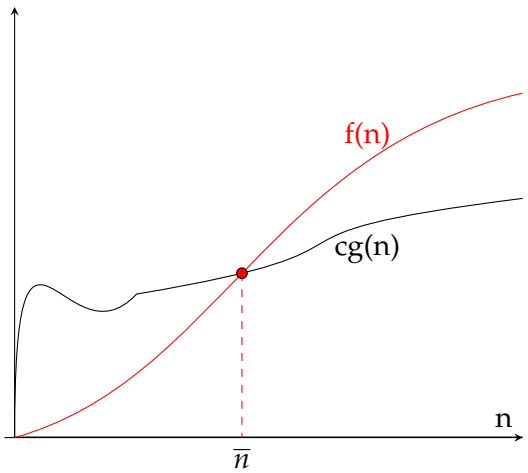
Se O dà un limite superiore asintotico,Ω fornisce un **limite inferiore asintotico**, la notazione è:

$$f(n) \in \Omega(g)$$

descrive che la funzione f asintoticamente non sarà mai "al di sotto" di g moltiplicata per una costante. In termini matematici questa descrizione può essere riscritta come

$$\exists c > 0 \; \exists \overline{n} \; \forall n > \overline{n} \; f(n) \geq cg(n)$$

Esiste un c maggiore di zero, esiste una $\overline{n}$ tale che per tutte le n¿$\overline{n}$ la funzione f sarà maggiore della funzione g moltiplicata per una costante c. Quando diciamo che il tempo di esecuzione di un algoritmo è $\Omega(n)$, intendiamo che per qualsiasi valore di n, il tempo di esecuzione di quell'input è almeno pari a una costante moltiplicata per g(n), con n sufficientemente grande. Ovvero stiamo assegnando un limite inferiore al tempo di esecuzione nel **caso migliore di un algoritmo**.



2.3.1 Esercizi

Dimostrare la seguente affermazione:

$$f \in O(g) \iff g \in \Omega(f)$$

Utilizziamo le definizioni della notazioni specificate:

$$\exists c \; \exists \overline{n} \forall n > \overline{n} f(n) \leq cg(n) \tag{2.1}$$

siano c ed $\overline{n}$ i due valori che soddisfano la formula, vogliamo dimostrare che vale

$$\exists c_1 \; \exists \overline{n_1} \forall n > \overline{n_1} g(n) \geq c'g(n)$$

dividiamo nella disequazione 2.1 per c:

$$\frac{f(n)}{c} \leq g(n)$$

scrivendola in un altro modo

$$g(n) \geq \frac{f(n)}{c}$$

Scegliendo come costante $\mathbf{c'}=\frac{1}{c}$ si ha

$$g(n) \geq c'f(n)$$

Altro esercizio

Dimostra la seguente affermazione:

$$f_1 \in O(g) \wedge f_2 \in O(g) \rightarrow f_1 + f_2 \in O(g)$$

Utilizziamo la definizione di O(g) per le funzioni f_1, f_2:

- $\exists c_1 \; \exists \overline{n_1} \forall n > \overline{n} f(n) \leq c_1 g(n)$;
- $\exists c_2 \; \exists \overline{n_2} \forall n > \overline{n} f(n) \leq c_2 g(n)$;

Quindi abbiamo

$$\exists c_1 \; \exists c_2 \; \exists \overline{n_1} \; \overline{n_2} (\forall n \geq \overline{n_1} f(n) \leq c_1 g(n) \wedge (\forall n > \overline{n} f(n) \leq c_2 g(n))$$

Per dimostrare questa dobbiamo prendere un valore ognuno di $c_1, c_2, \overline{n_1}, \overline{n_2}$ e ricondurci alla forma:

$$\exists c \exists \overline{n} \forall n \geq \overline{n} f_1(n) + f_2(n) \leq cg(n)$$

Per $\overline{n}$ intendiamo il punto dove le funzioni f_1, f_2 diventano minori di g per una costante, ovviamente prendiamo $\overline{n}$=max($\overline{n_1}$,$\overline{n_2}$) perché se sia f_1 che f_2 seguono la notazione O(g(n)) significa che tutte e due per qualche $\overline{n_x}$ siano inferiori di $c_x g(n)$. Avendo scelto per le due

funzioni $\overline{n_1}, \overline{n_2}$ come punti in cui le rispettive $\forall n \geq \overline{n_1} f_1 \leq c_1 g(n)$ e $\forall n \geq \overline{n_2} c_2 g(n)$ prendiamo come valore $\overline{n}$ il massimo fra $\overline{n_1}$ e $\overline{n_2}$. Determinata $\overline{n}$ abbiamo dimostrato che per qualche valore le due funzioni coesistono per cui proviamo ad unire le disequazioni:

$$\begin{gathered} f_1(n) \leq c_1 g(n) \\ f_2(n) \leq c_2 g(n) \\ f_1(n) + f_2(n) \leq \underbrace{(c_1 + c_2)}_{c} g(n) \end{gathered}$$

Quindi $c_1 + c_2$ rappresenta la nostra **c** e quindi l'affermazione era corretta. Le tre notazioni citate possono essere riscritte come:

- f $\in$ O(g) $\iff$ $\lim_{n\to\infty} \frac{f}{g} < \infty$;
- f $\in \Theta$(g) $\iff$ $\lim_{n\to\infty} \frac{f}{g}$ compresa tra 0 e ∞;
- f $\in \Omega$(g) $\iff$ $\lim_{n\to\infty} \frac{f}{g} > 0$;

2.4 Relazione tra Algoritmo e Problema

Sia **A** un algoritmo

- A $\in$ O(f): la complessità di A è una funzione g $\in O(f)$, peggio di f non può fare;
- A $\in \Omega$(f): esiste uno schema di input tale che la complessità di A sullo schema è una funzione g $\in \Omega$(f), l'algoritmo non può fare meglio di f;
- A $\in \Theta$(f) la mia stima è accurata;

Sia **P** un problema:

- P $\in$ O(f): $\exists A \in O(f)$ che risolve P, esiste un modo per risolvere il problema, non esiste un tempo più basso di f;
- P $\in \Omega$(f) $\forall A$ che risolve P, A $\in \omega(f)$, non esistono algoritmi che fanno meglio di f

3 Divide et Impera. Risoluzione delle equazioni di ricorrenza

- **Divide**: questo passo divide il problema in un insieme di sottoproblemi che sono istanze più piccole dello stesso problema;
- **Impera**: i sottoproblemi vengono risolti in modo ricorsivo. Quando hanno una dimensione sufficientemente piccola, essi vengono risolti direttamente;
- **Combina**: le soluzioni dei sottoproblemi vengono combinate per generare la soluzione completa;

Quando i sottoproblemi sono abbastanza grandi da essere risolti in modo ricorsivo, si ha il **caso ricorsivo**, mentre quando hanno dimensioni sufficientemente piccoli che la ricorsione ha "toccato il fondo" si ha il **caso base**.

3.1 Ricorrenza

Una **ricorrenza** è un'equazione o disequazione che descrive una funzione in termini del suo valore con input più piccoli(Equazione 3.1). Consideriamo il caso del fattoriale

```
1 fatt(n)
2     if n=0
3         ret 1
4     else
5         ret n*fatt(n-1)
```

bisogna trovare la dimensione del problema, nel caso in cui siamo in grado di determinarlo possiamo calcolare la sua complessità. Definisco **T(n)** la complessità in base al tempo. Se n=0 l'algoritmo termina in un tempo costante, altrimenti impiegherà 1+T(n-1).

$$\begin{cases} 1 \; n = 0 \\ 1 + T(n-1) \end{cases} \tag{3.1}$$

Proviamo a determinarne la complessità:

$$T(n) = 1 + T(n-1)$$

sfruttando il fatto che T(n) non prenda ancora il ramo else si può riscrivere come

$$\begin{aligned} T(n) &= 1+1+T(n-2) \\ &= 1+1+1+T(n-3) \\ &\cdots\cdots \\ &= \underbrace{1+1+1+\ldots+1}_{i} + T(n-i) = n + T(0) = n+1 \end{aligned}$$

quindi il nostro risultato appartiene a Θ(n). Abbiamo utilizzato un abuso della notazione, le classi di equivalenza Θ li posso definire come oggetti. Misurare la complessità col valore di un oggetto è rischiosa, poiché con n abbastanza grande l'utilizzo della memoria è esponenziale. Esistono 3 metodi per risolvere le ricorrenze:

- **Metodo di sostituzione**: ipotizziamo un limite e poi usiamo l'induzione matematica per dimostrare che la nostra ipotesi è corretta;
- **Albero di ricorsione**: converte la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione;
- **Metodo dell'esperto**: prevede di risolvere determinati tipi di ricorrenze con determinate caratteristiche.;

3.2 Metodo di sostituzione

Il metodo di sostituzione per risolvere le ricorrenze richiede due passi:

- Ipotizzare la forma della soluzione;
- Usare l'induzione matematica per trovare le costanti e dimostrare che la soluzione funziona.

Il nome del metodo deriva dalla sostituzione della soluzione ipotizzata al posto della funzione. Questo metodo è potente, ma ovviamente può essere solamente applicato nei casi in cui sia facile immaginare la forma della soluzione. Consideriamo i seguenti 3 casi e proviamo a calcolarne la complessità:

1. $\boldsymbol{T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n}$.
   Per procedere al calcolo, dobbiamo considerare che quando n è piccolo la complessità è costante, il caso base c'è sempre quindi viene dato per implicito. Per risolvere questa equazione di ricorrenza, ci viene data la soluzione al problema e utilizzando il metodo di sostituzione si riesce a risolvere l'equazione. La soluzione fornitaci è $T(n) \in O(n \log n)$. Quindi ricordando la notazione O, stiamo cercando qualche c che soddisfi la condizione $T(n) \leq cn \log n$. Il metodo consiste nel dimostrare che T(n)$\leq cn \log n$. Supponiamo che questo limite sia valido per $\lfloor n/2 \rfloor$, ovvero che

$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$. Applico la sostituzione sulla definizione citata sopra

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2\left(c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor\right) + n \end{aligned}$$

Quindi se $2\left(c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor\right) + n$ è maggiore di T(n) significa anche che $2\left(c\frac{n}{2} \log \frac{n}{2}\right) + n$ sarà maggiore di T(n), essendo che $\left\lfloor \frac{n}{2} \right\rfloor \leq \frac{n}{2}$. Quindi si può riscrivere come

$$\begin{aligned} &2\left(c\frac{n}{2} \log \frac{n}{2}\right) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - (cn \log 2 - n) \end{aligned}$$

Arrivati a questo punto devo domandarmi quando $cn \log n - (cn \log 2 - n) \leq cn \log n$:

$$\begin{aligned} cn \log n - (cn \log 2 - n) &\leq cn \log n \\ cn \log 2 - n &\geq 0 \\ n(c \log 2 - 1) &\geq 0 \\ c &\geq \frac{1}{\log 2} \end{aligned}$$

2. $\boldsymbol{T(n) = T(\lfloor \frac{n}{2} \rfloor + T(\lceil \frac{n}{2} \rceil) + 1}$.
Ci sono casi in cui è possibile ipotizzare correttamente il limite asintotico, ma i calcoli matematici non tornano nell'induzione. Di solito, il problema è che l'ipotesi effettuata non è abbastanza forte per determinarne il limite esatto. Spesso basta correggere l'ipotesi sottraendo un termine di ordine inferiore, come in questo caso. La soluzione fornita è $T(n) \in O(n)$.Per definizione esiste qualche c tale per cui $T(n) \leq cn$ Prima di iniziare è opportuno ricordare questa proprietà algebrica:

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$$

Proseguo con i calcoli:

$$\begin{aligned} T(n) &\leq cn \\ T(n) &\leq c \overbrace{\left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil}^{n} + 1 \\ &= cn + 1 \end{aligned}$$

Come prima devo dimostrare che $cn + 1 < cn$, ma ciò non è **mai vero**.Intuitivamente la nostra ipotesi è quasi esatta: non vale soltanto a causa della costante 1, un termine

di ordine inferiore. Per ovviare a questo problema consideriamo la soluzione O(cn-b):

$$\begin{aligned} T(n) &\leq cn - b \\ T(n) &\leq c\left\lceil \frac{n}{2} \right\rceil - b + c\left\lceil \frac{n}{2} \right\rceil - b + 1 \\ &= cn - 2b + 1 \\ &= cn - b - (b-1) \end{aligned}$$

Quindi stiamo valutando che $cn - b - (b-1)$ sia minore di $cn - b$:

$$cn - b - (b-1) \leq cn - b \rightarrow b \geq 1$$

Proviamo a dimostrare che la soluzione sia anche T(n) $\in \Omega$(n). Seguendo la definizione esiste una costante c tale per cui $T(n) \geq cn$. Sostituendo nell'equazione ottengo:

$$\begin{aligned} T(n) &\geq cn \\ T(n) &\geq c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1 \geq cn \end{aligned}$$

Ciò vale sempre.

3. $\boldsymbol{T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n}$.
In questo caso proviamo a procedere per induzione

$$\begin{aligned} &= n + 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) \\ &= n + 3\left(\left\lfloor \frac{\frac{n}{4}}{4} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor\right) \\ &= n + 3\left(\left\lfloor \frac{n}{4^2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor\right) \\ &= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 3^2 T\left(\frac{n}{4^2}\right) \\ &= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 3^2\left(3T\left(\left\lfloor \frac{\lfloor \frac{n}{4^2} \rfloor}{4} \right\rfloor + \left\lfloor \frac{n}{4^2} \right\rfloor\right)\right) \\ &= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 3^2\left\lfloor \frac{n}{4^2} \right\rfloor + \ldots + 3^{i-1}\left\lfloor \frac{n}{4^{i-1}} \right\rfloor + 3^i T\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right) \end{aligned}$$

Il problema come si può vedere inizia a diventare complesso, e ci si pone una domanda: *quanto deve essere grande i per arrivare a un caso base?*

$$\begin{aligned} \frac{n}{4^i} &\leq 1 \\ 4^i &\geq n \\ i &\geq \log_4 n \end{aligned}$$

Quindi

$$\leq n + 3\left\lfloor \frac{n}{4} \right\rfloor + 3^2\left\lfloor \frac{n}{4^2} \right\rfloor + \ldots + 3^{i-1}\left\lfloor \frac{n}{4^{i-1}} \right\rfloor + 3^{\log 4^{n-1}}\left\lfloor \frac{n}{4^{\log_4 n-1}} \right\rfloor$$

$$\leq n + \underbrace{\frac{3}{4}n + \left(\frac{3}{4}\right)^2 n + \left(\frac{3}{4}\right)^{\log_4 n-1}}_{\text{serie geometrica con q=}\frac{3}{4}} * c$$

$$\leq \frac{1}{1-\frac{3}{4}} + c3^{\log_4 n}$$

$$= 4n + c3^{\log_4 n}$$

$$3^{\log_4 n} = n^{log_n(3^{log_4 n})}$$

$$= n^{log_4 n * log_n 3}$$

$$= n^{log_4 3}$$

Nella terzultima riga ho utilizzato la formula del cambio di base, riguardo alla base da scegliere scelgo quella più conveniente poiché l'ordine di grandezza non cambia. Quindi la complessità di questo problema è $\boldsymbol{\Theta(n)}$. Eravamo partiti da

$$T(n) = 3T\left(\left\lceil \frac{n}{4} \right\rceil\right) + n$$

e i numeri che abbiamo qui, li abbiamo anche nel risultato

$$T(n) = \alpha T\left(\left\lceil \frac{n}{b} \right\rceil\right)$$

.

3.3 Metodo dell'albero di ricorsione

In un **albero di ricorsione** ogni nodo rappresenta il costo di un singolo sottoproblema da qualche parte nell'insieme delle chiamate ricorsive di funzione. Sommiamo i costi all'interno di ogni livello dell'albero per ottenere un insieme di costi per livello; poi sommiamo tutti i costi per livello per determinare il costo totale di tutti i livelli. Consideriamo la seguente equazione di ricorrenza : $T(n) = T(n/3) + T(2n/3) + n$. Proviamo ora a rappresentare due livelli dell'albero di ricorsione:

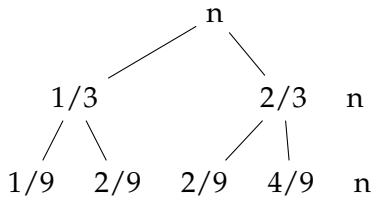


Figura 3.1: albero di ricorsione

Ogni livello ha un costo uguale ad n. Il più lungo cammino semplice dalla radice a una foglia(nodo senza figli) è $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$. Quindi poiché $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, intuitivamente l'altezza dell'albero è $\log_{3/2} n$.

3.4 Metodo Principale (Master theorem)

Sia $a \geq 1$ e $b \geq 1$,$f(n)$ una funzione e $T(n)$ definita per gli interi non negativi

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3.2}$$

La ricorrenza (3.2) descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sottoproblemi, ciascuno di dimensione n/b. dove con $\frac{n}{b}$ interpretiamo anche le approssimazioni $\lceil \frac{n}{b} \rceil$ e $\lfloor \frac{n}{b} \rfloor$. Allora abbiamo tre casi:

1. Se $f(n) \in O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
   In altre parole prendo f(n) e lo confronto con $n^{\log_b a}$, se è più grande questo' ultimo, allora la soluzione dell'equazione di ricorrenza è $\Theta(n^{\log_b a})$.

   $$f(n) * n^{\epsilon} < n^{\log_b a}$$

2. Se f(n) $\in \Theta(n^{\log_b a})$ T(n) = $\Theta(f(n) \log n)$. Se

3. Se f(n) $\in \Omega(n^{\log_b a + e})$ T(n)=$\Theta(f(n))$ con $\epsilon > 0$. Confronto f(n) con $n^{\log_b a}$ e se quest'ultima è polinomialmente più piccola di f(n) il teorema vale

   $$f(n) > n^{\log_b a + \epsilon}$$

   Però bisogna verificare questo caso

   $$af\left(\frac{n}{b}\right) \leq cf(n)$$

   con $c < 1, n \geq \overline{n}$

In ciascuno dei tre casi, confrontiamo la funzione $f(n)$ con la funzione $n^{\log_b a}$. Se, come nel caso 1, la funzione $n^{\log_b a}$ è più grande, allora la soluzione è $T(n) = \Theta(n^{\log_b a})$. Se, come nel caso 3, la funzione $f(n)$ è la più grande, allora la soluzione è $T(n) = \Theta(f(n))$.Se, come nel caso 2, le due funzioni hanno la stessa dimensione, moltiplichiamo per un fattore logaritmico e la soluzione è $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

3.4.1 Applicazione del metodo dell'esperto

Proviamo a determinare nei seguenti 4 esempi se il metodo del maestro può essere applicato per determinare la complessità della ricorrenza.

Considero

T(n)=$9T\left(\frac{n}{3}\right) + n$.

La ricorrenza si presenta con $a = 9$, $b = 3$, $f(n) = n$, e quindi $n^{\log_b a} = n^2$. Quindi devo prendere un qualsiasi ϵ tale per cui una delle 3 condizioni del teorema valga. Poiché $n^{\log_b a}$ è più grande di f(n)=n, posso scegliere di utilizzare il primo caso del metodo principale

$$n \in O(n^{2-\epsilon})$$

quindi un epsilon tale per cui

$$\begin{gathered} n > n^{2-\epsilon} \\ 1 > 2 - \epsilon \\ \epsilon < 1 \end{gathered}$$

perciò prendendo $\epsilon = \frac{1}{2}$ la condizione

$$n \in O(n^{\frac{3}{2}})$$

è verificata.

Un altro modo di operare invece è quello di considerare che $f(n) = n^{\log_3 9-\epsilon}$, dove $\epsilon = 1$, possiamo quindi applicare il caso 1 del teorema dell'esperto e concludere che la soluzione è $T(n) = \Theta(n^2)$.

Adesso consideriamo

T(n)=$T\left(\frac{2}{3}n\right) + 1$.

I dati di questo problema di ricorrenza sono $a = 1$, $b = \frac{3}{2}$, $f(n) = 1$, mentre per $n^{\log_b a}$ si ha $n^{\log_{\frac{2}{3}} 1} = n^0 = 1$. Dato che $n^{\log_b a} = f(n)$ si applica il secondo caso e perciò $T(n) = \Theta(1 * \log n) = (\log n)$.

Proseguiamo con la ricorrenza

T(n)=$3T\left(\frac{n}{4}\right) + n \log n$.

I dati del problema sono $a = 3$, $b = 4$, $f(n) = n \log n$ da cui trovo che $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$. Dato che $n^{\log_b a}$ è minore di $f(n) = n \log n$ è giusto pensare di utilizzare il caso 3 se riusciamo a dimostrare che f(n) soddisfa la condizione di regolarità $af(n/b) \leq cf(n)$.

$$\begin{gathered} 3\left(\frac{n}{4} \log \frac{n}{4}\right) \leq cn \log n \\ \frac{3}{4} \log \frac{n}{4} \leq c \log n \\ \frac{3}{4} (\log n - \log 4) \leq \log n \end{gathered}$$

Scegliendo $c = \frac{3}{4}$, la disequazione sarà verificata: $\frac{3}{4} \log \frac{n}{4} \leq \frac{3}{4} \log n$. Un altro modo di operare è quello di utilizzare la definizione di limite asintotico Ω

$$n \log n \in \Omega(n^{\log_4 3+\epsilon})$$

Sto cercando un ϵ tale che

$$n \log n > n^{\log_4 3+\epsilon}$$

Quindi per grado di funzioni

$$\epsilon + \log_4 3 < 1$$

da cui

$$\epsilon < 1 - \log_4 3$$

Scegliendo $\epsilon = \frac{1-\log_4 3}{2}$ si ha

$$n \log n \in \Omega\left(n^{\log_4 3+\frac{1-\log_4 3}{2}}\right)$$

Continuiamo con l'equazione

T(n)=2T$\left(\frac{n}{2}\right) + n \log n$.

I dati di questo problema sono $a = 2$, $b = 2$, $f(n) = n \log n$ e quindi $n^{\log_b a} = n$. Potremmo ritenere che si possa applicare il caso 3, in quanto f(n)= $n \log n$ è asintoticamente più grande di $n^{\log_b a} = n$; il problema è che non è **polinomialmente più grande**. Se dovessimo continuare con i calcoli

$$f(n) > n^{\log_b a+\epsilon}$$
$$n \log n > n^{1+\epsilon}$$
$$n \log n > n * n^{\epsilon}$$
$$\log n > n^{\epsilon}$$

e ciò non è mai vero per qualsiasi $\epsilon > 0$, quindi il teorema non può essere applicato.

Procediamo con un' altro esempio

T(n)=8$T(n/2) + n^2$

I dati dell'equazione di ricorrenza equivalgono ad $a = 8$,$b = 2$, $f(n) = n^2$ e $n^{\log_b a} = n^3$. Dato che f(n)¡$n^{\log_b a}$ è ragionevole utilizzare il metodo 1, essendo più precisi $f(n) = O(n^{\log_b a-\epsilon} = O(n^{3-1}$, scegliendo un $\epsilon = 1$ la richiesta è stata soddisfatta, quindi T(n)=$\Theta(n^3)$.

Ultimo esempio

T(n)=2$T(n/2) + n$

I dati sono $a = 2$, $b = 2$, $f(n) = n$ e $n^{\log_b a} = n^{\log_2 2} = n$. $f(n)$ ha lo stesso grado polinomiale di $n^{\log_b a}$ quindi si procede con il secondo caso e perciò $T(n) = \Theta(n \log n)$.

4 Problemi di Ordinamento

- **INPUT**: sequenza $a_1, \ldots, a_n$ di oggetti su cui è definita una relazione di ordinamento totale[1];
- **OUTPUT** una permutazione $(a'_1, \ldots, a'_n)$ di $(a_1, \ldots, a_n)$ tale che $\forall ij\ i < j\ a'_i < a'_j$, cioè il risultato dell'ordinamento sarà una sequenza di oggetti in ordine crescente;

I numeri da ordinare sono anche detti **chiavi**. Il problema dell'ordinamento non potrà mai fare al di sotto di n: $\boldsymbol{\Omega(n)}$

4.1 Insertion Sort

Il tempo richiesto dall'Insertion Sort può variare in base a due elementi: la lunghezza dei dati in input e la disposizione degli elementi all'interno dell'array di partenza. Nel primo caso più aumenta il numero di elementi che dobbiamo ordinare più impiegherà l'algoritmo, nel secondo caso invece se i dati che ci sono forniti in input sono già in forma ordinata allora l'algoritmo impiegherà un tempo inferiore rispetto ad una sequenza di elementi non ordinati. L'algoritmo ordina gli elementi **sul posto**: i numeri sono risistemati all'interno dell'array A avendo, in ogni istante, al più un numero costante di essi memorizzati all'esterno dell'array. Una delle sue caratteristiche è la **proprietà invariantiva**: dopo n iterazione, i primi n elementi sono ordinati.

4.1.1 PseudoCodice

Prende come parametro un array $\mathbf{A}[1 \ldots n]$ contenente una sequenza di lunghezza n che deve essere ordinata.

```
1 INSERTION-SORT(A)
2  for j <- 2 to length(A)
3       KEY <- A[j]
4       i <- j-1
5       while i>0 AND A[i]>KEY
6            A[i+1]<-A[i]
7            i <- i-1
8       A[i+1]<-KEY
```

Consideriamo un esempio A=$\langle 5, 2, 4, 6, 1, 3 \rangle$, l'indice j si sposta da destra a sinistra su tutto l'array, ad ogni iterazione del ciclo **for** più esterno, l'elemento A[j] è copiato fuori dall'array, quindi cominciando dalla posizione j-1, gli elementi sono spostati ad uno ad uno in una posizione a destra finché non viene trovata la posizione per A[j].

[1] Dati due oggetti qualsiasi siamo in grado di confrontarli: maggiore, minore o uguale.

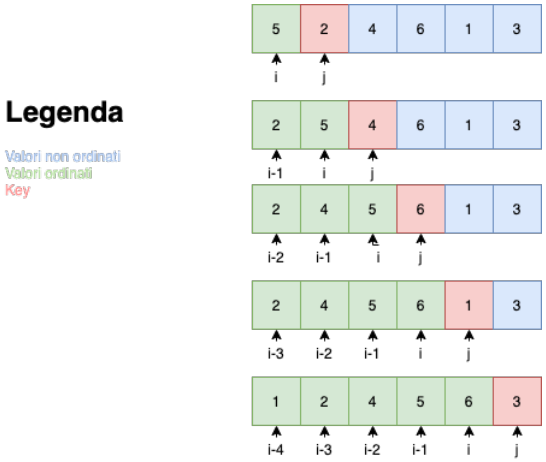


Figura 4.1: Funzionamento Insertion Sort

4.1.2 Analisi dell'algoritmo

Nel caso in cui l'array che viene fornito in input è già ordinato. Il caso peggiore invece è quando i dati di partenza sono ordinati in modo decrescente poiché dobbiamo confrontare ogni elemento A[j] con ogni elemento dell'intero sottoarray ordinato A[1..j-1] e quindi avremo bisogno di due cicli innestati con complessità finale dell'algoritmo. Il caso medio la maggior parte delle volte equivale a quello peggiore e anche in questo caso ciò avviene:

- **Caso peggiore**: O(n^2), ordinato decrescente;
- **Caso medio**: O(n^2);
- **Caso Ottimo**: O(n), cioè tutto ordinato;

Le caratteristiche principale dell'Insertion Sort sono due: l'**ordinamento in loco** e la **stabilità**. L'ordinamento in loco significa che l'algoritmo utilizza una memoria aggiuntiva costante anche quando varia l'input, ciò non sarà molto scontato negli algoritmi che vedremo successivamente. La stabilità invece determina che l'algoritmo non cambia l'ordine di oggetti uguali.

4.2 Merge Sort

Molti problemi sono più facili da risolvere riducendoli in piccoli sotto problemi, questo modo di operare prende il nome di **Divide et Impera**. Questo paradigma prevede tre passi a ogni livello di ricorsione:

- **Divide** il problema in più sotto problemi che sono istanze più piccole dello stesso problema;
- **Impera**: risolve i sotto problemi in modo ricorsivo. Quando hanno una dimensione sufficientemente piccola, essi vengono risolti direttamente;

- **Combina**: combina le soluzioni dei sotto problemi per generare la soluzione del problema originale;

Anche il MergeSort applica questo paradigma e opera nel seguente modo:

- **Divide**: divide la sequenza degli n elementi da ordinare in due sottosequenze di n/2 elementi ciascuna;
- **Impera**: ordina le due sotto sequenze utilizzando in modo ricorsivo il MergeSort;
- **Combina**: unisce le soluzioni per generare la sequenza ordinata;

4.2.1 PseudoCodice

```
1   MERGESORT(A, p, r)
2       B <- nuovo array di dimensione r-p+1
3       IF p < r
4           q <- (p+r)/2
5           MERGESORT(A,p,q)
6           MERGESORT(A,q+1,r)
7           MERGE(A,p,q,r)
```

IL MERGESORT continua a dividere l'array di partenza finché non si raggiungono sottoarray di 1 valore($p = r$), quando ciò accade le chiamate a MERGE che si occupa di ordinare gli array forniti in input iniziano ad essere invocate. La dimensione dei sottoarray raddoppia ad ogni chiamata di MERGE, fino ad arrivare alla soluzione generale. Una illustrazione è data dalla figura 4.3.

```
1   MERGE(A, p, q, r)
2       i <- 1
3       j <- p
4       k <- q+1
5       while j<= q OR k<=r
6           IF j<= q AND (k>r OR A[J] <= A[K])
7               B[i]<- A[j]
8               j++
9           ELSE
10              B[i]<-A[k]
11              k++
12          i++
13      FOR i<- 1 TO r-p+1
14          A[p+i-1] <- B[i]
```

In dettaglio la procedura MERGE opera nel seguente modo: nelle riga 2 assegno ad i il valore 1 che rappresenta il primo indice dell'array B che conterrà gli elementi in ordine, successivamente nelle righe 3 4 inizializziamo i due indici che scorreranno rispettivamente la prima metà dell'array A e la seconda. Il while della riga 5 serve a scorrere tutte e due le metà di

ciascuno sotto array. Dalla riga 6 alla riga 12 ci occupiamo di determinare l'ordine degli elementi: nel primo if si controlla la prima metà del sotto array con la condizione $j <= q$ e la seconda parte dell'AND serve per capire se l'indice di destra ha già scorso il sotto array di destra che determinerebbe che i dati di sinistra sono tutti maggiori di quelli di sinistra oppure che l'elemento puntato da j sia minore di quello di k. Quindi se l'if a riga 6 è vero inseriamo nell'array B A[j] e aumentiamo j, altrimenti inseriamo A[k] e aumentiamo k, successivamente aumentiamo i di 1. Continuiamo così finché i due indici non hanno scorso tutti i due sotto array, quando ciò avviene si cicla per tutto B e si inserisce in A partendo dalla posizione p+i-1-esima l'elemento i-esimo di B, questo perché l'indice p potrebbe essere diverso da 1.

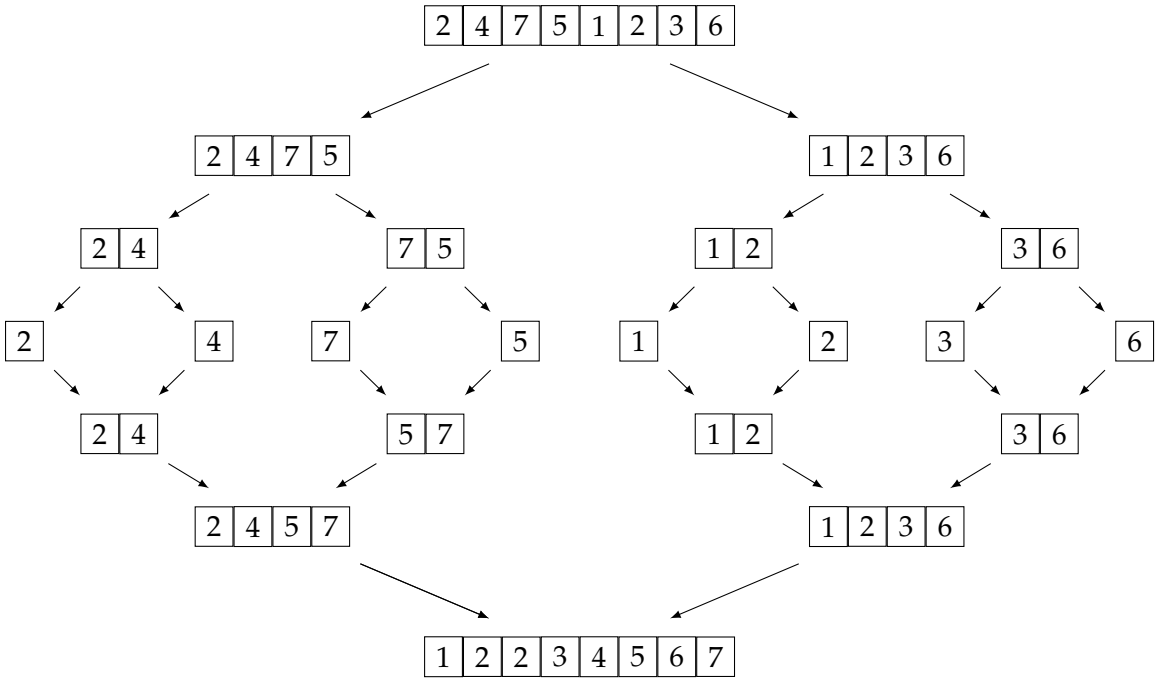


Figura 4.2: Rappresentazione grafica dell'algoritmo MergeSort: l'array iniziale continua a essere diviso in 2 parti uguali fino a quando gli array generati dalla divisione sono formati da un solo elemento che è il caso base. Arrivati a ciò gli array vengono ordinati ricorsivamente fin quando la soluzione corrisponde all'array iniziale, ma ordinato

4.2.2 Analisi dell'algoritmo

A differenza dell' Insertion Sort, il Merge Sort **non opera in loco**, poiché ad ogni chiamata del metodo *Merge* viene creato un array temporaneo, lo spazio aggiuntivo dato da questa variabile è proporzionale alla grandezza del problema dato in input. L'algoritmo, invece, è stabile poiché l'ordine degli elementi uguali non cambia. Per calcolare la complessità computazionale del Mergesort partiamo prima calcolando quella del Merge: le **righe 2-4** hanno una complessità costante poiché si sta soltanto inizializzando delle variabili. Continuando con il ciclo che è nel range delle righe 5-13: le operazioni che vengono eseguite sono dei controlli

oppure degli incrementi di variabili e quindi sono costanti, quindi ci dobbiamo focalizzare sul numero di volte che il ciclo viene ripetuto e ciò è n volte. Infine le **righe 13-14** hanno una complessità costante n. Sommando i fattori arriviamo a determinare che la complessità totale del Merge è $\Theta(n)$. Il MergeSort inizialmente calcola il centro dell'array e ciò ha complessità costante, poi risolve in modo ricorsivo i due sotto problemi, ciascuno di dimensione $n/2$, ciò contribuisce con $2T(n/2)$ al problema. Sommato alla procedura del Merge ci riconduciamo all'equazione di ricorrenza $T(n) = 2T(n/2) + n$. Utilizzando il teorema dell'esperto abbiamo che $a = 2$,$b = 2$,$f(n) = n$, $n^{\log_b a} = n$. $f(n)$ e $n^{\log_b a}$ hanno lo stesso ordine polinomiale perciò ci troviamo nel secondo caso del teorema dell'esperto. La soluzione è quindi $\Theta(n \log n)$. L'algoritmo non ha caso migliore e peggiore poiché comunque dividerebbe in qualsiasi caso l'array in due sottogruppi fino a quando non si arrivi ad un unico elemento, ciò può essere ricondotto ad un albero di decisione che come spiegheremo nella sezione dell' Heapsort ha $\log n$ livelli e dato che ogni livello impiega un tempo n cioè la somma dei tempi di esecuzione dei Merge sui sottoarray la soluzione sarà $\Theta(n \log n)$.

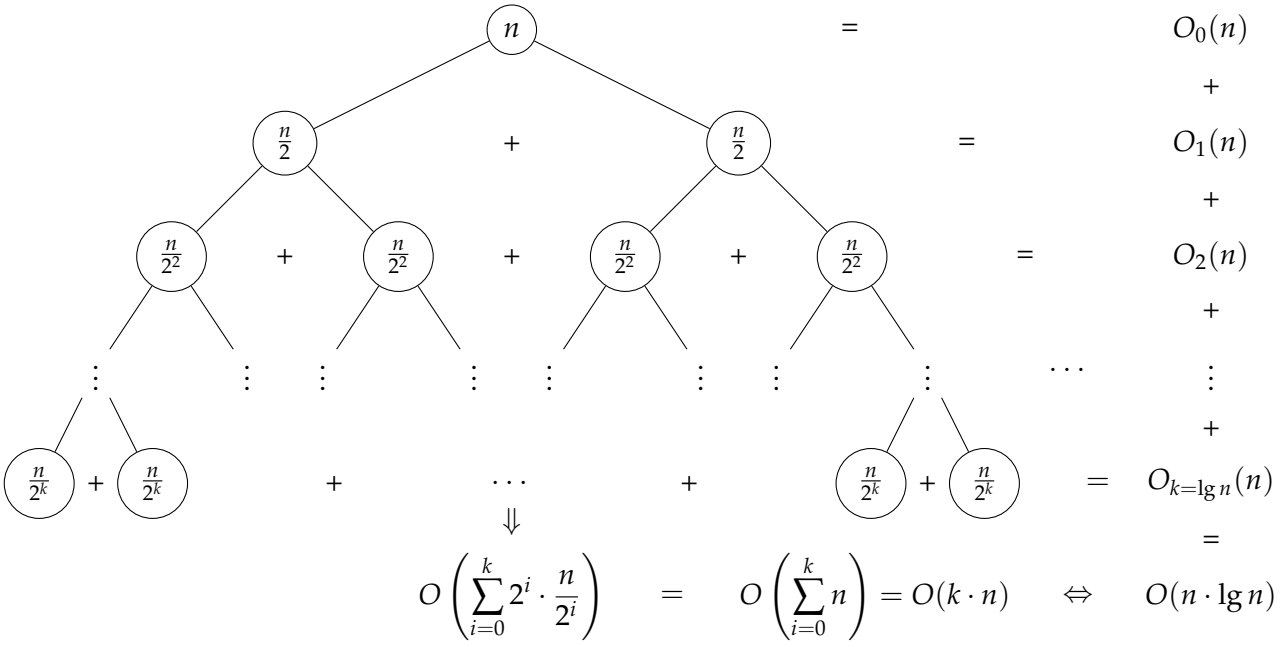


Figura 4.3: Albero di ricorsione del MergeSort

4.3 Quicksort

Quicksort come il Merge sort è basato sul paradigma Divide et Impera. L'idea che sta dietro è quella di dividere in due parti l'array iniziale A[p..r], calcolare una funzione di Partition che con l'utilizzo di due indici e calcolato un pivot porta gli elementi minori o uguali del pivot fino all'indice j e da j ad r ci stanno gli elementi maggiori del pivot. Le parti principali sono quindi 3:

- **Divide**: Divide l'array in due possibili sotto array A[p..q], A[q+1..r] tale che ogni

elemento di A][p..q] sia minore o uguale di A[p] ed ogni elemento di A[q+1..r] sia maggiore di A[p]. Calcola inoltre l'indice q;

- **Impera**: Ordino i due sottoarray con chiamate ricorsive al Quicksort;
- **Combina**: Dato che i due sottoarray sono ordinati, li unisco ed ho l'array ordinato;

4.3.1 Pseudocodice

La seguente procedura implementa il Quicksort:

```
1 QUICKSORT(A, p, r)
2     IF p < r
3         q <- PARTITION(A, p, r)
4         QUICKSORT(A, p, q)
5         QUICKSORT(A, q+1, r)
```

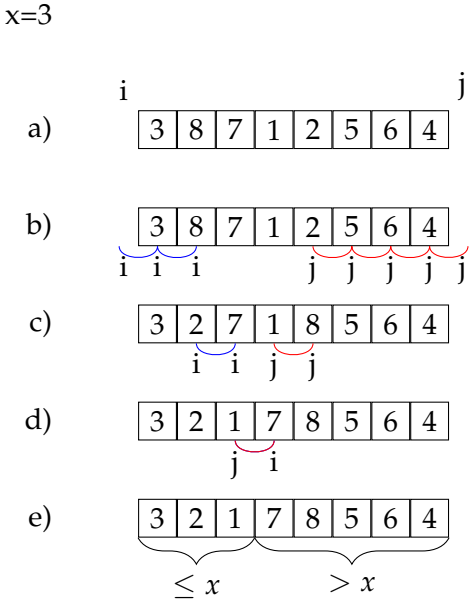


Figura 4.4: Rappresentazione grafica dell'algoritmo Partition su un array. a) L'indice i parte da 0, mentre l'indice j inizia da n+1,la variabile x indica il pivot che è il primo elemento dell'array: x=3 . b) i avanza nell'array finché non trova un elemento maggiore di x, mentre j decrementa fino a quando non trova un elemento minore di x, queste due condizioni portano a selezionare 8 e 2. c) 2 e 8 vengono scambiati e continua l'avanzamento degli indici, vengono trovati altri due numeri da scambiare 1 e 7. d) Gli elementi 1 e 7 si scambiano e gli indici continuano ad avanzare, l'indice j ha sovrapposto l'indice i e la funzione partition finisce qua, ritornando il valore di j. e) Gli elementi a fine partition dell'array sono composti da due blocchi: quelli minori o uguali di x e quelli maggiori di x.

```
1     PARTITION(A, p, r)
```

```
2          x <- A[p]
3          i <- p-1
4          j <- r+1
5          while TRUE
6               REPEAT
7                    j <- j-1
8               UNTIL A[j] <= x
9               REPEAT
10                   i <- i+1
11              UNTIL A[i]>=x
12              IF i<j
13                   Scambia(A[i],A[j])
14              ELSE
15                   RETURN j
```

I REPEAT se non si riescono a capire possono essere sostituti da

```
1     DO
2          j <- j-1
3     WHILE A[j]> x
4     DO
5          i <- i+1
6     WHILE A[i]<x
```

4.3.2 Prestazioni del Partition

La procedura del Partition può essere suddivisa in 3 gruppi:

- **Righe 2-4**: sono principalmente degli assegnamenti delle variabili, il loro lavoro risulta costante;
- **Ciclo while TRUE**: la condizione While(TRUE) è sempre vera, perciò bisogna determinare quando si può uscire dal ciclo. Ciò avviene quando la condizione $i < j$ è falsa, cioè j ed i si sono sovrapposti, questo evento avviene quando i due indici hanno ciclato n posizioni dell'array. Ricollegandoci alla figura 4.4, i ha valore 4 e j ha valore 3, considerando che i è partito da -1 e j da 9, i passi che i due indici hanno fatto è $9 - 4 + 0 + 4$=8, cioè la lunghezza dell'array. Il costo è quindi n;
- **RIGHE 18-21**: rappresentano lo scambio fra due variabili e questa operazione è costante;

Nel calcolare la complessità di un algoritmo, le costanti risultano per la maggior parte superflue, quindi se ci concentrassimo solamente sul valore polinomiale avremo che $Partition \in \Theta(n)$.

4.3.3 Complessità computazionale

Dopo aver determinato la complessità dell'algoritmo di partizione si può procedere determinando quello del QuickSort. Dato che la sequenza iniziale di oggetti viene divisa in due parti di lunghezza l e $n-l$ e viene chiamato il QuickSort su ognuno di essi, la formula di ricorrenza può essere scritta come:

$$T(n) = n + T(l) + T(n-l)$$

Il **caso peggiore** si presenta quando l è 1, cioè quando il pivot scelto è il più piccolo, ciò crea una partizione di due array di dimensioni (n-1) e 1. La formula di ricorrenza per questo caso è la seguente:

$$T(n) = n + T(n-1) + T(1)$$

Cerchiamo di calcolare la complessità dell'equazione di ricorrenza attraverso l'induzione matematica:

$$\begin{aligned} &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + T(n-3) \\ &\dots \\ &= n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2} = \Theta(n^2) \end{aligned}$$

Nel **caso migliore**, ovvero la partizione ha suddiviso l'array iniziale in due sottoarray ognuno di dimensione massima n/2, il tempo di ricorrenza sarà, cioè il pivot scelto è il mediano.

$$T(n) = n + 2T(\frac{n}{2}) = \Theta(n \log n)$$

L'algoritmo non è stabile, se prendo un'array di elementi uguali avviene che la prima metà è nella seconda e viceversa però ordina in loco.

4.3.4 Versione Random

Anziché utilizzare il primo elemento dell'array come pivot, ne utilizziamo uno random, poiché il pivot è preso a caso ci aspettiamo che dell'array di input potraà essere ben bilanciata in media, a differenza del caso precedente dove c'erano alcuni casi come l'array ordinato in modo decrescente o crescente che portavano ad una complessità computaizonale di $\Theta(n^2)$.

```
RANDOMIZED_PARTITION(A,p,r)
    i <- RAND(p,r)
    Scambia(A[i],A[p])
    return PARTITION(A,p,r)
```

Quanto costa a livello computazionale?

Se consideriamo che le probabilità di scelta dell'algoritmo siano uguali per tutti i numeri, l'equazione di ricorrenza dell'algoritmo diventa:

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

Partition restituirà sempre due array, in qualsiasi caso. Questa equazione può essere riscritta raggruppando $\frac{1}{n}$

$$= n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n-i))$$

E il risultato sarà $n \log n$, quindi concludendo utilizzando il RANDOMIZED-PARTITION il tempo che ci aspettiamo del quicksort è $O(n \log n)$.

4.3.5 Limiti dello stack

Un problema di cui non abbiamo parlato è che le chiamate ricorsive occupano spazio nello Stack, poiché ogni volta che la funzione è invocata i parametri devono essere salvati da qualche parte.Il caso peggiore richiederà O(n) spazio nello stack. Per ovviare a questo problema si potrebbe rimuovere l'ultima chiamata:

```
QUICK_SORT(A,p,r)
    while p<r
        q <- PARTITION(A,p,r)
        QUICK_SORT(A,p,q)
        p <- q+1
```

Questo metodo di eliminazione dell'ultima chiamata ricorsiva di una funzione prende il nome di **tail call elimination**. Abbiamo ridotto di metà le chiamate ricorsive, però si può fare ancora di meglio: si può pensare di iterare il sottoarray più grande che richiederebbe più chiamate ricorsive se implementato in modo standard, mentre per l'altro sottoarray che sarà più piccolo si utilizzerà la ricorsione:

```
1  QUICK_SORT(A,p,r)
2      while p<r
3          q <- PARTITION(A,p,r)
4          if(q-p+1 > r-q)
5              QUICK_SORT(A,q+1,r)
6              r <- q
7          else
8              QUICK_SORT(A,p,q+1)
9              p <- q+1:
```

La dimensione massima del sottoarray più piccolo sarà sempre la metà totale, quindi ad ogni chiamata la dimensione dell'array più piccola sarà dimezzata (1 chiamata metà dell'array, 2 chiamate metà della metà, ...)

Chiamate Ricorsive	lunghezza massima array più piccolo
0	n
1	$\leq \frac{n}{2}$
2	$\leq \frac{n}{4}$
i	$\leq \frac{n}{2^i}$

Tabella 4.1: Dimensione dell'array più piccolo al variare delle chiamate ricorsive

Quindi per quale valore di i si avrà che la dimensione del sotto array vale 1?

$$\frac{n}{2^i} \leq 1 \rightarrow i = \log_2 n$$

Ci troviamo in un caso base, non si può avere uno stack di attivazione più grande **O(logn)**.

4.4 Heap

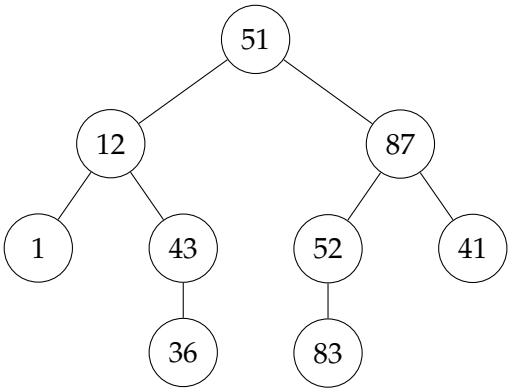


Figura 4.5: Esempio Albero Binario

La struttura dati heap è un array che può essere visto come un albero binario(Figura 4.5). Ogni nodo dell'albero corrisponde ad un elemento dell'array. L'elemento più in alto è chiamato **radice**, ogni può avere dei figli, massimo 2: **LEFT** e **RIGHT**. Se il nodo non ha figli è chiamato **foglia**. Un albero è detto **semi completo** se l'ultimo livello è semi riempito da sinistra verso destro. Un caso speciale di albero semi completo è l'albero completo.Una caratteristica molto importante è che se prendiamo l'indice *i* di un nodo, si può semplicemente calcolare gli indici dei figli e del genitore:

- Parent(i)= $\lfloor i/2 \rfloor$;
- Left(i) = 2i;
- Right(i) 2i+1;

Dentro ai nodi vogliamo mettere un' informazione, di solito una chiave di ordinamento.

Uno heap è un albero binario semi completo con chiavi associate ai nodi tale che per ogni nodo x le chiavi associate ai figli di x sono minori o uguali alla chiave associata a x. Da ricordare la differenza dell' Heapsize e la lunghezza dell'array:

- **Heapsize(A)**: indica il numero degli elementi dell'heap che sono registrati nell'array A;
- **Length(A)**: indica il numero di elementi nell'array;

Anche se ci possono essere dei numeri memorizzati in tutto l'array A[1 . . . A.length], soltanto i numeri in A[1. . . A.Heapsize] sono elementi validi dell'heap.

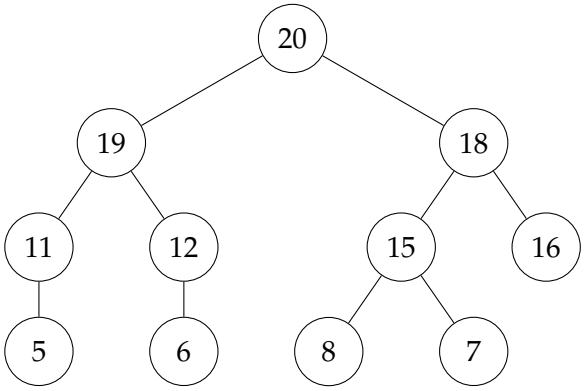


Figura 4.6: Esempio Heap

Ci sono due tipi di heap binari: **max-heap** e **min-heap**. In entrambi i tipi, i valori dei nodi soddisfano una proprietà dell'heap:

- **max-heap**: A[parent] $\geq$ A[i];
- **min-heap**: A[parent] $\leq$ A[i];

Per gli algoritmi che utilizzeremo partiremo con i max-heap, l'**altezza di un nodo** è il numero di archi nel cammino semplice più lungo che dal nodo scende fino ad una foglia.L'**altezza** di un heap è l'altezza di una radice ed è $\Theta(\log n)$ poiché essendo che ogni genitore può avere al più due figli, ogni livello avrà al più 2 volte i nodi del livello superiore. Le operazione che operano su un heap avranno quindi complessità $O(\log n)$

4.4.1 Conservare la proprietà dell'Heap

Per mantenere la proprietà di un max-heap useremo la procedura **MAX-HEAPIFY**. Quando viene chiamata, MAX-HEAPIFY assume che gli alberi binari con radici in LEFT(i) e RIGHT(i) siano max-heap, ma che A[i] possa essere più piccolo di uno dei sui figli, violando così la proprietà dell'intero HEAP. A ogni passo, viene determinato il più grande tra gli elementi $A[i], A[LEFT(i)]$ e $A[RIGHT(i)]$; il suo indice viene memorizzato in massimo. Se $A[i]$ è più grande, allora il sottoalbero con radice nel nodo i è un max-heap e la procedura termina. Altrimenti, uno dei due figli ha l'elemento più grande e $A[i]$ viene scambiato con $A[massimo]$;

in questo modo, il nodo i e i suoi figli soddisfano la proprietà del max-heap. Il nodo con indice *massimo*, però, adesso ha il valore originale $A[i]$ e, quindi, il sottoalbero con radice in *massimo* potrebbe violare la proprietà del max-heap. Di conseguenza, bisogna chiamare ricorsivamente la subroutine MAX-HEAPIFY per questo sottoalbero.

```
1  MAX-HEAPIFY(A, i)
2      l <- LEFT(i)
3      r <- RIGHT(i)
4      if l <= HEAPLENGTH(A) && A[l]>A[i]
5          largest = l
6      else
7          largest = i
8      if r <= HEAPLENGTH(A) && A[r]>A[i]
9          largest = r
10     if largest != i
11         Scambia(A[i],A[largest])
12         MAX-HEAPIFY(A, largest)
```

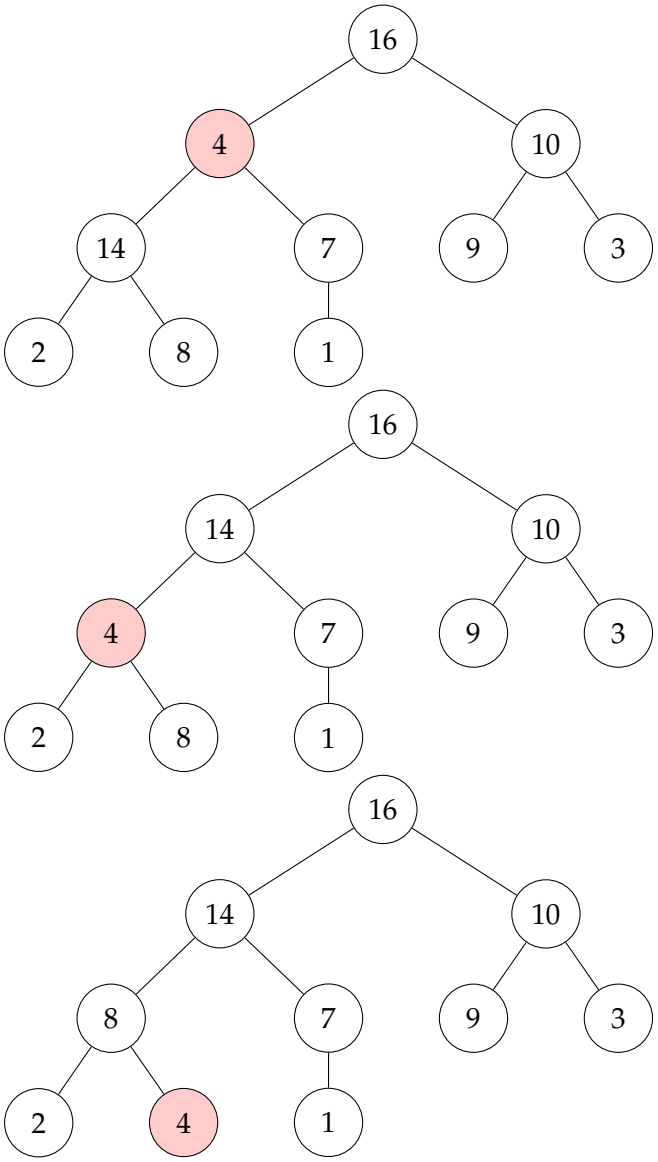


Figura 4.7: Rappresentazione MAX-HEAPIFY

Il tempo di esecuzione di MAX-HEAPIFY in un sottoalbero di dimensione n con radice in un nodo i è pari al tempo $\Theta(1)$ per sistemare le relazioni fra gli elementi A[i], A[LEFT(i)], A[RIGHT(i)], più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del nodo i. Il caso pessimo è quando l'ultimo livello è completato a metà $T(n) \leq T(\frac{2}{3}n) + \Theta(1)$ Utilizzando il teorema del maestro:

$$a = 1,\ b = \frac{3}{2},\ f(n) = 1$$
$$n^{\log_{\frac{3}{2}} 1} = 1 = f(n)$$

L'ordine di grandezza è uguale, perciò la complessità sarà $O(\log n)$

4.4.2 Costruire un heap

Possiamo utilizzare la procedura MAX-HEAPIFY per dal basso verso l'alto per convertire un' array in un max-heap.

```
1 BUILD–MAX–HEAP(A)
2     A.heap-size=A.length
3     for i = ⌊A.length/2⌋ downto 1
4         MAX–HEAPIFY(A, i)
```

Fino ad A.length/2 poiché l'ultimo livello è formato da solo foglie che rappresentano di per sè un heap di un solo elemento che utilizzeremo come punto di partenza, perciò non c'è bisogno di applicare l'algoritmo su di esse.

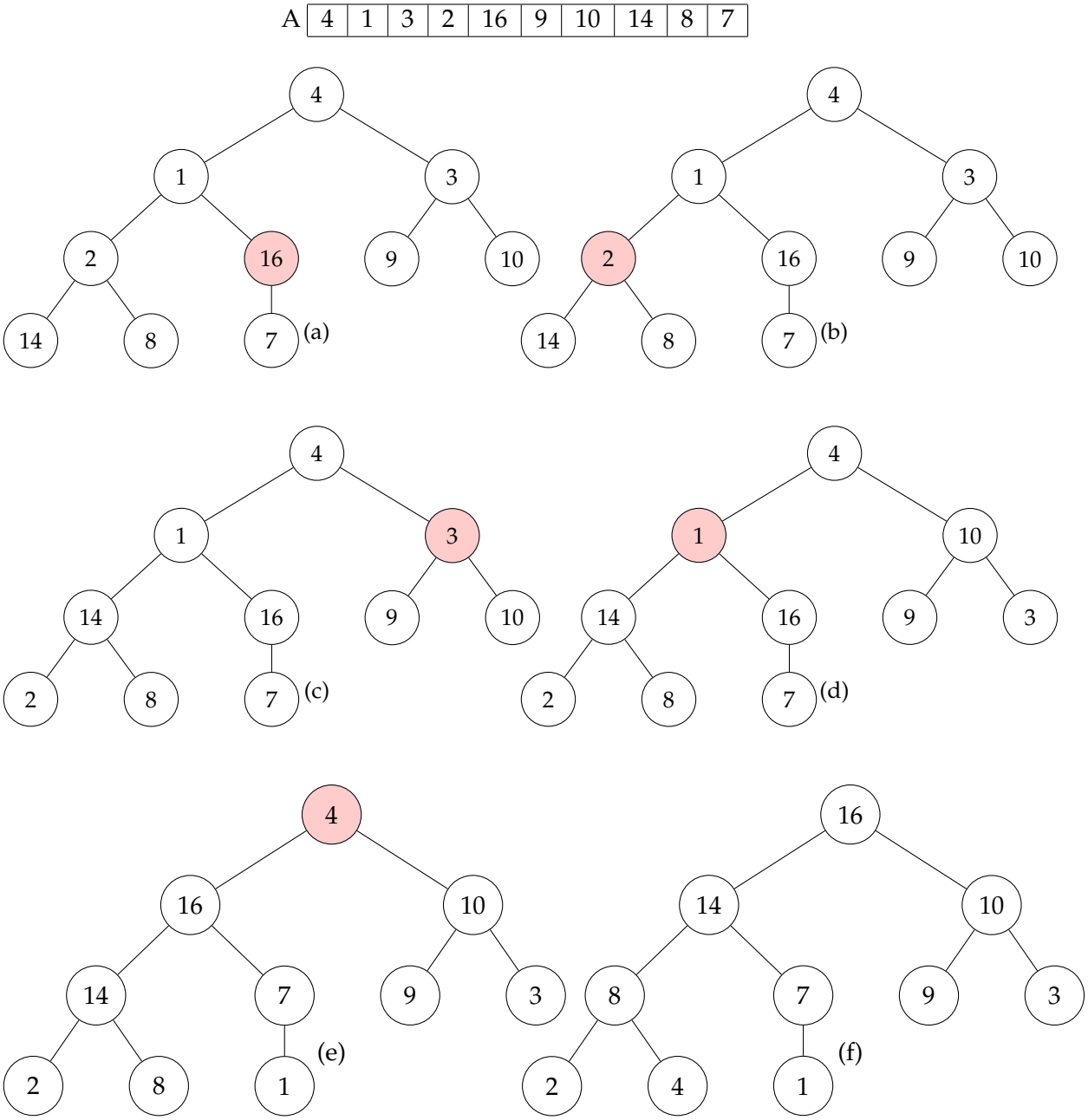


Figura 4.8: Funzionamento BUILD-MAX-HEAPSORT sull'array A

Ad ogni chiamata di MAX-HEAPIFY costa un tempo O(logn) e ci sono O(n) di queste chiamate. Quindi il tempo di esecuzione è O(nlogn), questo limite superiore non è sinotticamente stretto. L'analisi più rigorosa si basa che un heap di altezza n, avrà una lunghezza di $\lfloor \log n \rfloor$ e, per ogni h al massimo $\lceil n/2^{h+1} \rceil$ nodi di altezza h. Il tempo che impiega MAX-HEAPIFY

per un nodo h è O(h). Quindi MAX-HEAP è limitato superiormente da:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h) = \left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

Ricordando che è possibile ottenere altre formule integrando e derivando le precedenti formule. Per esempio, derivando entrambi i lati della serie geometrica infinita e moltiplicando per x, si ha

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

per $|x| < 1$ Quindi scegliendo x=$\frac{1}{2}$ ed applicando la formula si ha

$$n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Quindi, il tempo di esecuzione di BUILD-MAX-HEAP può essere limitato così:

$$O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Dunque possiamo costruire un max-heap da un array non ordinato in un tempo lineare.

4.4.3 HeapSort

L'algoritmo heapsort inizia utilizzando BUILD-MAX-HEAP per costruire un max-heap nell'array di input A.Poiché l'elemento più grande sarà sempre nella radice A[1], esso può essere inserito nella sua posizione finale scambiandolo con A[n], se ora diminuiamo la grandezza dell'HEAPSIZE(rimuovo il nodo n dall'heap) i figli della radice restano max-heap, ma potrebbero violare la proprietà di questa struttura quindi rievoco MAX-HEAPIFY e continuo finché l'heap non arriva ad una dimensione di n=2:

```
1 HEAPSORT(A)
2       BUILD–MAX–HEAP(A)
3       for i = A.length downto 2
4             scambia (A[1],A[i])
5             HEAPSIZE(A)--;
6             MAX–HEAPIFY(A,1)
```

La complessità dell'algoritmo è O(nlog n) poiché BUILD-MAX-HEAP impiega un tempo n, il ciclo è effettuato n volte e MAX-HEAPIFY ha complessità O(log n). L'algoritmo ordina in loco, ma non è stabile.

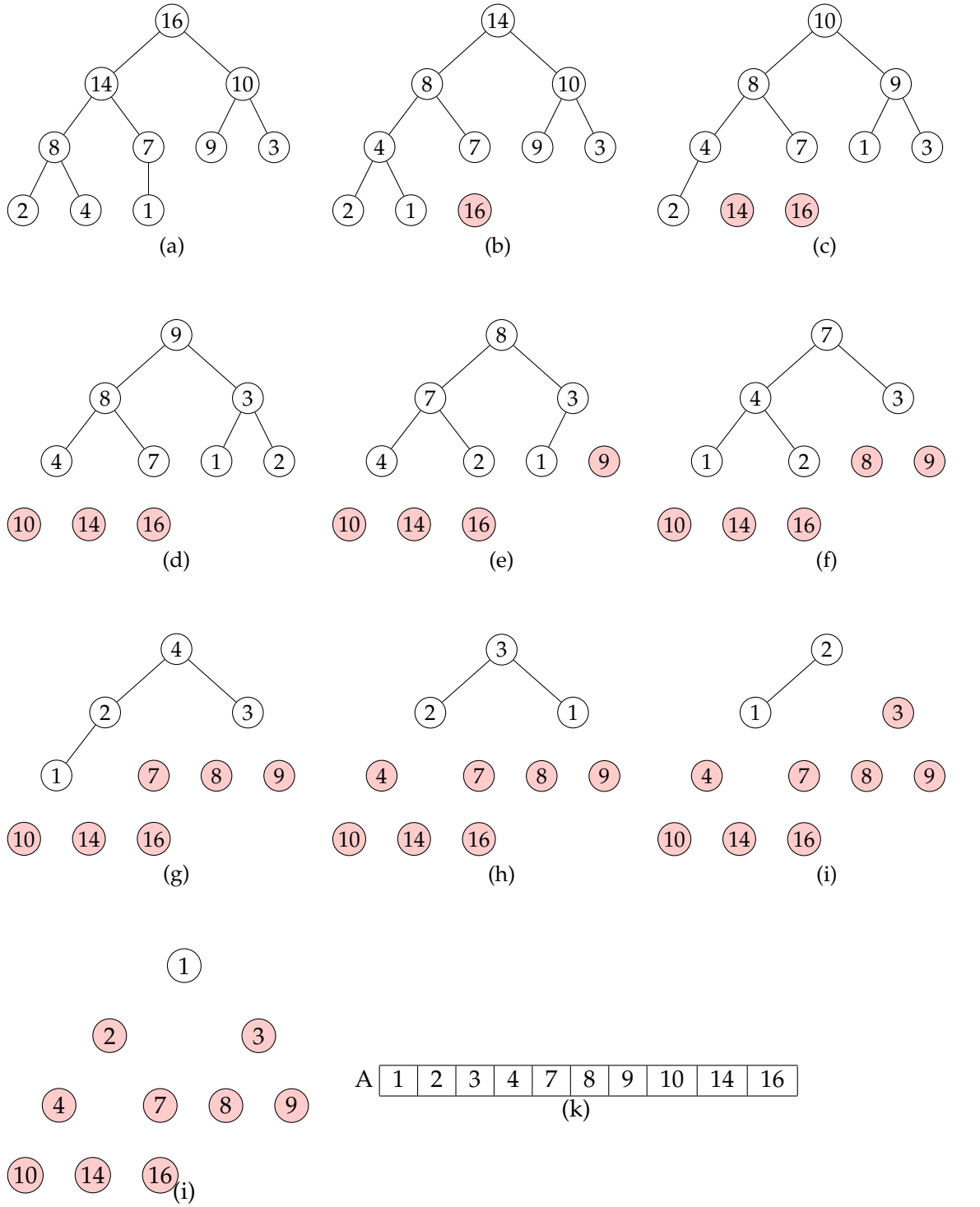


Figura 4.9: Esempio funzionamento HeapSort

4.4.4 Estrarre il valore massimo da un Heap

```
1 HEAP–EXTRACT–MAX(A)
2       if HEAPSIZE(A)<1
3           then errore
4       max = A[1]
5       A[1] = A[HEAPSIZE(A)]
6       HEAPSIZE(A)--
7       MAX–HEAPIFY(A,1)
8       return max
```

Il tempo che impiega **HEAP-EXTRACT-MAX** è O(log n) poiché esegue soltanto del lavoro costante oltre a chiamare MAX-HEAPIFY che è di ordine log n.

4.4.5 Limiti dell'ordinamento

Gli algoritmi citati fino ad ora utilizzano un confronto tra operandi, questo metodo può essere visto in modo astratto come un **albero di decisione** che è un albero completo che rappresenta i confronti tra gli elementi effettuati da un determinato tipo di algoritmo di ordinamento in un input di dimensione nota.

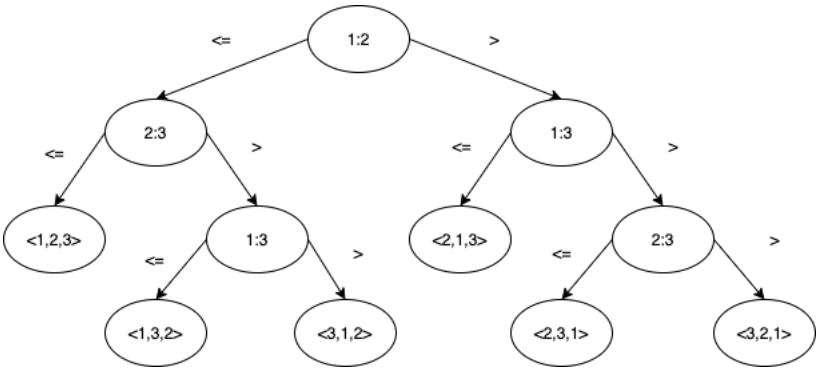


Figura 4.10: Albero di decisione

La figura4.10 è un'implementazione dell'insertion sort su tre elementi.Ogni nodo ha al suo interno due valori $i : j$. La decisione a sinistra indica che il confronto è minore, a destra maggiore. La lunghezza del percorso più lungo dalla radice di un albero di decisione ad una delle foglie rappresenta il caso peggiore. Conseguentemente il caso peggiore dell'algoritmo è uguale all'altezza dell'albero di decisione (log n). Proprio per questo quando baso la mia decisione su dei confronti, il programma deve eseguire almeno il logaritmo delle situazioni totali possibili. Come citato in precedenza, un algoritmo di ordinamento è effettuato su una sequenza di oggetti su cui è definita una relazione di ordinamento. La soluzione che troveremo sarà data da una permutazione dell'input.

$$\#\text{permutazioni di n} = n!$$

La profondità minima dell'albero di decisione è $\log n!$, e il percorso più lungo appartiene alla famiglia $\Theta(n \log n)$:

$$\log n! \in (\log n^n) = \Theta(n \log n)$$

Quindi non è possibile avere un ordinamento migliore di nlog n. Quindi il problema dell'ordinamento $\in \Omega(n \log n)$ se è necessario eseguire dei confronti.

4.5 Ordinare in tempo Lineare

Fino ad ora abbiamo utilizzato algoritmi di ordinamento che si basavano su una relazione di confronto che permetteva di determinare l'ordine degli oggetti. Si è anche visto che il limite inferiore è $\Theta(\log n)$ e asintoticamente non è possibile con il metodo utilizzato fino ad ora (confronto tra oggetti) fare di meglio. Un'idea può essere di cambiare punto di vista e cercare di non ordinare gli elementi attraverso un confronto tra essi, ma con altri tipi di proprietà.

4.5.1 Counting Sort

Supponiamo di dover ordinare un'array di n numeri naturali in $1, 2, \ldots, k$ con k fissato a priori. Allora quando k=O(n), l'algoritmo impiegherà $\Theta(n)$. L'idea di base del counting sort è di determinare per ogni elemento dell'array x, il numero di elementi minore di x. Nel codice del counting sort, assumiamo che l'input dato è un array A[1...n] e che length(A)=n. Abbiamo bisogno di due array: B[1...n] che conterrà la sequenza data in input ordinata e l'array C[1...k] che sarà utilizzato temporaneamente per identificare quanti numeri sono maggiori di un certo valore i.

```
1  COUNTING-SORT(A,B,k)
2  FOR i <- 1 TO k
3      C[i]=0
4  FOR j <- 1 TO LENGTH(A)
5      C[A[J]]++
6  FOR i <- 2 TO k
7      C[i] <- C[i]+C[i-1]
8  FOR J <- LENGTH(A) DOWN TO 1
9      B[C[A[J]]] <- A[J]
10     C[A[J]]--
```

Il codice del counting sort è formato da 4 cicli **for**. Nelle righe 2-3 si inizializza l'array C. Nelle righe 4-5 si ispeziona ogni elemento dell'array A, se il valore dell'array è j, si aumenta di 1 il valore nella cella j di C (C[j]) che C[j] conterrà le ricorrenze degli elementi dell'array A. Successivamente nelle righe 6-8 si effettua un'operazione di accumulo ovvero si determina per ogni i=0,1,...,k quanti elementi dell'array sono minori o uguali ad i, infine nelle righe 9-11, partendo dall'ultimo elemento di A j=Length(A)...1, si inserisce ogni elemento nella corrispettiva cella di B segnata dal valore C[A[j]] che come detto prima rappresenta quanti valori sono minori o uguali di A[j], quindi C(A[j]) rappresenta l'indice dove inserire l'elemento in B. Eseguita l'operazione di inserimento, si decrementa C(A[j]).

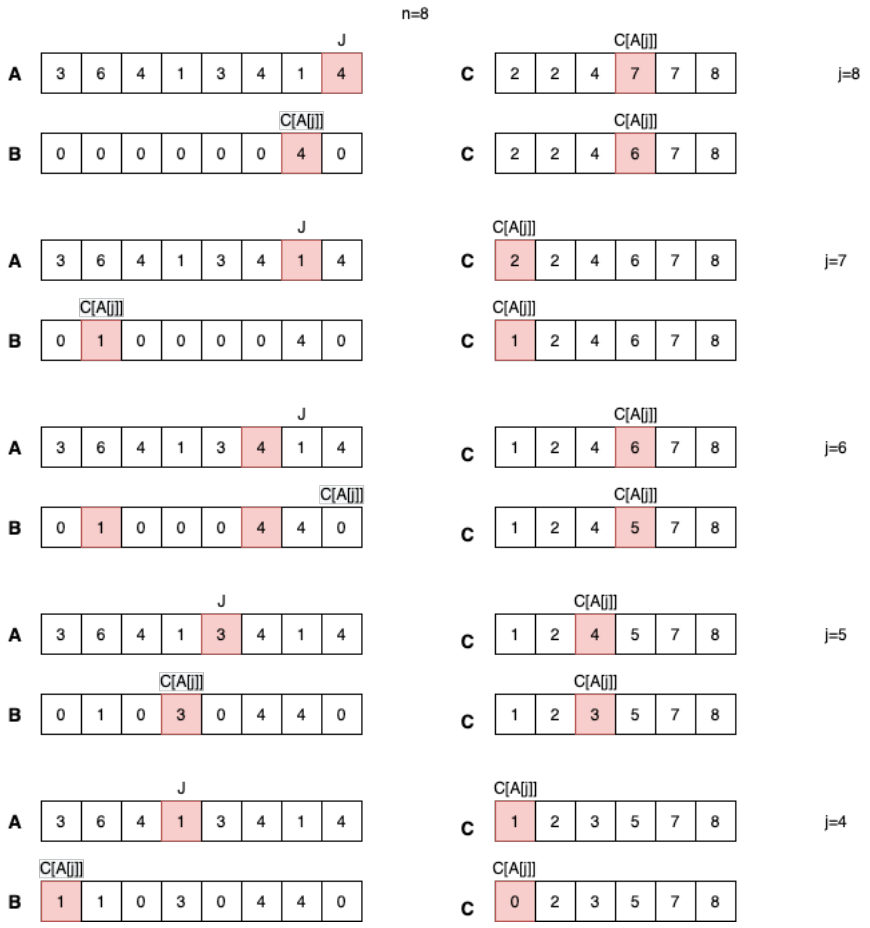
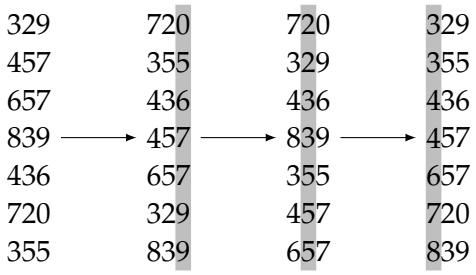


Figura 4.11: Rappresentazione semicompleta del Counting sort

Il primo **for** impiega $\Theta(k)$, il secondo **for** $\Theta(n)$, il terzo **for** $\Theta(k)$ e il quarto $\Theta(n)$. Utilizziamo Θ, poiché siamo sicuri che i cicli finiranno in quel preciso istante. Il costo finale è $\Theta(k+n)$ e se quindi scegliessimo k=n, il costo sarebbe $\Theta(n)$. Un'importante proprietà di counting sort è la **stabilità**: i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input.

4.5.2 Radix Sort

Radix Sort è l'algoritmo utilizzato dalle macchine per ordinare le schede perforate. Idealmente si partirebbe confrontando la cifra più significativa, però il Radix sort ordina prima le schede in base alla cifra **meno significativa**. Le schede vengono poi combinate in un unico mazzo e vengono ordinate in base alla seconda cifra meno significativa e ricombinate in maniera analoga. Il processo continua finché le schede saranno ordinate rispetto a tutte le d cifre. Per eseguire l'algoritmo è essenziale che gli ordinamenti delle cifre siano stabili.



Il codice per Radix sort è semplice. La seguente procedura prevede che ogni elemento all'interno dell'array A di n elementi abbia d cifre, dove la cifra 1 è quella di ordine più basso e la cifra d quella di ordine più alto.

```
1 RADIX–SORT(A,d)
2     for i <– 1 downto d
3         Counting Sort oppure un ordinamento stabile per
4         ordinare l'array A sulla cifra i
```

La correttezza di Radix Sort si basa sul tipo di algoritmo di ordinamento che si utilizza per ogni cifra. Ogni passaggio su n numeri di d cifre richiede un tempo $\Theta(n+k)$. Poiché ci sono d passaggi, il tempo totale di Radix sort è $\Theta(d(n+k))$. Con d costante per tutti i numeri e k=O(n) il Radix sort viene eseguito in tempo lineare.

Il Radix sort è veramente così conveniente?

La risposta è dipende, se prendessimo ad esempio il Quicksort, il Radix sort ha una complessità minore rispetto a quest'ultimo però sebbene il Radix sort richieda meno passaggi del Quicksort, ogni suo passaggio **potrebbe richiedere più tempo**. Inoltre la versione di Radix Sort che usa Counting sort come ordinamento stabile intermedio non effettua l'ordinamento sul posto, come molti algoritmi $\Theta(\log n)$ e inoltre se lo spazio nella memoria principale è limitato, potrebbe essere preferibile utilizzare un altro tipo di algoritmo.

4.5.3 Bucket Sort

Bucket sort assume che l'input sia estratto da una distribuzione uniforme e ha tempo di esecuzione medio $\Theta(n)$. Come Counting sort, Bucket sort effettua un'ipotesi sull'input, ovvero che l'input sia generato da un processo casuale che distribuisce gli elementi uniformemente e indipendentemente. Bucket sort divide l'intervallo in n sottointervalli della stessa dimensione, detti **bucket**, e poi distribuisce gli n numeri di input nel bucket. La distribuzione di probabilità dei valori è nota. Poiché gli input sono uniformemente distribuiti non ci aspettiamo molti valori all'interno degli stessi bucket. Per produrre l'output, semplicemente ordiniamo i numeri in ogni bucket e poi scorriamo ordinatamente i bucket, elencando gli elementi in ciascuno di essi.L'algoritmo è **Stabile**. Per analizzare il costo del bucket sort, bisogna determinare quello delle chiamate dell'insertion sort. Indichiamo con n_i la variabile casuale che rappresenta il numero degli elementi che vengono inseriti nel bucket B[i]. Il tempo di esecuzione di bucket sort è:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i)^2$$

Analizziamo ora il tempo di esecuzione nel caso medio calcolando il **valore atteso** del tempo di esecuzioni:

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i)^2\right] \quad \text{(per la linearità del valore atteso)}$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O\left[E(n_i)^2\right]$$

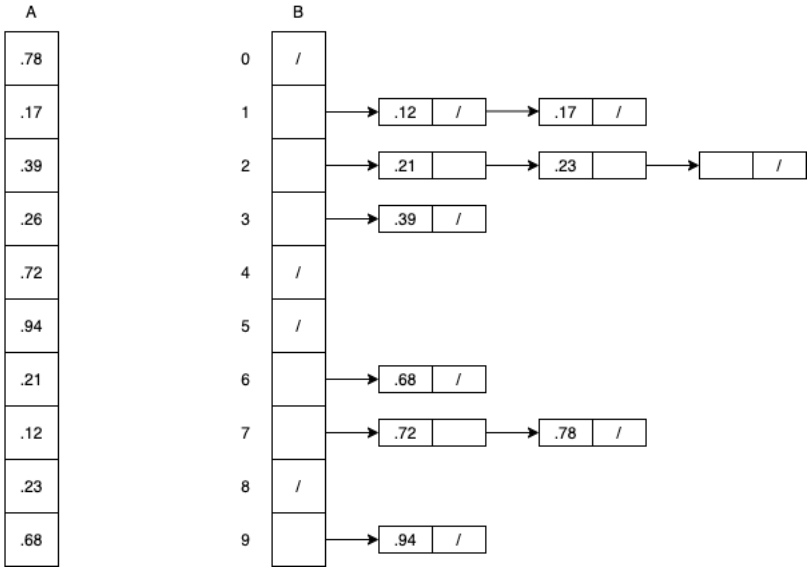


Figura 4.12: BucketSort su un intervallo [0,1) con 10 bucket

Considero la variabile indicatrice $X_i = I$A[j] ricade nel bucket i. Quindi il numero totale di elementi nel bucket i è rappresentato dalla sommatoria:

$$n_i = \sum_{i=0}^{n} X_i$$

Per calcolare $E[n_i^2]$ esprimiamo il quadrato e raggruppiamo i termini:

$$E[n_i^2] = E\left[\left(\sum j = 1^n X_{ij}\right)^2\right]$$
$$= E\left[\sum j = 1^n \sum k = 1^n X_{ij} X_{ik}\right]$$

Ricordando la proprietà:

$$\sum j = 1^n \sum j = k^n x_i X_j = \sum_{i=1}^{n} x_i^2 + \sum_{i,j=1, i \neq j} x_i x_j$$

Utilizziamola:

$$= E\left[\sum j = 1^{n} X_{ij}^2 + \sum_{1\le j\le n} \sum_{1\le k\le n, k\neq j} X_{ij}X_{ik}\right]$$
$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n} \sum_{1\le k\le n, k\neq j} E[X_{ij}X_{ik}]$$

La variabile casuale indicatrice X_{ij} vale 1 con probabilità $1/n$ e 0 negli altri casi quindi

$$E[X_{ij}^2] = 1 * \frac{1}{n} + 0\left(1 - \frac{1}{n}\right)$$

Quando $k \neq j$, le variabili X_{ij} e X_{ik} sono indipendenti, quindi

$$E[X_{ij}X_{ik}] = \frac{1}{n} * \frac{1}{n} = \frac{1}{n^2}$$

Sostituendo questi due valori attesi nella equazione, si ha

$$E[n_i^2] = 2 - \frac{1}{n}$$

Utilizzando questo valore atteso, concludiamo che il tempo di esecuzione nel caso medio di bucket sort è $\Theta(n) + nO(2 - 1/n) = \Theta(n)$

4.6 Riassunto Algoritmi di Ordinamento

Algoritmo	pessimo	medio	migliore	Stabilità	In loco	Spazio di supporto
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	si	si	$\Theta(1)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	si	no	$O(n)$
Quick sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no	si	$O(n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	no	si	$\Theta(1)$
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$	si	no	O(k)
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$	si	no	O(n)
Bucket sort	$O(n^2)$	$\Theta(n)$	$\Theta(n)$	si	no	O(n)

Tabella 4.2: Tabella di riassunto per gli algoritmi di ordinamento

5 Selezione

L'i-esima **statistica d'ordine** di un insieme di n elementi è l'i-esimo elemento più piccolo. Il **minimo** di un insieme di elementi è la prima statistica di ordine (i=1) e il **massimo** è l'n-esima statistica d'ordine (i=n). La **mediana** è il 'punto di mezzo'. Il **problema della selezione** può essere definito nel seguente modo:

> **Input**: un insieme A di n numeri(distinti) e un intero i, con $1 \leq i \leq n$ su cui è definita una relazione di ordinamento.
>
> **Output**: l'elemento x $\in$ A che è maggiore esattamente di altri $i - 1$ elementi di A.

Il problema della selezione lo possiamo risolvere nel tempo O($n \log n$), perché possiamo utilizzare un algoritmo di ordinamento di quell'ordine (MergeSort, QuickSort, HeapSort) e selezionarne l'elemento i-esimo.

5.1 Minimo e massimo

Quanti confronti sono necessari per trovare il minimo di un insieme con n elementi? Facilmente si potrebbe dire n-1 confronti poiché esaminiamo, uno alla volta, gli elementi dell'insieme e teniamo traccia dell'ultimo elemento più piccolo trovato:

```
1 MINIMIUM(A)
2     min <- A[1]
3     for i <- 2 to A.length
4         if min > A[i]
5             min <- A[i]
6     return min
```

Ovviamente anche il massimo può essere trovato nello stesso modo:

```
1 MAXIMUM(A)
2     max <- A[1]
3     for i <- 2 to A.length
4         if max < A[i]
5             max <- A[i]
6     return max
```

E n-1 confronti è il massimo che possiamo fare perché è il limite inferiore per determinare il minimo e/o il massimo.

5.1.1 Minimo e massimo simultanei

L'idea iniziale sarebbe quella di determinare il minimo e il massimo separatamente con un costo di $\Theta(2n-2)$, ma si potrebbe fare di meglio. Possiamo farlo mantenendo aggiornati il minimo e il massimo trovati mentre scorriamo nell'array. Confrontiamo prima due elementi di input l'uno con l'altro e poi il più piccolo dei due con il minimo corrente e il più grande dei due col massimo corrente, con un costo di 3 confronti per ogni 2 elementi.

- Se n è dispari assegniamo al minimo e al massimo il valore del primo elemento e poi elaboriamo i restanti elementi in coppia. Si svolgono $3\lfloor n/2 \rfloor$ confronti;
- Se n è pari, effettuiamo un confronto tra i primi 2 elementi per determinare i valori iniziali del minimo e massimo.Si svolgono $3n/2-3$ confronti;

In qualsiasi caso il numero di confronti effettuati è al massimo $3\lfloor n/2 \rfloor$.

5.2 Selezione in tempo atteso lineare

Il problema della selezione può sembrare più difficile rispetto a quello della selezione del minimo, ma vedremo che asintoticamente equivarrà $\Theta(n)$. Il primo algoritmo che andremo a vedere è il RANDOMIZED-SELECT che è molto simile al Quicksort.

```
1 RANDOMIZED-SELECT(A, p, r, i)
2     if p == r
3         return A[p]
4     q <- RANDOMIZED-PARTITION(A, p, r)
5     k <- q-p+1
6     if i<= k
7         return RANDOMIZED-SELECT(A, p, q, i)
8     else
9         return RANDOMIZED-SELECT(A, q+1, r, i-k)
```

La procedura RANDOMIZED-SELECT opera nel seguente modo. La riga 2 serve a identificare il caso base, se viene dato in input un'array di un elemento lo ritorna. La riga 4 divide l'array A[p..r] in due sotto array A[p..q], A[q+1..r] dove il primo sottoarray ha tutti gli elementi minori di A[q] e il secondo ha tutti gli elementi maggiori di A[q]. La riga 5 serve a calcolare il numero di elementi all'interno del primo sotto array più uno ovvero il pivot. La riga 6 controlla se l'elemento i è nel primo sottoarray oppure nel secondo, se è nel primo si richiama il RANDOMIZED-SELECT su A[p..q], mentre se dovesse essere nel secondo si richiama su A[q+1..r] e ricordando che conosciamo già k valori che sono più piccoli dell'i-esimo elemento più piccolo di A[p..r]- gli elementi di A[p..q], l'elemento desiderato è in (i-k).

Il caso ottimo sarebbe quello dove l'array dato in partenza è già ordinato e il RANDOMIZED-SELECT rappresenterebbe il caso specifico della ricerca dicotomica che ha un'equazione di ricorrenza pari a $T(n) = n + T(\frac{n}{2})$ e se provassimo a calcolarne la complessità con il metodo dell'esperto la soluzione sarebbe $\Theta(n)$. Consideriamo i dati che ci vengono forniti $a = 1$, $b = 2$, $f(n) = n$, $n^{\log_b a} = n^0 = 1$, $f(n) = O(n^{1-\epsilon}$ con $\epsilon = 1$, quindi la soluzione è $T(n) \in \Theta(n)$.

5.3 Selezione in tempo lineare nel caso peggiore

Esaminiamo adesso un algoritmo di selezione il cui tempo di esecuzione è O(n) nel caso peggiore. Come RANDOMIZED-SELECT anche l'algoritmo SELECT ritorna trova l'elemento desiderato però l'idea di base è quella di garantire una buona partizione dell'array. I passi di questo algoritmo sono 5:

1. Dividere gli n elementi dell'array di input in $\lfloor n/5 \rfloor$ gruppi di 5 elementi e al massimo un gruppo da meno di 5 elementi;
2. Trovare la mediana di ciascuno degli $\lceil n/5 \rceil$ gruppi, effettuando prima un ordinamento per inserimento degli elementi di ogni gruppo e poi scegliendo la mediana in ognuno dei gruppi;
3. Usare SELECT ricorsivamente per trovare la mediana x delle $\lceil n/5 \rceil$ mediane trovate nel passo 2;
4. Partizionare l'array utilizzando come pivot la mediana x trovata nel passo 3;
5. Se i = k restituisce k, altrimenti utilizzare SELECT ricorsivamente per trovare l'i-esimo elemento più piccolo nel lato basso se i $<$ k, oppure l'(i-k)-esimo elemento più piccolo nel lato i $>$ k;

Per analizzare il tempo di esecuzione di SELECT, determiniamo prima un limite inferiore sul numero di elementi che sono maggiori dell'elemento di partizionamento x. Almeno la metà delle mediane sono maggiori o uguali della mediana x, ciò significa che almeno metà dei $\lceil n/5 \rceil$ gruppi contribuiranno contribuisce con 3 elementi che sono maggiori di x, tranne quel gruppo che ha meno di 5 elementi e quel gruppo che contiene x. Se escludiamo questi due gruppi, allora il numero di elementi maggiori di x è

$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil\right) \geq \frac{3n}{10} - 6$

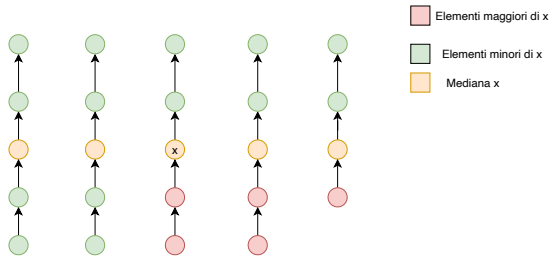


Figura 5.1: Rappresentazione grafica di SELECT

Quindi nel caso peggiore SELECT viene chiamato ricorsivamente per almeno $\frac{7n}{10} + 6$ elementi nel passo 5. I passi 1, 2, 4 impiegano un tempo O(n)(il passo 2 è formato da O(n) chiamate di Insertion Sort su insiemi di dimensione O(1)). Il passo 3 impiega $T(\lceil n/5 \rceil)$.

T(n)=$T\left(\left\lceil \frac{n}{5}\right\rceil\right) + T(\frac{7}{10} + 6) + \Theta(n)$

Dato che stiamo cercando di creare un algoritmo di selezione in tempo lineare ci aspettiamo che il risultato sia di tipo lineare perciò proviamo ad utilizzare il metodo della sostituzione per vedere se l'ipotesi che abbiamo fatto è corretta.

$$T(n) \leq cn$$
$$T(n) \leq c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7n}{10} + 6\right) + \Theta(n)$$

Ciò si può solo utilizzare se $7n/10 + 6 < n$, facendo i calcoli risulta che è vero per n$>$ 20. Quindi proseguendo con i calcoli e ricordando che $\lceil n/5 \rceil = n/5 + 1$:

$$\begin{aligned} &\leq \frac{n}{5} + c + \frac{7}{10}cn + 6c + O(n) \\ &= \frac{9}{10}cn + 7c + O(n) \\ &= cn - \underbrace{\left(\frac{1}{10} - 7c\right)}_{>O(n)} \\ &\leq cn \end{aligned}$$

Per un c abbastanza grande il problema ha soluzione $O(n)$ per differenza di funzioni lineari abbiamo dedotto il risultato, però visto che è la prima volta che lo facciamo mostriamo il perché di tale scelta:

$$\begin{aligned} &\leq cn - \left(\frac{1}{10} - 7c\right) + c'n \\ &= cn - \left(\frac{1}{10}cn - 7c - c'n\right) \\ &= cn - \left(\left[\frac{1}{10} - c'\right] - 7c\right) \end{aligned}$$

Quando questa cosa è minore di cn?

$$\left[\frac{1}{10} + c'\right] n - 7c \geq 0$$

Se scelgo $c = 10c'$ arriviamo ad un risultato $\approx 70n$ che asintoticamente risulta più piccolo di un $n \log n$, ma solo per valori estremamente grandi, ad esempio per un array contente 10^{21} valori il logaritmo sarebbe più veloce di 70n.

6 Strutture Dati

6.1 Strutture Dati elementari

6.1.1 Stack

Lo stack è un insieme dinamico che addotta la politica LIFO(*Last In First Out*) per gestire i dati al suo interno cioè l'ultimo elemento che è messo all'interno dell'insieme è il primo ad uscire. Le operazioni principali sono 2:**PUSH** che inserisce all'interno della pila e **POP** che prende l'ultimo elemento della pila. Un metodo utilizzato molto è il **Top** che ritorna l'indice dell'elemento inserito più recente.

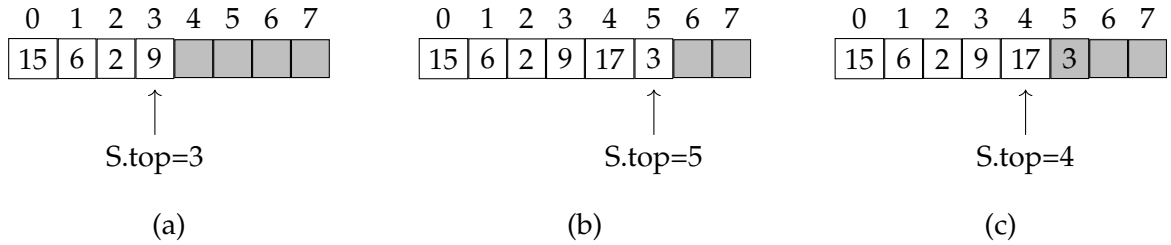


Figura 6.1: Implementazione di uno stack S con un array. Gli elementi all'interno dello Stack sono quelli in chiaro, mentre le celle in scuro rappresentano gli spazi vuoti disponibili.

Lo Stack può essere implementato in molti modi, noi negli esempi, ma anche nel corso degli appunti baseremo questa struttura dati su un'array. Ricordiamoci che in caso di POP se non ci sono elementi all'interno dello Stack ci sarà un'errore di **underflow**, mentre se lo stack è pieno e si cerca di eseguire una PUSH ci sarà un caso di **overflow** Proponiamo lo pseudocodice delle funzioni più richieste quando si parla di Stack. Consideriamo S lo stack e x il valore che dobbiamo inserire nello Stack allora:

```
1 STACK–EMPTY(S)
2     if S.top == 0
3         return TRUE
4     return FALSE

1 PUSH(S, x)
2     S.top <- S.top +1
3     S[S.top] <- x

1 POP(S)
2     if STACK–EMPTY(S)
```

```
3         error "underflow"
4     else S.top = S.top -1
5         return S[S.top+1]
```

6.1.2 Code

Le operazioni principali sono **ENQUEUE** e **DEQUEUE** che rappresentano le operazioni di INSERT e DELETE di un elemento della coda, sono in analogia con i metodi PUSH e POP dello stack. LA coda ha un inizio(*head*) e una fine(*tail*). Quando un elemento viene inserito, nella coda, prende posto alla fine della coda, esattamente come una fila di persone. L'elemento rimosso è sempre all'inizio della coda. L'attributo Q.tail indica la prossima posizione in cui l'ultimo elemento sarà inserito nella coda, mentre l'attributo Q.head indica l'inizio della coda.

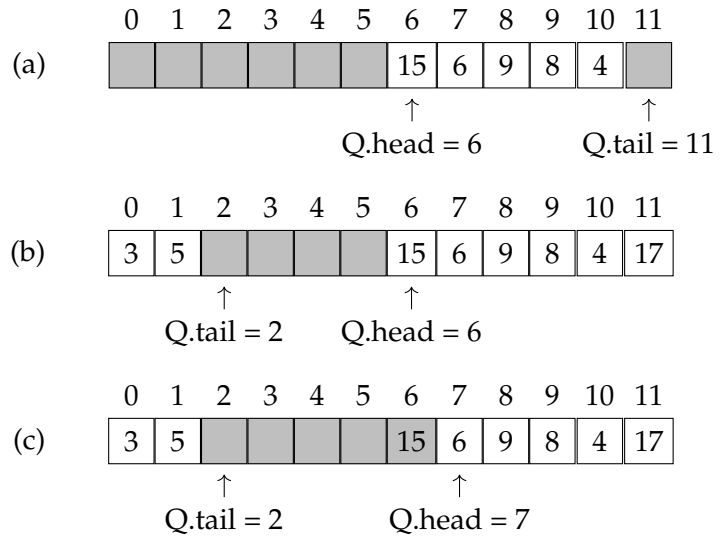


Figura 6.2: Una coda implementata con un array. Gli elementi di una coda appaiono soltanto nelle posizioni con sfondo bianco. (a) La coda ha 5 elementi nelle posizioni Q[7..1]. Nel caso (b) ci sono 8 elementi nella coda dopo le chiamate ENQUEUE(Q,17), ENQUEUE(Q,3), ENQUEUE(Q,5). Nell'ultimo caso (c) è effettuata una DEQUEUE(Q).

Se non si implementa una coda circolare, allora quando Q.tail==Q.head la coda è vuota, se si cerca di eseguire la DEQUEUE su una coda vuota c'è un errore di **underflow**, mentre se la struttura fosse piena e si cercasse di inserire altri elementi con la **ENQUEUE** ci sarebbe un caso di **overflow**. Le ricerche del minimo e del primo valori sono lineari, mentre l'operazione di inserimento è costante $\Theta(1)$.

```
1 ENQUEUE(Q,x)
2     Q[Q.tail] <- x
3     if Q.tail == Q.length
4         Q.tail <- 1
```

```
5       else
6           Q.tail <- Q.tail +1

1 DEQUEUE(Q)
2       x <- Q.head;
3       if Q.head == Q.length
4           Q.head <- 1
5       else
6           Q.head <- Q.head+1
7       return x
```

6.1.3 Liste concatenate

Una **lista concatenata** è una struttura dati i cui oggetti sono disposti in ordine lineare, esso è determinato da un puntatore in ogni oggetto: se la ogni oggetto della lista ha due puntatori che puntano rispettivamente all'elemento precedente e quello successivo la lista si chiama **doppiamente concatenata**, se invece è presente solo un puntatore è definita **concatenata**. Gli attributi puntatori sono chiamati *prev* che punta all'elemento precedente e *next* che punta a quello successivo. Consideriamo x un elemento appartenente alla lista, se x.*prev*== NULL x è detto elemento **testa**, se x.*next* == NULL allora x è chiamato elemento **coda** o **tail**. Ogni oggetto ha al suo interno un'attributo chiave che prende il nome di **key**. Una lista può essere ordinata in base ad un valore chiave: l'elemento minimo è la testa della lista e l'elemento massimo è la coda. Una lista può anche non essere ordinata, questi due modi per implementare le liste offrono delle complessità computazionali diverse sui tipi di operazione che vi effettuiamo: nella lista ordinata la ricerca del minimo e del primo elemento inserito è costante mentre l'inserimento è lineare $\Theta(n)$ poiché nel caso peggiore dobbiamo scorrere tutta la lista. Nel caso della lista non ordinata l'inserimento è costante e la ricerca del minimo e del primo elemento inserito è costante. La **priority queue** è una coda dove gli oggetti hanno una chiave su cui è basata una relazione di ordinamento, l'heap è una coda di priorità dove tutti le operazioni che vengono effettuate al suo interno hanno costo logaritmico.

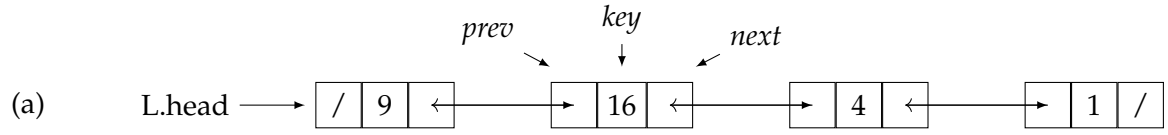


Figura 6.3: (a)Una lista doppiamente concatenata L. Ogni elemento della lista è un oggetto con attributi per la chiave e per i puntatori *prev* e *next* che puntano all'oggetto precedente e a quello successivo. L'attributo prev della testa e l'attributo next della coda valgono NULL.

La procedura LIST-SEARCH prende in input una lista concatenata L e una chiave k, se esiste un oggetto all'interno della lista con chiave k lo ritorna, altrimenti ritorna NULL.

```
1 LIST-SEARCH(L, k)
2       x <- L.head
```

```
3       while x NOT NULL AND x.key NOT k
4           x <- x.next
5       return x
```

Per effettuare una ricerca in una lista di n oggetti il tempo di esecuzione è $\Theta(n)$ poiché nel caso peggiore bisogna scorrere tutta gli n elementi. L'inserimento nella lista avviene con la procedura LIST-INSERT che prende in input una lista L e un oggetto x che deve essere inserito. L'inserimento può essere effettuato in testa oppure in coda: nel primo caso la complessità è O(1), mentre nel secondo è $\Theta(n)$.

```
1 LIST-INSERT-IN-CODA(L, x)
2       curr <- L.head
3           while curr.next NOT NULL
4               curr <- curr.next
5       curr.next <- x
6       x.prec <- curr
```

```
1 LIST-INSERT-IN-TESTA(L, x)
2       x.next <- L.head
3       x.prev <- NULL
4       if L.head == NULL
5           L.head <- x
6       else
7           L.head.prev <- x
```

La procedura LIST-DELETE rimuove un elemento x da una lista concatenata L. Deve ricevere un puntatore a x; poi elimina x dalla lista aggiornando i puntatori.

```
1 LIST-DELETE(L, x)
2       if x.prev NOT NULL
3           x.prev.next <- x.next
4       else
5           L.head <- NULL
6       if x.next NOT NULL
7           x.next.prev <- x.prev
```

Qui il ragionamento diventa più complesso, però iniziamo controllando se l'elemento x non è la testa: se è così l'elemento precedente ad x dovrà puntare all'elemento successivo di x, se dovesse essere la testa della lista allora la impostiamo a NULL. Successivamente controlliamo se x non è la coda: se dovesse essere così il puntatore *prev* dell'elemento successivo di x dovrà puntare all'elemento precedente di x.

6.2 Alberi

Rappresentiamo i singoli nodi di un albero con un oggetto. Analogamente alle liste concatenate, supponiamo che ogni nodo contenga i campi *key* e dei puntatori che puntano ad altri nodi.

6.2.1 Alberi binari

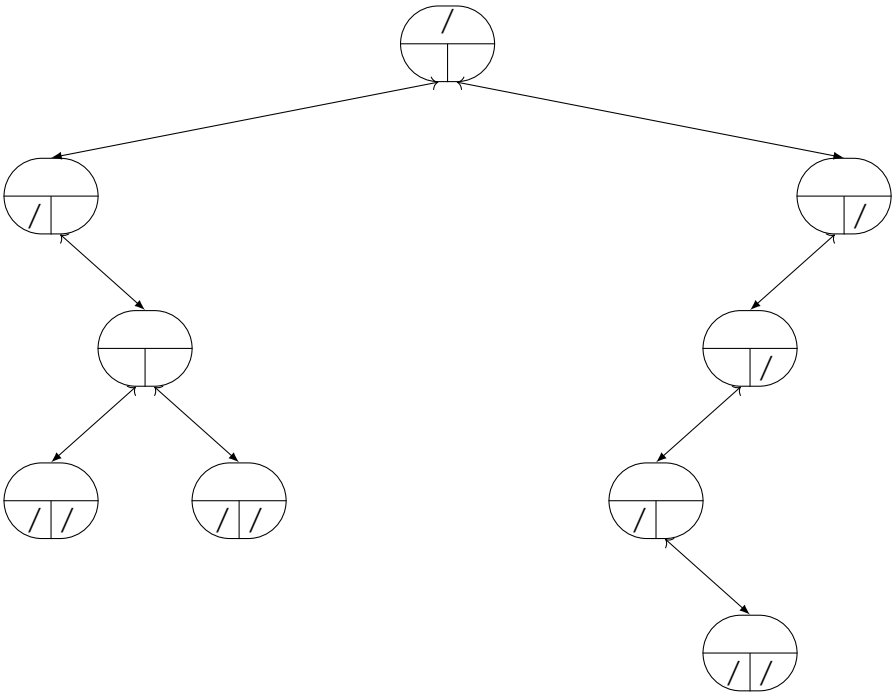


Figura 6.4: La rappresentazione di un albero binario T. Ogni nodo x ha i campi $p[x]$(in alt),$left[x]$(in basso a sinistra) e $right[x]$(in basso a destra).

Utilizziamo il campo $p[x]$ per memorizzare il puntatore del padre, $l[x]$ al figlio sinistro, $r[x]$ al figlio destro. Se $p[x] == NULL$ allora il nodo x è la radice. Se il nodo x non ha figlio sinistro $left[x] = NULL$ e simmetricamente per quello destro $right[x] = NULL$. Un'altra rappresentazione può essere data dagli con un numero arbitrario di figli, come prima ogni nodo contiene un puntatore p al padre. Ogni nodo x ha soltanto due puntatori:

- $left - child[x]$: punta al figlio più a sinistra del nodo x;
- $right - sibiling[x]$: punta al fratello di x immediatamente a destra.

6.2.2 Alberi binari di ricerca

Un albero binario di ricerca è organizzato come un albero binario, ma le chiavi di ogni nodo sono memorizzate in modo tale da far rispettare la **proprietà degli alberi binari di ricerca**: per ogni nodo x le chiavi del sotto albero sinistro sono minori o uguali della chiave x che è minore uguale delle chiavi del sotto albero destro. Ogni nodo dell'albero oltre al campo *key* ha altri 3 puntatori:p, l, r che rispettivamente rappresentano il puntatore al padre, al figlio sinistro e al figlio destro. Un albero di ricerca si dice **bilanciato** se rispetta questi tre canoni:

- la altezza del sottoalbero sinistro non differisce al più di uno dell'altezza del sottoalbero destro;

- il sottoalbero sinistro è bilanciato;
- il sottoalbero destro è bilanciato;

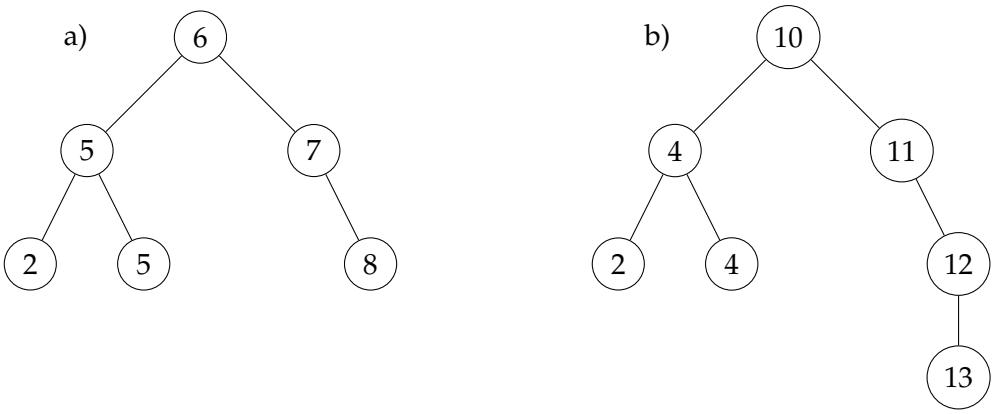


Figura 6.5: In questa figura vengono rappresentati due tipi di alberi: a) è un albero binario di ricerca, ciò si può notare dal fatto che il sottoalbero sinistro è composto da nodi che hanno chiave minore o uguale del nodo root e il sottoalbero destro con nodi con chiave maggiori o uguale di quella root. b) è anch'esso un albero binario di ricerca. Questi due alberi sono tutti e due bilanciati poiché la differenza di altezza di ciascun dei due sottoalberi che li forma è al più 1.

Ricerca

Una procedura molto utilizzata è la cercare un nodo con una data chiave in un albero binario di ricerca. L'implementazione di tale algoritmo è molto semplice:

```
1 TREE–SEARCH(x, k)
2     if (x == NULL or k == x.key)
3         return x
4     if k < x.key
5         return TREE–SEARCH(x.left, k);
6     return TREE–SEARCH(x.right, k);
```

La ricerca parte dalla radice e segue un cammino semplice verso il basso. Per ogni nodo x che incontra controlla se la chiave di x e k sono uguali, se ciò è vero ritorna il nodo, altrimenti se la chiave di x è maggiore di k, esegue il controllo sul sottoalbero sinistro di x, simmetricamente se la chiave di x è minore di k continua il controllo sul sottoalbero destro di x. Queste due scelte avvengono per la proprietà dell'albero di ricerca: per ogni nodo, il sottoalbero sinistro ha nodi con chiave minore uguale, mentre quello destro maggiori o uguali. Il **tempo di esecuzione** del TREE-SEARCH è O(h) dove h rappresenta l'altezza dell'albero.

Ricerca del minimo e massimo

Un elemento minimo di un albero binario di ricerca può essere trovato partendo dalla radice seguendo il percorso formato dai figli a sinistra(*left*) fino a quando non viene incontrato un

valore NULL.Il **tempo di esecuzione** di tutti e due gli algoritmi è dato dall'altezza dell'albero binario di ricerca, quindi TREE-MAXIMUM e TREE-MINIMUM $\in O(h)$.

```
1 TREE–MINIMUM(x)
2     while(x.left != NULL )
3         x <– x.left
4     return x
```

La ricerca dell'elemento massimo è pressoché identica a quella citata sopra, l'unico cambiamento è che il percorso da seguire è formato dai nodi a destra, cioè dal puntatore *right*.

```
1 TREE–MAXIMUM(x)
2     while(x.right != NULL)
3         x  <– x.right
4     return x
```

Inserimento in un albero binario di ricerca bilanciato

Il nostro obbiettivo è quello di cercare una struttura con inserimenti in tempo logaritmico. Gli alberi bilanciati permettono di effettuare l'inserimento di un nodo in tempo logaritmico però non sono grado di mantenere la struttura di albero bilanciato in tempo non lineare. Consideriamo il seguente esempio formato da un albero bilanciato senza nodo destro. Vogliamo inserire all'interno della struttura un nodo x con chiave più piccola di tutti gli altri (Figura 6.6). Per rendere l'elemento di nuovo bilanciato è necessario che tutti gli elementi dell'albero devono essere spostati, cioè all'interno di ogni nodo devo sistemare almeno un campo puntatore, dato che all'interno dell'albero ci sono n nodi, il **tempo minimo richiesto** è n.

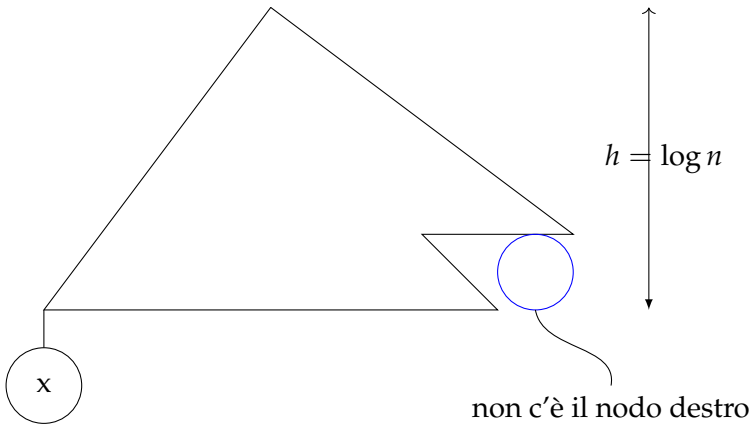


Figura 6.6: Esempio inserimento di un nodo all'interno di un albero bilanciato. Mantenere la struttura bilanciata in tempo logaritmico non è possibile o almeno non ne siamo ancora in grado.

6.2.3 Alberi RB

Un **albero rosso-nero** è un albero binario di ricerca dove ogni nodo ha un campo in più che determina il suo colore: rosso o nero. Su questo tipo di albero imponiamo condizioni **locali**: consideriamo un nodo generico x all'interno dell'albero, allora tutti i nodi adiacenti ad x devono avere delle determinate caratteristiche. Ci sono inoltre condizioni **globali**, cioè l'intero albero deve avere determinate caratteristiche. Un altro che permette di differirsi dagli alberi di ricerca è se mancasse un figlio o un padre ad un nodo, verrebbe inserito un nodo NIL. Ci sono quattro proprietà che descrivono un albero RB:

1. ogni nodo è rosso o nero;
2. ogni foglia è nera;
3. i figli di un nodo rosso sono neri, non è possibile avere due nodi rossi adiacenti;
4. per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri;

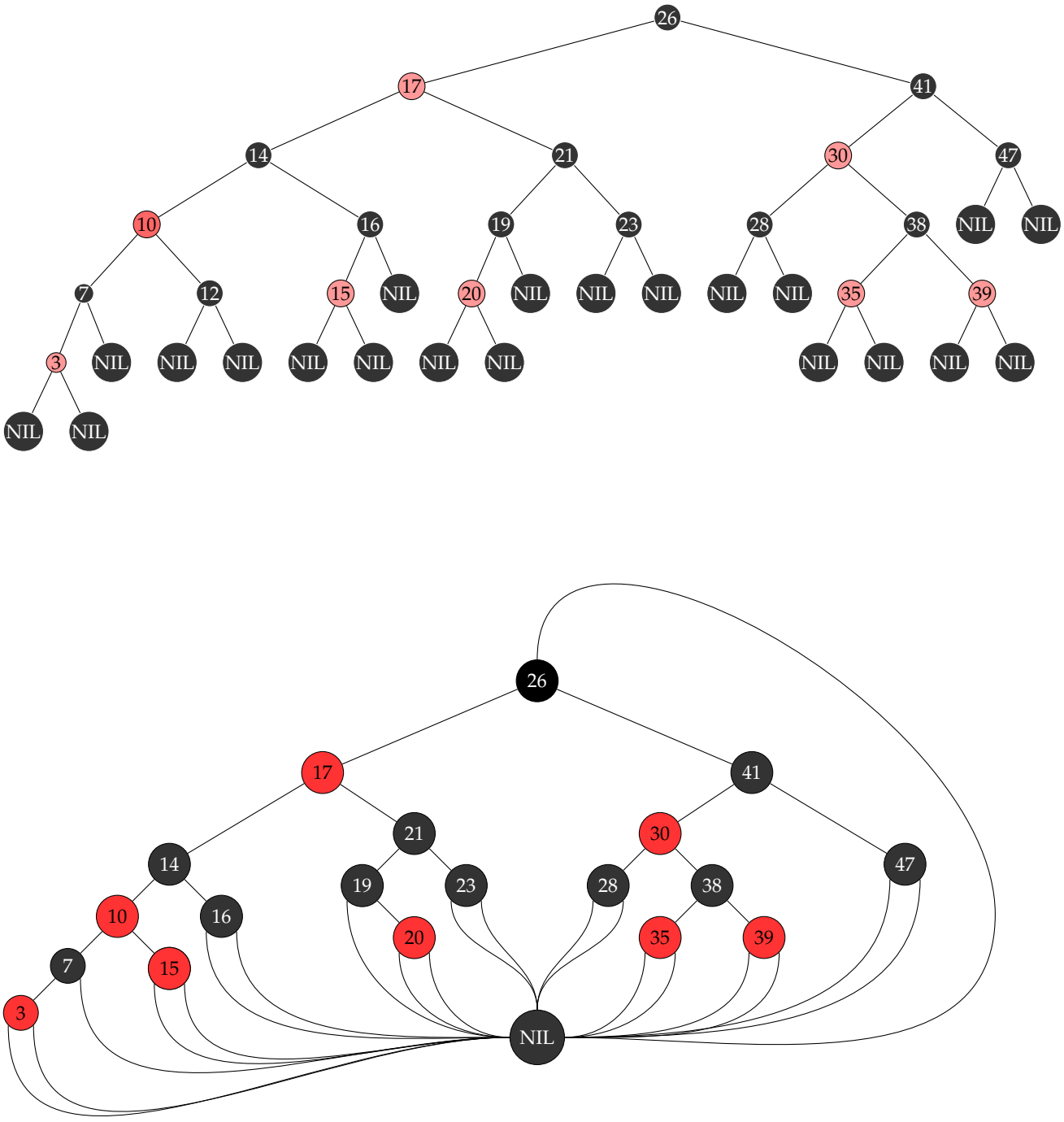
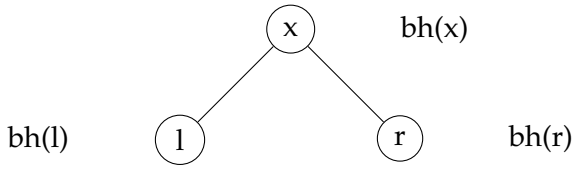


Figura 6.7: Ogni foglia rappresentata da NIL è nera. Nella seconda parte dell'immagine l'albero è lo stesso di prima dove tutti i nodi NIL sono stati sostituiti dall'unica sentinella NIL, che è sempre nera. NIL inoltre è il padre di root

L'implementazione mostrata dalla Figura 6.7 porta a un grande dispendio di memoria poiché con l'aggiunta dei nodi NIL nelle foglie, il numero dei nodi totali raddoppia. Quindi, un metodo più efficacie è che tutti i nodi che non hanno figli puntano al nodo sentinella NIL, così facendo non è necessario raddoppiare la struttura dati. L'uso della sentinella ci permette di trattare il nodo NIL come un nodo generico x di cui però non è possibile determinarne il padre. Definiamo **altezza nera** di un nodo x, indicata da $bh(x)$, il numero di nodi neri lungo un cammino semplice che inizia dal nodo x (ma non lo include) e finisce in una foglia.

Lemma: Per ogni nodo x il sottoalbero radicato in x contiene almeno $2^{bh(x)} - 1$ nodi interni. Scrivendola in un altro modo potrebbe dire che numero nodi interni $\geq 2^{bh(x)} - 1$ Per induzione sulla profondità h:

- h=0: $\underbrace{0}_{\text{numero nodi interni}} \geq 2^0 - 1$;
- $h = l + 1$:



$bh(l) \geq bh(x) - 1$ e $bh(l) \geq bh(x) - 1$.

Quindi se i nodi interni di x sono dati dalla somma dei nodi interni di l e r +1, la formula si può riscrivere come

$$\text{nodi interni di x} \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$$

Teorema 6.2.1. L'altezza massima di un albero rosso-nero con n nodi interni è $2\log(n+1)$.

Dimostrazione

Consideriamo T un RB-Albero di n nodi e h la sua profondità. Sia x la radice di T. Si nota che $bh(x) \geq \frac{h}{2}$ poiché ad ogni nodo rosso ce n'è uno nero. Quindi

$$n \geq 2^{bh(x)} - 1 \geq 2^{\frac{h}{2}} - 1$$
$$n + 1 \geq n^{\frac{h}{2}}$$
$$h \leq 2\log n + 1$$

Operazioni di Rotazione

Un tipo di operazione che utilizzeremo successivamente nell'inserimento e nella cancellazione di un nodo all'interno di un albero RB è la **rotazione** che può essere sinistra o destra. Dato che una è simmetrica rispetto all'altra ne implementeremo soltanto una. Consideriamo T un albero binario RB e x il nodo su cui vogliamo effettuare la rotazione. Inoltre assumiamo che la radice dell'albero sia presente.

```
RIGHT–ROTATE(T,x)
    y <- x.left          //Imposta y come figlio sinistro di x
    x.left <- y.right
    y.right.p <- x
    y.p <- x.p          //Collega il padre di x ad y
    y.right <- z
    x.p <- y
    IF y.p.right == x
        y.p.right <- y
    ELSE
        y.p.left <- y
```

Quando eseguiamo una rotazione a destra sul nodo x, supponiamo che suo figlio destro y non sia NIL. La rotazione destra "fa perno" sul collegamento tra x e y; il nodo y diventa la nuova radice del sottoalbero, mentre x diventa figlio destro di y e il figlio destro di x diventa figlio sinistro di x.

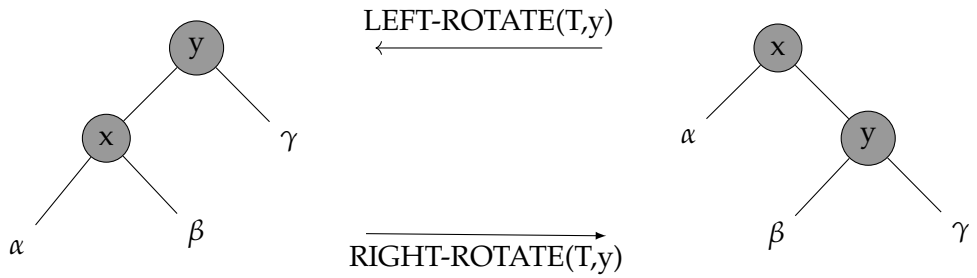


Figura 6.8: Rotazioni in un albero binario di ricerca. L'operazione LEFT-ROTATE trasforma la configurazione di destra con quella di sinistra e la configurazione a sinistra può essere trasformata nella configurazione a destra con l'operazione inversa RIGHT-ROTATE.

L'algoritmo di rotazione è implementato per mantenere le proprietà che caratterizzano un albero RB. Le operazioni effettuate modificano solamente i puntatori coinvolti, quindi la procedura verrà effettuata in tempo O(1).

Inserimento

L'inserimento in un albero RB di n nodi può essere effettuato nel tempo O($\log n$). L'algoritmo dell'inserimento di un nodo in un albero RB è presente sul libro, ma ci interesseremo della parte che permette di mantenere le 4 condizioni di un albero RB. Consideriamo un albero T e il nodo appena aggiunto z di colore rosso.

```
RB–INSERT–FIXUP(T,z)
    while z.p.pcolor == RED
        if z.p == z.p.p.left
            y <- z.p.p.right
                if y.color == RED
```

```
6                       z.p.color <- BLACK          //Caso 1
7                       y.color <- BLACK.           //Caso 1
8                       z.p.p.color <- RED          //Caso 1
9                       z <- z.p.p                  //Caso 1
10                  else
11                      if z == z.p.right
12                          z <- z.p.               //Caso 2
13                          LEFT-ROTATE(T,z)        //Caso 2
14                      z.p.color <- BLACK          //Caso 3
15                      z.p.p.color <- RED          //Caso 3
16                      RIGHT-ROTATE(T,z.p.p)       //Caso 3
17          else
18              (come la clausola precedente,
19              ma con "right" e "left" scambiati)
20      z <- root
```

Quali proprietà degli alberi RB possono essere violate prima della chiamata RB-INSERT-FIXUP, considerando che essa venga chiamata dopo aver inserito un nodo rosso nell'albero? La proprietà 1 continua ad essere valida, come la proprietà 2 dato che entrambi i figli del nuovo nodo rosso sono la sentinella T.NIL. Anche la proprietà 4 è garantita poiché il numero di nodi neri nel sistema è rimasto lo stesso. Quindi l'unica violazione avviene nei confronti della proprietà 3. L'algoritmo basa il proprio funzionamento su prendere l'anomalia e portarla nella radice dell'albero dove l'anomalia non sarà più presente. Focalizzandoci ora sul codice troviamo il ciclo **while**(2-19) che mantiene vera, all'inizio di ogni iterazione del ciclo, la seguente variante formata da tre parti:

a) Il nodo z è rosso;

b) Se $z.p$ è la radice,

c) Se ci sono violazioni delle proprietà degli alberi RB. Ce ne può essere al massimo una e riguarda il fatto che la radice non sia nera oppure la proprietà 3 che indica la presenza di un nodo rosso con un figlio rosso cioè che z e $z.p$ sono rossi.

Per descrivere l'algoritmo dobbiamo focalizzarci in ordine su inizializzazione, conservazione e conclusione. Dobbiamo dimostrare che le tre parti dell'invariante sono vere nell'istante in cui viene chiamata RB-INSERT-FIXUP:

- **Inizializzazione**: partiamo da un albero RB senza violazioni a cui abbiamo aggiunto un nodo rosso:

  a) Quando viene chiamata la funzione, z è il nodo rosso appena aggiunto;

  b) Se $z.p$ è la radice, allora $z.p$ non cambia colore e rimane nero prima della chiamata alla funzione;

  c) : l'albero RB rispetta 3 delle 4 condizioni prima della chiamata RB-INSERT-FIXUP;

L'unica violazione possibile è della proprietà 3 ed è attribuibile al fatto che z(*il nodo appena inserito*) e $z.p$(*il parent di z*) sono tutti e due di colore rosso.

- **Conservazione**: ci sarebbero sei casi da confrontare all'interno del **while**, ma tre di essi sono simmetrici agli altri tre, a seconda che il padre $z.p$ di p sia un figlio destro o sinistro del nonno $z.p.p$ di z(*riga 3*). Quindi il nodo $z.p.p$ esiste, e riguardo alla parte b) dell'invariante, se $z.p$ è la radice $z.p$ è nero.
- **Conclusione**: quando il ciclo termina accade quando $z.p$ è nero. Quindi non c'è più la violazione della proprietà 3.

Ora che abbiamo dimostrato che l'invariante è sempre vera ci possiamo dedicare sulla descrizione dell'algoritmo. Ci sono 3 casi, segnati anche dai commenti nel codice:

- **Caso 1: lo zio y di z è rosso**

  Questo caso viene eseguito se $z.p$ e y sono entrambi rossi. Poiché il nonno di z è nero, possiamo colorare di nero $z.p$ e y, risolvendo così il problema che z e $z.p$ sono entrambi rossi. Infine coloriamo di rosso $z.p.p$ per conservare la proprietà 4. Poi ripetiamo il ciclo **while** con il nonno di z come nuovo nodo z.

- **Caso 2: lo zio y di z è nero e z è un figlio destro**

  Caso 3: lo zio y di z è nero e z è un figlio sinistro

  Nei casi 2 e 3 il colore dello zio è nero, l'unica distinzione fra i due casi è se z è figlio sinistro di $z.p$ o destro. Nel caso 2 z è figlio destro ed effettuiamo subito una rotazione sinistra per trasformare la situazione nel caso 3 in cui il nodo z è figlio sinistro. Poiché z e $z.p$ sono tutti e due rossi, la rotazione non influisce nè sull'altezza nera e quindi non viola la proprietà 4. Nel caso 3, prima coloriamo il nodo $z.p$ di nero e successivamente coloriamo il nonno di z, $z.p$ di rosso. Effettuiamo una rotazione destra per mantenere la proprietà 4; dopodiché, dal momento che non abbiamo più due nodi rossi consecutivi, l'algoritmo è concluso.

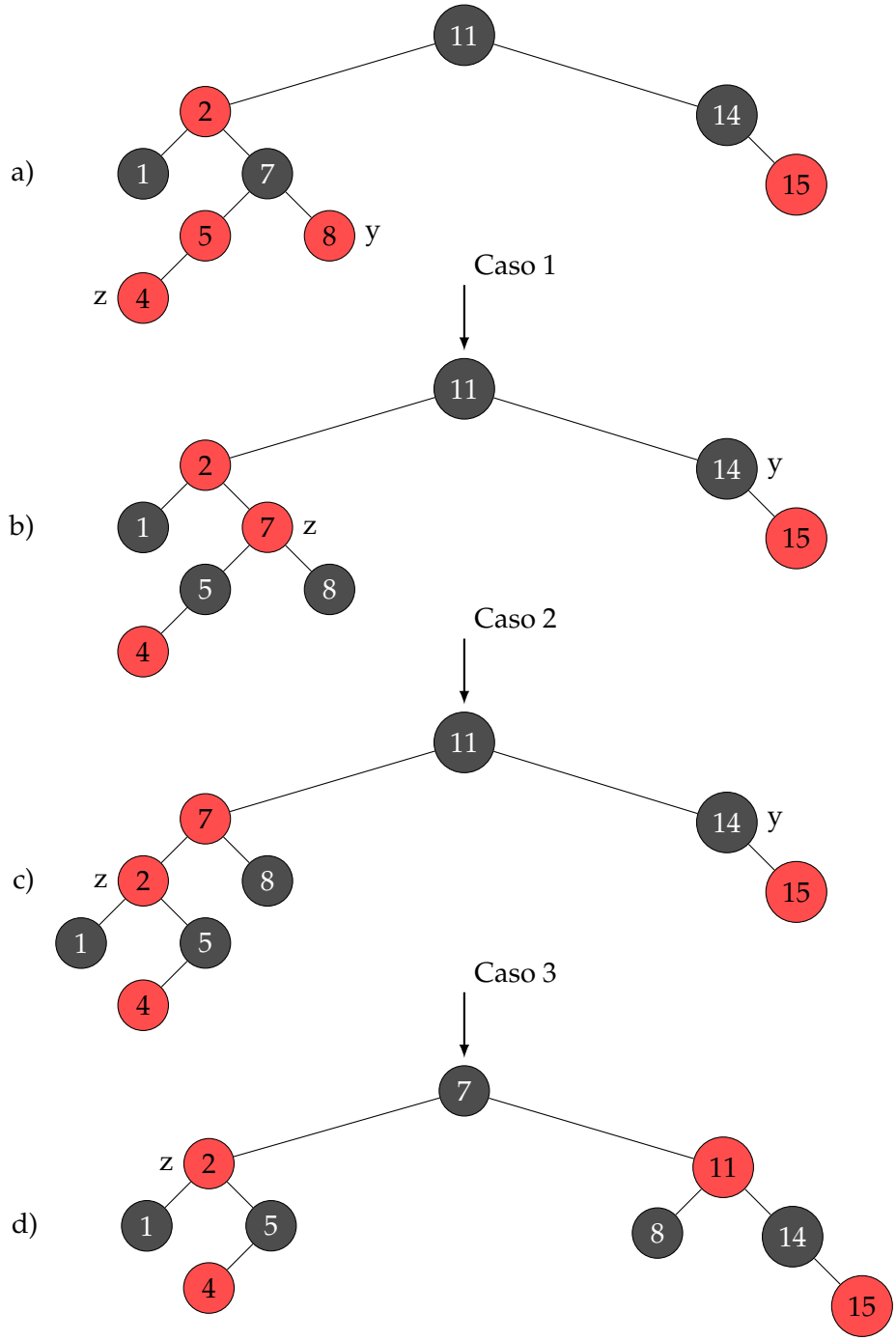


Figura 6.9: Il funzionamento di RB-INSERT-FIXUP. a) Un nodo z dopo l'inserimento. Poiché z e suo padre $z.p$ sono entrambi rossi, si verifica una violazione della proprietà 3. Poiché lo zio y di z è rosso, si applica il caso 1 del codice. I nodi sono ricolorati e il puntatore z si sposta verso l'alto nell'albero. b) Ancora una volta, z e suo padre sono entrambi rossi, ma lo zio y di z è nero. Poiché z è il figlio destro di $z.p$, si applica il caso 2. Viene effettuata una rotazione sinistra; l'albero risultante è illustrato in c. Adesso z è il figlio sinistro di suo padre e si applica il caso 3. Una ricolorazione e una rotazione destra genera l'albero illustrato in d), che è un valido albero RB.

Cancellazione

Un'altra operazione fondamentale effettuata sugli alberi RB è la cancellazione di un nodo. Analogamente al caso dell'inserimento, la cancellazione è effettuata in tempo $O(\log n)$. Considereremo soltanto il caso di ripristino dell'albero dopo una rimozione del nodo e daremo per assunto che l'eliminazione sia già stata effettuata e dobbiamo soltanto convalidare le proprietà di un albero RB che sono state violate.

Consideriamo ora il nodo y come il nodo che è stato eliminato, x come il nodo che si sposta sulla posizione originale del nodo y. Se il colore del nodo eliminato è nero ci sono 2 casi da risolvere:

- se entrambi x e $x.p$ erano rossi, allora è stata violata la proprietà 3;
- spostando y all'interno dell'albero, qualsiasi cammino semplice che prima conteneva y ora ha un nodo nero in meno e quindi la proprietà 4 non è più conservata. Un'idea per ripristinare questa proprietà è quella di assegnare al nodo x, che occupa la posizione originale di y, un colore nero extra, così il percorso a cui prima mancava un nodo nero, ora ce l'ha. Il nero extra in un nodo è legato al fatto che si punta a tale nodo, non al valore dell'attributo color che avrà sempre due scelte *rosso* e *nero*.

Il concetto base della procedura del ripristino dell'albero dopo una eliminazione è che in ciascun caso il numero dei nodi neri(compreso quello extra) rimanga invariato dopo qualsiasi trasformazione, ad esempio se il percorso da una radice ad una foglia prima della trasformazione era uguale a 2 anche dopo le operazioni il numero dei nodi neri di quel percorso era uguale a 2. Il nodo x se punta a nodi neri li rende **doppiamente neri**, mentre se punta a nodi rossi in **rosso e nero**, contribuisce rispettivamente con 1 e 2 al conteggio dei nodi neri. Consideriamo ora i quattro casi su cui su basa l'algoritmo di ripristino:

Caso 1: il fratello w di x è rosso

Il caso 1 si verifica quando il fratello di x, w, è rosso. Per la proprietà 3, un nodo rosso deve avere esclusivamente figli neri, perciò scambiamo il colore del nodo w con il colore del padre di x($x.p$) e poi si effettua una rotazione sinistra su $x.p$.Il numero dei nodi neri in ogni percorso è rimasto invariato prima e dopo la trasformazione e il fratello di x ora è nero e quindi abbiamo trasformato il caso 1 nel caso 2, 3 o 4.

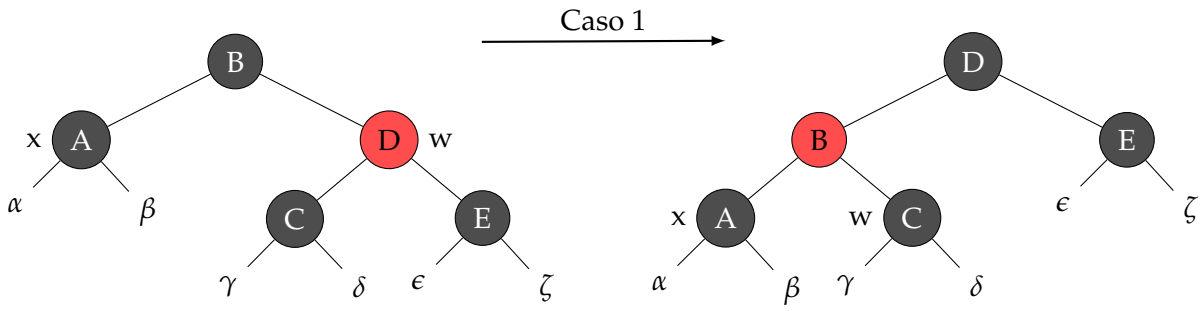


Figura 6.10: Il caso 1 è trasformato in caso 2,3 o 4 scambiando i colori dei nodi B e D ed effettuando una rotazione sinistra.

Caso 2: il fratello di w è nero ed entrambi i figli di w sono neri

Nel caso 2 entrambi i figli di w sono neri. Dato che anche w è nero, togliamo un *nero* sia da x che da w e aggiungiamo un nero extra a $x.p$ che inizialmente poteva essere rosso o nero. La proprietà 4 non viene violata e il numero di nodi neri in ogni percorso è rimasto invariato. Il caso 2 dilaga l'anomalia verso l'alto ed è l'unico fra i quattro che può ripetersi nel ciclo **while** e termina solamente quando il valore c dell'attributo color del nuovo x è RED, cioè se il padre iniziale di x aveva colore rosso.

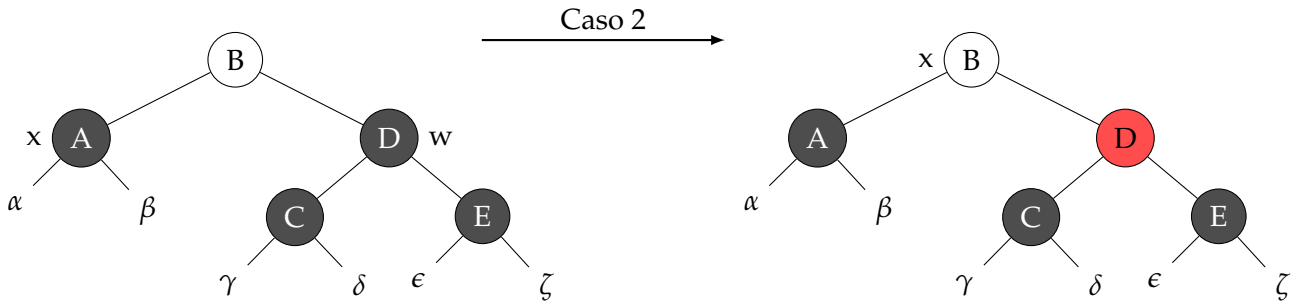


Figura 6.11: Nel caso 2, il nero extra viene spostato verso l'alto nell'albero, colorando di nero il nodo D e puntando x al nodo B. Se entrassimo nel caso 2 dopo il caso 1 il ciclo termina poiché il nodo B è sia nero che rosso.

Caso 3: il fratello di w è nero, il figlio sinistro di w è rosso e il figlio destro di w è nero Il caso 3 si verifica quando w è nero, suo figlio sinistro è rosso e quello destro nero. Scambiamo colore tra $w.left$ e w e successivamente effettuiamo una rotazione destra di w, rendendo $w.left$ la radice e w il suo nodo destro. Dato che al nodo w abbiamo assegnato il colore rosso e a $w.left$ nero, ci siamo ricondotti al caso 4.

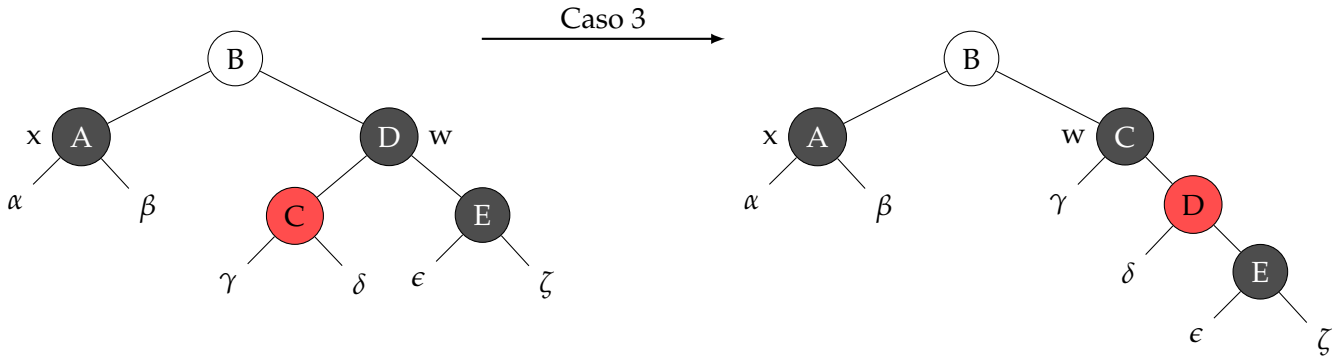


Figura 6.12: Il caso 3 è trasformato nel caso 4 scambiando i colori dei nodi C e D ed effettuando una rotazione destra su D.

Caso 4: il fratello di w è nero e il figlio destro di w è rosso e quello sinistro è nero Il caso 4 si verifica quando w, fratello di x è nero e il suo figlio destro è rosso. Al nodo w

assegniamo il colore del padre $x.p$ e a quest'ultimo il colore nero. Coloriamo di nero anche il figlio destro di w ed effettuiamo una rotazione sinistra di $x.p$ e puntiamo x alla radice. Con queste operazioni abbiamo rimosso il nero extra da x, rendendolo singolarmente nero, senza violare nessuna proprietà degli alberi RB. Se alla fine x punta alla radice, l'algoritmo conclude.

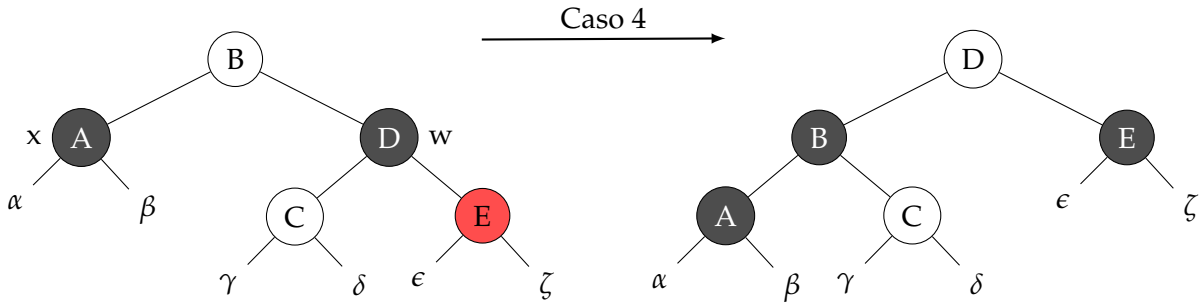


Figura 6.13: Il nero extra associato a x può essere rimosso cambiando il colore del nodo D col colore di suo padre, successivamente si colora il nodo E di nero e si effettua una rotazione sinistra sul nodo D.

Metodi per visualizzare un albero

Gli alberi possono essere visitati in diversi modi:

- **Preordine**: prima si visita il nodo e poi i suoi sottoalberi;
- **Inordine**: prima si visita il sottoalbero sinistro, poi il nodo e infine il sottoalbero destro;
- **Postordine**: prima si visitano i sottoalberi, poi il nodo;

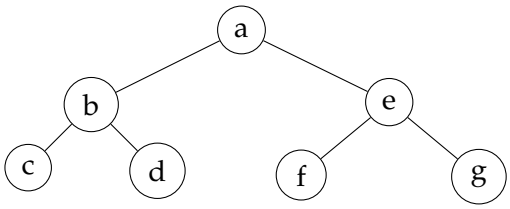


Figura 6.14: Albero d'esempio per effettuare gli attraversamenti

```
1 PreOrder(T)
2     if T != NIL
3         "visita T"
4         PreOrder(T.left)
5         PreOrder(T.right)
```

Sequenza: a b c d e f g

```
1 InOrder(T)
```

```
2     if T != NIL
3         InOrder(T.left)
4         "visita T"
5         InOrder(T.right)
```

Sequenza: c b d a f e g

```
1 PostOrder(T)
2     if T != NIL
3         PostOrder(T.left)
4         PostOrder(T.right)
5         "visita T"
```

Sequenza: c d b f g e a

6.2.4 Aumentare strutture dati

Per alcuni problemi è sufficiente una struttura dati elementari, ma per molti altri problemi occorre aumentare una struttura dati elementare memorizzando in essa delle informazioni aggiuntive.

Statistiche d'ordine dinamiche

Nella sezione 6 abbiamo introdotto il concetto di statistica d'ordine. In questo paragrafo, vedremo come modificare gli alberi RB in modo che qualsiasi tipo di statistica d'ordine possa essere calcolata in tempo logaritmico O(log n), insieme al calcolo del **rango** che verrà eseguito anch'esso in tempo logaritmico. Un **albero per statistiche d'ordine** T è un albero RB con un' informazione aggiuntiva in ogni nodo.

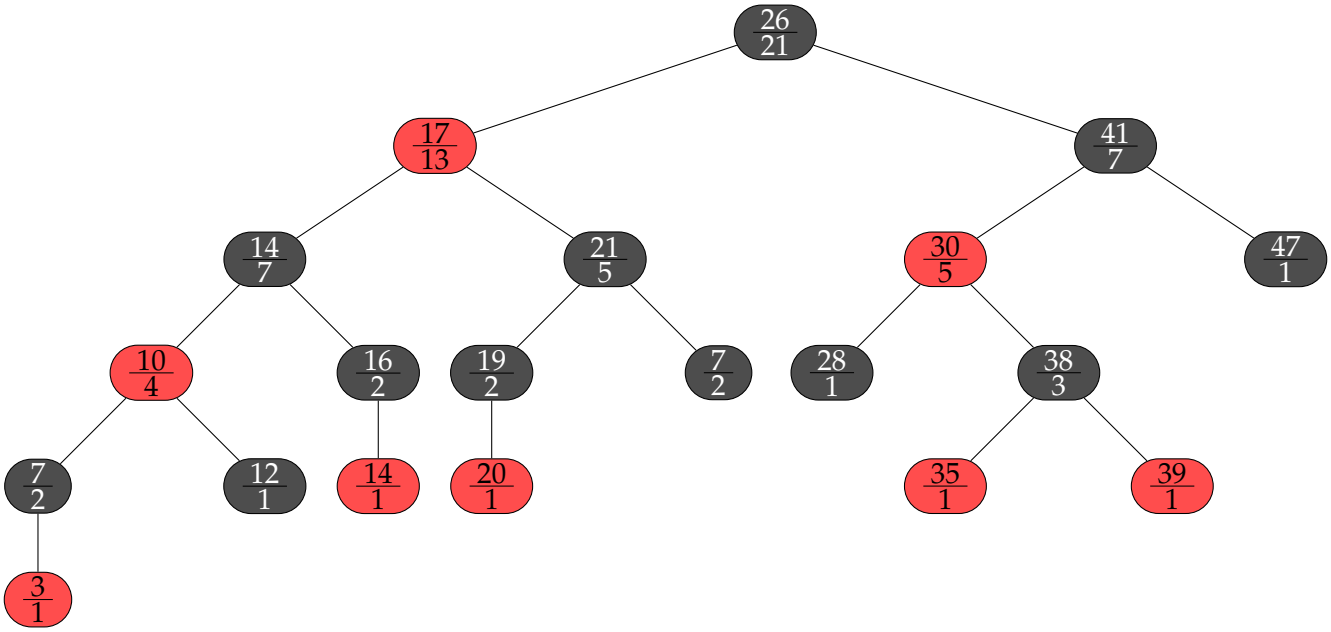


Figura 6.15: Un albero per statistiche d'ordine; è un albero RB aumentato. Oltre ai suoi attributi usuali, ogni nodo x ha un attributo $x.size$, che è il numero di nodi nel sottoalbero con radice in x

In un nodo x di un albero RB, oltre ai soliti $x.left$, $x.right$, $x.p$, $x.key$, $x.color$ è presente un'ulteriore campo $x.size$ che contiene il numero di nodi(interni) nel sottoalbero con radice in x (incluso lo stesso x), cioè la dimensione del sottoalbero. Se definiamo che la dimensione della sentinella è 0, ovvero impostiamo $T.nil.size = 0$, allora abbiamo l'identità

$$x.size = x.left.size + x.right.size + 1$$

Per esempio nella figura 6.15 la chiave 14 memorizzata in un nodo nero ha rango 5 e la chiave 14 memorizzata in un nodo rosso ha rango 6.

Ricerca di un elemento con un dato rango

La procedura OS-SELECT(x, i) restituisce un puntatore al nodo che contiene l'$i-esima$ chiave più piccola nel sottoalbero con radice in x. Per trovare il nodo con l'$i-esima$ chiave più piccola in un albero per statistiche di ordine T, chiamiamo OS-SELECT($T.root, i$)

```
1 OS-SELECT(x, i)
2     r <- x.left.size + 1
3     if i == r
4         return x
5     elseif i < r
6         return OS-SELECT(x.left, i)
7     else
8         return OS-SELECT(x.right, i - r)
```

Nella riga 2 di OS-SELECT calcoliamo r, il rango del nodo x nel sottoalbero di radice x. Il valore di $x.left.size$ è il numero di nodi che precedono x in un attraversamento simmetrico del sottoalbero con radice in x. Quindi, $x.left.size + 1$ è il rango di x all'interno del sottoalbero con radice in x. Se $i = r$ il nodo x è il nodo $i - esimo$ più piccolo, quindi la riga 4 restituisce x. Se $i < r$ significa che l'elemento $i - esimo$ si troverà nel sottoalbero sinistro che conterrà gli elementi più piccoli di x. Poiché ci sono r elementi nel sotto albero sinistro con radice in x che precedono il sottoalbero destro di x in un attraversamento simmetrico, se $i > r$, l'elemento $i - esimo$ più piccolo nel sottoalbero con radice in x è l'$(i - r)$-esimo elemento più piccolo nel sottoalbero con radice $x.right$.

Esempio

Consideriamo la ricerca del 17-esimo elemento più piccolo nell'albero per statistiche d'ordine nella figura 6.15. Iniziamo con x che punta alla radice, la cui chiave è 26, e con $i = 17$. Poiché la dimensione del sottoalbero sinistro di 26 è 12, il suo rango è 13. Quindi, sappiamo che il nodo con rango 17 è il quarto(17-13=4) elemento più piccolo nel sottoalbero destro di 26. Dopo la chiamata ricorsiva, x è il nodo con chiave 41 e $i = 4$. Poiché la dimensione del sottoalbero sinistro di 41 è 5, il suo rango all'interno del suo sottoalbero è 6. Quindi, sappiamo che il nodo con rango 4 è il quarto elemento più piccolo nel sottoalbero sinistro di 41. Dopo la chiamata ricorsiva, x è il nodo con chiave 30 e il suo rango all'interno del suo sottoalbero è 2. Quindi, effettuiamo di nuovo la ricorsione per trovare il secondo (4-2 = 2) elemento più piccolo nel sottoalbero con radice nel nodo con chiave 38. Adesso il suo sottoalbero sinistro ha dimensione 1; questo significa che esso è il secondo elemento più piccolo. Quindi, la procedura restituisce un puntatore al nodo con chiave 38. Poiché per ogni chiamata ricorsiva si scende di un livello nell'albero per statistiche d'ordine, il tempo totale di OS-SELECT, nel caso peggiore, è proporzionale all'altezza dell'albero. Poiché l'albero è un albero RB, la sua altezza è $O(\log n)$, dove n è il numero di nodi. Quindi, il tempo di esecuzione di OS-SELECT è $O(\log n)$ per un insieme dinamico di n elementi.

Determinare il rango di un elemento

Dato un puntatore a un nodo x in un albero per statistiche d'ordine T, la procedura OS-RANK restituisce la posizione di x nell'ordinamento lineare determinato da un attraversamento simmetrico dell'albero T.

```
1 OS–RANK(T, x)
2     r <- x.left.size +1
3     y <- x
4     while y != T.root
5         if y == y.p.right
6             r <- r + y.p.left.size +1
7         y <- y.p
```

Il rango di x può essere considerato come il numero di nodi che prendono x in un attraversamento simmetrico dell'albero, più 1 per x stesso. All'inizio di ogni del ciclo **while** (righe 4-7), r è il rango di $x.key$ nel sottoalbero con radice nel nodo y.

Inizializzazione: prima della prima iterazione, la riga 1 assegna a r il rango di $x.key$ all'interno del sottoalbero con radice in x. L'assegnazione $y = x$ meòòa riga 3 rende l'invariante vera la prima volta che viene eseguito il test nella riga 4.

Conservazione: alla fine di ogni iterazione del ciclo **while**, poniamo $y = y.p$. Quindi dobbiamo dimostrare che, se r è il rango di $x.key$ nel sottoalbero con radice in y all'inizio del corpo del ciclo, allora r è il rango di $x.key$ nel sottoalbero con radice in $y.p$ alla fine del corpo del ciclo. In ogni iterazione del ciclo **while** consideriamo il sottoalbero con radice in $y.p$, quindi dobbiamo aggiungere i nodi del sottoalbero con radice nel fratello di y che precedono x in un attraversamento simmetrico, più 1 per $y.p$, se anche questo nodo precede x. Se y è un figlio sinistro, allora né $y.p$ né altri nodi nel sottoalbero destro di $y.p$ precedono x, quindi lasciamo r invariato. Altrimenti, y è un figlio destro e tutti i nodi del sottoalbero sinistro di $y.p$ precedono x, come pure lo stesso $y.p$ Quindi, nella riga 6, aggiungiamo $y.p.left.size + 1$ al valore corrente di r.

Conclusione: il ciclo termina quando $y = T.root$, sicché il sottoalbero con radice in y è l'intero albero. Dunque, il valore di r è il rango di $x.key$ nell'intero albero.

Se eseguissimo OS-RANK nell'albero per statistiche d'ordine della figura 6.15 per trovare il rango del nodo con chiave 38, otteniamo la seguente sequenza di valori per $y.key$ e r all'inizio del ciclo **while**:

iterazione	y.key	r
1	38	2
2	30	4
3	41	4
4	26	17

Tabella 6.1: Il nodo $y.key$ è 38 ed r è 1+1 = 2, si entra nel ciclo e dato che il nodo y è nodo destro, r viene ricalcolato e vale 4. Viene assegnata ad y il valore del padre e si procede con la seconda iterazione. Ora y è 30 e dato che è figlio sinistro, il valore di r nella seconda iterazioni rimane invariato. Alla terza iterazione $y.key$ è 41, ed essendo nodo destro il rango viene ricalcolato e vale 17(valore vecchio+size del fratello sinistro+1 che rappresenta il nodo padre), y punta ora al nodo root e si procede con la quarta iterazione che finisce poiché il nodo y è il nodo root. y

Gestione delle dimensioni dei sottoalberi

Dato l'attributo *size* in ogni nodo, OS-SELECT e OS-RANK possono calcolare rapidamente le informazioni sulle statistiche d'ordine. Tuttavia, questo lavoro risulterebbe inutile se tali attributi non potessero essere gestiti con efficienza nelle operazioni di base che modificano gli alberi rosso-neri. Come detto nel paragrafo a riguardo agli alberi, l'inserimento di un nodo si svolge in due fasi. Nella prima fase, si discende dalla radice dell'albero, inserendo il nuovo nodo come figlio di un nodo esistente. Nella seconda fase si risale, cercando di rimuovere l'anomalia creata con la prima fase per conservare la proprietà degli alberi rosso-neri. Nella prima fase, per gestire le dimensioni dei sottoalberi, incrementiamo semplicemente $x.size$ per ogni nodo x nel cammino semplice attraversato dalla radice fino alle foglie. Il nuovo

nodo che viene aggiungo ha l'attributo *size* pari a 1. Come sappiamo, la profondità di un albero rosso-nero è logaritmica rispetto al numero di nodi presenti in esso, quindi nel percorso per andare ad x è O(log n), quindi il costo addizionale per gestire gli attributi *size* è O(log n). Nella seconda fase, le uniche modifiche strutturali dell'albero rosso-nero di base sono provocate dalle rotazioni, che sono al massimo due. Inoltre, una rotazione è un' operazione locale: soltanto due nodi hanno gli attributi *size* invalidati. Al codice della procedura LEFT-ROTATE(T,x) aggiungiamo le seguenti righe:

```
1 y.size = x.size
2 x.size = x.left.size + x.right.size + 1
```

Poiché vengono effettuate al massimo due rotazioni durante l'inserimento in un albero rosso-nero, occorre soltanto un tempo addizionale O(1) per aggiornare gli attributi *size* nella seconda fase. Quindi, il tempo totale per completare l'inserimento in un albero per statistiche d'ordine di n nodi è O(log n), che è asintoticamente uguale a quello di un normale albero rosso-nero. Anche la cancellazione da un albero rosso-nero è formata da due fasi: la prima opera sull'albero di ricerca sottostante; la seconda provoca al massimo tre rotazioni, senza altre modifiche strutturali. La prima fase o rimuove un nodo y oppure lo sposta più in alto nell'albero. Per aggiornare le dimensioni dei sottoalberi, seguiamo un cammino semplice dal nodo y (partendo dalla sua posizione originale nell'albero) fino alla radice, riducendo il valore dell'attributo size per ogni nodo che incontriamo. Poiché questo cammino semplice ha una lunghezza O(log n) in un albero rosso-nero di n nodi, il tempo aggiuntivo che viene impiegato per gestire gli attributi *size* nella prima fase è O(log n). Le O(1) rotazioni nella seconda fase della cancellazione possono essere gestite come è stato fatto nell'inserimento. Quindi, le operazioni di inserimento e cancellazione, inclusa la gestione degli attributi *size*, richiedono un tempo O(log n) su di un albero per statistiche d'ordine di n nodi.

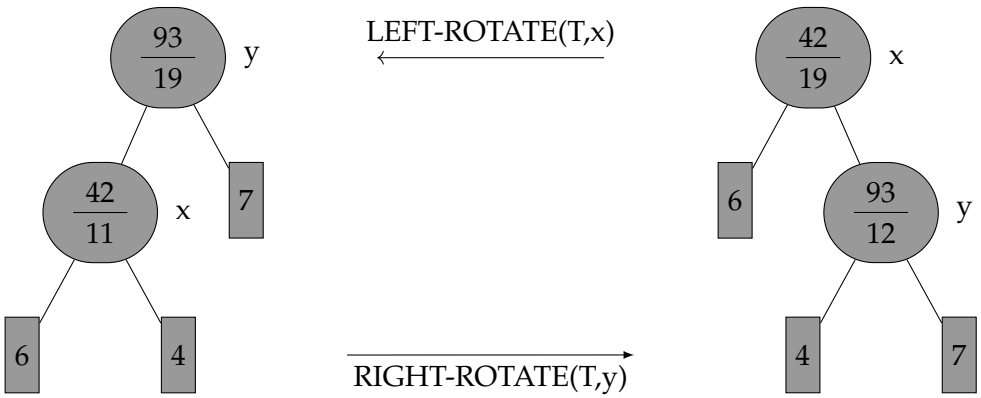


Figura 6.16: Aggiornamento delle dimensione dei sottoalberi durante le rotazioni. I due nodi i cui attributi *size* devono essere aggiornati sono quelli uniti dal collegamento attorno ai quali si effettua la rotazione. Gli aggiornamenti sono locali, richiedendo soltanto le informazioni *size* memorizzate in x, y e nelle radici dei sottoalberi

7 Programmazione dinamica

La programmazione dinamica risolve i problemi combinando le soluzioni dei sottoproblemi (in questo contesto, con il termine "programmazione" facciamo riferimento a un metodo tabulare, non alla scrittura del codice per un calcolatore). Se gli algoritmi *divide et impera* suddividono un problema in sottoproblemi indipendenti, risolvono in modo ricorsivo i sottoproblemi e poi combinano le soluzioni per risolvere il problema originale, la programmazione dinamica **può essere applicata** quando i sottoproblemi non sono indipendenti, ovvero quando i sottoproblemi hanno in comune dei sottoproblemi. Un algoritmo di programmazione dinamica risolve ciascun sottoproblema una sola volta e salva la sua soluzione in una tabella, evitando così il lavoro di ricalcolare la soluzione ogni volta che si presenta il sottosottoproblema. La programmazione dinamica, tipicamente, si applica ai **problemi di ottimizzazione**. Ogni soluzione ha un valore e si vuole trovare una soluzione con il valore ottimo, ricordandoci che ci possono essere più soluzioni che raggiungono il valore ottimo. Il processo di sviluppo di un algoritmo di programmazione dinamica può essere suddiviso in una sequenza di quattro fasi:

1. Caratterizzare la struttura di una soluzione ottima;
2. Definire in modo ricorsivo il valore di una soluzione ottima;
3. Calcolare il valore di una soluzione ottima, di solito con uno schema bottom-up;
4. Costruire una soluzione ottima dalle informazioni calcolate;

Le fasi 1-3 formano la base per risolvere un problema applicando la programmazione dinamica. La fase 4 può essere omessa se è richiesto soltanto il valore di una soluzione ottima. Quando dobbiamo eseguire la fase 4, a volte memorizziamo delle informazioni aggiuntive durante il calcolo della fase 3 per semplificare la costruzione di una soluzione ottima.

7.1 Moltiplicare una sequenza di matrici

Data una sequenza di n matrici $\langle A_1, A_2, \ldots, A_n \rangle$, vogliamo calcolare il prodotto

$$A_1 A_2 \ldots A_n$$

La moltiplicazione delle matrici è associativa, quindi tutte le parentesizzazioni forniscono lo stesso prodotto. Un prodotto di matrici è **completamente parentesizzato** se è una singola matrice oppure è il prodotto, racchiuso tra parentesi, di due prodotti di matrici completamente parentesizzati. Se la sequenza delle matrici è $\langle A_1, A_2, A_3, A_4 \rangle$, il prodotto $A_1 A_2 A_3 A_4$ può essere completamente perentesizzato in cinque modi distinti:

- $(A_1(A_2 A_3 A_4))$;

- $(A_1(A_2A_3)A_4)$;
- $((A_1A_2A_3)A_4)$;
- $((A_1A_2)(A_3A_4)$;

Il modo in cui parentesizziamo una sequenza di matrici può avere un impatto notevole sul costo per calcolare il prodotto. L'algoritmo standard per moltiplicare due matrici è il seguente

```
1  MATRIX–MULTIPLY(A, B)
2      if A.columns ≠ B.rows
3          error "dimensioni non compatibili"
4      else Sia C una nuova matrice A.rows x B.columns
5          for i = 1 to A.rows
6              for j = 1 to B.columns
7                  c_ij = 0
8                  for k = 1 to A.columns
9                      c_ij = c_ij + a_ik * b_kj
10         return C
```

Possiamo moltiplicare due matrici A e B soltanto se **sono compatibili**: il numero di colonne di A deve essere uguale al numero di righe di B. Se A è una matrice $p \times q$ e B è una matrice $q \times r$, la matrice risultante C è una matrice $p \times r$. Il tempo per calcolare C è dominato dal numero di prodotti scalari nella riga 8, che è pqr. Per mostrare come il costo per moltiplicare le matrici dipenda dallo schema di parentesizzazione, consideriamo il problema di moltiplicare una sequenza di tre matrici $\langle A_1, A_2, A_3\rangle$. Le dimensioni delle tre matrici sono rispettivamente, $10 \times 100, 100 \times 5$ e 5×50. La sequenza di matrici può anche essere scritta come segue

$$\langle A_1, A_2, A_3\rangle = \langle 10, 100, 5, 50\rangle$$

Con questa notazione è possibile identificare le tre matrici in maniera abbastanza semplice, se assegniamo la lettera p alla sequenza e con un' indice i è possibile scorrerla, allora:

- $A_1 : p_0 * p_1 = 10 \times 100$;
- $A_2 : p_1 * p_2 = 100 \times 5$;
- $A_3 : p_2 * p_3 = 5 \times 50$;

Quindi una matrice A_i appartenente alla sequenza p può essere determinata come $p_{i-1} * p_i$. Ritornando al costo, possiamo moltiplicare in due modi diversi:

- $((A_1A_2)A_3)$: eseguiamo $10 * 100 * 5 = 5000$ prodotti. scalari per calcolare la matrice 10×5 risultante del prodotto A_1A_2, più altri $10 * 5 * 50 = 2500$ per moltiplicare la matrice del prodotto tra A_1A_2 e A_3. Il costo totale è quindi 7500 ;
- $(A_1(A_2A_3))$: eseguiamo $100 * 5 * 50 = 25000$ prodotti scalari per calcolare la matrice 100×50 risultante del prodotto A_2A_3, più altri $10 * 100 * 50 = 50000$ prodotti scalari per moltiplicare A_1 per questa matrice, per un totale di 75000 prodotti scalari;

Come si può vedere su un input anche abbastanza semplice formato solamente da tre matrici il costo aumenta di 10 volte.

Il **problema della moltiplicazione di una sequenza di matrici** può essere definito in questo modo: data una sequenza di n matrici $\langle A_1, A_2, \ldots, A_n \rangle$, dove la matrice A_i ha dimensione $p_{i-1} \times p_i$ per i = $1, 2, \ldots, n$, determinare lo schema di parentesizzazione completa del prodotto $A_1 A_2 \ldots A_n$ che minimizza il numero di prodotti scalari.

Contare il numero di parentesizzazioni Vogliamo dimostrare che un controllo esaustivo di tutti i possibili schemi di parentesizzazione non ci consente di ottenere un algoritmo efficiente. Indichiamo con $P(n)$ il numero di parentesizzazioni alternative di una sequenza di n matrici. Quando $n = 1$, c'è una sola matrice e, quindi, un solo schema di parentesizzazione. Quando $n \geq 2$, un prodotto di matrici completamente parentesizzato è il prodotto di due sottoprodotti di matrici completamente parametrizzati e la suddivisione fra i due sottoprodotti può avvenire fra la k-*esima* e la (k+1)-esima per qualsiasi $k = 1, 2, \ldots, n-1$. Quindi, otteniamo la ricorrenza

$$P(n) = \begin{cases} 1 \text{ se } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) \text{ se } n \geq 2 \end{cases}$$

L'equazione di ricorrenza ha soluzione $\omega(2^n)$ cioè il numero di soluzioni è esponenziale in n; pertanto la tecnica di utilizzare la forza bruta (*bruteforce*) è una strategia inadeguata.

7.1.1 Applicare la programmazione dinamica

Per utilizzare il metodo della programmazione dinamica bisogna seguire le quattro fasi citati precedentemente:

1. Caratterizzare la struttura di una soluzione ottima;
2. Definire in modo ricorsivo il valore di una soluzione ottima;
3. Calcolare il valore di una soluzione ottima;
4. Costruire una soluzione ottima dalle informazioni calcolate;

Fase 1: struttura di una parentesizzazione ottima

La prima fase consiste nel trovare la sottostruttura ottima e poi utilizzare questa sottostruttura per costruire una soluzione ottima del problema delle soluzioni ottime dei sottoproblemi. Adottiamo la notazione $A_{i\ldots j}$, dove $i \leq j$, per la matrice che si ottiene calcolando il prodotto $A_i A_{i+1} \ldots A_j$. Se il problema non è banale, cioè $i < j$, allora qualsiasi parentesizzazione del prodotto $A_i \ldots A_j$ deve suddividere il prodotto fra A_k e A_{k+1} per qualche intero k nell'intervallo $i \leq k \leq j$; ovvero per qualche valore di k, prima calcoliamo le matrici $A_{i\ldots k}$ e $A_{k+1\ldots j}$ e poi, le moltiplichiamo per ottenere il prodotto finale $A_{i\ldots j}$:

$$A_i \ldots A_j = ((A_i A_{i+1} \ldots A_k)(A_{k+1} \ldots A_j))$$

Il costo totale di questa parentesizzazione è il costo per calcolare la matrice $A_{i\ldots k}$, più il costo per calcolare la matrice $A_{k+1\ldots j}$, più il costo per moltiplicare queste due matrici. Supponiamo che una parentesizzazione ottima di $A_i A_{i+1} \ldots A_j$ suddivida il prodotto fra A_k

e A_{k+1}. Allora la parentesizzazione della prima sottosequenza $A_i A_{i+1} \dots A_k$ all'interno di questa parentesizzazione ottima di $A_i A_{i+1} \dots A_j$ deve essere una parentesizzazione ottima di $A_i A_{i+1} \dots A_k$[1]. Abbiamo visto che qualsiasi soluzione di un'istanza non banale del problema della moltiplicazione di una sequenza di matrici richiede che il prodotto venga suddiviso in due sottoprodotti; inoltre qualsiasi soluzione ottima contiene al suo interno soluzioni ottime delle istanze dei sottoproblemi. Possiamo costruire una soluzione ottima di un'istanza del problema della moltiplicazione di una sequenza di matrici suddividendo il problema in due sottoproblemi, trovando le soluzioni ottime delle istanze dei sottoproblemi e, infine, combinando le soluzioni ottime dei sottoproblemi.

Fase 2: una soluzione ricorsiva

Per il problema della moltiplicazione di una sequenza di matrici, scegliamo come sottoproblemi i problemi di determinare il costo minimo di una parentesizzazione di $A_i A_{i+1} \dots A_j$ per $1 \leq i \leq j \leq n$. Sia $m[i, j]$ il numero minimo di prodotti scalari richiesti per calcolare la matrice $A_{i\dots j}$; per il problema principale, il costo del metodo più economico per calcolare $A_{1\dots n}$ sarà quindi $m[1, n]$. Possiamo definire $m[i, j]$ ricorsivamente in questo modo. Se $i = j$, il problema è banale; la sequenza è formata da una matrice $A_{i\dots i} = A_i$, quindi $m[i, i] = 0$ per $i = 1, 2, \dots, n$. Quando $i < j$ sfruttiamo la struttura di una soluzione ottima ottenuta nella fase 1. Supponiamo che la parentesizzazione ottima suddivida il prodotto $A_i A_{i+1} \dots A_j$ fra A_k e A_{k+1}, dove $i \leq k < j$. Quindi $m[i, j]$ è uguale al costo minimo per calcolare i sottoprodotti $A_{i\dots k}$ e $A_{k+1\dots j}$, più il costo per moltiplicare queste due matrici. Ricordando che ogni matrice A_i è $p_{i-1} \times p_i$, il calcolo del prodotto delle matrici $A_{i\dots k} A_{k+1\dots j}$ richiede $p_{i-1} p_k p_j$ prodotti scalari. Quindi otteniamo

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

I valori possibili per k sono $j - i$, poiché la parentesizzazione ottima deve utilizzare uno di questi valori di k, dobbiamo semplicemente controllarli tutti per trovare il migliore. Quindi la nostra definizione ricorsiva per il costo minimo di una parentesizzazione del prodotto $A_i A_{i+1} \dots A_j$ diventa

$$m[i, j] = \begin{cases} 0 \text{ se } i = j \\ \min m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \text{ se } i < j \end{cases}$$

Fase 3: calcolo dei costi ottimi

Osserviamo che ci sono relativamente pochi problemi distinti: un problema per ogni possibile scelta di i e j, con $1 \leq i \leq j \leq n$. Un algoritmo ricorsivo può incontrare ciascun sottoproblema più volte nelle varie diramazioni del suo albero di ricorsione, questa caratteristica è la seconda peculiarità dell'applicabilità della programmazione dinamica. Implementeremo il metodo bottom-up tabulare con la procedura MATRIX-CHAIN-ORDER riportata più sotto. Questa procedura assume che la matrice A_i abbia dimensioni $p_{i-1} \times p_i$ per

[1] La dimostrazione è possibile trovarla sul libro a pagina 309, ma semplicemente afferma che se la sequenza da $A_i \dots A_j$ è ottima, se dovesse esistere un'altra sequenza $A_i \dots A_k$ migliore di quella contenuta in $A_i \dots A_j$ e si effettuasse uno scambio tra esse, allora esisterebbe un'altra sequenza $A_i \dots A_j$ migliore di quella ottima perciò contraddizione non può esistere

$i = 1, 2, \ldots, n$. L'input è una sequenza $p = \langle p_0, p_1, \ldots, p_n \rangle$, dove $p.length = n + 1$. La procedura usa una tabella ausiliaria $m[1 \ldots n, 1 \ldots n]$ per memorizzare i costi $m[i, j]$ e una tabella ausiliaria $s[1 \ldots n, 1 \ldots n]$ che registra l'indice k cui corrisponde il costo ottimo nel calcolo di $m[i, j]$.

```
1  MATRIX–CHAIN–ORDER(p)
2      n = p.length – 1
3      Siano m[1...n,1...n] ed s[1...n,1...n] due nuove tabelle
4      for i = 1 to n
5          m[i,i] = 0
6      for l = 2 to n // l è la lunghezza della sequenza
7          for i = 1 to n – l + 1
8              j = i + l – 1
9              m[i,j] = ∞
10             for k = i to j – 1
11                 q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
12                 if q < m[i,j]
13                     m[i,j] = q
14                     s[i,j] = k
```

L'algoritmo prima calcola $m[i, i] = 0$ per $i = 1, 2, \ldots, n$ (i costi minimi per le sequenze di lunghezza $l = 1$) nelle righe 3-4. Poi calcola m[i, i + 1] per $i = 1, 2, \ldots, n - 1$ (i costi minimi per sequenze di lunghezza $l = 2$) e così via fin quando non si raggiunge n. In ciascun passo, il costo calcolato $m[i, j]$ nelle righe 10-13 dipende soltanto dagli elementi della tabella $m[i, k]$ e $m[k + 1, j]$ già calcolati. Poiché abbiamo definito m[i,j] soltanto per $i \leq j$, viene utilizzata solamente la posizione della tabella m che si trova subito sopra la diagonale principale. Da un semplice esame della struttura annidata dei cicli della procedura MATRIX-CHAIN-ORDER si deduce che il tempo di esecuzione dell'algoritmo è pari a $O(n^3)$ e richiede lo spazio $\Theta(n^2)$ per memorizzare le tabelle m e s. Quindi, la procedura MATRIX-CHAIN-ORDER è molto più efficiente del metodo con tempo esponenziale che elenca tutte le possibili parentesizzazioni controllandole una per una.

Fase 4: costruire una soluzione ottima La procedura MATRIX-CHAIN-ORDER determina il numero ottimo di prodotti scalari richiesti per moltiplicare una sequenza di matrici, ma non dimostra direttamente come moltiplicare le matrici. La tabella $s[1 \ldots n, 1 \ldots n]$ ci fornisce le informazioni per farlo. Ogni posizione $s[i, j]$ registra quel valore di k per il quale la parentesizzazione ottima di $A_i A_{i+1} \ldots A_j$ suddivide il prodotto fra A_k e A_{k+1}. Quindi, sappiamo che il prodotto finale delle matrici nel calcolo ottimale di $A_{1 \ldots n}$ è $A_{1 \ldots s[1,n]} A_{s[1,n]+1 \ldots n}$. I prodotti precedenti possono essere calcolati ricorsivamente, perché $s[1, s[1, n]]$ determina l'ultimo prodotto nel calcolo di $A_{s[1,n]+1 \ldots n}$. La seguente procedura ricorsiva produce una parentesizzazione ottima di $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, dati gli indici i e j e la tabella s calcolata da MATRIX-CHAIN-ORDER. La chiamata iniziale di PRINT-OPTIMAL-PARENS(s,1,n) produce una parentesizzazione ottima di $\langle A_1, A_2, \ldots, A_n \rangle$

```
1  PRINT–OPTIMAL–PARENS(s, i, j)
2      if i == j
```

```
3           Stampa“A“_i
4       else
5           Stampa  “(“
6           PRINT–OPTIMAL–PARENS(s, i, s[i, j])
7           PRINT–OPTIMAL–PARENS(s, s[i, j]+1, j)
8           Stampa  “)“
```

Esempio

Consideriamo il seguente caso: ci viene fornita una sequenza di cinque matrici di rispettiva dimensione $4 \times 10, 10 \times 3, 3 \times 12, 12 \times 20, 20 \times 7$ compatibili. Calcolare la parentesizzazione migliore con il costo ottimo.

Determiniamo la sequenza di matrici $p = \langle 4, 10, 3, 12, 20, 7 \rangle$ e creiamo la tabella di memoizzazione che verrà utilizzata in maniera iterativa per essere riempita:

	1	2	3	4	5	
	0					1
		0				2
			0			3
				0		4
					0	5

Figura 7.1: Matrice del costo della parentesizzazione. Gli 0 indicano tutti i casi banali per cui $i = j$, l'indice j rappresenta le righe, mentre i le colonne.

Iniziamo a calcolare la sequenza composta dalla moltiplicazione da due matrici e i casi sono ben 4:

- $m[1, 2]$: rappresenta la moltiplicazione tra $A_1 A_2$ e come ben sappiamo una parentesizzazione ottima è composta da due moltiplicazioni fra due matrici che possono essere prodotti di altri moltiplicazioni. Le due matrici da moltiplicare sono separate da un indice k che in questo caso indica il numero di righe di A_2. Il costo della moltiplicazione è:
$$m[1, 2] = p_0 * p_1 * p_2 = 4 \times 10 \times 3 = 120$$
Il procedimento è identico anche per i seguenti, quindi scriverò solamente il calcolo.;
- $m[2, 3] = p_1 * p_2 * p_3 = 10 \times 3 \times 12 = 360$;

- $m[3,4] = p_2 * p_3 * p_4 = 12 \times 12 \times 20 = 720$;
- $m[4,5] = p_3 * p_4 * p_5 = 12 \times 20 \times 7 = 1680$;

Questi risultati indicano il costo ottimo per moltiplicare due matrici compatibili e li inseriamo nella tabella di memoizzazione:

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Continuiamo con il calcolo per la moltiplicazioni di tre matrici, qui avremo 3 casi:

- $m[1,3] = \min \begin{cases} m[1,2] + m[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \times 3 \times 12 = 264 \\ m[1,1] + m[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \times 10 \times 3 = 480 \end{cases}$
- $m[2,4] = \min \begin{cases} m[2,3] + m[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \times 12 \times 20 = 2760 \\ m[3,4] + m[2,2] + p_1 p_2 p_4 = 0 + 720 + 10 \times 3 \times 20 = 1320 \end{cases}$
- $m[3,5] = \min \begin{cases} m[3,4] + m[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \times 20 \times 7 = 1140 \\ m[4,5] + m[3,3] + p_2 p_3 p_5 = 1680 + 0 + 3 \times 12 \times 7 = 1932 \end{cases}$

Prendiamo i casi minimi di questi calcoli e inseriamoli negli apposti spazi nella tabella

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Continuiamo con la lunghezza delle sequenza da moltiplicare pari a 4:

- $m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4 \times 10 \times 20 = 2120 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 = 265 + 0 + 4 \times 12 \times 20 = 1224 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 = 120 + 720 + 4 \times 3 \times 20 = 1080 \end{cases}$
- $m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \times 3 \times 7 = 1350 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \times 20 \times 7 = 2720 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \times 12 \times 7 = 2880 \end{cases}$

Procediamo come prima e inseriamo i dati nella tabella

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Infine ci manca l'ultimo caso

- $m[1,5]$ min $\begin{cases} m[1,1]+m[2,5]+p_0p_1p_5 = 0+1350+4\times10\times7 = 1630 \\ m[1,2]+m[3,5]+p_0p_2p_5 = 120+1140+4\times3\times7 = 1344 \\ m[1,3]+m[4,5]+p_0p_3p_5 = 264+1680+4\times12\times7 = 2206 \\ m[1,4]+m[5,5]+p_0p_4p_5 = 1080+0+4\times20\times7 = 1544 \end{cases}$

Scegliamo come sempre il risultato migliore ed inseriamo nella tabella:

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

L'esercizio ora si può ritenere concluso, la parte della costruzione della soluzione ottimale la lasciamo svolgere da parte del lettore.

7.2 Elementi della programmazione dinamica

Sottostruttura ottima

La prima fase del processo di risoluzione di un problema di ottimizzazione consiste nel caratterizzare la struttura di una soluzione ottima. Un problema presenta una **sottostruttura ottima** se una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi. Quando un problema presenta una sottostruttura ottima, ciò potrebbe essere un buon indizio dell'applicabilità della programmazione dinamica. Quando costruiamo una soluzione ottima del problema dalle soluzioni ottime dei sottoproblemi, dobbiamo essere sicuri che l'insieme dei sottoproblemi considerati includa quelli utilizzati in una soluzione ottima. Nel paragrafo precedente abbiamo osservato che una parentesizzazione ottima di $A_iA_{i+1}\dots A_j$ che suddivide il prodotto fra A_k e A_{k+1} contiene al suo interno soluzioni ottime dei problemi di parentesizzazione di $A_iA_{i+1}\dots A_k$ e $A_{k+1}A_{k+2}\dots A_j$. Quando si cerca di scoprire la sottostruttura ottima di un problema, si segue sempre uno schema comune:

1. Dimostrate che una soluzione del problema consiste nel fare una scelta (nel caso della parentesizzazione, un indice in corrispondenza del quale suddividere la sequenza delle matrice). Questa lascia uno o più sottoproblemi da risolvere.

2. Per un dato problema, supponete di conoscere la scelta che porta a una soluzione ottima, di come sia stata determinata questa scelta non ci interessa;
3. Fatta la scelta, determinate quali sottoproblemi ne derivano e quale sia il modo migliore per caratterizzare lo spazio di sottoproblemi risultante;
4. Dimostrate che le soluzioni dei sottoproblemi che avete utilizzato all'interno della soluzione ottima del problema devono essere necessariamente ottime, adottando una tecnica "taglia e incolla": prima supponete che ciascuna delle soluzioni dei sottoproblemi non sia ottima e, poi, arrivate a una contraddizione. In particolare, "tagliando" la soluzione non ottima di un sottoproblema e "incollando" quella ottima, dimostrate che potete ottenere una soluzione migliore del problema originale, contraddicendo l'ipotesi di avere già una soluzione ottima.

Per caratterizzare lo spazio dei sottoproblemi, una buona regola consiste nel cercare di mantenere tale spazio quanto più semplice possibile, per poi espanderlo, se necessario. Supponiamo di limitare lo spazio dei sottoproblemi per la moltiplicazione di una sequenza di matrici a quei prodotti matriciali della forma $A_1A_2 \dots A_j$. Una parentesizzazione ottima deve suddividere questo prodotto fra A_k e A_{k+1} per qualche $1 \leq k < j$. A meno che non garantiamo che k sia sempre uguale a $j-1$, troveremo che i sottoproblemi hanno la forma $A_1A_2 \dots A_k$ e $A_{k+1}A_{k+2} \dots A_j$ e che quest' ultimo sottoproblema non ha la forma $A_1A_2 \dots A_j$. Per questo problema, è stato necessario consentire ai sottoproblemi di variare in "entrambe le estremità" ovvero consentire sia i che a j di variare nel sottoproblema $A_iA_{i+1} \dots A_j$. La sottostruttura ottima varia in funzione del tipo di problema in due modi:

1. per il numero di sottoproblemi che sono utilizzati in una soluzione ottima del problema originale;
2. per il numero di scelte che possiamo fare per determinare quale sottoproblema utilizzare in una soluzione ottima.

La moltiplicazione di una sequenza di matrici relativa alla sottosequenza $A_iA_{i+1} \dots A_j$ è un esempio di un caso in cui ci sono due sottoproblemi e $j-i$ scelte. Per una data matrice A_k, abbiamo due sottoproblemi: parentesizzazione di $A_iA_{i+1} \dots A_K$ e parentesizzazione di $A_{k+1}A_{k+2} \dots A_j$ e dobbiamo trovare le soluzioni ottime per entrambi i sottoproblemi. Una volta trovate queste soluzioni, scegliamo l'indice k fra $j-i$ candidati. Il tempo di esecuzione di un algoritmo di programmazione dinamica dipende dal prodotto di due fattori: il numero di sottoproblemi da risolvere complessivamente e il numero di scelte da considerare per ogni sottoproblema. Nella moltiplicazione di una sequenza di matrici c'erano $\Theta(n^2)$ sottoproblemi complessivamente e per ciascuno di essi avevano al massimo $n-1$ scelte, con un tempo di esecuzione pari a $O(n^3)$. Spesso la programmazione dinamica usa la sottostruttura ottima secondo uno schema *bottom-up*(dal basso verso l'alto); ovvero, prima vengono trovate le soluzioni ottime dei sottoproblemi e, dopo aver risolti i sottoproblemi, viene trovata una soluzione ottima del problema. Trovare una soluzione ottima del problema significa scegliere i sottoproblemi da utilizzare per risolvere il problema. Il costo della soluzione del problema, è pari ai costi per risolvere i sottoproblemi più un costo che è direttamente imputabile alla scelta stessa. Nella moltiplicazione di una sequenza di matrici, prima determiniamo le parentesizzazioni ottime delle sottosequenze di $A_iA_{i+1} \dots A_j$; poi scegliamo la matrice A_k, in

corrispondenza della quale suddividere il prodotto. Il costo imputabile alla scelta è il termine $p_{i-1}p_kp_j$.

Ripetizione dei sottoproblemi

Il secondo ingrediente che un problema di ottimizzazione deve avere affinché la programmazione dinamica possa essere applicata è che lo spazio dei sottoproblemi deve essere "piccolo", nel senso che un algoritmo ricorsivo per il problema risolve ripetutamente gli stessi sottoproblemi, anziché generare nuovi sottoproblemi. Quando un algoritmo ricorsivo rivisita più volte lo stesso problema, diciamo che il problema di ottimizzazione ha dei **sottoproblemi ripetuti**[2].

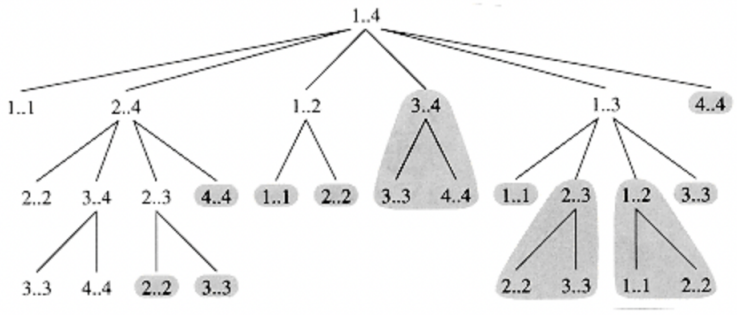


Figura 7.2: L'albero di ricorsione per il calcolo di RECURSIVE-MATRIX-CHAIN(p, 1, 4). Ogni nodo contiene i parametri i e j. I calcoli svolti in un sottoalbero con sfondo verde sono sostituiti da una singola ricerca con la programmazione dinamica

Gli algoritmi di programmazione dinamica tipicamente sfruttano i sottoproblemi ripetuti risolvendo ciascun sottoproblema una sola volta e, poi, memorizzando la soluzione in una tabella da cui può essere recuperata quando serve, impiegando un tempo costante per la ricerca. Per vedere la ripetizione degli elementi, consideriamo la seguente procedura ricorsiva (inefficiente) che calcola $m[i, j]$, il numero minimo di prodotti scalari richiesti per calcolare il prodotto di una sequenza di matrici $A_{i\ldots j} = A_iA_{i+1}\ldots A_j$.

```
1  RECURSIVE–MATRIX–CHAIN(p, i, j)
2      if i == j
3          return 0
4      m[i,j] = ∞
5      for k = 1 to j − 1
6          q = RECURSIVE–MATRIX–CHAIN(p, i, k) +
7          RECURSIVE–MATRIX–CHAIN(p,k+1,j) + p_{i−1}p_k p_j
8          if q < m[i,j]
9              m[i,j] = q
10     return m[i,j]
```

[2]Due sotto problemi dello stesso problema sono indipendenti se non condividono le stesse risorse. Due sottoproblemi sono ripetuti se sono esattamente lo stesso sottoproblema che si ripresenta come sottoproblema di problemi differenti

Quando un albero di ricorsione per la soluzione ricorsiva naturale di un problema contiene più volte lo stesso sottoproblema e il numero totale di sottoproblemi differenti è piccolo, la programmazione dinamica può migliorare l'efficienza in modo anche eclatante.

8 Algoritmi Golosi

Gli algoritmi per i problemi di ottimizzazione, tipicamente, eseguono una sequenza di passi, con una serie di scelte a ogni passo. Per molti problemi di ottimizzazione è uno spreco applicare le tecniche della programmazione dinamica per effettuare le scelte migliori; è preferibile utilizzare algoritmi più semplici ed efficienti. Un **algoritmo** goloso fa sempre la scelta che sembra ottima in un determinato momento, ovvero fa una scelta *localmente ottima*, nella speranza che tale scelta porterà a una soluzione globalmente ottima.

8.1 Problema della selezione di attività

Il problema della programmazione di più attività in competizione che richiedono l'uso esclusivo di una risorsa comune, con l'obbiettivo di selezionare il più grande insieme di attività mutuamente compatibili. Supponiamo di avere un insieme $S = \{a_1, a_2, \ldots, a_n\}$ di n **attività** che devono utilizzare la stessa risorsa. Ogni attività a_i ha un **tempo di inizio** s_i e un **tempo di fine** f_i, con $0 \leq s_1 < f_i < \infty$. Quando viene selezionata, l'attività a_i si svolge durante l'intervallo semiaperto $[s_i, f_i)$. Le attività a_i e a_j sono **compatibili** se gli intervalli $[s_i, f_i)$ e $[s_j, f_j)$ non si sovrappongono. Il **problema della selezione di attività** consiste nel selezionare il sottoinsieme che contiene il maggior numero di attività mutuamente compatibili. Supponiamo che le attività siano ordinate in modo monotonicamente crescente rispetto ai tempi di fine.

Per esempio, consideriamo seguente insieme S di attività:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Per questo esempio, il sottoinsieme $\{a_3, a_9, a_{11}\}$ è formato da attività mutuamente compatibili, ma non è il sottoinsieme di dimensione massima, perché il sottoinsieme $\{a_1, a_4, a_8, a_{11}\}$ è più grande. In effetti a_1, a_4, a_8, a_{11} è un sottoinsieme massimo di attività mutuamente compatibili; un altro sottoinsieme massimo è $\{a_2, a_4, a_9, a_{11}\}$.

La sottostruttura ottima del problema della selezione di attività

Indichiamo con S_{ij} l'insieme delle attività che iniziano dopo la fine dell'attività a_i e finiscono prima dell'inizio dell'attività a_j. Supponiamo di voler trovare un insieme massimo di attività mutuamente compatibili in S_{ij}; supponiamo inoltre che A_{ij} sia un insieme massimo che include una certa attività a_k. Includendo a_k in una soluzione ottima, restano due sottoproblemi: trovare le attività mutuamente compatibili nell'insieme S_{ik} e trovare le attività mutuamente compatibili nell'insieme S_{kj}. Siano $A_{ik} \cap S_{ik}$ e $A_{kj} = A_{ij} \cap S_{kj}$, in modo che A_{ik} contenga le attività in A_{ij} che finiscono prima dell'inizio di a_k e $A_{ij} = A_{ik} \cup a_k \cup A_{kj}$ e, quindi, l'insieme

massimo A_{ij} di attività mutuamente compatibili in S_{ij} è formato da $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ attività. Questo modo di caratterizzare la sottostruttura ottima ci suggerisce di risolvere il problema della selezione di attività tramite la programmazione dinamica. Se indichiamo con $c[i, j]$ la dimensione di una soluzione ottima per l'insieme S_{ij}, otteniamo la ricorrenza

$c[i, j] = c[i, k] + c[k, j] + 1$

Se non sapessimo che una soluzione ottima per l'insieme S_{ij} include l'attività a_k, dovremmo esaminare tutte in S_{ij} per trovare quella da scegliere, pertanto

$$c[i, j] = \begin{cases} 0 \text{ se } S_{ij} = \emptyset \\ max\ c[i, k] + c[k, j] + 1 \text{se } S_{ij} \neq \emptyset \end{cases}$$

Fare la scelta golosa

Che cosa accadrebbe se potessimo scegliere l'attività da aggiungere alla nostra soluzione ottima senza dover risolvere prima tutti i sottoproblemi? In effetti, per il problema della selezione di attività, occorre considerare una sola scelta: la scelta golosa. L'intuito ci dice, quindi, di scegliere l'attività in S che finisce per prima, perché così la risorsa resterebbe disponibile per il maggior numero possibile di attività successive. In altre parole, poiché le attività sono ordinare in modo crescente rispetto ai tempi di fine, la scelta golosa è l'attività a_1. Se facciamo la scelta golosa, ci resta un solo sottoproblema da risolvere: trovare le attività che iniziano dopo la fine di a_1. Dato che tutte le attività sono ordine in modo monotonicamente crescente, solamente le attività che sono dopo di a_1 saranno compatibile con essa perché finiranno un periodo dopo di a_1 e perciò tutte le attività precedenti sono inutili al nostro caso. Sia $S_k = a_i \in S : s_i \geq f_k$ l'insieme delle attività che iniziano dopo la fine dell'attività a_k. Se facciamo la scelta golosa dell'attività a_1, S_1 resta l'unico problema da risolvere e quindi una soluzione ottima del problema originale è formata dall'attività a_1 e da tutte le attività in una soluzione ottima del problema S_1.

Teorema 8.1.1. Consideriamo un sottoproblema non vuoto S_k e sia a_m l'attività in S_k che ha il primo tempo di fine; allora l'attività a_m è inclusa in qualche sottoinsieme massimo di attività mutuamente compatibili di S_k.

Dimostrazione

Supponiamo che A_k sia un sottoinsieme massimo di attività mutuamente compatibili di S_k e sia a_j un attività in A_k con il più piccolo tempo di fine. Se $a_j = a_m$, abbiamo finito, perché abbiamo dimostrato che l'attività a_m è utilizzata in qualche sottoinsieme massimo di attività mutuamente compatibili di S_k. Se $a_j \neq a_m$, sia $A'_k = A_k - a_j \cup a_m$ l'insieme A_k con la sostituzione di a_m con a_j. Le attività in A'_k sono disgiunte, perché lo sono le attività in A_k, a_j è l'attività che finisce per prima in A_k ed $f_m \leq f_i$. Poiché $|A'_k| = |A_k|$, concludiamo che A'_k è un sottoinsieme massimo di attività mutuamente compatibili di S_k che include a_m.

Il modo di procedere sarà di selezionare ripetutamente l'attività che finisce per prima, mantenendo soltanto le attività compatibili con questa attività, e ripetendo il procedimento finché non resteranno altre attività. Poiché possiamo scegliere sempre l'attività con il primo tempo di fine, i tempi di fine delle attività che scegliamo devono essere strettamente crescenti.

Un algoritmo per risolvere il problema della selezione di attività non ha bisogno di operare dal basso verso l'alto come un algoritmo di programmazione dinamica basato su tabelle. Esso può operare dall'alto verso il basso(top-down) scegliendo un'attività da inserire nella selezione ottima e poi risolvendo il sottoproblema di scegliere le attività tra quelle che sono compatibili con le attività già scelte. Tipicamente gli algoritmi golosi seguono questo schema top-down: fare una scelta e risolvere un sottoproblema, diversamente dalla tecnica bottom-up che risolve i sottoproblemi prima di fare una scelta.

Un algoritmo goloso ricorsivo

La procedura RECURSIVE-ACTIVITY-SELECTOR riceve i tempi di inizio e fine delle attività, rappresentati come array s ed f, l'indice k che definisce il sottoproblema S_k da risolvere e la dimensione n del problema originale. Restituisce un insieme massimo di attività mutuamente compatibili in S_k. Supponiamo che le n attività di input siano già ordinate in modo monotonicamente crescente rispetto ai tempi di fine. Se non lo fossero, possiamo ordinarle in tal modo nel tempo O($n \log n$). Per iniziare, aggiungiamo l'attività a_0 con $f_0 = 0$, in modo che il sottoproblema S_0 rappresenti tutto l'insieme delle attività S. La chiamata iniziale, che risolve il problema intero, è RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n).

```
1 RECURSIVE−ACTIVITY−SELECTOR(s, f, k, n)
2     m <− k + 1
3     while m ≤ n and s[m] < f[k]
4         m <− m + 1
5     if m ≤ n
6         return a_m ∪ RECURSIVE−ACTIVITY−SELECTOR(s, f, m, n)
7     else return ∅
```

In una chiamata ricorsiva, il ciclo **while**(righe 2-3) cerca la prima attività in S_k. Il ciclo esamina $a_{k+1}, a_{k+2}, \ldots, a_n$, finché non trova la prima attività a_m che è compatibile con a_k; tale attività ha $s_m \geq f_k$. Se il ciclo termina perché non ha trovato tale attività, la procedura restituisce l'unione di a_m e del sottoinsieme massimo di S_m restituito dalla chiamata ricorsiva di RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). In alternativa, il ciclo può terminare perché $m > n$, nel qual caso abbiamo esaminato tutte le attività in S_k senza trovarne una compatibile con a_k. In questo caso $S_k = \emptyset$, e quindi la procedura restituisce $\emptyset$

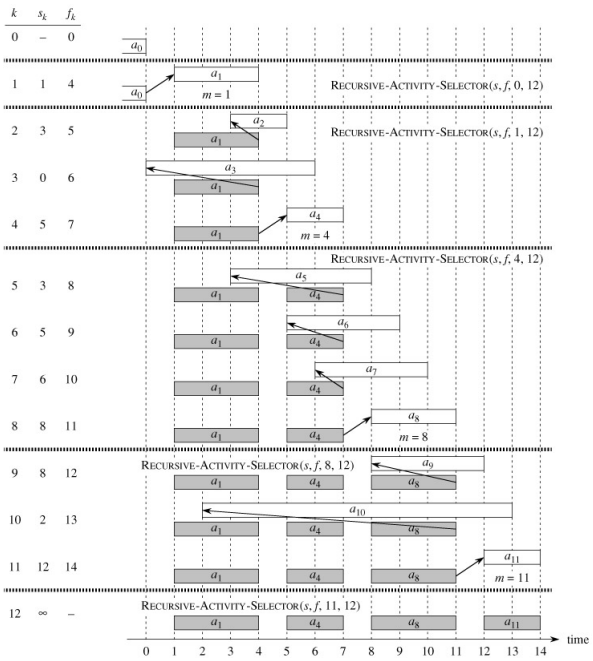


Figura 8.1: Il funzionamento di RECURSIVE-ACTIVITY-SELECTOR con le 11 attività date in precedenza. Le attività considerate in ciascuna chiamata ricorsiva sono rappresentate fra le righe orizzontali. L'attività fittizia a_0 finisce nel tempo 0 e, nella chiamata iniziale, RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11), viene selezionata l'attività a_1. In ogni chiamata ricorsiva le attività già selezionate sono su sfondo grigio; l'attività su sfondo bianco è quella in esame. Se un'attività ha il tempo di inizio che precede il tempo di fine dell'ultima attività che è stata inserita viene scartata, altrimenti viene selezionata. L'ultima chiamata ricorsiva, RECURSIVE-ACTIVITY-SELECTOR(s, f,11,11) restituisce $\emptyset$. L'insieme risultante delle attività selezionate è a_1, a_4, a_8, a_{11}

Un algoritmo goloso iterativo

La procedura GREEDY-ACTIVITY-SELECTOR- è una versione iterativa della procedura RECURSIVE-ACTIVITY-SELECTOR. Si suppone che le attività di input siano ordinate in modo monotonicamente crescente rispetto ai tempi di fine.

```
1 GREEDY–ACTIVITY–SELECTOR(s, f)
2     n = s.length
3     A = {a_1}
4     k = 1
5     for m = 2 to n
6         if s[m] ≥ f[k]
7             A = A ∪ a_m
8             k = m
9     return A
```

La variabile k indicizza l'ultimo inserimento in A, che corrisponde all'attività a_k nella versione ricorsiva. Poiché le attività vengono considerate in base all'ordine monotonicamente crescente dei loro tempi di fine, f_k è sempre il massimo tempo di fine di qualsiasi attività in A; ovvero

$f_k = \max\{f_i : a_i \in A_i\}$

Le righe 2-3 selezionano l'attività a_1, inizializzano A in modo che contenga proprio questa attività e inizializzano k per indicizzare questa attività. Il ciclo **for** trova l'attività che finisce per prima in S_k. Il ciclo esamina a turno ogni attività a_m e inserisce a_m in A, se è compatibile con tutte le attività precedentemente selezionate; questa attività è quella che finisce per prima in S_k. Per vedere se l'attività a_m è compatibile con tutte le attività che si trovano correntemente in A, basta controllare che il suo tempo di inizio s_m non preceda il tempo di fine f_k dell'ultima attività che è stata inserita in A. Se l'attività a_m è compatibile, allora inserisce l'attività a_m in A e si cambia l'indice k ad m. GREEDY-ACTIVITY-SELECTOR programma un insieme di n attività nel tempo $\Theta(n)$, nell'ipotesi che le attività siano inizialmente ordinate rispetto ai loro tempi di fine.

9 Alberi binomiali e heap binomiali

9.1 Alberi binomiali

Un albero binomiale B_k è un albero ordinato definito ricorsivamente. L'albero binomiale B_k consiste di due alberi binomiali B_{k-1} collegati assieme: la radice di uno dei due alberi è il figlio più a sinistra della radice dell'altro. Le proprietà degli alberi binomiali sono 4:

1. i nodi dell'albero sono 2^k.
2. l'altezza dell'albero è k.
3. i nodi a profondità i sono esattamente $\binom{k}{i}$ per $i = 0, 1, \ldots, k$.
4. la radice dell'albero ha grado k, il grado della radice è maggiore del grado di ogni altro nodo; inoltre se $k-1, k-2, \ldots, 0$ è una enumerazione, da sinistra verso destra, dei figli della radice, allora il figlio numero i è la radice di un sottoalbero binomiale B_i.

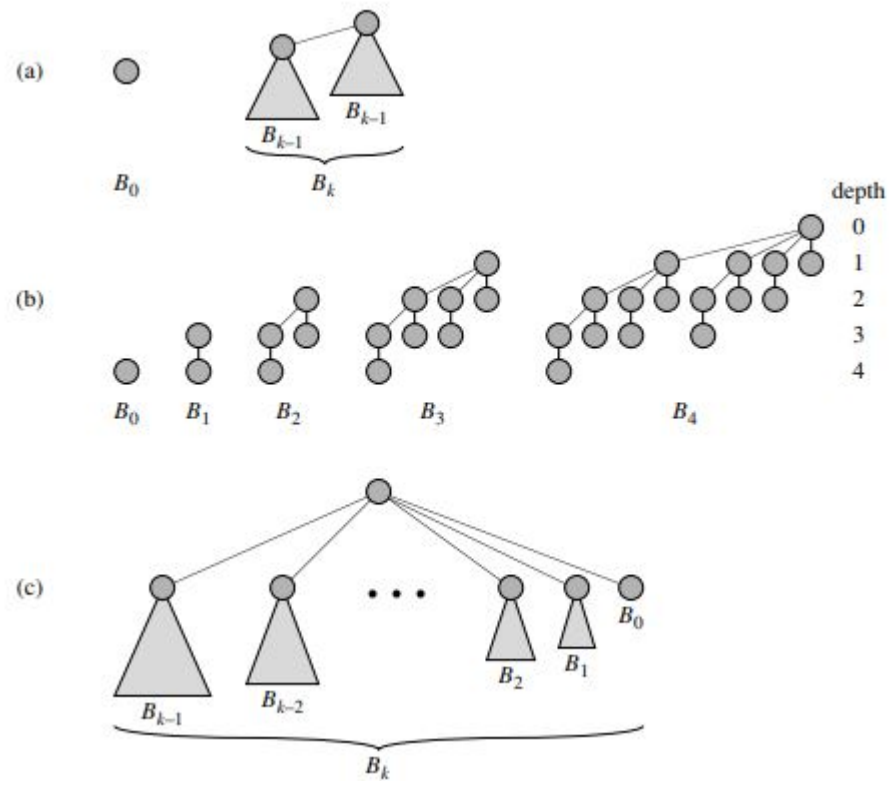


Figura 9.1: (a)La definizione ricorsiva dell'albero binomiale B_k, i triangoli rappresentano i sotto alberi con radice. (b) Alberi binomiali da B_0 a B_4. Viene mostrata la profondità dei nodi dell'albero B_4. (c) Un modo alternativo di considerare l'albero binomiale B_k.

Procediamo a dimostrare tutti questi punti per induzione su k. Per dimostrare i passi induttivi, assumiamo che il lemma sia verificato per l'albero B_{k-1}:

1. L'albero binomiale B_k consiste di due copie B_{k-1}, quindi B_k ha $2^{k-1} + 2^{k-1} = 2^k$ nodi.
2. L'albero B_k è costituito da due copie dell'albero B_{k-1} collegate assieme, pertanto la profondità massima di un nodo dell'albero B_k si ottiene sommando 1 alla profondità massima di B_{k-1}. Per l'ipotesi induttiva, la massima profondità dell'albero B_k risulta essere (k-1) + 1 = k.
3. Sia $D(k, i)$ il numero dei nodi a profondità i dell'albero binomiale B_k. Per costruzione l'albero B_k è costituito da due copie dell'albero B_{k-1} collegate assieme, pertanto un nodo a profondità i in B_{k-1} nell'albero B_k appare una volta a profondità i e l'altra volta a profondità $i + 1$. In altri termini, abbiamo che il numero dei nodi a profondità i nell'albero B_k è dato dalla somma del numero dei nodi a profondità i di B_{k-1} con il numero dei nodi a profondità $i - 1$ in B_{k-1}.Quindi

   $D(k, i) = D(k-1, i) + D(k-1, i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$

4. Il solo nodo con un grado più alto in B_k rispetto a quello che aveva in B_{k-1} è il nodo radice che ha un figlio in più di quanti ne aveva in B_{k-1}. Dato che la radice di B_{k-1} ha grado $k-1$, segue che la radice di B_k ha grado k. Dall'ipotesi induttiva, e come è mostrato dall'esempio di figura 34(c), i figli della radice di B_{k-1} sono, da sinistra a destra, le radici di $B_{k-2}, B_{k-3}, \ldots, B_0$. Pertanto, quando si collegano assieme i due albero B_{k-1} abbiamo che i figli della radice risultante diventano $B_{k-1}, B_{k-2}, \ldots, B_0$.

Il massimo grado di ogni nodo in un albero binomiale con n nodi è $\log n$.

9.2 Heap binomiali

Uno **heap binomiale** H è un'insieme di alberi binomiali che soddisfa le seguenti proprietà dette proprietà dello heap binomiale:

1. Ogni albero binomiale in H ha la proprietà di **ordinamento parziale dello heap**: la chiave di un nodo è maggiore o uguale della chiave del nodo padre.
2. Non esistono due alberi binomiali in H le cui radici hanno lo stesso grado.

La prima proprietà garantisce che la chiave della radice di uno heap binomiale sia la più piccola chiave dello heap. La seconda proprietà implica che uno heap binomiale con n nodi consiste al massimo di $\lfloor \log n \rfloor + 1$ alberi binomiali. Infatti si osservi come la rappresentazione binaria del numero n contenga un numero di bit uguale a $\lfloor \log n \rfloor + 1$; supponiamo che questi bit siano $\langle b_{\lfloor \log \rfloor}, b_{\lfloor \log n \rfloor - 1}, \ldots, b_0 \rangle$ e quindi $n = \sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i$. Per la proprietà 1 degli alberi binari abbiamo che l'albero binomiale B_1 è presente in H se e solo se $b_i = 1$. Pertanto, lo heap binomiale H contiene al massimo $\lfloor \log n \rfloor + 1$ alberi binomiali.

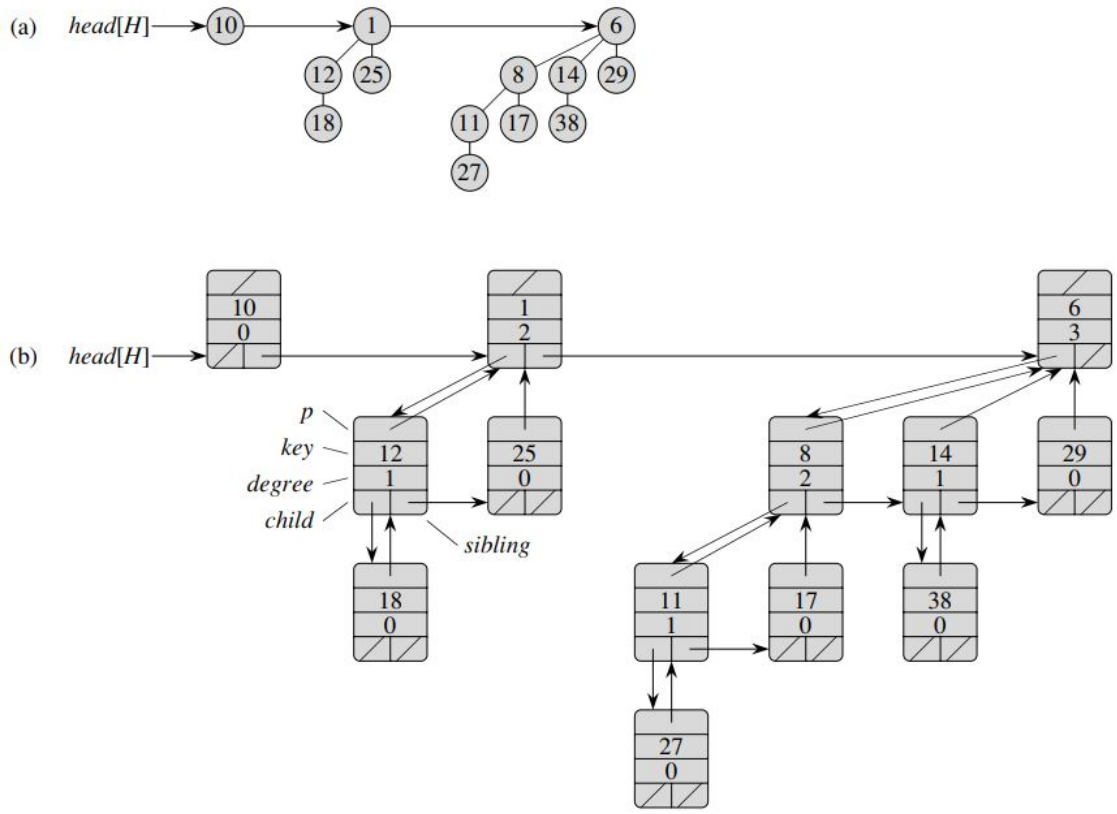


Figura 9.2: Uno heap binomiale H con n = 13 nodi. (a) Lo heap è costituito dall'insieme di alberi binomiali B_0, B_2 e B_3, rispettivamente con 1,4 e 8 nodi, per un totale di 13 nodi. Ogni albero binomiale rispetta l'ordinamento dello heap, quindi la chiave di un nodo qualunque non è più piccola della chiave del nodo padre. La figura mostra anche la lista delle radici, che contiene tutte le radici ordinate per grado crescente. (b) Una rappresentazione più dettagliata dello heap binomiale H. Ogni albero binomiale è memorizzato usando la rappresentazione figlio-sinistro, fratello-destro; inoltre in ogni modo viene memorizzato il grado del nodo

Ogni albero binomiale è memorizzato con la rappresentazione figlio-sinistro, fratello-destro. Ogni nodo è caratterizzato da un insieme di informazioni ed anche contiene un certo numero di puntatori: p[x] al nodo padre, child[x] al suo figlio più a sinistra e il puntatore sibling[x] al primo fratello di destra del nodo x. Infine ogni nodo x contiene un ulteriore campo per indicare il numero dei sui figli, cioè il grado del nodo.

9.2.1 Operazioni su heap binomiali

Creazione di un nuovo heap binomiale

La procedura MAKE-BINOMIAL-HEAP crea uno heap binomiale vuoto: la procedura restituisce un oggetto H con $head[H] = NIL$, dopo avergli assegnato lo spazio di memoria opportuno. Il tempo di esecuzione è$\Theta(1)$.

Ricerca della chiave minima

La procedura BINOMIAL-HEAP-MINIMUM ha come parametro di ingresso uno heap binomiale H con n nodi, e restituisce come risultato il puntatore al nodo con la chiave minima.

```
1  BINOMIAL–HEAP–MINIMUM(H)
2      y ← NIL
3      x ← head[H]
4      min ← ∞
5      while x ≠ NIL
6          do if key[x] < min
7              then min ← key[x]
8                   y ← x
9              x ← sibling[x]
10     return y
```

Uno heap binomiale soddisfa la proprietà dell'ordinamento heap e, pertanto, la chiave minima deve necessariamente essere associata a un nodo radice. La procedura BINOMIAL-HEAP-MINIMUM esamina tutte le radici, il cui numero è al massimo $\lfloor \log n \rfloor + 1$ e memorizza nella variabile *min* il minimo corrente e nella variabile y il puntatore al minimo corrente. Poiché la procedura esamina al massimo $\lfloor \log n \rfloor + 1$ radici, il suo tempo di esecuzione è $O(\log n)$ **Unione di due heap binomiali**

La procedura UNION collega assieme tutti gli alberi binomiali le cui radici hanno lo stesso grado. La seguente procedura collega assieme l'albero B_{k-1} di radice y con l'albero B_{k-1} con radice z: ovvero z diviene il padre di y. Pertanto z diviene la radice di un albero B_k.

```
1  BINOMIAL–LINK(y,z)
2      p[y] ← z
3      sibling[y] ← child[z]
4      child[z] ← y
5      grado[z] ← grado[z] + 1
```

La procedura BINOMIAL-LINK inserisce il nodo y in testa alla lista dei figli del nodo z in tempo O(1). L'unione è divisa in due fasi: la prima fase fonde assieme in un'unica lista, ordinata in modo crescente rispetto al grado dei nodi, le liste delle radici degli heap binomiali. La seconda fase, invece prende nota di questi puntatori alla lista delle radici dell'heap binomiale:

- x è il puntatore alla radice in esame;
- prev[x] è il puntatore alla radice che precede x nella lista delle radici;
- next[x] è il puntatore alla radice che segue x nella lista delle radici;
- grado[x] è il grado della radice x;

```
1 UNION(H1,H2)
2      H ← MAKE–BINOMIAL–HEAP()
3      head[H] ← UNISCI–DUE–HEAP(H1,H2)
```

```
4   libera gli oggetti H1 e H2 ma non le lista cui si riferiscono
5   if head[H] = NIL
6       then return H
7   prev[x] ← NIL
8   x ← head[H]
9   next[x] ← sibling[x]
10  while next ≠ NIL
11      do if grado[x] ≠ grado[next[x]] oppure
12                  (sibling[next[x]] ≠ NIL e
13                  e grado[sibling[next[x]]] = grado[x])
14              then prev[x] ← x                          ▷ Casi 1 e 2
15                  x ← next[x]                           ▷ Casi 1 e 2
16              else if key[x] ≤ key[next[x]]
17                  then sibling[x] ← sibling[next[x]]    ▷ Caso 3
18                      BINOMIAL–LINK(next[x],x)          ▷ Caso 3
19                  else if prev[x] = NIL                 ▷ Caso 4
20                      then head[H] ←next[x]             ▷ Caso 4
21                      else sibling[prev[x]]←next[x]     ▷ Caso 4
22                  BINOMIAL–LINK(x,next[x])              ▷ Caso 4
23                  x ← next[x]
24      next[x] ← sibling[x]
25  return H
```

Esistono quattro casi per questo algoritmo:

- Il caso 1, si presenta quando grado[x]≠ grado[next[x]], ovvero quando x è la radice di un albero B_k, e next[x] è la radice di un albero B_l per qualche $l > k$. I nodi x e $next[x]$ non vengono collegati ed i puntatori sono fatti avanzare di una posizione nella lista H.
- Il caso 2, si presenta quando x è la prima di tre radici con lo stesso grado, cioè quando

  $grado[x] = grado[next[x]] = grado[sibling[next - x]]$

  Questa situazione viene affrontata in modo analogo al caso 1: i puntatori sono fatti avanzare di una posizione nella lista H. L'istruzione della linea 10 fa i controlli necessari sia per il caso 1 che per il caso 2;
- I casi 3 e 4 si presentano quando x è la prima di due radici con lo stesso grado:

  $grado[x] = grado[next[x]] \neq grado[sibling[next[x]]]$

  Questi due casi si possono presentare al passo di interazione che segue un caso qualsiasi. I due casi sono sostanzialmente simmetrici e dipendono dal controllo di chi tra x e $next[x]$ ha la chiave più piccola: questo controllo stabilisce quale sia il nodo che diventerà il nodo radice dopo che i ude nodi sono stati collegati assieme. Nel caso 3, $key[x] \leq key[next[x]]$ e quindi $next[x]$ è collegato a x.
- Nel caso 4, $next[x]$ ha la chiave più piccola, e quindi x è collegato a $next[x]$.

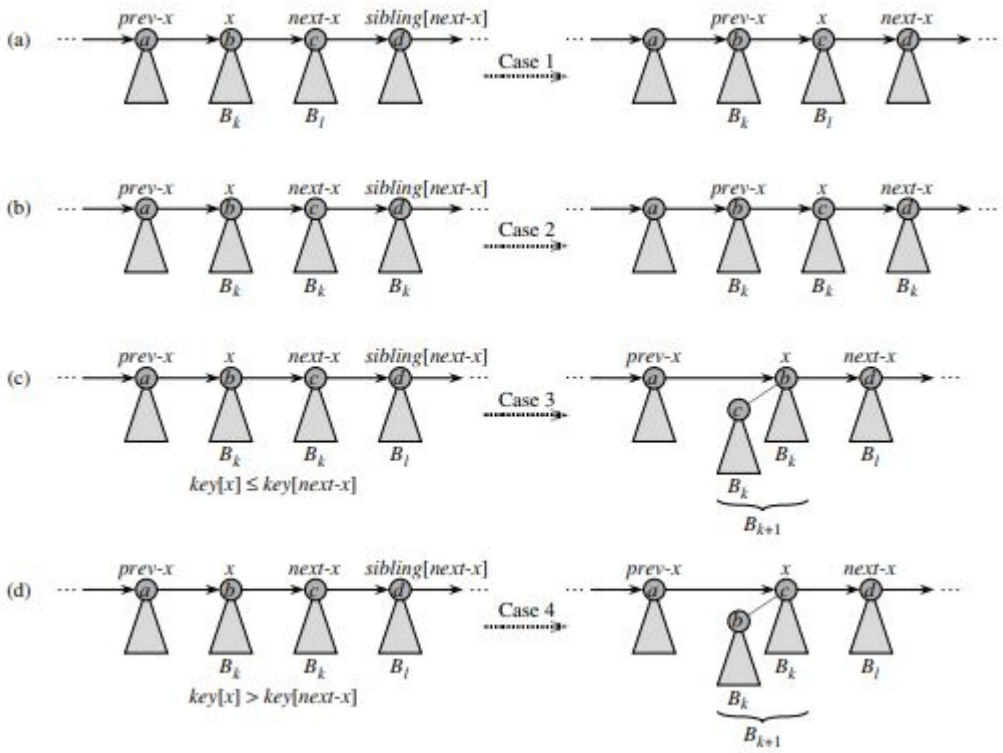


Figura 9.3: I quattro casi che si presentano durante l'esecuzione della procedura UNION. Le etichette a,b,c e d sono utilizzate per identificare le radici coinvolte nelle operazioni; queste etichette non indicano nè il grado nè le chiavi associate ai nodi. In tutti casi esaminati x è radice di un albero B_k e $l > k$.(a) Caso 1: $grado[x] \neq grado[next[x]]$, i puntatori vengono avanzati di una posizione nella lista delle radici. (b) Caso 2: $grado[x] = grado[next[x]] = grado[sibling[next[x]]]$. Anche in questo caso i puntatori vengono avanzati di una posizione nella lista delle radici e alla successiva iterazione vengono eseguite le istruzioni del caso 3 o del caso 4. (c) Caso 3: $grado[x] = grado[next[x]] \neq grado[sibling[next[x]]]\&\&key[x] \leq key[next[x]]$. Si eleminiza $next[x]$ dalla lista delle radici e lo si collega a x, creando in questo modo un albero B_{k+1}. (d) Caso 4: $grado[x] = grado[next[x]] \neq grado[sibling[next[x]]]\&\&key[x] > key[next[x]]$. Si elimina x dalla lista delle radici e lo si collega a $next[x]$: di nuovo questa operazione crea un albero B_{k+1}

Il tempo di esecuzione della procedura UNION è $O(\log n)$ dove n è il numero complessivo di nodi degli heap binomiali.

Inserimento di un nodo La procedura crea in tempo O(1) uno heap binomiale composto da un solo nodo, successivamente unisce questo heap con lo heap complessivo in tempo $O(\log n)$.

```
1 BINOMIAL–HEAP–INSERT(H, x)
2     H' ← MAKE-BINOMIAL–HEAP()
3     p[x] ← NIL
```

```
4       child[x] ← NIL
5       sibling[x] ← NIL
6       grado[x] ← 0
7       head[H'] ← x
8       H ← UNION(H,H')
```

Estrazione del nodo con la chiave minima

La procedura è composta da cinque punti:

1. Trova la radice x con la chiave minima nella lista delle radici di H ed elimina x dalla lista delle radici di H;
2. Si crea uno heap binario vuoto;
3. si inverte l'ordine della lista dei figli di x e assegna come radice allo heap generato precedentemente il puntatore al primo elemento della lista risultante;
4. Si unisce l'heap creato al complessivo;
5. ritorna x

10 Strutture dati per insiemi disgiunti

Alcune applicazioni richiedono di raggruppare n elementi distinti in una collezione di insiemi disgiunti[1]. Queste applicazioni spesso richiedono l'esecuzione di due particolari operazioni: trovare l'unico insieme che contiene un determinato elemento e unire due insiemi. Questo capitolo esamina i metodi per mantenere una struttura dati che supporta queste operazioni.

10.1 Operazioni con gli insiemi disgiunti

Una **struttura dati per insiemi disgiunti** mantiene una collezione $\epsilon = \{S_1, S_2, \ldots, S_k\}$ di insiemi dinamici disgiunti. Ciascun insieme è identificato da un **rappresentante**, che è un elemento dell'insieme. Non è importante specificare quale elemento debba essere rappresentante, uno qualsiasi va più che bene. Come in altre implementazioni degli insiemi dinamici finora analizzate, ogni elemento di un insieme è rappresentato da un oggetto. Indicando con x un oggetto, vogliamo supportare le seguenti operazioni:

- **MAKE-SET(x)**. Crea un nuovo insieme il cui unico elemento (e rappresentante) è x. Poiché gli insiemi sono disgiunti, x non può trovarsi in qualche altro insieme.
- **UNION(x,y)**. Unisce gli insiemi dinamici che contengono x e y, per esempio S_x e S_y, in un nuovo insieme che è l'unione di questi due insiemi. SI suppone che i due insiemi siano disgiunti prima dell'operazione. Il rappresentante dell'insieme risultante è un elemento qualsiasi di $S_x \cup S_y$, sebbene molte implementazioni di UNION scelgano specificamente il rappresentante di S_x o quello di S_y, come nuovo rappresentante.
- **FIND-SET(x)**. Restituisce un puntatore al rappresentante dell'insieme che contiene x.

In questo capitolo analizzeremo i tempi di esecuzione delle strutture dati per gli insiemi disgiunti in funzione di due parametri:

1. **n**. Il numero di operazioni MAKE-SET.
2. m. Il numero totale di operazioni MAKE-SET, UNION e FIND-SET.

Poiché gli insiemi sono disgiunti, ciascuna operazioni UNION riduce di un'unità il numero degli insiemi. Dopo $n - 1$ operazioni UNION, quindi, resta un solo insieme. Ne consegue che il numero di operazioni UNION è al più $n - 1$. Poiché le operazioni MAKE-SET sono incluse nel numero totale di operazioni m, allora $m \geq n$. Si suppone che le n operazioni MAKE-SET siano le prime n operazioni eseguite.

[1]Nella teoria degli insiemi la disgiunzione è la relazione che sussiste fra due insiemi che non hanno alcun elemento in comune. In altre parole, due insiemi A e B sono disgiunti se la loro intersezione è l'insieme vuoto cioè $A \cap B = \varnothing$

10.2 Rappresentazione di insiemi disgiunti tramite liste concatenate

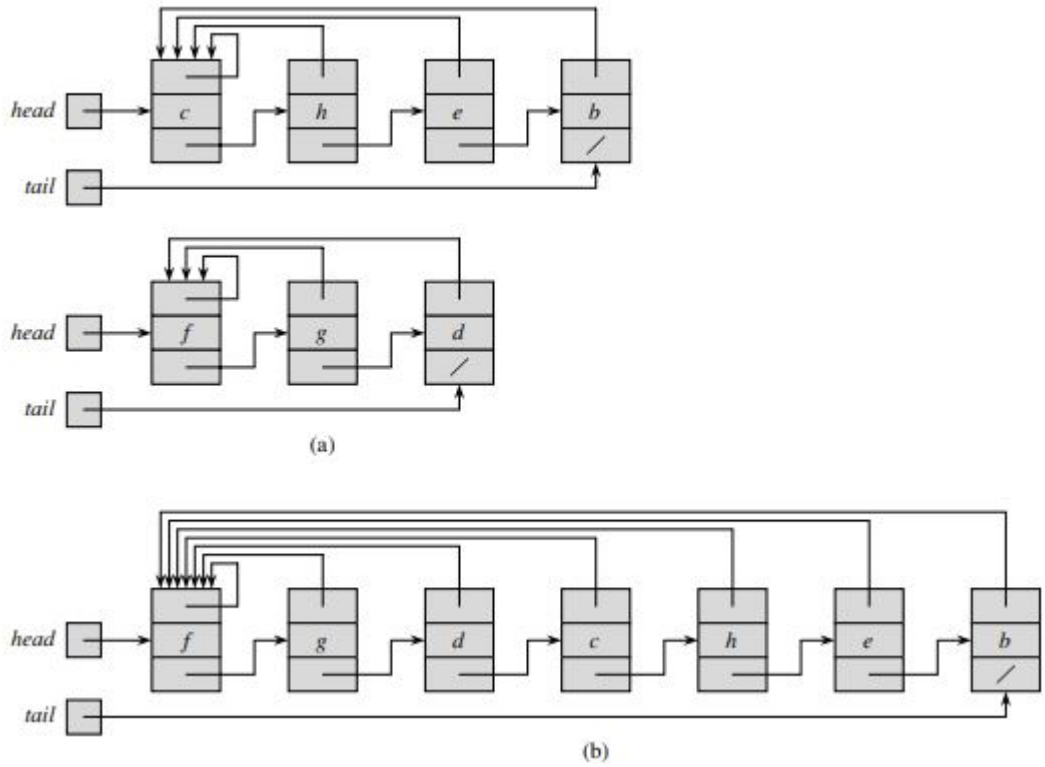


Figura 10.1: (a) Rappresentazione con liste concatenate di due insiemi. L'insieme S_1 contiene gli elementi d, f e g, con f come rappresentante. L'insieme S_2 contiene gli elementi b, c, e e h, con c come rappresentante. Ogni oggetto nella lista contiene un elemento dell'insieme, un puntatore al successivo oggetto nella lista e un puntatore che ritorna all'oggetto dell'insieme. Ogni oggetto dell'insieme ha i puntatore *head* e *tail* rispettivamente al primo e all'ultimo oggetto. (b) Il risultato dell'operazione UNION(g,e), che aggiunge la lista concatenata che contiene e alla lista concatenata che contiene g. Il rappresentante dell'insieme risultante è f. L'oggetto dell'insieme per la lista S_2 di e viene eliminato.

La figura 10.1 mostra un semplice modo di implementare una struttura dati per gli insiemi disgiunti: ciascun insieme è rappresentato dalla sua lista concatenata. L'oggetto di ciascun insieme ha gli attributi *head*, che punta al primo oggetto della lista, e *tail* che punta all'ultimo oggetto. Ogni oggetto nella lista contiene un elemento dell'insieme, un puntatore al successivo oggetto della lista e un puntatore che ritorna all'oggetto dell'insieme. All'interno di ciascuna lista concatenata, gli oggetti possono apparire in qualsiasi ordine. Il rappresentante è l'elemento dell'insieme nel primo oggetti della lista. Con la rappresentazione tramite liste concatenate, entrambe le operazioni MAKE-SET e FIND-SET sono semplici da realizzare e richiedono un tempo O(1). Per realizzare l'operazione MAKE-SET(x), creiamo una nuova

lista concatenata il cui onico oggetto è x. Per l'operazione FIND-SET(x), basta seguire il puntatore da x per arrivare all'oggetto del suo insieme e poi ritornare all'elemento nell'oggetto cui punta *head*.

Semplice implementazione dell'operazione di unione

Come illustra la figura 10.1(b), noi eseguiamo UNION(x,y) aggiungendo la lista di y alla fine della lista di x. Il rappresentante della lista di x diventa il rappresentante dell'insieme risultante. Purtroppo, dobbiamo aggiornare il puntatore all'oggetto dell'insieme per ogni oggetto che originariamente si trovava nella lista di y; questo richiede un tempo lineare nella lunghezza della lista di y.Nella figura 10.1, l'operazione UNION(g, e) fa sì che i puntatori siano aggiornati negli oggetti di b, c, e e h. In effetti non è difficile trovare una sequenza di m operazioni su n oggetti che richiede un tempo $\Theta(n^2)$. Supponiamo di avere gli oggetti $x_1, x_2, \ldots, x_n$. Eseguiamo la sequenza di n operazioni MAKE-SET seguite dalle $n-1$ operazioni UNION, quindi $m = 2n-1$. Impieghiamo un tempo $\Theta(n)$ per eseguire le n operazioni MAKE-SET. Poiché l'i-esima operazione UNION aggiorna i oggetti, il numero totale di oggetti aggiornati da tutte le $n-1$ operazioni UNION è

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

Il numero totale di operazioni è $2n-1$, pertanto ciascuna operazione richiede un tempo $\Theta(n)$. Dunque il tempo ammortizzato di un'operazione è $\Theta(n)$.

Euristica dell'unione pesata

Nel caso peggiore, la precedente implementazione della procedura UNION richiede un tempo $\Theta(n)$ per chiamata, perché potremmo appendere una lista più lunga a una più corta; dobbiamo aggiornare il puntatore all'oggetto dell'insieme per ogni elemento della lista più lunga. Supponiamo che ogni lista includa anche la lunghezza della lista; supponiamo inoltre di appendere sempre la lista più piccola a quella più lunga, risolvendo in modo arbitrario i casi di liste aventi la stessa lunghezza. Con questo passaggio, una singola operazione UNION può ancora impiegare un tempo $\Omega(n)$, se entrambi gli insiemi hanno $\Omega(n)$ elementi. Tuttavia una sequenza di m operazioni MAKE-SET, UNION e FIND-SET, n delle quali sono operazioni MAKE-SET, impiega un tempo $O(m + n \log n)$.

Dimostrazione

Poiché ogni operazione UNION unisce due insiemi disgiunti, vengono eseguite al più $n-1$ operazioni UNION. Iniziamo calcolando, per ogni oggetto, un limite superiore al numero di volte che viene aggiornato il puntatore di un oggetto all'oggetto del suo insieme. Consideriamo un particolare oggetto x; sappiamo che ogni volta che il puntatore di x viene aggiornato, x deve trovarsi nell'insieme più piccolo. Quindi la prima volta che il puntatore di di$x$ viene aggiornato l'insieme risultante deve avere almeno 2 elementi. Analogamente, la seconda volta che il puntatore di x viene aggiornato, l'insieme risultante deve avere almeno 44 elementi. Procedendo così, osserviamo che per ogni $k \leq n$, dopo che il puntatore di x viene aggiornato $\lceil \log k \rceil$ volte, l'insieme risultante deve avere almeno k elementi. Dato che l'insieme più grande ha al più n elementi, il puntatore in ciascun oggetto è stato aggiornato al più $\lceil \log n \rceil$

volte in tutte le operazioni UNION. Quindi il tempo totale speso per aggiornare i puntatori degli oggetti durante le operazioni UNION è $O(n \log n)$. Ciascuna operazione MAke-SET e FIND-SET impiega un tempo $O(1)$ e ci sono $O(m)$ di queste operazioni; il tempo totale per l'intera sequenza è quindi $O(m + n \log n)$.

# Cambi	Dimensione minima della lista
1	2
2	4
3	8
...	...
i	2^i

Tabella 10.1: Tabella che rappresenta la corrispondenza tra il numero di cambiamenti riguardanti i puntatori degli oggetti e la dimensione della lista risultante. Se si suppone che il numero totale di MAKE-SET è n, il numero di cambi è dato da $2^i \leq n$ da cui $i \leq \log n$. Cioè il numero massimo di cambi che è possibile fare per n oggetti è $\log n$

10.3 Foreste di insiemi disgiunti

In una **foresta di insiemi disgiunti**, ogni elemento punta soltanto a suo padre. Il nodo radice di ogni albero contiene il rappresentante ed è padre di sé stesso. Come vedremo più avanti, sebbene gli algoritmi semplici che usano questa rappresentazione non siano più veloci di quelli che usano la rappresentazione con le liste concatenate. Introducendo l'unione per rango e la compressione del cammino possiamo ottenere una struttura dati per insiemi disgiunti asintoticamente ottimale.

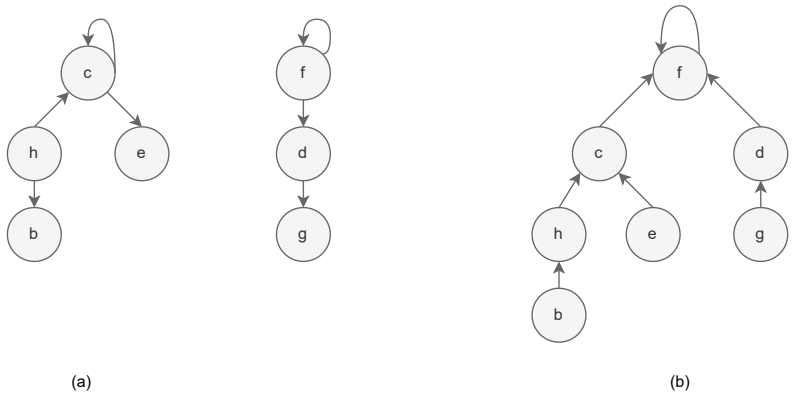


Figura 10.2: Una foresta di insiemi disgiunti. (a) Due alberi che rappresentano i due insiemi della figura 10.1. L'albero a sinistra rappresenta l'insieme $\{b, c, e, h\}$, con c come rappresentante dell'insieme; l'albero a destra rappresenta l'insieme $\{d, f, g\}$., con f come rappresentante dell'insieme. (b) Il risultato dell'operazione UNION(e, g)

Realizziamo le tre operazioni degli insiemi disgiunti nel seguente modo. Un'operazione

MAKE-SET crea semplicemente un albero con un solo nodo. Un'operazione FIND-SET segue i puntatori ai padri finché non trova la radice dell'albero. I nodi visitati in questo cammino semplice verso la radice costituiscono il **cammino di ricerca**

Euristiche per migliorare il tempo di esecuzione

La prima euristica, **unione per rango** è simile all'euristica dell'unione pesata che abbiamo utilizzato con la rappresentazione delle liste concatenate. L'idea consiste nel fare in modo che la radice dell'albero con meno nodi punti alla alla radice dell'albero con più nodi. Per ogni nodo manteniamo un **rango** che è un limite superiore per l'altezza del nodo. Nell'unione per rango, la radice con il rango più piccolo viene fatta puntare alla radice con il grado più grande durante un'operazione UNION. La seconda euristica viene utilizzata durante le operazioni FIND-SET per fare in modo che ciascun nodo nel cammino di ricerca punti direttamente alla radice. La compressione del cammino non cambia i ranghi.

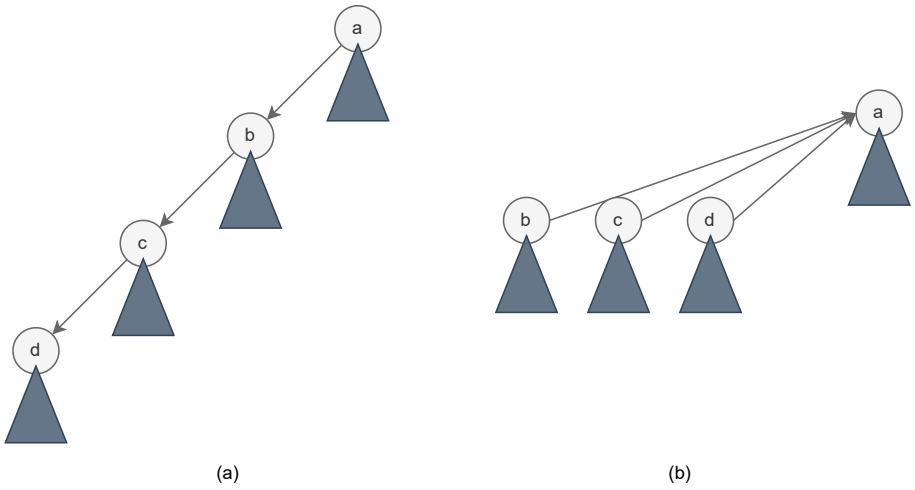


Figura 10.3: La compressione del cammino durante l'operazione FIND-SET. Le frecce e i cappi delle radici sono stati omessi. (a) Un albero che rappresenta un insieme prima di eseguire FIND-SET(a). I triangoli rappresentano i sottoalberi cui radici sono i nodi indicati nella figura. Ogni nodo ha un puntatore a suo padre. (b) Lo stesso insieme dopo l'esecuzione di FIND-SET(a). Ogni nodo lungo il cammino di ricerca adesso punta direttamente alla radice.

```
1 FIND–SET(x)
2     if x ≠ x.p
3         x.p = FIND–SET(x.p)
4     return x.p
```

La procedura FIND-SET è un metodo a doppio passaggio: durante il primo passaggio, risale il cammino di ricerca per trovare la radice; durante il secondo passaggio, discende il cammino di ricerca per aggiornare i nodi in modo che puntino direttamente alla radice.

Sia l'unione per rango sia la compressione del cammino se usate separatamente migliorano il tempo di esecuzione delle operazioni con le foreste di insiemi disgiunti; il miglioramento è

ancora più grande se le due euristiche sono utilizzate insieme. Con la sola unione per rango si ottiene un tempo di esecuzione pari a $O(m \log n)$. Quando utilizziamo sia l'unione per rango sia la compressione del cammino, il tempo di esecuzione nel caso peggiore è $O(m\alpha(n))$, dove $\alpha(n)$ è l'inversa della funzione di Ackermann. In qualsiasi applicazione di una struttura dati per insiemi disgiunti, $\alpha(n) \leq 4$.

11 Hashing

Una tavola hash è una struttura dati efficace per implementare i dizionari. Sebbene la ricerca di un elemento in una tavola hash richieda, nel tempo peggiore, lo stesso tempo $\Omega(n)$ richiesto per ricercare un elemento nella lista concatenata, l'hashing si comporta molto bene nella pratica. Sotto ipotesi ragionevoli, il tempo medio per cercare un elemento in una tavola hash è $O(1)$.

11.1 Tavole a indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo. Supponiamo che un'applicazione abbia bisogno di un insieme dinamico in cui ogni elemento ha una chiave estratta dall'universo $U = \{0, 1, \ldots, m-1\}$, dove m non è troppo grande. Supponiamo inoltre che due elementi non possono avere la stessa chiave. Per rappresentare l'insieme dinamico, utilizziamo un array o **tavola a indirizzamento diretto**, che indicheremo con $T[0 \ldots m-1]$, dove ogni posizione o **cella** corrisponde a una chiave nell'universo U.

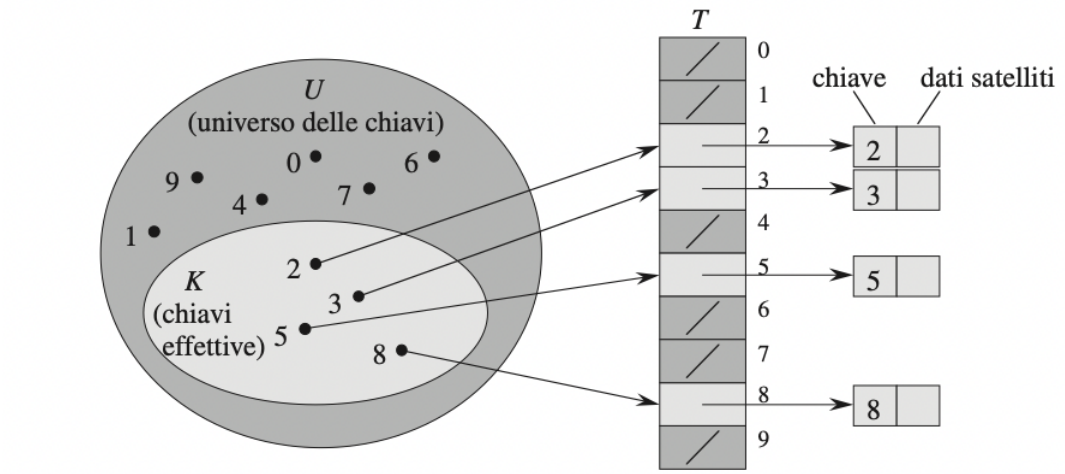


Figura 11.1: Implementazione di un insieme dinamico tramite la tavola a indirizzamento diretto T. Ogni chiave nell'universo $U = \{0, 1, \ldots, 9\}$ corrisponde a un indice della tavola. L'insieme $K = \{2, 3, 5, 8\}$ delle chiavi effettive determina le celle nella tavola che contengono i puntatori agli elementi. Le altre celle contengono la costante NIL

Ciascuna delle operazioni SEARCH, INSERT, DELETE richiedono tempo $O(1)$.

11.2 Tavole Hash

La difficoltà dell'indirizzamento diretto è se l'universo delle chiavi U è troppo grande, memorizzare una tavola T di dimensione $|U|$ può essere impraticabile o impossibile. Inoltre, può essere che l'insieme K delle chiavi effettivamente utilizzate sia così piccolo rispetto a U che la maggior parte dello spazio allocato per la tavola T sarebbe sprecato. Quando l'insieme K delle chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le chiavi possibili, una tavola hash richiede molto meno spazio di una tavola a indirizzamento diretto. Lo spazio richiesto può essere ridotto a $\Theta(|k|)$, mantenendo il vantaggio di ricercare un elemento nella tavola hash nel tempo $O(1)$. Il limite vale per il *tempo medio*, mentre nell'indirizzamento diretto vale per il *tempo nel caso peggiore*. Con l'indirizzamento diretto, un elemento con chiave k è memorizzato nella cella k. Con l'hashing, questo elemento è memorizzato nella cella $h(k)$; cioè, utilizziamo una **funzione hash** h per calcolare la cella dalla chiava k. h associa l'universo U delle chiavi alle celle di una **tavola hash** $\mathrm{T}[0, \ldots m-1]$:

$h : U \mapsto 0, 1, \ldots, m-1$

dove la dimensione m della tavola hash è generalmente molto più piccola della cardinalità di U.

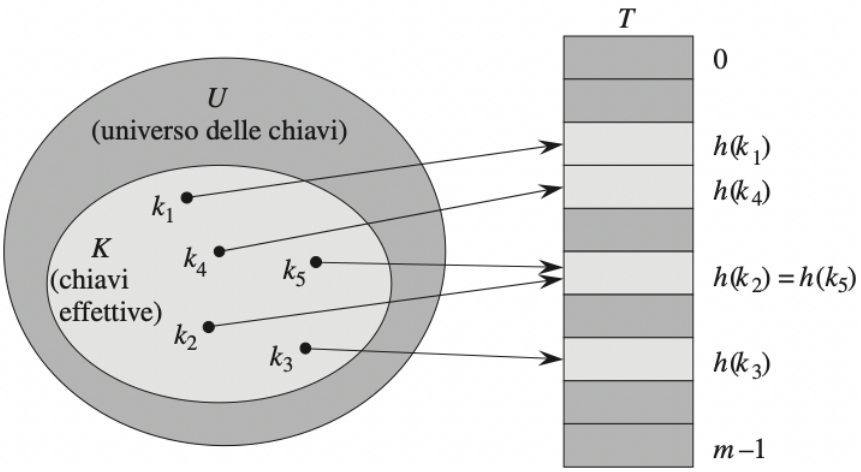


Figura 11.2: L'uso di una funzione hash h per associare le chiavi alle celle della tavola hash Le chiavi k_2 e k_2 sono associate alla stessa cella, quindi sono in collisione.

Come si può intravedere dalla figura, due chiavi possono essere mappate nella stessa cella. Questo evento si chiama **collisione**. La soluzione ideale sarebbe di evitare qualsiasi tipo di collisione e si potrebbe tentare di farlo scegliendo un'opportuna funzione hash h in modo che essa sembri casuale evitando le collisioni o portandole al numero più piccolo. L'altro metodo è utilizzare liste concatenate.

11.2.1 Concatenamento

Nel **concatenamento** poniamo tutti gli elementi che sono associati alla stessa cella in una lista concatenata. La cella j contiene un puntatore alla testa della lista di tutti gli elementi memorizzati che vengono mappati in j; se non ce ne sono, la cella j contiene la costante NIL.

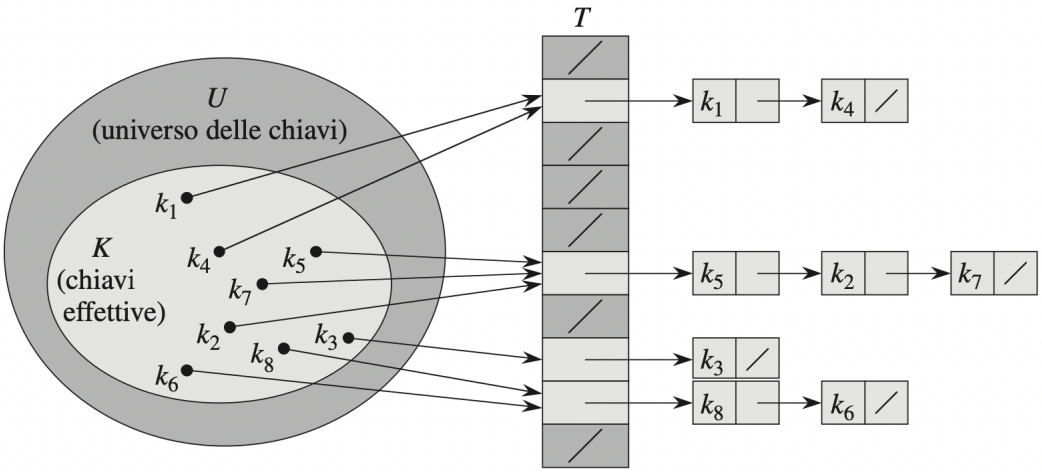


Figura 11.3: Risoluzione delle collisioni mediante concatenamento. Ogni cella $T[j]$ della tavola hash contiene una lista concatenata di tutte le chiavi il cui valore hash è j. Per esempio, $h(k_1) = h(k_4)$.

INSERT e DELETE hanno complessità $O(1)$.

Analisi dell'hasing con concatenamento

Data una tavola hash T con m celle dove sono memorizzati n elementi, definiamo **fattore di carico** α della tavola T il rapporto n/m, ossia il numero medio di elementi memorizzati in una lista. Studiamo il comportamento in base al caso in cui ci troviamo:

- **Peggiore**. Tutte le n chiavi sono associate alla stessa cella, creando una lista di lunghezza n. Il tempo di esecuzione della ricerca è quindi $\Theta(n)$ più il tempo per calcolare la funzione hash.
- **Medio**. Dipende dal modo in cui la funzione hash h distribuisce mediamente l'insieme delle chiavi da memorizzare tra le m celle. Una ricerca richiede un tempo $\Theta(1 + \alpha)$, nell'ipotesi che la distribuzione dei valori è uniforme nelle celle. Ricordiamo che quel 1 è perché supposizione di hashing da costo costante.
- **Migliore**. Ogni cella ha un solo valore, quindi il lavoro richiesto è costante $O(1)$.

11.3 Funzione hash

Una buona funzione hash soddisfa (approssimativamente) l'ipotesi dell'hashing uniforme semplice: ogni chiave ha la stessa probabilità di essere mandata in una qualsiasi delle m

celle.

11.3.1 Metodo della divisione

Quando si applica il **metodo della divisione** per creare una funzione hash, una chiave k viene associata a una delle m celle prendendo il resto della divisione fra k e m

$$h(k) = k \mod m$$

Quando utilizziamo il metodo della divisione, di solito, evitiamo certi valori di m. Per esempio, m non dovrebbe essere una potenza di 2, perché se $m = 2^p$, allora $h(k)$ rappresenta proprio i p bit meno significativi di k. Un numero primo non troppo vicino a una potenza esatta di 2 è spesso una buona scelta per m.

11.3.2 Metodo della moltiplicazione

Il **metodo della moltiplicazione** per creare funzioni hash si svolge in due passi. Prima moltiplichiamo la chiave k per una costante A nell'intervallo $0 < A < 1$ ed estraiamo la parte frazionaria di kA. Poi moltiplichiamo questo valore per m e prendiamo la parte intera inferiore del risultato.

$$h(k) = m(kAmod1)$$

dove $kAmod1$ rappresenta la parte frazionaria di kA, cioè $kA - \lceil kA \rceil$. Un vantaggio del metodo della moltiplicazione è che il valore di m non è critico, ma dipende completamente dal valore A. Tipicamente, lo scegliamo come una potenza di 2, il che rende semplice implementare la funzione hash nella maggior parte dei calcolatori. Sebbene questo metodo funzioni con qualsiasi valore della costante A, tuttavia con qualche valore funziona meglio che con altri.

$$A = (\sqrt{5} - 1) = 0,6180339887\ldots$$

11.3.3 Indirizzamento aperto

Nell'**indirizzamento aperto**, tutti gli elementi sono memorizzati nella tavola hash stessa; ogni cella della tavola contiene un elemento dell'insieme dinamico o la costante NIL. Diversamente dal concatenamento, non ci sono liste né elementi memorizzati all'esterno della tavola. Quindi, nell'indirizzamento aperto, la tavola hash può "riempirsi" a tal punto che non possono essere effettuati altri inserimenti; una conseguenza è che il fattore di carico α non supera mai 1. Per effettuare un inserimento mediante l'indirizzamento aperto, esaminiamo in successione le posizioni della tavola hash(**ispezione**), finché non troviamo una cella vuota in cui inserire la chiave. Anziché seguire sempre lo stesso ordine $0, 1, \ldots, m-1$ (che richiede un tempo di ricerca $\Theta(n)$), la sequenza delle posizioni esaminate durante una ispezione **dipende dalla chiave da inserire**. Per determinare quali celle esaminare, estendiamo la funzione hash in modo da includere l'ordine di ispezione (a partire da 0) come secondo input. Quindi, la funzione hash diventa:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

Con l'indirizzamento aperto si richiede che, per ogni chiave k, la **sequenza di ispezione**

$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$

sia una permutazione di $\langle 0, 1, \ldots, m-1\rangle$, in modo che ogni posizione della tavola hash possa essere considerata come possibile cella in cui inserire una nuova chiave mentre la tavola si riempie. Supponiamo che gli elementi della tavola hash T siano chiavi senza dati satelliti; la chiave k è identica all'elemento che contiene la chiave k. La procedura HASH-INSERT ritorna il numero della cella in cui ha memorizzato la chiave k oppure segnala un errore se la tavola era già piena

```
1  HASH-INSERT(T, k)
2       i <- 0
3       do
4            j = h(k,i)
5            if T[j] == NIL
6                 T[J] <- k
7                 return j
8            else
9                 i <- i +1
10      while i≠ m
```

L'algoritmo che ricerca la chiave k esamina la stessa sequenza di celle che ha esaminato l'algoritmo di inserimento quando ha servito k. Quindi, la ricerca può terminare (senza successo) quando trova una cella vuota. L'algoritmo di ricerca è identico a quello Insert, quindi non pacco. La cancellazione da una tavola hash a indirizzamento aperto è un'operazione difficile. Quando cancelliamo una chiave dalla cella i, non possiamo semplicemente marcare questa cella come vuota. Così facendo, potrebbe essere impossibile ritrovare qualsiasi chiave k nel cui inserimento abbiamo esaminato la cella i e l'abbiamo trovata occupata.

Ispezione lineare

Data una funzione hash ordinaria $h' : U \to \{0, 1, \ldots, m-1\}$ che chiameremo **funzione hash ausiliaria**, il metodo dell'**ispezione lineare** usa la funzione hash

$h(k,i) = (h'(k,i) + i) \mod m$

per $i = 0, 1, \ldots, m-1$. Data la chiave k, la prima cella esaminata è $T[h'(k)]$, che è la cella data dalla funzione hash ausiliaria; la seconda cella esaminata è $T[h'(k)+1]$ fino a $T[h'(k)-1]$. Poiché la prima cella ispezionata determina l'intera sequenza di ispezioni, ci sono soltanto m sequenze di ispezione distinte. Una rappresentazione grafica può essere la seguente:

T
h(k,0)
h(k,1)
h(k,2)

Tabella 11.1: Tabella hash ad indirizzamento aperto. Si sceglie un indice dato dalla funzione hash per iniziare a cercare un posto vuoto, quando si trova si inserisce il valore

Molto semplice da implementare, presenta un problema noto come **addensamento primario** (*agglomerazione*): si formano lunghe file di celle occupate, che aumentano il tempo medio di ricerca.

Ispezione quadratica

L'**ispezione quadratica** usa una funzione hash della forma

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$

dove h' è una funzione hash ausiliaria, c_1 e $c_2 \neq 0$ sono costanti ausiliarie e $i = 0, 1, \ldots, m-1$. La posizione iniziale esaminata è $T[h'(k)]$; le posizioni successivamente esaminate sono distanziate da quantità che dipendono in modo quadratico dal numero d'ordine di ispezione i. Funziona meglio dell'ispezione lineare, ma i valori c_1 ed m non si possono scegliere arbitrariamente. Anche questa implementazione porta ad una forma più lieve di addensamento, che chiameremo **addensamento secondario**.

Doppio Hashing

Il doppio hashing è uno dei metodi migliori disponibili per l'indirizzamento aperto, perché le permutazioni prodotte hanno molte delle caratteristiche delle permutazioni scelte a caso. Il **doppio hashing** usa una funzione hash della forma

$h(k,i) = (h_1(k) + ih_2(k)) \mod m$ dove h_1 e h_2 funzioni hash ausiliarie. L'ispezione inizia dalla posizione $T[h_1(k)]$; le successive posizioni sono distanziate dalle precedenti ispezioni di una quantità $h_2(k) \mod m$. Quindi, diversamente dal caso dell'ispezione lineare o quadratica, la sequenza di ispezione qui dipende in due modi dalla chiave k, perché possono variare sia la posizione iniziale di ispezione sia la distanza fra due posizione successive di ispezione.

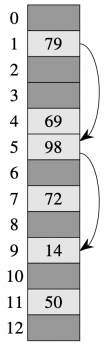


Figura 11.4: Inserimento con doppio hashing. La tavola hash ha dimensione 13 con $h_1(k) = k \mod 13$ e $h_2(k) = 1 + (k \mod 11)$. Poiché $14 = 1(\mod 13)$ e $14 = 3(mod 11)$, la chiave 14 viene inserita nella cella vuota 9, dopo che le celle 1 e 5 sono state esaminate e trovate occupate

Il valore $h_2(k)$ deve essere relativamente primo con la dimensione m della tavola hash perché venga ispezionata l'intera tavola hash. Un modo pratico per garantire questa condizione

è scegliere m potenza di 2 e definire h_2 in modo che produca sempre un numero dispari. Quando la tabella hash si riempie, si raddoppia la sua dimensione m e si ricalcolano tutti gli hash relativi alle chiavi presenti all'interno della tabella e si reinseriscono.

12 Algoritmi elementari per grafi

I grafi sono strutture dati molto comuni in informatica e gli algoritmi che operano con essi sono di fondamentale importanza in questo campo. Nel descrivere il tempo di esecuzione per un dato grafo $G = (V, E)$, di solito, esprimiamo la dimensione dell'input in funzione del numero di vertici $|V|$ e del numero di archi $|E|$ del grafo. Ovvero, ci sono due parametri importanti che descrivono la dimensione dell'input, non uno solo. Nella notazione asintotica e *soltanto* all'interno di questa notazione, il simbolo V indica $|V|$ e il simbolo E indica $|E|$. Ad esempio, se l'algoritmo è eseguito nel tempo $O(VE)$, intendiamo che l'algoritmo viene eseguito nel tempo $O(|V||E|)$. Un'altra convenzione che adottiamo riguarda la pseudo codifica. Indichiamo con G.V l'insieme dei vertici di un grafo G e con $G.E$ l'insieme degli archi del grafo.

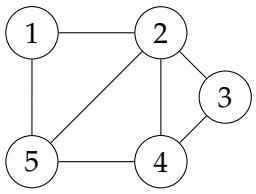


Figura 12.1: Grafo G = (V,E) dove $|V| = 5$ e $|E| = 7$

12.1 Rappresentazione dei grafi

Ci sono due metodi standard per rappresentare un grafo G = (V, E): come una collezione di liste di adiacenza o come una matrice di adiacenza. La rappresentazione con liste di adiacenza permette di rappresentare in *modo compatto* i grafi **sparsi**, quelli in cui il numero di Archi($|E|$) è molto più piccolo del numero dei vertici al quadrato($|V^2|$). Tuttavia potrebbe essere preferibile una rappresentazione con matrice di adiacenza quando il grafo è **esempio** - $|E|$ prossimo a $|V|^2$ - o quando dobbiamo essere in grado di dire rapidamente se c'è un arco che collega due vertici particolari.

La **rappresentazione con liste di adiacenza** di un grafo $G = (V, E)$ consiste in un array Adj di $|V|$ liste, una per ogni vertice in V. Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici v tali che esista un arco $(u, v) \in E$. Ovvero $Adj[u]$ include tutti i vertici adiacenti a u in G.

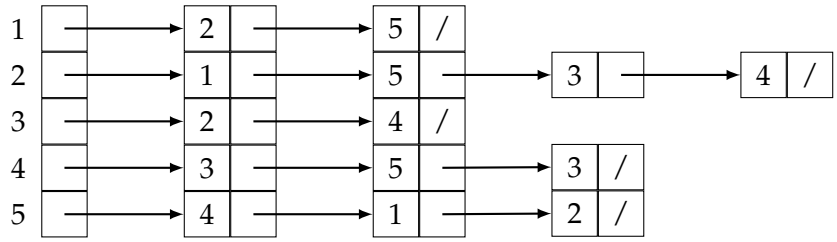


Figura 12.2: Rappresentazione del grafo non orientato (Fig:12.1) con liste di adiacenza

.

Consideriamo ora il caso di un grafo orientato:

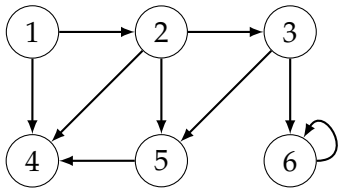


Figura 12.3: Grafo orientato G = (V,E) dove $|V| = 6$ e $|E| = 9$

La rappresentazione con le lista di adiacenza per tale grafo è la seguente:

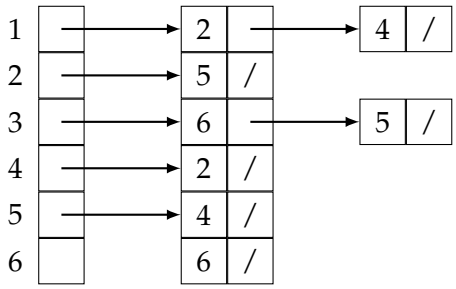


Figura 12.4: Rappresentazione con liste di adiacenza di (Fig:12.3)

Se G è un grafo orientato, la somma delle lunghezze di tutte le lista di adiacenza è $|E|$, perché un arco della forma (u, v) è rappresentato inserendo v in $Adj[u]$. Se G è un grafo non orientato, la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, perché se (u, v) è un arco non orientato, allora u appare nella lista di adiacenza di v e viceversa. Per i grafi orientati e non orientati, la rappresentazione con liste di adiacenza ha l'interessante proprietà che la quantità di memoria richiesta è $\Theta(V + E)$. Le liste di adiacenza possono essere facilmente adattate per rappresentare i **grafi pesati**, cioè, i grafi per i quali ogni arco ha un **peso** associato, tipicamente dato da una **funzione peso** $w : E \mapsto \mathbb{R}$. La rappresentazione con liste di adiacenza è molto robusta, nel senso che può essere modificata per supportare molte altre varianti di grafi. Uno svantaggio potenziale della rappresentazione con liste di adiacenza

è che non c'è modo più veloce per determinare se un particolare arco (u, v) è presente nel grafo che cercare v nella lista di adiacenza $Adj[u]$.

Per la **rappresentazione con matrice di adiacenza di un grafo G =** (V, E) si suppone che i vertici siano numerati $1, 2, \ldots, |V|$ in modo arbitrario. La rappresentazione con matrice di adiacenza di un grafo G costante in una matrice di dimensioni $|V| \times |V|$ tale che

$$a_{ij} = \begin{cases} 1 \text{ se } (1, j) \in E \\ 0 \text{ negli altri casi} \end{cases}$$

Consideriamo ora la matrice di adiacenza del grafo in figura 12.1

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figura 12.5: Rappresentazione con matrice di adiacenza del grafo G (Fig : 12.1)

Nel caso del grafo orientato:

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figura 12.6: Rappresentazione con matrice di adiacenza del grafo G (Fig : 12.3)

Osservate la simmetria rispetto alla diagonale principale della matrice di adiacenza. Poiché (u, v) e (v, u) rappresentano lo stesso arco in un grafo non orientato, la matrice di adiacenza A di un grafo non orientato è uguale alla sua trasposta $A = A^T$. In alcune applicazioni conviene memorizzare soltanto gli elemento che si trovano sopra e lungo la diagonale della matrice di adiacenza, riducendo così la memoria richiesta per memorizzare il grafo quasi della metà. La matrice di adiacenza di un grafo richiede $\Theta(V^2)$ indipendentemente dal numero di archi di un grafo. Sebbene la rappresentazione con liste di adiacenza sia asintoticamente efficiente almeno quanto la rappresentazione con matrice di adiacenza, tuttavia quando

i grafi sono abbastanza piccoli potrebbe essere preferita la matrice di adiacenza per la sua semplicità.

12.2 Visita in ampiezza

La **visita in ampiezza** è uno dei più semplici algoritmi di ricerca nei grafi. Dato un grafo G = (V, E) e un vertice distinto s, detto **sorgente**, la visita in un ampiezza ispeziona sistematicamente gli archi di G per "scoprire" tutti i vertici che sono raggiungibili da s. Calcola la distanza (il minor numero di archi) da s a ciascun vertice raggiungibile. Per ogni vertice v raggiungibile da s, il cammino semplice nell'albero BF che va da s a v corrisponde a un "cammino minimo" da s a v in G, cioè un cammino che contiene il minor numero di archi. L'algoritmo opera su grafi orientati e non orientati. La visita in ampiezza è chiamata così perché espande la frontiera fra i vertici scoperti e quelli da scoprire in maniera uniforme lungo l'ampiezza della frontiera. Ovvero l'algoritmo scopre tutti i vertici che si trovano a distanza k da s, prima di scoprire i vertici a distanza $k + 1$. Per tenere traccia del lavoro svolto, la visita in ampiezza colora i vertici di bianco, di grigio o di nero. Inizialmente tutti i vertici sono bianchi; dopo possono diventare grigi e poi neri. Un vertice viene **scoperto** quando viene incontrato per la prima volta durante la visita; in quel momento cessa di essere un vertice bianco. I vertici grigi e neri, quindi, sono vertici che sono stati scoperti, ma l'algoritmo li mantiene distinti per fare in modo che la visita proceda in ampiezza. Se $(u, v) \in E$ e il vertice u è nero, allora il vertice v è grigio oppure nero; ovvero tutti i vertici adiacenti ai vertici neri sono stati scoperti. I vertici grigi possono avere qualche vertice bianco adiacente; essi rappresentano la frontiera fra i vertici scoperti e quelli da scoprire. La visita in ampiezza costruisce l'albero BF che, inizialmente contiene soltanto la sua radice, che è il vertice sorgente s. Quando un vertice bianco v viene scoperto durante l'ispezione della lista di adiacenza di un vertice u già scoperto, il vertice v e l'arco (u, v) vengono aggiunti all'albero. Per convenzione $V(G)$ indica tutti i vertici del grafo G e $E(G)$ i suoi archi. L'attributo $color[u]$ indica il colore del nodo u, $d[u]$ la distanza dal nodo u al nodo s e $\pi[u]$ il nodo da cui siamo arrivati.

BFS(G, s)

```
1   ∀u ∈ V(G)
2       color[u] ← white
3       d[u] ← ∞
4       π[u] ← NIL
5   d[s] ← 0
6   color[s] ← GRAY
7   Q ← {s}
8   WHILE Q ≠ ∅
9       u ← HEAD{Q}
10      ∀v ∈ Adj[u]
11          IF color[v] = WHITE
12              color[v] ← GRAY
13              d[v] ← d[u] + 1
14              π[v] ← u
```

```
15                  ENQUEUE(Q, v)
16          color[u] ← BLACK
17          DEQUEUE(Q)
```

Le righe $1-4$ colorano di bianco tutti i vertici, assegnano all'attributo $d[u]$ il valore infinito per ogni vertice u e assegnano al padre di ogni vertice il valore NIL. La riga 5 inizializza $d[s]$ a 0 poiché la distanza di un nodo a sè stesso è 0, la riga 6 colora s di grigio e nella 7 si crea una coda FIFO dove si inserisce il vertice s. Il ciclo **while** (righe 8-17) si ripete finché restano dei vertici grigi, che sono vertici scoperti le cui liste di adiacenza non sono state ancora completamente esaminate. Questo ciclo **while** conserva la seguente invariante:

Quando viene eseguito il test della riga 8, la coda Q è formata dall'insieme dei vertici grigi.

Prima della prima iterazione l'unico vertice grigio, e l'unico vertice in Q è il vertice s e ogni iterazione del ciclo conserva questa invariante. La riga 9 preleva dalla coda il primo elemento inserito. La riga 10 è un ciclo **for** che esamina ciascun vertice v nella lista di adiacenza di u. Se v è bianco, significa che v non è stato visitato e quindi si assegna a v il colore *grigio*, $d[v]$ ora è $d[u]+1$ poiché la distanza da s ad arrivare a u più 1 poiché v è vertice adiacente di u. $\pi[u]$ è assegnato il valore di u poiché è il vertice da cui siamo partiti per arrivare a v. Fatte queste assegnazioni, sempre all'interno del ciclo, inseriamo nella coda FIFO il vertice v. Con questa ultima operazione, si sono modificati tutti i parametri con cui potevamo operare di v, quindi continuiamo con la successiva iterazione e prendiamo un altro nodo adiacente a u, quando non ce ne saranno più togliamo u dalla coda e gli assegniamo il colore nero (righe 16-17).

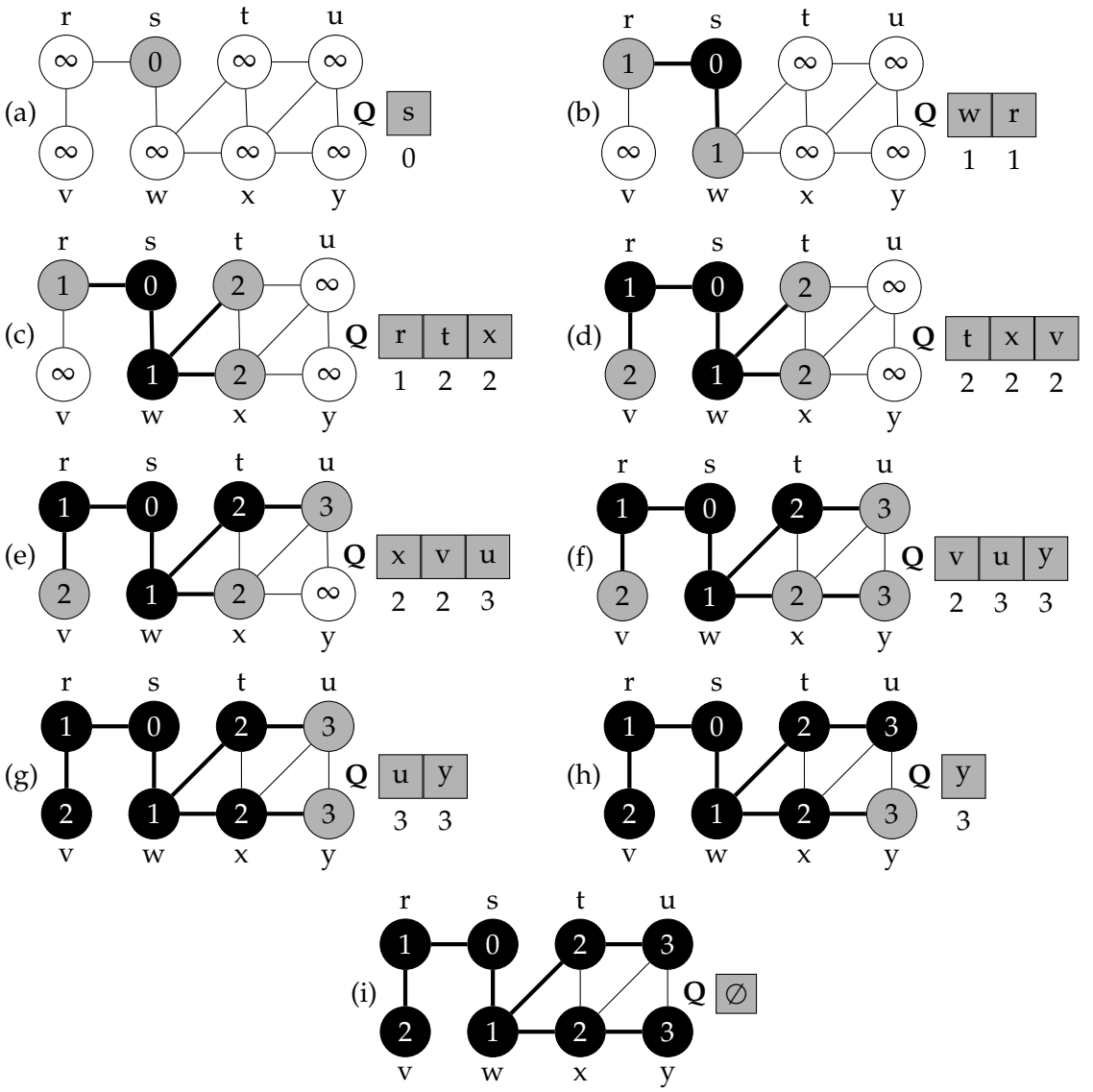


Figura 12.7: Il funzionamento della procedura BFS con un grafo non orientato. Gli archi sono evidenziati quando vengono prodotti da BFS. All'interno di ciascun vertice u è indicato il valore di $d[u]$. La coda Q è rappresentata all'inizio di ogni iterazione del ciclo **while**. In corrispondenza dei vertici nella coda sono annotate le distanze dai vertici

Analisi

Dopo l'inizializzazione, nessun vertice sarà colorato di bianco, quindi il test a riga 11 garantisce che ciascun vertice venga accodato al più una volta e, di conseguenza, venga eliminato dalla coda al più una volta e, di conseguenza, venga eliminato dalla coda al più una volta. Le operazioni di inserimento e cancellazione della coda richiedono un tempo $O(1)$, quindi il tempo totale dedicato alle operazioni con la coda è $O(V)$. Poiché la lista di adiacenza di cia-

scun vertice viene ispezionata soltanto quando il vertice viene rimosso dalla coda, ogni lista di adiacenza viene ispezionata al più una volta. Poiché la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(E)$, il tempo totale impiegato per ispezionare le liste di adiacenza è $O(E)$. Il costo aggiuntivo di inizializzazione è $O(V)$, quindi il tempo di esecuzione totale di BFS è $O(V + E)$.

Cammini minimi

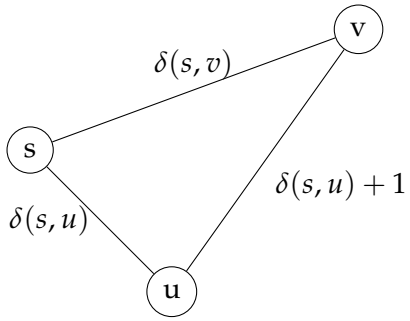
Definiamo la **distanza di cammino minimo** $\delta(s, v)$ da s a v come il numero minimo di archi in un cammino qualsiasi che va dal vertice s al vertice v; se non c'è un cammino che va da s a v, allora $\delta(s, v) = \infty$. Un cammino di lunghezza $\delta(s, v)$ da s a v è detto **cammino minimo** da s a v.

Lemma 1

Se G = (V, E) è un grafo orientato o non orientato e se $s \in V$ è un vertice arbitrario, allora per qualsiasi arco $(u, v) \in E$ si ha

$\delta(s, v) \leq \delta(s, u) + 1$

Dimostrazione Se u è un vertice raggiungibile da s, allora lo è anche da v. In questo caso, il cammino minimo da s a v non può essere più lungo del cammino minimo da s a u seguito dall'arco (u, v), quindi la disuguaglianza è valida. Se u non è raggiungibile da s, allora $\delta(s, v) = \infty$ e la disuguaglianza è valida.



Lemma 2

Sia G=(V, E) un grafo orientato o non orientato e supponiamo che BFS venga eseguita sul grafo G da un dato vertice sorgente $s \in V$. Allora, al termine della procedura, per ogni vertice $v \in V$, il valore $d[v]$ calcolato da BFS soddisfa la relazione $d[v] \geq \delta(s, v)$

Dimostrazione Applichiamo l'induzione sul numero di operazioni ENQUEUE. La nostra ipotesi induttiva è che $d[v] \geq \delta(s, v)$ per ogni $v \in V$. Il caso base dell'induzione è la situazione che si ha dopo subito dopo che il vertice s è stato inserito nella coda. L'ipotesi induttiva è vera qui, perché $d[s] = 0 = \delta(s, s)$ e $d[v] = \infty \geq \delta(s, v)$ per ogni $v \in V - \{s\}$. Per il passo induttivo, consideriamo un vertice bianco v che viene scoperto durante la visita a partire da

un vertice u. L'ipotesi induttiva implica che $d[u] \geq \delta(s, u)$:

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) \end{aligned}$$

Il valore del vertice $d[v]$ non cambierà più poiché viene inserito al più una volta nella coda e quindi l'ipotesi induttiva è vera.

Lemma 3

Supponiamo che durante l'esecuzione di BFS su un grafo G = (V, E), la coda Q contenga i vertici $\langle v_1, v_2, \ldots, v_r \rangle$ dove v_1 è l'inizio della coda Q e v_r è la fine. Allora, $d[v_r] \leq d[v_1] + 1$ e $d[v_i] \leq d[v_i + 1]$ per $i = 1, 2, \ldots, r - 1$.

Dimostrazione La dimostrazione è per induzione sul numero di operazioni eseguite con la coda. Inizialmente, quando la coda contiene soltanto s, il lemma è certamente valido. Per il passo induttivo, dobbiamo provare che il lemma è valido dopo l'inserimento e la cancellazione di un vertice nella coda. Se il vertice v_1 viene eliminato dalla coda, v_2 diventa il nuovo inizio della coda. Per l'ipotesi induttiva $d[v_1] \leq d[v_2]$; ma allora abbiamo $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$ e le restanti disuguaglianze rimangono inalterate. Quando inseriamo nella coda un vertice v, esso diventa v_{r+1}. In quel momento, abbiamo già eliminato dalla coda Q il vertice u, la cui lista di adiacenza è quella correntemente ispezionata e, per l'ipotesi induttiva, il nuovo inizio della coda v_1 ha $d[v_1] \geq d[u]$. Pertanto $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. Per l'ipotesi induttiva, abbiamo anche che $d[v_r] \leq d[u] + 1$, ovvero $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$, e le restanti disuguaglianze restano inalterate.

Corollario 1 Supponiamo che i vertici v_i e v_j siano inseriti nella coda durante l'esecuzione di BFS e che v_i sia inserito nella coda prima di v_j. Allora $d[v_i] \leq d[v_j]$ nell'istante in cui v_j viene inserito nella coda.

Dimostrazione Conseguenza immediata del lemma 3 e della proprietà che ogni vertice avente un valore finito d al più una volta durante l'esecuzione di BFS.

Teorema 12.2.1 (correttezza della visita in ampiezza). Sia G = (V, E) un grafo orientato o non orientato e supponiamo che la procedura BFS venga eseguita sul grafo G dato un vertice sorgente $s \in V$. Allora, durante la sua esecuzione, BFS scopre tutti i vertici $v \in V$ che sono raggiungibili dalla sorgente s e, alla fine dell'esecuzione, $d[v] = \delta(s, v)$ per ogni $v \in V$. Inoltre, per qualsiasi vertice $v \neq s$ che è raggiungibile da s, uno dei cammini minimi da s a v è un cammino minimo da s a $\pi[v]$ seguito dall'arco $(\pi[v], v)$.

Dimostrazione Supponiamo, per assurdo, che qualche vertice riceva un valore d che non è uguale alla distanza del suo cammino minimo. Sia v il vertice con il minimo $\delta(s, v)$ che riceve questo valore errato d; chiaramente $v \neq s$. Per il Lemma 2, $d[v] \geq \delta(s, v)$ e quindi $d[v] \geq \delta(s, v)$. Il vertice v deve essere raggiungibile da s, perché se non lo fosse, allora $\delta(s, v) = \infty \geq d[v]$. Sia u il vertice che precede immediatamente v in un cammino minimo da s a v, cosicché $\delta(s, v) = \delta(s, u)$. Ponendo insieme queste proprietà, si ha

$$d[v] \geq \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \tag{12.1}$$

Adesso consideriamo il momento in cui BFS sceglie di eliminare il vertice u dalla coda Q. In quel momento, il vertice v può essere bianco, grigio o nero. Se il vertice v è bianco, allora $d[v] = d[u] + 1$ che contraddice la disuguaglianza sopra. Se il vertice v è nero, allora era stato già rimosso dalla coda, e per il **Corollario 1**, si ha $d[v] \leq d[u]$ che contraddice ancora la disuguaglianza. Se il vertice v è grigio, allora era stato colorato di grigio dopo la cancellazione della coda di qualche vertice w, che era stato cancellato da Q prima di u e per il quale $d[u] = d[w] + 1$. Per il **corollario 1**, però, $d[w] \leq d[u]$, e quindi si ha $d[v] = d[w] + 1 \leq d[u] + 1$, che contraddice la disuguaglianza sopra. Dunque, possiamo concludere che $d[v] = \delta(s, v)$ per ogni $v \in V$. Tutti i vertici raggiungibili da s devono essere scoperti, perché se non lo fossero, avrebbero $\infty = d[v] > \delta(s, v)$. Per concludere la dimostrazione del teorema, osserviamo che se $\pi[v] = u$, allora $d[v] = d[u] + 1$. Quindi possiamo ottenere un cammino minimo da s a v prendendo un cammino minimo da s a $\pi[v]$ e poi attraversando l'arco $(\pi[v], v)$.

12.3 Visita in profondità

La strategia adottata dalla visita in profondità consiste nel visitare il grafo sempre più in "profondità" se possibile. Gli archi vengono ispezionati a partire dall' ultimo vertice scoperto v che ha ancora archi non ispezionati che escono da esso. Quando tutti gli archi di v sono stati ispezionati, la visita fa "marcia indietro" per ispezionare gli archi che escono dal vertice dal quale v era stato scoperto. Diversamente dalla visita in ampiezza, il cui sottografo dei predecessori forma un albero, il sottografo dei predecessori prodotto da una visita in profondità può essere formato da **più alberi** perché la visita può essere ripetuta da più sorgenti[1] Il seguente pseudocodice ' l'algoritmo di base che effettua una visita in profondità. Il grafo di input G può essere orientato o non orientato. La variabile *time* è una variabile globale che utilizziamo per registrare le informazioni temporali.

```
1 DFS(G)
2       ∀u ∈ V[G]
3             color[u] ← WHITE
4             π[u] ← NIL
5       TIME ← 0
6       ∀u ∈ V[G]
7             IF color[u] = WHITE
8                   DFS_VISIT(u)
```

Ogni vertice v ha due informazioni temporali: $d[u]$ e $f[u]$ che rispettivamente indicano il momento in cui il vertice v è stato scoperto e il momento in cui la visita completa l'ispezione della lista di adiacenza del vertice v. Queste informazioni temporali sono utilizzate in molti

[1]Sebbene, in teoria, una visita in ampiezza possa procedere da più sorgenti e una visita in profondità possa essere limitata a una sola sorgente, tuttavia il nostro approccio riflette il modo in cui vengono tipicamente utilizzati i risultati di queste visite. La visita in ampiezza, di solito, è utilizzata per trovare le distanze dei cammini minimi da una sorgente data. La visita in profondità, di solito, è una subroutine in un altro algoritmo

algoritmi che operano con i grafi, e in generale, agevolano l'analisi del comportamento della visita in profondità. La procedura DFS registra nell'attributo $d[u]$ il momento in cui scopre il vertice u e $f[u]$ il momento in cui completa la visita del vertice u. Queste informazioni temporali sono numeri interi compresi fra 1 e $2|V|$, perché ciascuno dei $|V|$ vertici può essere scoperto una sola volta e la sua visita può essere completata una sola volta. Per ogni vertice si ha $d[u] < f[u]$. Il seguente pseudocodice è la implementazione di DFS_VISIT, prende come input un nodo u di colore bianco.

```
1  DFS_VISIT(u)
2      color[u] ← GRAY
3      d[u] ← TIME ← TIME + 1
4      ∀v ∈ Adj[u]
5          IF color[v] = WHITE
6              π[v] ← u
7              DFS_VISIT(v)
8      color[u] ← BLACK
9      f[u] ← TIME ← TIME + 1
```

La procedura DFS opera nel seguente modo. Le righe 2-4 colorano di bianco tutti i vertici e inizializzano i loro attributi π a NIL. La riga 5 azzera il contatore globale del tempo. Le righe 6-8, controllano, uno alla volta, tutti i vertici in V e, quando trovano un vertice bianco, lo visitano utilizzando la procedura DFS_VISIT. Ogni volta che viene chiamata la procedura DFS_VISIT(u), il vertice u diventa la radice di un nuovo albero della foresta DF. Quando la procedura DFS termina, a ogni vertice u è stato assegnato un **tempo di scoperta** $d[u]$ e un **tempo di completamento** $f[u]$.

In ogni chiamata di DFS_VISIT(u), il vertice u è inizialmente bianco. La righe 2-3 colorano di grigio il nodo u e incrementano il valore di TIME assegnandolo a $d[u]$. Le righe 4-7 ispezionano ogni vertice v adiacente a u e visitano in modo ricorsivo il vertice v, se è bianco. Quando un vertice $v \in Adj[u]$ viene esaminato nella riga 4, diciamo che l'arco (u,v) è stato **ispezionato** dalla visita in profondità. Infine, dopo che tutti i nodi che escono da u sono stati ispezionati, le righe 8-9 colorano di nero u, incrementano *time* e registrano $f[u]$ il tempo di completamento della visita. Notate che i risultati della visita in profondità potrebbero dipendere dall'ordine in cui i vertici sono ispezionati nella riga 6 della procedura DFS e dall'ordine in cui i vicini di un vertice vengono visitati nella riga 4 dalla procedura DFS_VISIT. I cicli nelle righe 2-4 e nelle righe 6-8 di DFS impiegano un tempo $\Theta(V)$, escluso il tempo per eseguire le chiamate DFS_VISIT. Tale procedura è chiamata esattamente per ogni vertice $v \in V[G]$, perché DFS_VISIT viene invocata soltanto se un vertice è bianco e la prima cosa che fa è colorare di grigio il vertice. Durante un'esecuzione di DFS_VISIT, il ciclo nelle righe 4-7 viene eseguito $|Adj[v]|$ volte. Poiché

$$\sum_{v \in V[G]} |Adj[v]| = \Theta(E)$$

Quindi il tempo di esecuzione di DFS è dunque $\Theta(V + E)$.

Proprietà della visita in profondità

La visita in profondità fornisce informazioni preziose sulla struttura di un grafo. La più importante proprietà della visita in profondità è che il sottografo dei predecessori G_π forma effettivamente una foresta di alberi, in quando la struttura degli alberi DF rispecchia esattamente la struttura delle chiamate ricorsive DFS_VISIT. Ovvero, $u = \pi[v]$ se e soltanto se DFS_VISIT(v) è stata chiamata durante una visita della lista di adiacenza di u. In aggiunta, il vertice v è un discendente del vertice u nella foresta DF se e soltanto se v viene scoperto durante il periodo in cui u è grigio. Un'altra importante proprietà è che i tempi di scoperta e di completamento hanno una **struttura a parentesi**. Se rappresentiamo la scoperta del vertice u con una parentesi aperta, allora la storia delle scoperte e dei completamenti produce un'espressione ben formata, nel senso che le parentesi sono opportunamente annidate.

Teorema delle parentesi

In un qualsiasi visita in profondità di un grafo G = (V, E) (orientato o non orientato), per ogni coppia di vertici u e v, è soddisfatta una sola delle seguenti tre condizioni:

- Gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono completamente disgiunti; e inoltre u e v non sono discendenti l'uno dell'altro nella foresta DF.
- L'intervallo $[d[u], f[u]]$ è interamente contenuto nell'intervallo $[d[v], f[v]]$; inoltre u è discendente di v in un albero DF.
- L'intervallo $[d[v], f[v]]$ è interamente contenuto nell'intervallo $[d[u], f[u]]$; inoltre v è discendente di u in un albero DF.

Iniziamo con il caso $d[u] < d[v]$. Ci sono due sottocasi da considerare:

1. $d[v] < f[u]$: v è stato scoperto mentre u era ancora grigio. Questo implica che v è un discendente di u. Inoltre, poiché v è stato scoperto più recentemente di u, vengono ispezionati tutti i suoi archi uscenti e l'ispezione di v viene completata prima che la visita riprenda da u e venga completata l'ispezione di u. In questo caso quindi, l'intervallo $[d[v], f[v]]$ è interamente contenuto nell'intervallo $[d[u], f[u]]$.
2. $f[u] < d[v]$. Dato che la visita del nodo u è iniziata prima del nodo v e che l'istante in cui l'ispezione di u finisce avviene prima della scoperta del nodo v si deduce che v è un nodo bianco non discendente da u, quindi l'intervallo $[d[v], f[v]]$ non è contenuto in $[d[u], f[u]]$.

L'altro caso $d[v] < d[u]$ è simile, basta scambiare u con v.

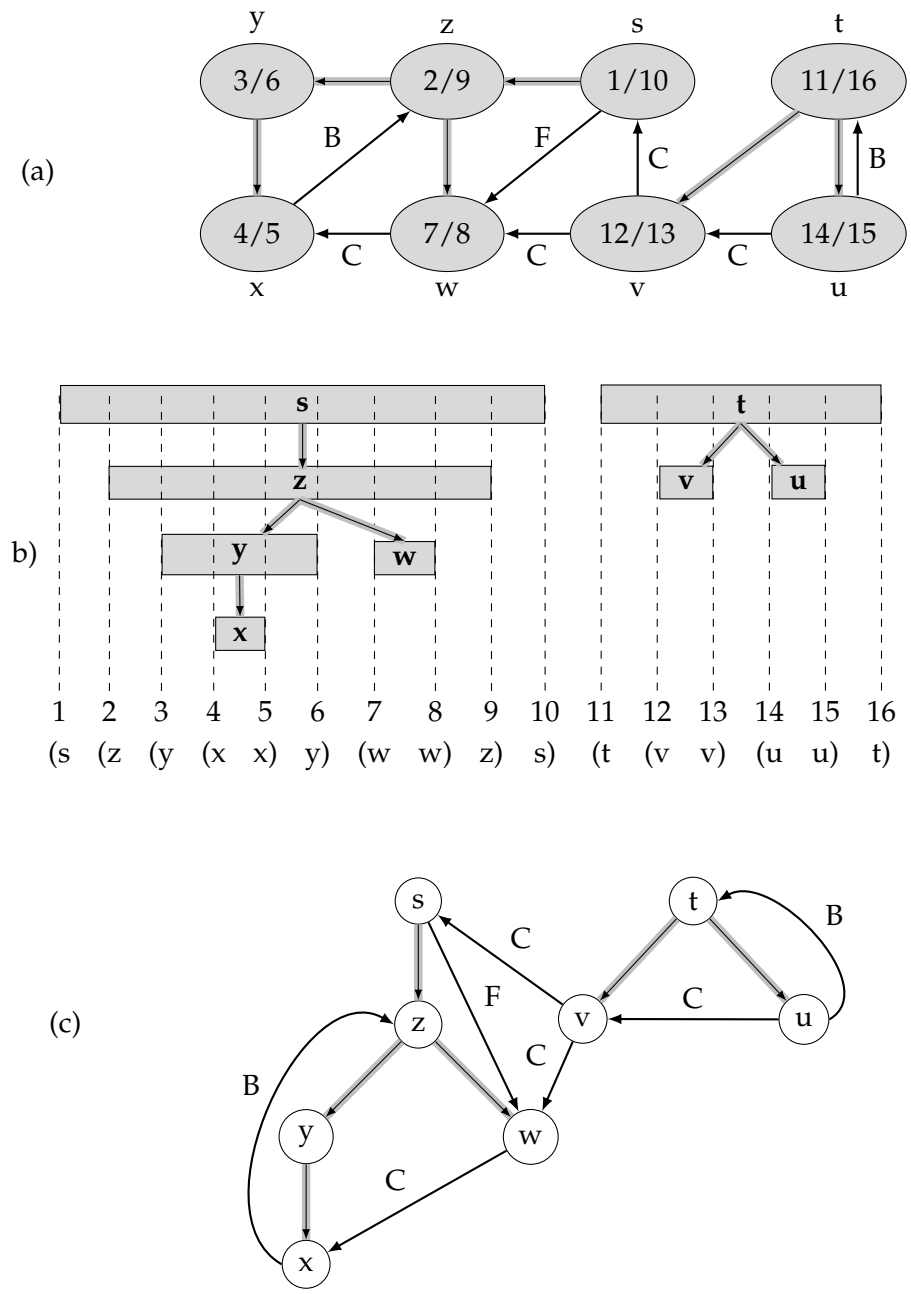


Figura 12.8: Proprietà della visita in profondità. (a) Il risultato di una visita in profondità di un grafo orientato. I vertici includono le informazioni temporali e i tipi di archi. (b) Gli intervalli fra il tempo di scoperta e il tempo di completamento di ciascun vertice corrispondono alla parentesizzazione illustrata. Ogni rettangolo si estende nell'intervallo definito dai tempi di scoperta e di completamento del corrispondente vertice. Sono illustrati gli archi d'albero. Se due intervalli si sovrappongono, allora uno viene annidato all'interno dell'altro e il vertice che corrisponde all'intervallo più piccolo è un discendente del vertice che corrisponde all'intervallo più grande. (c) Il grafo della parte (a) ridisegnato con tutti gli archi d'albero e in avanti che scendono in un albero DF e con tutti gli archi all'indietro che salgono da un discendente antenato

Corollario 12.3.0.1 (**Annidamento degli intervalli dei discendenti**). Il vertice v è un discendente proprio del vertice u nella foresta DF per un grafo G (orientato o non orientato) se e soltanto se $d[u] < d[v] < f[v] < f[u]$.

Dimostrazione è una conseguenza immediata del teorema delle parentesi.

Teorema 12.3.1 (**Teorema del cammino bianco**). In una foresta DF di un grafo G = (V, E) (orientato o non orientato), il vertice v è un discendente del vertice u se e soltanto se, al tempo $d[u]$ in cui viene scoperto u, il vertice v può essere raggiunto da u lungo un cammino che è formato esclusivamente da vertici bianchi.

Dimostrazione

- $\Rightarrow$): se $v = u$, allora il cammino da u a v contiene soltanto il vertice u, che è ancora bianco quando impostiamo il valore $d[u]$. Adesso, supponiamo che v sia un discendente proprio di u nella foresta DF. Per il Corollario 12.0.1, $d[u] < d[v]$, e quindi v è bianco nel tempo $d[u]$. Poiché v può essere un discendente qualsiasi di u, tutti i vertici lungo l'unico cammino semplice da u a v nella foresta DF sono bianchi al tempo $d[u]$.
- $\Leftarrow$): supponiamo che ci sia un cammino di vertici bianchi da u a v al tempo $d[u]$ e che v non diventi un discendente di u nell'albero DF. Senza perdere in generalità, supponiamo che tutti gli altri vertici diversi da v lungo il cammino diventino discendenti di u. Sia w il predecessore di v lungo il cammino, cosicché w è un discendente di u e per il corollario 12.0.1, si ha $f[u] \leq f[u]$. Poiché v deve essere scoperto dopo u, ma prima che sia completata la visita di w, si ha $d[u] < d[v] < f[w] < f[u]$. Il teorema delle parentesi allora implica che l'intervallo $[d[v], f[v]]$ sia interamente contenuto nell'intervallo $[d[u], f[u]]$. Per il corollario 12.0.1 v deve essere un discendente di u

Classificazione degli archi

Un'altra interessante proprietà della visita in profondità è che la visita può essere utilizzata per classificare gli archi del grafo di input G = (V, E). Possiamo definire quattro tipi di archi in base alla foresta DF prodotta da una visita in profondità del grafo G.

1. **Archi d'albero**: sono gli archi nella foresta DF. L'arco (u,v) è un arco d'albero se v viene scoperto la prima volta durante l'esplorazione di (u, v).
2. **Archi all'indietro**: sono quegli archi (u,v) che collegano un vertice u a un antenato v in un albero DF. I cappi, che possono presentarsi nei grafi orientati, sono considerati archi all'indietro.
3. **Archi in avanti**: sono gli archi (u,v) (diversi dagli archi d'albero) che collegano un vertice u a un discendente v in un albero DF (*scorciatoia tra due nodi*).
4. **Archi trasversali**: tutti gli altri archi. Possono connettere vertici nello stesso albero DF, purché un vertice non sia un antenato dell'altro, oppure possono connettere vertici di alberi DF differenti.

L'algoritmo DFS possiede le informazioni necessarie per classificare gli archi che incontra. L'idea chiave è che ogni arco (u,v) può essere classificato in base al colore del vertice v che viene raggiunto quando l'arco viene ispezionato per la prima volta:

1. **WHITE** indica un arco d'albero.
2. **GRAY** indica un arco all'indietro.
3. **BLACK** indica un arco in avanti o trasversale.

Il primo caso è immediato nella specifica dell'algoritmo. Per il secondo caso, notate che i vertici grigi formano sempre una catena lineare di discendenti che corrisponde allo stack delle chiamate attive di DFS_VISIT; il numero di vertici grigi è uno più della profondità dell'ultimo vertice scoperto nella foresta DF. L'ispezione procede sempre a partire dal vertice grigio più profondo, quindi un arco che raggiunge un altro vertice grigio raggiunge un antenato. Il terzo caso gestisce l'ultima possibilità; si può dimostrare che tale arco (u,v) è un arco in avanti se $d[u] < d[v]$. Se il nodo u è stato scoperto prima del nodo v e il nodo v è nero, significa che il nodo v è già stato ispezionato raggiungendo da un altro percorso. Dato che il predecessore di $\pi[v] \neq u$, si deduce che l'arco (u,v) è un arco in avanti poiché congiunge un antenato con un un suo discendente non adiacente ad esso. Se $d[u] > d[v]$ e il colore v è impossibile che (u,v) sia un arco all'indietro o d'albero poiché v è di colore nero e quindi è già stato visitato nella DFS. Supponiamo che $f[v] < d[u]$, ciò significa che la fine dell'ispezione al nodo v è avvenuta prima dell'inizio della scoperta del nodo u, e quindi dato che $d[v] < f[v] < d[u] < f[u]$ è anche impossibile che l'arco (u,v) sia un arco all'indietro, poiché u non è un discendente di v, quindi l'arco (u,v) è un arco trasversale.

12.4 Ordinamento topologico

Un **ordinamento topologico** di un grafo G = (V,E) aciclico (**DAG**, *directed acyclic graph*) è un ordinamento lineare di tutti i suoi vertici tale che, se G contiene un arco (u,v), allora u appare prima di v nell'ordinamento (se il grafo non è aciclico, allora non è possibile effettuare alcun ordinamento lineare). Un ordinamento topologico di un grafo può essere visto come un ordinamento dei suoi vertici lungo una linea orizzontale in modo che tutti gli archi orientati siano diretti da sinistra a destra. I grafi orientati sono utilizzati in molte applicazioni per indicare le precedenze fra gli eventi.

Esempio

Un professore deve indossare determinati indumenti prima di altri (per esempio, le calze prima delle scarpe). Altri indumenti possono essere indossati in qualsiasi ordine. Un' arco orientato (u,v) nel dag indica che l'indumento u deve essere indossato prima dell'indumento v. Un ordinamento topologico di questo dag, quindi, determina la sequenza ordinata in cui indossare gli indumenti per vestirsi.

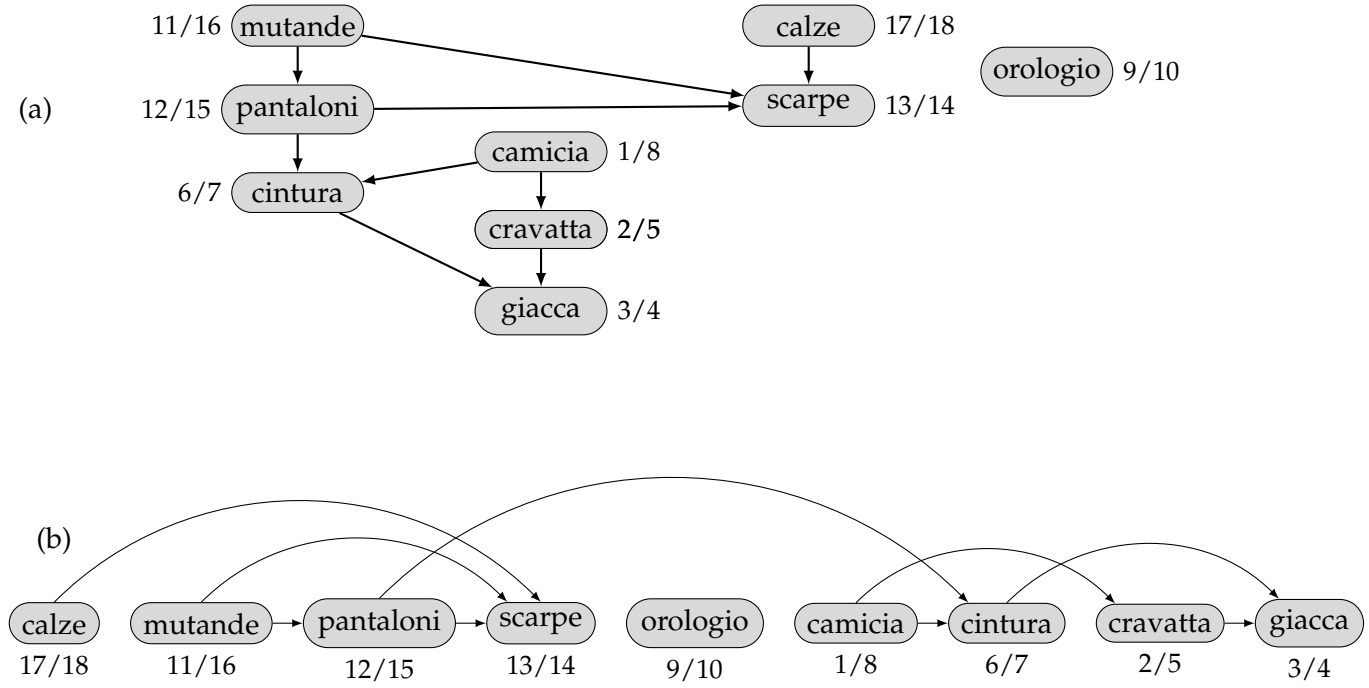


Figura 12.9: **(a)** Il professore ordina topologicamente i suoi indumenti quando si veste. Ogni arco orientato (u,v) significa che l'indumento u deve essere indossato prima dell'indumento v. **(b)** Lo stesso grafo rappresentato secondo l'ordinamento topologico. I suoi vertici sono ordinati da sinistra a destra in senso decrescente rispetto ai tempi di completamento. Notate che tutti gli archi orientati sono diretti da sinistra a destra.

La figura 12.9 illustra un caso che si verifica tutte le mattine quando il professore Bumstead si veste.

Il seguente semplice algoritmo ordina topologicamente un dag.

```
1 TOPOLOGICAL–SORT(G)
2     Chiamata DFS(G) per calcolare i tempi di completamento f[v]
3     per ciascun vertice v
4     Una volta completata l'ispezione di un vertice,
5         inserisce il vertice in testa a una lista concatenata
6     return la lista concatenata dei vertici
```

La figura 12.9(b) illustra come si presentano i vertici topologicamente in senso inverso rispetto ai loro tempi di completamento. Il tempo necessario per eseguire l'ordinamento topologico è $\Theta(V + E)$, perché la visita in profondità impiega un tempo $\Theta(V + E)$ e occorre un tempo $O(1)$ per inserire ciascuno dei $|V|$ vertici in testa alla lista concatenata.

Lemma 12.4.0.1. Un grafo orientato G è aciclico se e soltanto se una visita in profondità di G non genera archi all'indietro.

Dimostrazione

- $\Rightarrow$): supponiamo che ci sia un arco all'indietro (u,v). Allora, il vertice v è un antenato del vertice u nella foresta DF. Quindi, esiste un cammino che va da v a u nel grafo G e l'arco all'indietro (u,v) completa un ciclo.
- $\Leftarrow$): supponiamo che il grafo G contenga un ciclo c. Dimostriamo che una visita in profondità di G genera un arco all'indietro. Sia v il primo vertice che viene scoperto in c e sia (u,v) l'arco precedente in c. Al tempo $d[v]$, i vertici di c formano un cammino di vertici bianchi da v a u. Per la teoria del cammino bianco, il vertice u diventa un discendente di v nella foresta DF. Dunque, (u, v) è un arco all'indietro.

Teorema 12.4.1. TOLOGICAL-SORT(G) produce un ordinamento topologico di un grafo orientato aciclico G.

Dimostrazione Supponiamo che la procedura DFS venga eseguita su un dato dag G = (V,E) per determinare i tempi di completamento dei suoi vertici. È sufficiente dimostrare che per una coppia qualsiasi di vertici distinti $u, v \in V$, se esiste un arco in G che va da u a v, allora $f[v] < f[u]$. Consideriamo un arco qualsiasi (u, v) ispezionato da DFS(G). Quando questo arco viene ispezionato, il vertice v non può essere grigio, perché altrimenti v sarebbe un antenato di u e (u, v) sarebbe un arco all'indietro, contraddicendo il lemma 12.1.1. Quindi, il vertice v deve essere bianco o nero. Se v è bianco, diventa un discendente di u e quindi $f[v] < f[u]$. Se v è nero, la sua ispezione è stata già completata, quindi il valore di $f[v]$ è già stato impostato. Poiché stiamo ancora ispezionando dal vertice u, dobbiamo ancora assegnare un'informazione temporale a $f[u]$ e, quando lo faremo, avremo ancora $f[v] < f[u]$. Quindi, per qualsiasi arco (u, v) nel dag, si ha $f[v] < f[u]$, e questo dimostra il teorema.

12.5 Componenti fortemente connesse

Una componente fortemente connessa in un grafo orientato $G = (V, E)$ è un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici u e v in C, si ha $u \rightsquigarrow v$ e $v \rightsquigarrow u$; ovvero i vertici u e v sono raggiungibili l'uno l'altro. L'algoritmo per trovare le componenti connesse di un grafo $G = (V, E)$ utilizza il grafo trasposto di G, come il grafo $G^T = (V, E^T)$, dove $E^T = \{(u, v) : (v, u) \in E\}$. Ovvero E^T è formato dagli archi di G con le direzioni invertite. Data una rappresentazione con liste di adiacenza di G, il tempo richiesto per creare G^T è $O(V + E)$. Se G e G^T hanno esattamente le stesse componenti fortemente connesse, i vertici u e v sono raggiungibili l'uno dall'altro in G, se e soltanto se sono raggiungibili l'uno dall'altro in G^T. Il seguente algoritmo con tempo lineare $\Theta(V + E)$ calcola le componenti fortemente connesse di un grafo orientato G utilizzando due visite in profondità, una su G e una su G^T.

```
1 SCC(G)
2 Chiama DFS(G) per calcolare i tempi di completamento f[u]
3     per ciascun vertice u
4 Calcola G^T
5 Chiama DFS(G^T), ma nel ciclo principale di DFS,
6     considera i vertici in ordine decrescente
7     rispetto ai tempi f[u]
8 Genera l'output dei vertici di ciascun albero DF
```

```
9       che è stata prodotta come una singola
10      componente connessa
```

L'idea che sta alla base dell'algoritmo deriva da una proprietà fondamentale del **grafo delle componenti** $G^{SCC} = (V^{SCC}, E^{SCC})$, che è definito nel seguente modo. Supponiamo che G abbia le componenti fortemente connesse $C_1, C_2, \ldots, C_k$. L'insieme dei vertici V^{SCC} è $\{v_1, v_2, \ldots, v_k\}$ e contiene un vertice v_i per ogni componente connessa C_i di G. Esiste un arco $(v_i, v_j) \in E^{SCC}$ se G contiene un arco orientato (x, y) per qualche $x \in C_i$ e qualche $y \in C_j$. In altri termini, contraendo tutti gli archi i cui vertici incidenti sono all'interno della stessa componente fortemente connessa di G, si ottiene il grafico G^{SCC}. La proprietà fondamentale è che il grafo delle componenti è un grafo orientato aciclico.

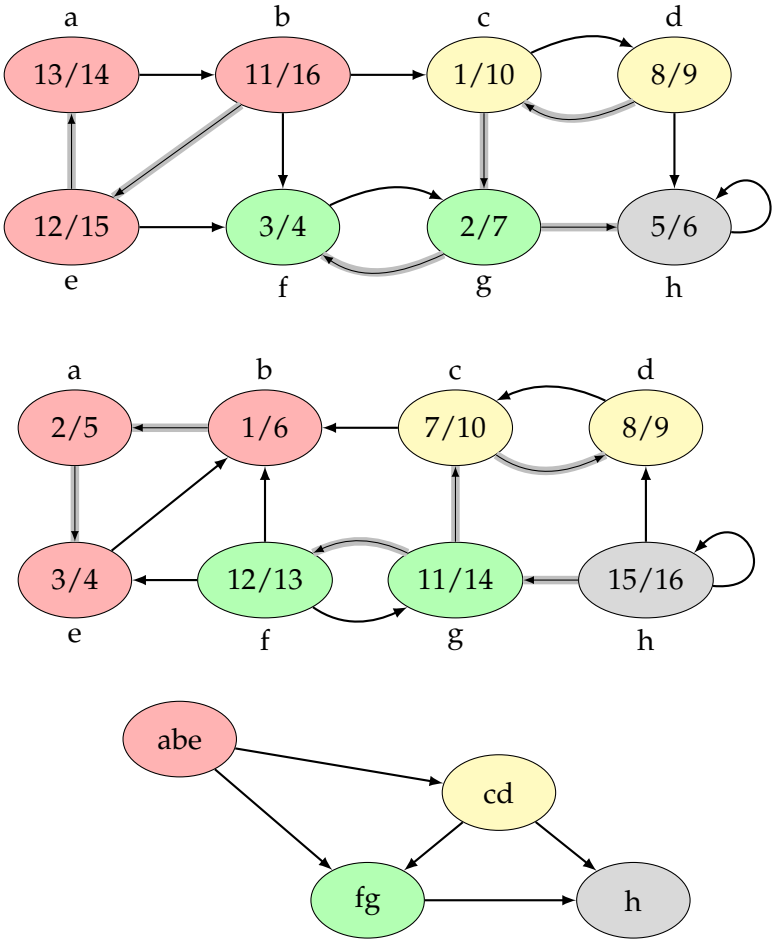


Figura 12.10: Un grafo orientato G. Le componenti fortemente connesse di G sono illustrate in base al colore dei nodi. Ci sono quattro componenti fortemente connesse: $\{a, b, c\}\{c, d\}, \{f, g\}, \{h\}$. Nella seconda immagine il grafo G^T e nella terza il grafo aciclico delle componenti G^{SCC} ottenuto contraendo tutti gli archi all'interno di ciascuna componente fortemente connessa di G in modo che resti un solo vertice in ciascuna componente.

Lemma 12.5.0.1. Siano C e C' due componenti fortemente connesse distinte nel grafo orientato G = (V, E). Se $u, v \in C$ e $u', v' \in C'$ e supponendo che ci sia un cammino $u \rightsquigarrow u'$ in G, allora non può esistere anche un cammino $v' \rightsquigarrow v$ in G

Se esiste un cammino $v' \rightsquigarrow v$ in G, allora esistono cammini $u' \rightsquigarrow v'$ e $v' \rightsquigarrow v \rightsquigarrow u$ in G. Quindi u e v' sono raggiungibili l'uno dall'altro, contraddicendo così le ipotesi che C e C′ siano componenti fortemente connesse e distinte. Estendiamo la notazione dei tempi di scoperta e di completamento agli insiemi dei vertici. Se $U \subseteq V$, allora definiamo $d(U) = \min_{u \in U}(d[u])$ e $f(U) = \max_{u \in U}(f[u])$. Ovvero $d(U)$ e $f(U)$ sono relativamente il primo tempo di scoperta e l'ultimo tempo di completamento di un vertice qualsiasi u.

Lemma 12.5.0.2. Siano C e C′ delle componenti fortemente connesse e distinte nel grafo orientato $G = (V, E)$. Supponiamo che esista un arco $(u, v) \in E$, dove $u \in C$ e $v \in C'$. Allora $f(C) > f(C')$.

Dimostrazione Ci sono due casi, a seconda di quale componente fortemente connessa, C o C′, contiene il primo vertice che viene scoperto durante la visita in profondità. Se $d(C) < d(C')$, indichiamo con x il primo vertice scoperto in C. Al tempo $d[x]$ tutti i vertici in C e C′ sono bianchi. Esiste un cammino in G da x a ciascun vertice in C che è formato da soltanto vertici bianchi. Poiché $(u, v) \in E$, per un vertice qualsiasi $w \in C'$, al tempo $d[x]$ esiste un cammino da x a w in G che è formato soltanto da vertici bianchi: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. Per il teorema del cammino bianco, tutti i vertici in C e C′ diventano discendenti di x nell'albero DF. Quindi per il corollario 12.0.1 $f[x] = f(C) > f(C')$.

Se, invece $d(C) > d(C')$, indichiamo con y il primo vertice scoperto in C'. Al tempo $d[y]$ tutti i vertici in C′ sono bianchi ed esiste un cammino in G da y a ciascun vertice in C′ che è formato soltanto da vertici bianchi. Per il teorema del cammino bianco, tutti i vertici in C′ diventano discendenti di y nell'albero DF e, per il corollario 12.0.1 $f[y] = f(C')$. Al tempo $d[y]$, tutti i vertici in C sono bianchi. Poiché esiste un arco (u, v) da C a C′. Per il Lemma 12.2.1 implica che non può esistere un cammino da C′ a C. Pertanto nessun vertice in C è raggiungibile da y. Al tempo $f[y]$, quindi, tutti i vertici in C sono ancora bianchi. Dunque, per un vertice qualsiasi $w \in C$, $f[w] > f[y]$, e questo implica che $f(C) > f(C')$.

Corollario 12.5.0.1. Siano C e C′ delle componenti fortemente connesse e distinte nel grafo orientato $G = (V, E)$. Supponiamo che esista un arco $(u, v) \in E^T$, dove $u \in C$ e $v \in C'$. Allora $f(C) < f(C')$

Dimostrazione Poiché $(u, v) \in E^T$, si ha $(v, u) \in E$. Poiché le componenti fortemente connesse di G e G^T sono le stesse. Il lemma 12.2.2 implica che $f(C) < f(C')$.

Il corollario 12.2.1 sopra è fondamentale per capire perché funziona la procedura SCC. Esaminiamo cosa accade quando eseguiamo la seconda visita in profondità, che si svolge su G^T. Iniziamo con la componente fortemente connessa C il cui tempo di completamento f(C) è massimo. La visita inizia da qualche vertice $x \in C$ e prosegue su tutti i vertici in C. Per il corollario 12.2.1 non ci sono archi in G^T che vanno da C a un'altra componente fortemente connessa, quindi la visita da x non passerà per i vertici di un'altra componente. Pertanto l'albero radicato in x contiene esattamente i vertici di C. Una volta completata la visita di tutti i vertici in C, l'algoritmo seleziona come radice un vertice di qualche altra componente

fortemente connessa C′, il cui tempo di completamento $f(C')$ è massimo rispetto a tutte le altre componenti, tranne C. Saranno visitati tutti i vertici in C', ma per il corollario 12.2.1 gli unici archi in G^T che vanno da C′ a un'altra componente devono essere diretti verso C, che è già stata visitata. In generale quando la visita in profondità di G^T si svolge su una componente fortemente connessa, tutti gli archi che escono da tale componente devono essere diretti verso componenti già visitate. Ciascun albero DF, quindi, rappresenterà esattamente una sola componente fortemente connessa.

Teorema 12.5.1. La procedura SSC(G) calcola correttamente le componenti fortemente connesse di un grafo orientato G.

Dimostrazione Dimostriamo per induzione sul numero di alberi DF trovati nella visita in profondità di G^T che i vertici di ciascun albero formano una componente fortemente connessa. L'ipotesi induttiva è che i primi k alberi prodotti siano componenti fortemente connesse. Il caso base dell'induzione, quando $k = 0$, è banale. Nel passo induttivo, supponiamo che ciascun dei primi k alberi DF prodotti siano una componente fortemente connessa e prendiamo in considerazione il $(k + 1)$-esimo albero prodotto. Supponiamo inoltre che la radice di questo albero sia il vertice u e che u si trovi nella componente fortemente connessa C. Considerando il modo in cui abbiamo scelto le radici nella visita di profondità abbiamo $f[u] = f(C) > f(C')$ per qualsiasi componente fortemente connessa C′ diversa da C che non è stata ancora visitata. Per l'ipotesi induttiva, nel momento in cui viene visitato il vertice u, tutti gli altri vertici di C sono bianchi. Per il teorema del cammino bianco, tutti gli altri vertici di C sono discendenti del vertice u nel suo albero DF. Inoltre, per l'ipotesi induttiva e per il corollario 12.2.1, tutti gli archi in G^T che escono da C devono essere diretti verso componenti fortemente connesse che sono già state visitate. Quindi, nessun vertice in qualsiasi componente fortemente connessa diversa da C sarà un discendente di u durante la visita in profondità di G^T. Dunque, i vertici dell'albero DF in G^T che è radicato in u formano solamente una sola componente fortemente connessa; questo completa il passo induttivo e la dimostrazione del teorema.

13 Alberi di connessione minimi

Preso un grafo connesso non orientato $G = (V, E)$ dove V è il numero dei nodi ed E è il numero degli archi. Ad ogni arco (u, v) è associato un peso ω che specifica il costo (può essere di qualsiasi natura) per collegare il nodo u al nodo v:

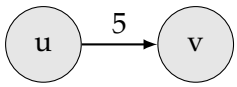


Figura 13.1: $\omega(u, v) = 5$

Vogliamo trovare un sottoinsieme aciclico (senza cicli) $T \subseteq E$ che collega tutti i vertici, il cui peso totale

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

sia **minimo**. Essendo che il sottoinsieme T è aciclico e che collega tutti i vertici, esso forma anche un **albero di connessione**[1] perché *connette* il grafo G. Il problema di trovare l'albero T è detto **problema dell'albero di connessione minimo**[2]

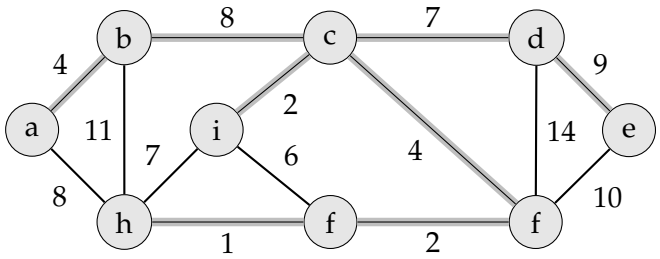


Figura 13.2: Un albero di connessione minimo per un grafo connesso. Gli archi di un albero di connessione minimo sono rappresentati su sfondo grigio. L'albero di connessione **può** non essere unico, ma ce ne possono essere di più con lo stesso peso totale.

Poiché un albero è un tipo di grafo, dobbiamo definire un albero in termini non soltanto dei suoi archi, ma anche dei suoi vertici, anche se in questo capitolo tratteremo gli alberi in base ai loro archi.

[1] Albero che contiene tutti i vertici del grafo e contiene soltanto un sottoinsieme degli archi, cioè solo quelli necessari per connettere tra loro tutti i vertici e con uno e un solo cammino.

[2] Il numero minimo di archi non viene ridotto perché esso sarà sempre dato da $|V| - 1$ archi.

13.1 Creare un albero di connessione minimo

Supponiamo di avere un grafo connesso non orientato $G = (V, E)$ con una funzione peso $\omega : E \to \mathbb{R}$ e di volere trovare un albero di connessione minimo per il grafo G. Il modo di operare è sintetizzato nel seguente "metodo generico", che fa crescere l'albero di connessione minimo di un arco alla volta. Il metodo generico gestisce un in insieme di archi A, conservando la seguente invariante

Prima di ogni iterazione, A è un sottoinsieme di qualche albero di connessione minimo

A ogni passo, determiniamo un arco (u, v) tale che $AU(u, v)$ è anche un sottoinsieme di un albero di connessione minimo. Tale arco è detto **arco sicuro** per A.

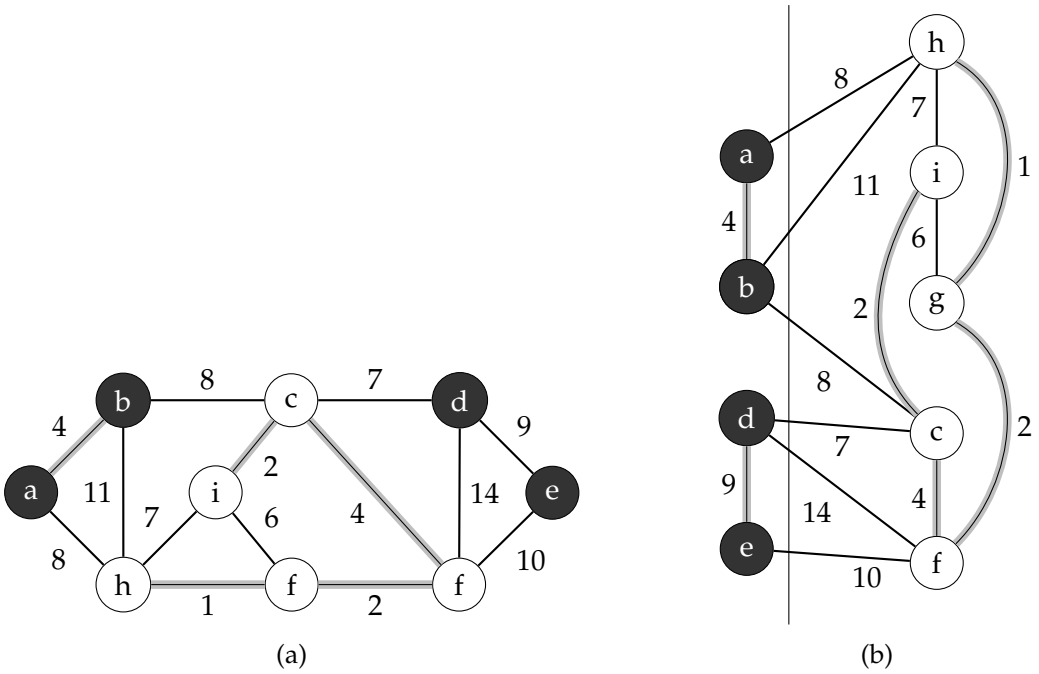


Figura 13.3: Un modo per vedere un taglio $(S, V - S)$ del grafo illustrato in figura (13.2).**(a)** I vertici nell'insieme S sono rappresentati in nero, quelli nell'insieme $V - S$ in bianco. Gli archi che attraversano il taglio sono quelli che collegano i vertici bianchi con i vertici neri. L'arco (d, c) è l'unico arco leggero che attraversa il taglio taglio. Un sottoinsieme A degli archi è rappresentato con uno sfondo grigio; notate che il taglio $(S, V - S)$ rispetta A, perché nessun arco di A attraversa il taglio.

Specifichiamo ora ulteriori definizioni:

- **Taglio**. $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V.
- **Attraversamento**. Si dice che un arco $(u, v) \in E$ **attraversa** il taglio $(S, V - S)$ se una delle sue estremità si trova in S e l'altra in $V - S$.

- Si dice che un taglio **rispetta** un insieme A di archi se nessun arco di A attraversa il taglio.
- **Arco leggero**. Un arco è un arco leggero per un taglio se il suo peso è il minimo fra i pesi degli altri archi che attraversano il taglio.

Teorema 13.1.1. Sia $G = (V, E)$ un grafo connesso non orientato con una funzione peso ω a valori reali definita in E. Sia A un sottoinsieme di E che è contenuto in qualche albero di connessione minimo per G, sia $(S, V - S)$ un taglio qualsiasi di G che rispetta A e sia (u, v) un arco leggero che attraversa $(S, V - S)$. Allora l'arco (u, v) è sicuro per A.

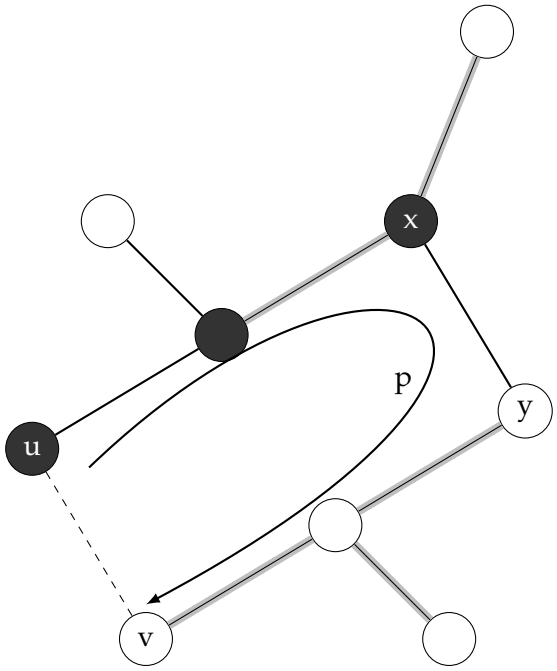


Figura 13.4: I vertici in S sono neri; i vertici in $V - S$ sono bianchi. Sono rappresentati gli archi dell'albero di connessione minimo T. Gli archi in A hanno uno sfondo grigio; (u, v) è un arco leggero che attraversa il taglio $(S, S - V)$. L'arco (x, y) è un arco nell'unico cammino semplice p che va da u a v in T.

Dimostrazione Sia T un albero di connessione minimo che contiene A e supponiamo che T non contenga l'arco leggero (u, v). Costruiremo un altro albero di connessione minimo T' che include $A \bigcup (u, v)$. L'arco (u, v) forma un ciclo con gli archi nel semplice p, raffigurato in (13.4). Poiché u e v si trovano sui lati opposti del taglio $(S, V - S)$, c'è almeno un altro arco in T che appartiene al cammino semplice[3] p e che attraversa il taglio. Sia (x, y) uno di questi archi. L'arco (x, y) non appartiene ad A, perché il taglio rispetta A. Eliminando (x, y), l'albero T si spezza in due componenti. Aggiungendo (u, v) le due componenti si ricongiungono per formare un nuovo albero di connessione $T' = T - (x, y)U(u, v)$. Poiché

[3]Nessun arco è percorso al più una volta.

(u, v) è un arco leggero che attraversa $(S, V - S)$ e anche l'arco (x, y) attraversa questo taglio, allora $\omega(u, v) \leq \omega(x, y)$. Quindi si ha

$$\begin{aligned} \omega(T') &= \omega(T) - \omega(x, y) + \omega(u, v) \\ &\leq \omega(T) \end{aligned}$$

Ma T è un albero di connessione minimo, quindi $\omega(T) \leq \omega(T')$; di conseguenza, anche T' deve essere un albero di connessione minimo. Infine, sappiamo che $A \subseteq T'$, in quanto $A \subseteq T$ e $(x, y) \notin A$; quindi $A \bigcup (u, v) \subseteq T'$. Di conseguenza, poiché T' è un albero di connessione minimo, (u, v) è un arco sicuro per A.

Durante l'esecuzione del metodo per la ricerca del cammino minimo, l'insieme A è sempre aciclico; altrimenti un albero di connessione minimo che include A conterrebbe un ciclo, e ciò sarebbe una contraddizione. Il grafo $G_A = (V, A)$ è una foresta e ciascuna delle componenti connesse di G_A è un albero. Inoltre, qualsiasi arco sicuro (u, v) per A collega componenti distinte di G_A, perché $A \bigcup (u, v)$ deve essere aciclico.

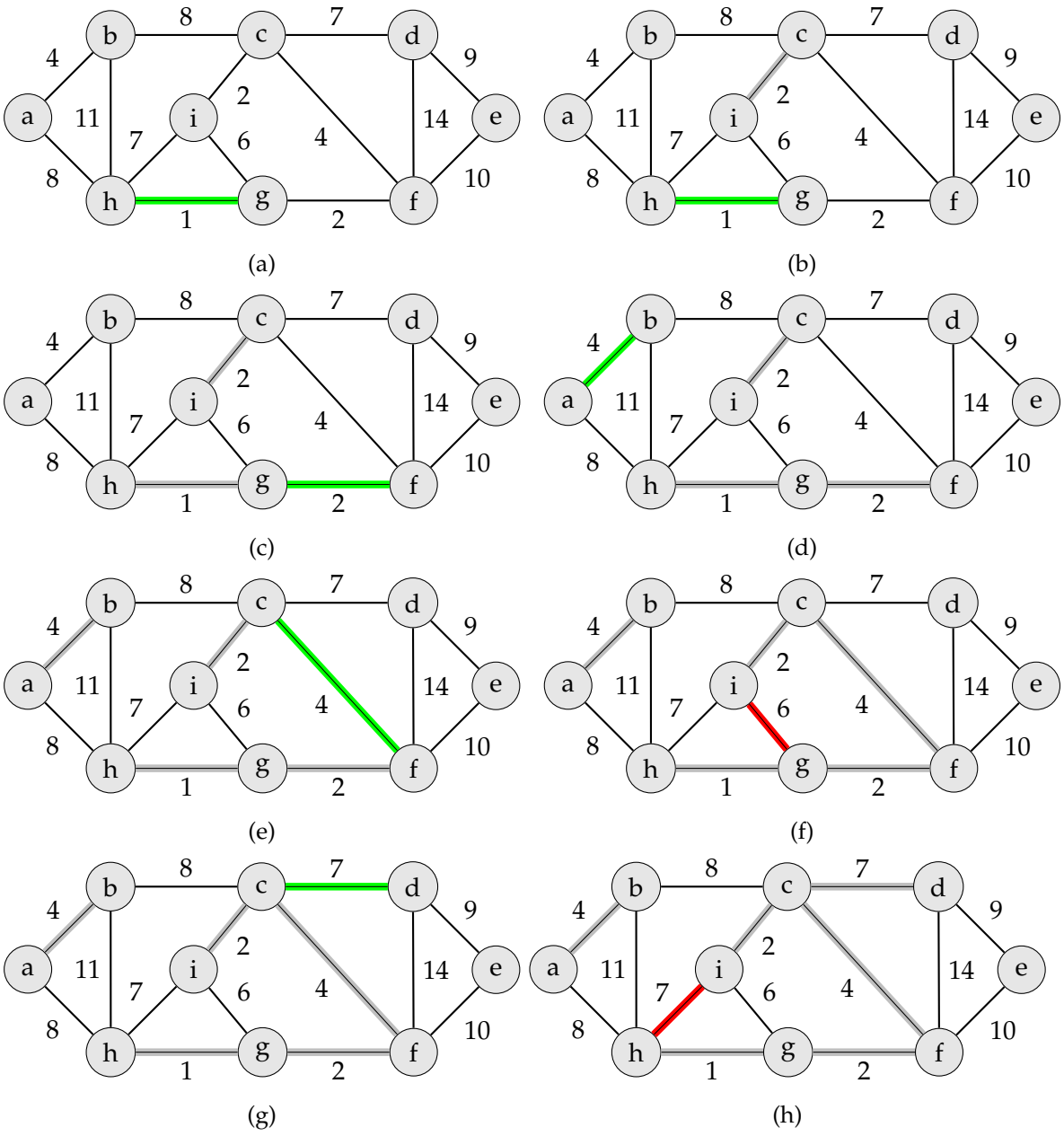
Corollario 13.1.1.1. Sia $G = (V, E)$ un grafo connesso non orientato con una funzione peso ω a valori reali definita in E. Sia A un sottoinsieme di E che è contenuto in qualche albero di connessione minimo per G e sia $C = (V_C, E_C)$ una componente connessa nella foresta $G_A = (V, A)$. Se (u, v) è un arco leggero che collega C a qualche altra componente in G_A, allora (u, v) è sicuro per A.

13.2 Algoritmo di Kruskal

L'algoritmo di Kruskal trova un arco sicuro da aggiungere alla foresta in costruzione scegliendo, fra tutti gli archi che collegano due alberi qualsiasi nella foresta, un arco (u, v) di peso minimo. Consideriamo C_1, C_2 i due alberi che sono collegati dall'arco (u, v). Poiché (u, v) deve essere un arco leggero che collega C_1 a qualche altro albero, per il Corollario 13.1.1.1, l'arco (u, v) è un arco sicuro per C_1. L'idea su cui si basa l'algoritmo è quella di aggiungere alla foresta, un arco con il minor peso possibile. Da come si può capire,Kruskal ha ideato un algoritmo *goloso* che sceglie l'opzione migliore con le informazioni che ha ad ogni passo. Usa una struttura dati per insiemi disgiunti per mantenere vari insiemi disgiunti di elementi; ogni insieme contiene i vertici di un albero della foresta corrente.

```
1  MST–KRUSKAL(G, ω)
2      A = ∅
3      ∀v ∈ G[V]
4          MAKE–SET(v)
5      ordina gli archi di G[E] in senso
6      non decrescente rispetto al peso ω
7      ∀(u, v) ∈ G[E] preso in ordine decrescente
8          if FIND–SET(u) ≠ FIND–SET(v)
9              A = A ⋃ (u, v)
10             UNION(u, v)
11     return A
```

Le righe 2-4 inizializzano l'insieme A come un insieme vuoto e creano $|V|$ alberi, al più uno per vertice. Il ciclo nelle righe 7-10 esamina gli archi dal più leggero al più pesante, per ogni arco (u, v) se i due nodi appartengono allo stesso albero, l'arco (u, v) non può essere aggiunto alla foresta senza generare un ciclo, quindi l'arco viene scartato. Altrimenti i due vertici appartengono ad alberi differenti. In questo caso, l'arco viene aggiunto all'insieme A.



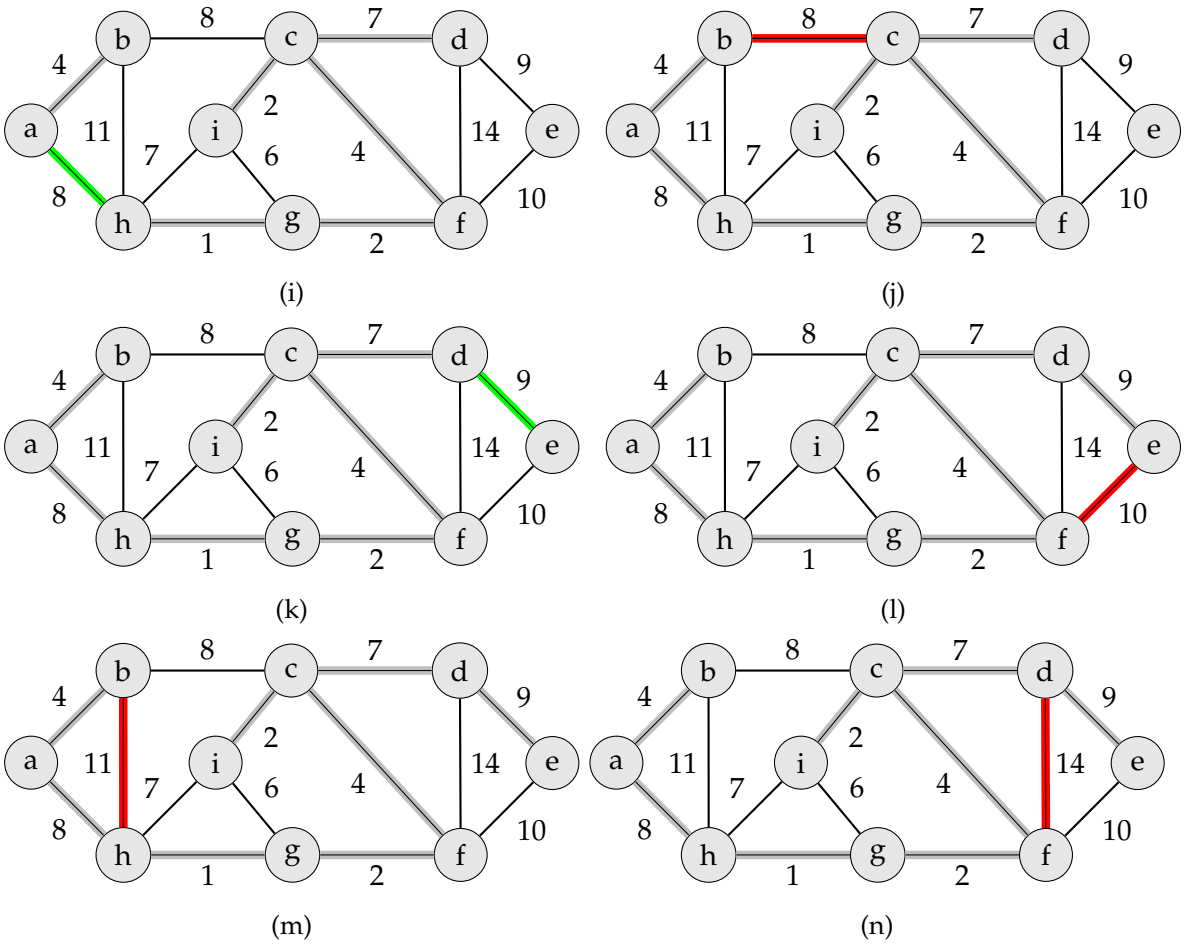


Figura 13.5: L'esecuzione dell'algoritmo di Kruskal con il grafo illustrato in figura 13.2. Gli archi su sfondo grigio appartengono alla foresta A che è in costruzione. Gli archi in verde sono correntemente considerati e collegano due alberi distinti, quelli in rosso identificano un arco che unisce due nodi all'interno dello stesso albero e ciò non può essere.

Il tempo d'esecuzione dell'algoritmo dipende dall'implementazione della struttura dati per gli insiemi disgiunti. Supponendo di utilizzare l'implementazione della foresta di insiemi disgiunti con *unione per rango* e *compressione dei cammini* otterremo la complessità migliore. L'inizializzazione dell'insieme A richiede tempo $O(1)$. Il tempo per ordinare gli archi è $O(E \log E)$. Il ciclo **for** esegue $O(E)$ operazioni FIND-SET e UNION. Tenendo conto anche delle $|V|$ operazioni Make-SET, si ottiene un tempo totale pari a $O((V + E)\alpha(V))$,dove α è al massimo 6. Poiché supponiamo che G sia un grafo connesso, abbiamo che il numero di archi è maggiore del numero di vertici $-1(|E| \geq |V| - 1)$ e quindi le operazioni con gli insiemi disgiunti richiedono un tempo $O(E\alpha(V))$. Inoltre, poichè $\alpha(|V|) = O(\log V) = O(\log E)$,il tempo di esecuzione totale dell'algoritmo è $O(E \log E)$. Osservando che $|E| < |V|^2$, si ha $\log |E| = O(\log V)$; quindi possiamo ridefinire il tempo di esecuzione come $O(E \log V)$.

13.3 Algoritmo di Prim

L'algoritmo di Prim ha la proprietà che gli archi nell'insieme A formano sempre un albero singolo. L'albero inizia da un arbitrario vertice radice r fornito in input e si sviluppa fino a coprire tutti i vertici in V. A ogni passo viene aggiunto all'albero A un arco leggero che collega A con un vertice isolato (vertice che non presente nell'albero A). Per il Corollario 13.1.1.1, questa regola aggiunge soltanto gli archi che sono sicuri per A. Nel seguente pseudocodice, il grafo connesso G e la radice r dell'albero di connessione minimo sono gli input per l'algoritmo. Durante l'esecuzione, tutti i vertici che **non** si trovano nell'albero risiedono in una coda di min-priorità Q basata sul campo *key*. In ogni vertice v, l'attributo $v.key$ è il peso minimo di un arco qualsiasi che collega v a un vertice nell'albero; per convenzione, $v.key = \infty$ se tale arco non esiste. L'attributo $v.\pi$ indica il padre di v nell'albero.

```
1  MST–PRIM(G, ω, r)
2      ∀u ∈ V(G)
3          u.key = ∞
4          u.π = NIL
5      r.key = 0
6      Q = V(G)
7      while Q ≠ ∅
8          u = EXTRACT–MIN(Q)
9          ∀v ∈ Adj[u]
10             if v ∈ Q and ω(u,v) < v.key
11                 v.π = u
12                 v.key = ω(u,v)
```

Le righe $2-6$ inizializzano per ogni vertice u del grafo, il campo $u.key$ ad infinito e $u.\pi$ a nullo, poiché nessun padre è stato ancora scoperto. Successivamente assegnano la chiave $r.key$ a 0, che sarà il primo nodo ad essere elaborato. Infine inizializza la coda di min-priorità Q. Ad ogni passo del ciclo **while** viene identificato un vertice $u \in Q$ tale che rappresenti un arco leggero attraversante un taglio $(V, V-Q)$. Dopo aver raccolto il vertice, si cicla in tutti i nodi che gli sono adiacenti nel **for** a righe $9-11$, se il nodo adiacente ad u è dentro alla coda e il peso per andare da u a v è minore della propria chiave $v.key$, si aggiornano i suoi valori. Le prestazioni dell'algoritmo dipendono dal modo in cui viene implementata la coda di min-priorità Q. Se Q è implementata come un min-heap binario, per creare la coda Q utilizziamo BUILD-MIN-HEAP per eseguire nel tempo O(V) l'inizializzazione delle righe 2-6. Il corpo del ciclo viene eseguito $|V|$ volte e dato che ogni operazione di EXTRACT-MIN richiede un tempo $O(\log v)$, il tempo totale è $O(V \log V)$.Il ciclo **for** (righe 10-12) viene eseguito $O(E)$ volte, in quanto la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$. A riga 12 c'è un DECREASE-KEY sul min-heap, che può essere impiegata in tempo $O(\log V)$. Quindi, il tempo totale dell'algoritmo è $O(V \log V + E \log V) = O(E \log V)$. Se si volesse utilizzare gli heap di Fibonacci, l'operazione di EXTRACT-MIN costerebbe $O(\log V)$ e DECREASE-KEY $O(1)$, quindi il tempo d'esecuzione dell'algoritmo migliora diventando $O(E + V \log V)$.

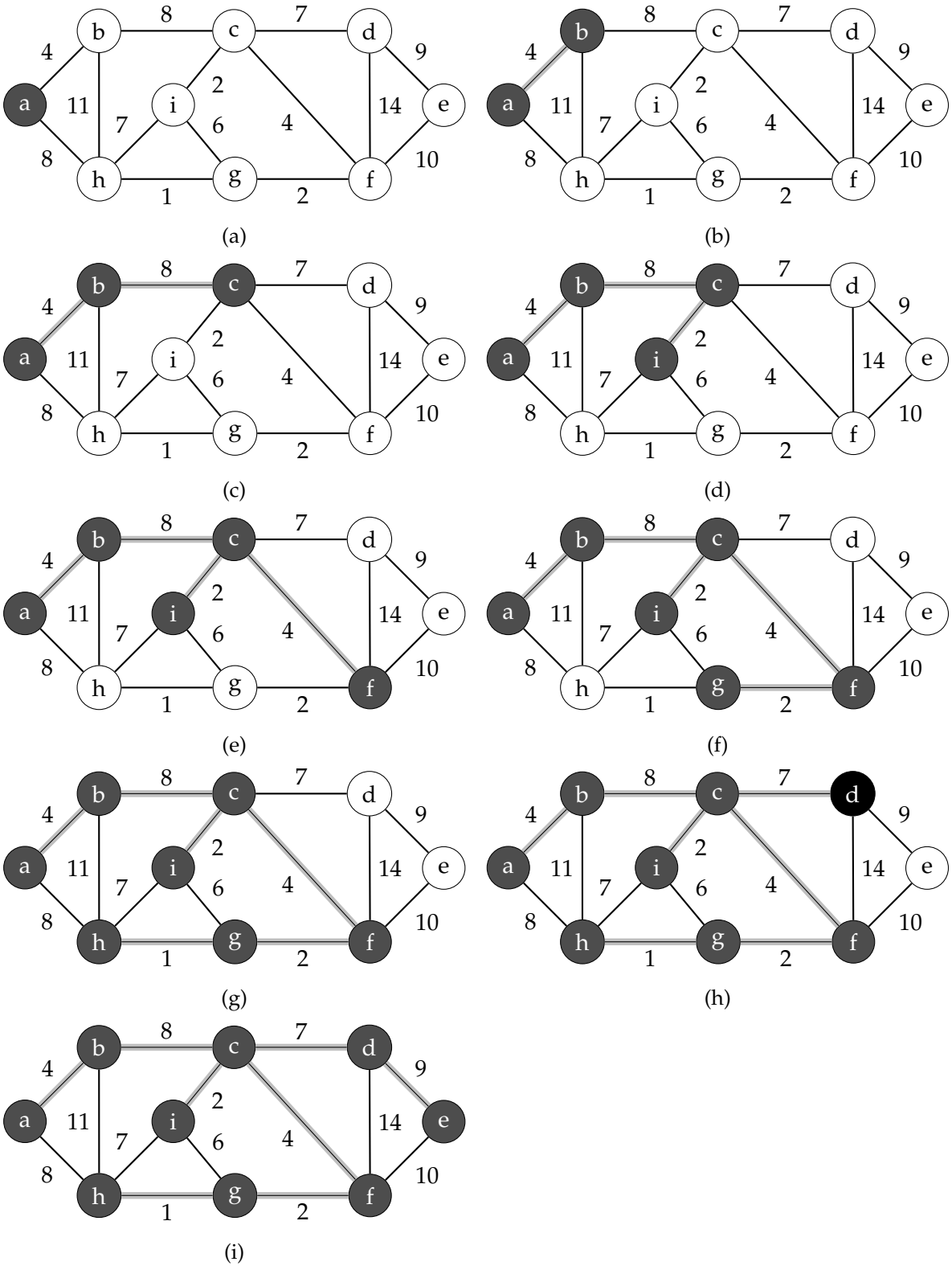


Figura 13.6: Il vertice radice è *a*. Gli archi con sfondo grigio appartengono all'albero in costruzione; i vertici dell'albero sono di colore nero. A ogni passo, i vertici dell'albero determinano un taglio del grafo e un arco leggero che attraversa il taglio viene aggiunto all'albero.

14 Cammini minimi da sorgente unica

14.1 Definizioni e proprietà dei cammini minimi

In un **problema dei cammini minimi** è dato un grafo orientato pesato $G = (V, E)$, con una funzione peso $\omega : E \to \mathbb{R}$ che associa agli archi dei pesi di valore reale. Il **peso** $\omega(p)$ del cammino $p = \langle v_0, V_1, \ldots, v_k \rangle$ è la somma dei pesi degli archi che lo compongono

$$\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

Il **peso del cammino minimo** $\delta(u, v)$ da u a v è definito in questo modo

$$\delta(u, v) = \begin{cases} \min\{\omega(p) : u \overset{p}{\rightsquigarrow} v\} & \text{se esiste un cammino da } u \text{ a } v \\ \infty \end{cases}$$

Un **cammino minimo** dal vertice u al vertice v è definito come un cammino qualsiasi p con peso $\omega(p) = \delta(u, v)$. I pesi degli archi possono essere intepretati come grandezze diverse dalle istanze (*tempi, costi, penalità, perdite*). L'algoritmo BFS è un algoritmo per cammini minimi che opera con i grafi non pesati, cioè i grafi in cui ogni arco può essere considerato di peso unitario.

In questo capitolo approfondiremo l'analisi del **problema dei cammini minimi da sorgente unica**: dato un grafo $G = (V, E)$, vogliamo trovare un cammino minimo che va da un dato vertice **sorgente** $s \in V$ a ciascun vertice $v \in V$.

Lemma 14.1.0.1 (I sottocamini di cammini minimi sono cammini minimi). Dato un grafo orientato pesato $G = (V, E)$ con la funzione peso $\omega : E \to \mathbb{R}$, sia $p = \langle v_0, v_1, \ldots, v_k \rangle$ un cammino minimo dal vertice v_0 al vertice v_k e, per qualsiasi i e j tali che $0 \leq i \leq j \leq k$, sia $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ il sottocammino di p dal vertice v_i al vertice v_j. Allora p_{ij} è un cammino minimo da v_i a v_j.

Dimostrazione Se scomponiamo il cammino p in $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, abbiamo $\omega(p) = \omega(p_{0i}) + \omega(p_{ij}) + \omega(p_{jk})$. Supponiamo che esista un cammino p'_{ij} da v_i a v_j con peso minore ($\omega(p'_{ij}) < \omega(p_{ij})$). Allora $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ è un cammino da v_0 a v_k il cui costo è minore di $\omega(p)$, che contraddice l'ipotesi che p sia un cammino minimo da v_0 a v_k.

Archi di peso negativo

Se il grafo $G = (V, E)$ non contiene cicli di peso negativo che sono raggiungibili dalla sorgente s, allora per ogni $v \in V$, il peso del cammino minimo $\delta(u, v)$ resta ben definito, anche se ha un valore negativo. Nessun cammino da s a un vertice del ciclo può essere un cammino

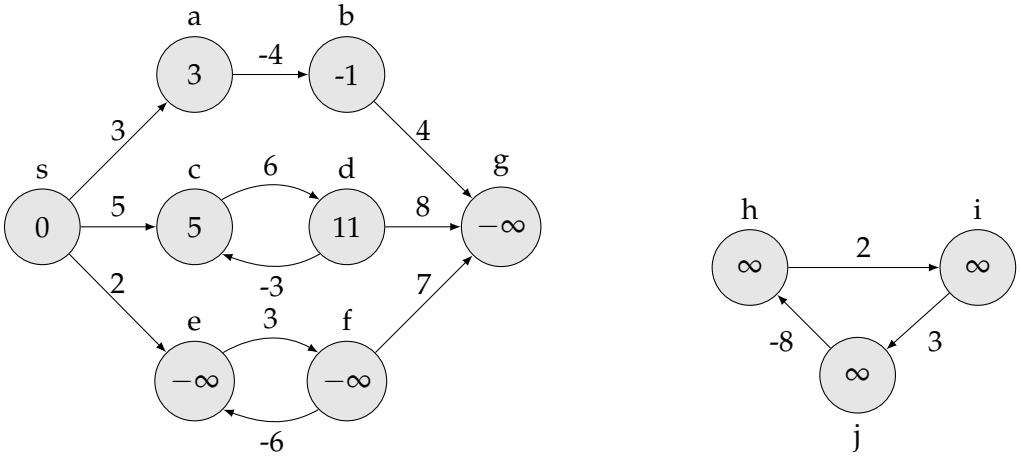


Figura 14.1: Archi con pesi negativi in un grafo orientato. All'interno di ciascun vertice è indicato il peso del suo cammino minimo dalla sorgente s. Poiché i vertici e ed f formano un ciclo di peso negativo che è raggiungibile da s, i pesi dei loro cammini minimi sono $-\infty$. I vertici h, i, j non possono essere raggiunti da s, pertanto i pesi dei loro cammini minimi sono ∞, anche se giacciono su un ciclo di peso negativo.

minimo poiché è sempre possibile trovare un cammino di peso minore che segue il cammino "minimo" proposto e poi attraversa il ciclo di peso negativo. Se esiste un ciclo di peso negativo in qualche cammino da s a v, definiamo $\delta(s, v) = -\infty$. **I cicli**

Un cammino minimo non può contenere neanche un ciclo di peso positivo, perché eliminando il ciclo dal cammino si ottiene un cammino con la stessa sorgente, la stessa destinazione e un peso più piccolo. Ovvero, se $p = \langle v_0, v_1, \ldots, v_l \rangle$ è un cammino e $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ è un ciclo di peso positivo in questo cammino ($v_i = v_j$ e $\omega(c) > 0$), allora il cammino $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_k \rangle$ ha peso $\omega(p') = \omega(p) - \omega(c) < \omega(p)$ e quindi p non può essere un cammino minimo da v_0 a v_k. Poiché un cammino aciclico qualsiasi in un grafo $G = (V, E)$ contiene al più $|V|$ vertici distinti, esso contiene al più $|V| - 1$ archi. Quindi possiamo limitare la nostra analisi ai cammini minimi che hanno al più $|V| - 1$ archi.

Rappresentazione dei cammini minimi

Dato un grafo $G = (V, E)$, manteniamo per ogni vertice $v \in V$ un **predecessore** $v.\pi$ che può essere un altro vertice o il valore NIL. Come nella visita in ampiezza, saremo interessati al **sottografo dei predecessori** $G_\pi = (V_\pi, E_\pi)$. Definiamo l'insieme dei vertici V_π come l'insieme dei vertici di G con predecessori non NIL, più la sorgente s:

$$V_\pi = \{v \in V : v.\pi \neq NIL\} \bigcup s$$

L'insieme degli archi orientati E_π è l'insieme degli archi per i vertici in V_π:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$$

I valori π prodotti dagli algoritmi di questo capitolo hanno la proprietà che al termine della loro esecuzione, G_π è un **albero di cammini minimi**, cioè un albero radicato che contiene

un cammino minimo dalla sorgente s a ogni vertice che è raggiungibile da s. Un albero di cammini minimi radicato in s è un sottografo orientato $G' = (V', E')$, dove $V' \subseteq V$ e $E' \subseteq E$, tale che

1. V' è l'insieme dei vertici raggiungibili da s in G
2. G' forma un albero con radice s
3. per ogni $v \in V'$, l'unico cammino semplice da s a v in G' è un cammino minimo da s a v in G

I cammini minimi non sono necessariamente unici nè lo sono gli alberi dei cammini minimi.

Rilassamento

Per ogni vertice $v \in V$, manteniamo un attributo $v.d$ che è un limite superiore per il peso di un cammino minimo dalla sorgente s a v. La seguente procedura con tempo $\Theta(V)$ inizializza le stme dei cammini minimi e dei predecessori.

```
1 INITALIZE–SINGLE–SOURCE(G, s)
2       ∀v ∈ V(G)
3             v.d = ∞
4             v.π = NIL
5       s.d = 0
```

Infine enunciamo il processo di **rilassamento** di un arco (u, v) che verifica se passando per il vertice u, è possibile migliorare il cammino minimo per v precedentemente trovato e, in caso affermativo, nell'aggiornare $v.d$ e $v.\pi$.

```
1 RELAX(u , v ,ω)
2       if  v.d > u.d + ω(u, v)
3             v.d = u.d + ω(u, v)
4             v.π = u
```

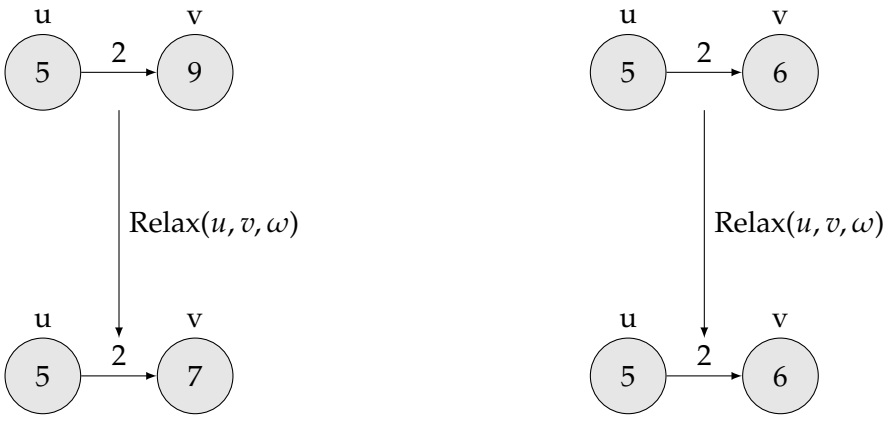


Figura 14.2: Rilassamento di un arco (u, v) con peso $\omega(u, v) = 2$. La stima del cammino minimo di ciascun vertice è illustrata all'interno del vertice.

14.2 Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema dei cammini minimi da sorgente unica un grafo pesato $G = (V, E)$ nel caso in cui tutti i pesi degli archi **non sono negativi**, quindi $\omega(u, v) \geq 0$ per ogni arco $(u, v) \in E$. L'algoritmo mantiene un insieme S di vertici i cui pesi finali dei cammini minimi dalla sorgente s sono già stati determinati, inoltre seleziona ripetutamente il vertice $u \in V - S$ con la stima minima del cammino minimo, aggiunge u a S e rilassa tutti gli archi che escono da u.

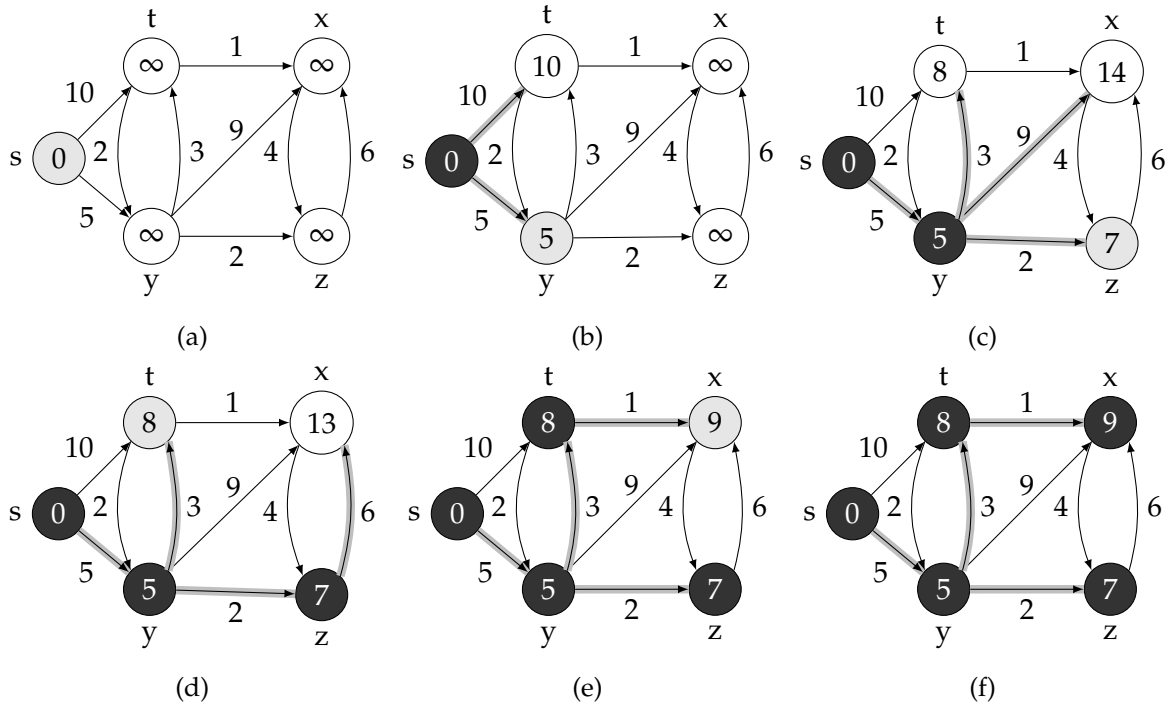


Figura 14.3: L'esecuzione dell'algoritmo di Dijkstra. La sorgente s è il vertice più a sinistra. Le stime dei cammini minimi sono illustrate all'interno dei vertici; gli archi ombreggiati indicano i valori dei predecessori. I vertici neri si trovano nell'insieme S e i vertici bianchi si trovano nella coda di min-priorità $Q = V - S$

```
1 DIJKSTRA(G, ω, s)
2     INITIALIZE–SINGLE–SOURCE(G, s)
3     S = ∅
4     Q = V(G)
5     while Q ≠ ∅
6         u = EXTRACT − MIN(Q)
7         S = S ⋃ {u}
8         ∀v ∈ Adj(u)
9             RELAX(u, v, ω)
```

La riga 2 inizializza i valori d e π; la riga 3 inizializza l'insieme S come l'insieme vuoto. L'algoritmo mantiene l'invariante che $Q = V - S$ all'inizio di ogni iterazione del ciclo **while** delle righe 5-9. Ogni volta che viene eseguito il ciclo, un vertice u viene estratto da $Q = V - S$ e aggiunto all'insieme S. Il vertice u ha la stima minima del cammino minimo di tutti i vertici in $V - S$. Le righe $8 - 9$ rilassano ogni arco (u, v) che esce da u e aggiornano la stima $d[v]$ e il predecessore $\pi[v]$, se il cammino minimo che arriva a v può essere migliorato passando per u. Il ciclo **while** delle righe 5-9 viene ripetuto esattamente $|V|$ volte.

Teorema 14.2.1 (Correttezza dell'algoritmo di Dijkstra). L'algoritmo di Dijkstra, eseguito su un grafo orientato pesato $G = (V, E)$ con funzione peso non negativa ω e sorgente s, termina con $d[u] = \delta(s, u)$ per tutti i vertici $u \in V$.

Dimostrazione Bisogna dimostrare che per ogni vertice $u \in V$, $d[u] = \delta(s, u)$ nell'istante in cui u viene aggiunto all'insieme S.

- **Inizializzazione**: inizialmente, $S = \emptyset$, quindi l'invariante è vera.
- **Conservazione**: vogliamo dimostrare che in ogni iterazione, $d[u] = \delta(s, u)$ per il vertice aggiunto all'insieme S. Supponiamo *per assurdo* che u sia il primo vertice per il quale $d[u] \neq \delta(s, u)$. Deve essere $u \neq s$, perché s è il primo vertice aggiunto all'insieme S e $d[s] = \delta(s, s) = 0$. Poiché $u \neq s$, è anche vero che $S \neq 0$ appena prima che u venga aggiunto a S. Deve esistere qualche cammino da s a u, perché altrimenti $d[u] = \delta(s, u) = \infty$ e questo violerebbe la nostra ipotesi che $d[u] \neq \delta(s, u)$. Poiché esiste almeno un cammino. deve esistere un cammino minimo p da s a u.

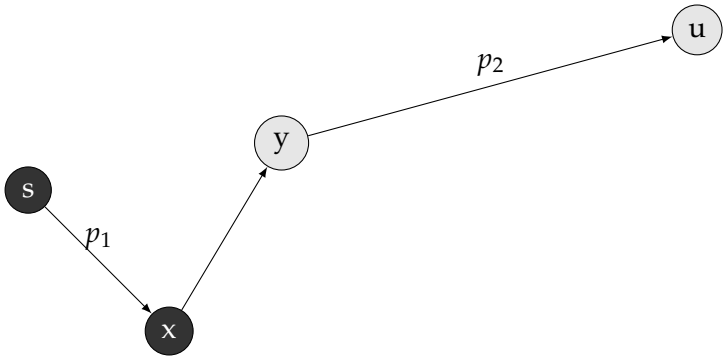


Figura 14.4: L'insieme S, composto dai nodi neri, non è vuoto appena prima che il vertice u sia aggiunti. ad esso. Un cammino minimo p dalla sorgente s al vertice u può essere scomposto in più cammini

Scomponendo il cammino p in percorsi più piccoli che collegano nodi ($p_1 = \langle s, \ldots, x \rangle$, $p_2 = \langle y, u \rangle$). Asseriamo che $d[y] = \delta(s, y)$ quando u viene aggiunto a S. Poiché u è stato scelto come il primo vertice per il quale $d[u] \neq \delta(s, u)$ quando esse viene aggiunto a S, era vero che $d[x] = \delta(s, x)$ quando x è stato aggiunto a S. Poiché y precede u in un cammino minimo da s a u e tutti i pesi degli archi non sono negativi, si ha $\delta(s, y) \leq \delta(s, u)$ e quindi $d[y] \leq d[u]$. Tuttavia, poiché entrambi i vertici u e y si trovano in $V - S$

quando u venne scelto, si ha $d[u] \leq d[y]$, di conseguenza, $d[u] = \delta(s,u)$ che contraddice la nostra scelta di u. Concludiamo che $d[u] = \delta(s,u)$.

- **Conclusione**: Al termine $Q = \emptyset$ che, insieme con la nostra precedente invariante che $Q = V - S$, implica che $S = V$. Dunque $d[u] = \delta(s,u)$ per tutti i vertici $u \in V$.

14.2.1 Analisi

L'operazione INSERT viene chiamata una sola volta per ogni vertice, come EXTRACT-MIN. Poiché ogni vertice $u \in V$ viene aggiunto all'insieme S esattamente una volta, ogni arco nella lista di adiacenza di u, viene esaminato nel ciclo **for** esattamente una volta durante l'esecuzione dell'algoritmo. Poiché il numero totale di archi è $|E|$, ci sono in totale $|E|$ iterazioni di questo ciclo **for**, e quindi l'algoritmo in totale chiama DECREASE-KEY al più $|E|$ volte. Il tempo d'esecuzione dipende dal modo in cui viene implementata la coda di min-priorità:

- **Array**. Ciascuna operazione INSERT e DECREASE-KEY impiega un tempo $O(1)$ e ciascuna operazione EXTRACT-MIN impiega un tempo $O(V)$, per un tempo totale pari a $O(V^2 + E) = O(V^2)$.
- **Min-Heap Binario**. Se il grafo è sufficientemente sparso è utile implementare la coda con un min-heap binario. Ogni operazione EXTRACT-MIN richiede un tempo $O(\log V)$. Ci sono $|V|$ operazioni come questa. Il tempo per costruire un min-heap binario è $O(V)$. Ogni operazione DECREASE-KEY richiede un tempo $O(\log V)$, e ci sono al più $|E|$ operazioni come questa. Il tempo totale di esecuzione è dunque $O((V + E) \log V)$, che è $O(\log V)$.
- **Heap di Fibonacci**. Si può ottenere un tempo di esecuzione pari a $O(V \log V + E)$ implementando la coda di min-priorità con un heap di Fibonacci. Il costo ammortizzato di ciascuna delle $|V|$ operazioni EXTRACT-MIN è $O(\log V)$ e ogni chiamata DECREASE-KEY richiede soltanto un tempo ammortizzato $O(1)$; queste chiamate non sono più di $|E|$.

14.3 L'algoritmo di Bellmann-Ford

Dato un grafo orientato pesato $G = (V, E)$ con sorgente s e funzione peso $\omega : E \to \mathbb{R}$, l'algoritmo di Bellmann-Ford restituisce un valore booleano che indica se esiste un ciclo di peso negativo che è raggiungibile dalla sorgente. Se tale ciclo esiste, il problema non ha soluzione. L'algoritmo usa il rilassamento, riducendo progressivamente il valore stimato $d[v]$ per il peso di un cammino minimo dalla sorgente s a ciascun vertice $v \in V$, fino a raggiungere il peso effettivo $\delta(s,v)$ di un cammino minimo.

```
1 BELLMAN–FORD(G, ω, s)
2     INITIALIZE–SINGLE–SOURCE(G, s)
3     for i ← 1 to |V(G)| − 1
4         ∀(u, v) ∈ E(G)
5             Relax(u, v, ω)
6     ∀(u, v) ∈ E
```

```
7            if  d[u] ≥ d[u] + ω(u, v)
8                RET  FALSE
9        RET  TRUE
```

Lemma 14.3.0.1. Sia $G = (V, E)$ un grafo orientato pesato con sorgente s e funzione peso $\omega : E \to \mathbb{R}$ e supponiamo che G non contenga cicli di peso negativo raggiungibili da s. Dopo $|V| - 1$ iterazioni del primo ciclo **for**, si ha $d[v] = \delta(s, v)$ per tutti i vertici v raggiungibili da s.

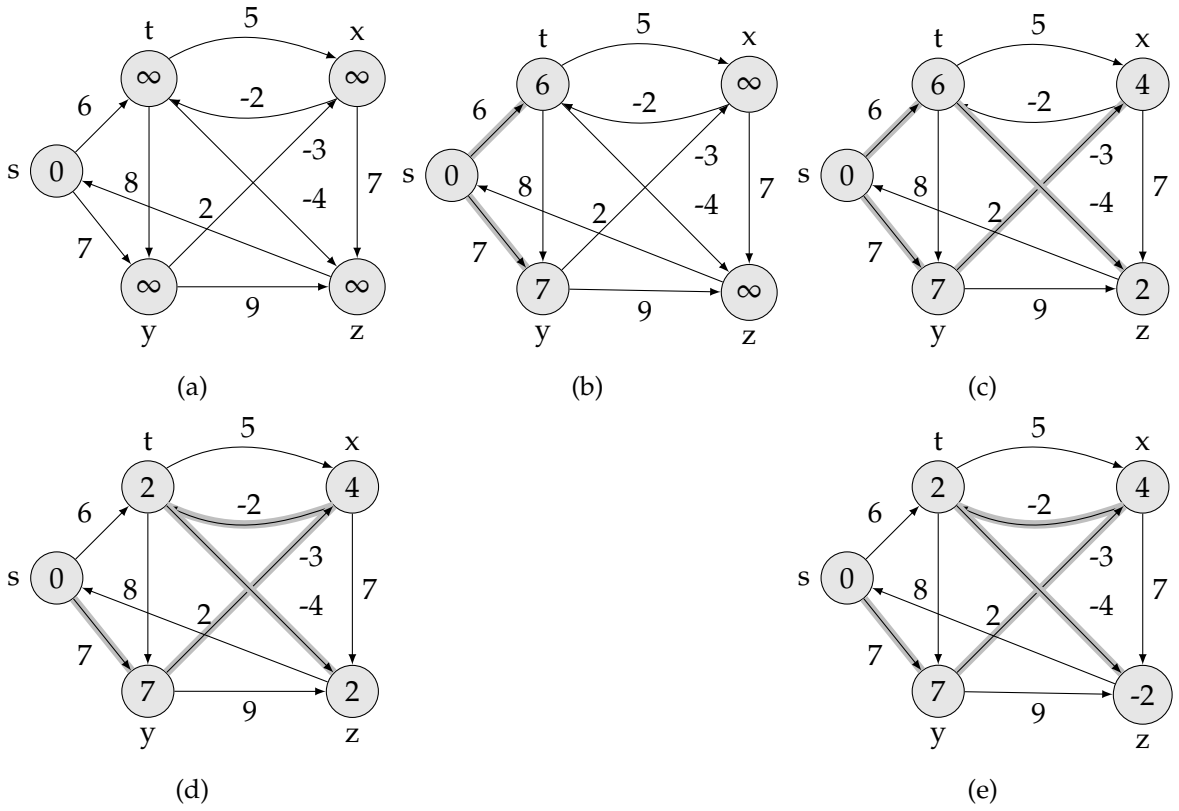


Figura 14.5: L'esecuzione dell'algoritmo di Bellman-Ford. La sorgente è il vertice s. I valori d sono annotati all'interno dei vertici; gli archi ombreggiati indicano i valori dei predecessori: se l'arco (u, v) è ombreggiato, allora $\pi[v] = u$

Corollario 14.3.0.1. L'algoritmo Bellman-Ford viene eseguito nel tempo $O(VE)$. Sia $G = (V, E)$ un grafo orientato pesato con sorgente s e funzione peso $\omega : E \to \mathbb{R}$. Per ogni vertice $v \in V$, esiste un cammino da s a v se e soltanto se l'algoritmo BELLMAN-FORD termina con $d[v] \leq \infty$ quando viene eseguito sul grafo G.

Teorema 14.3.1 (Correttezza dell'algoritmo di Bellman-Ford). Supponiamo di eseguire l'algoritmo di BELLMAN-FORD su un grafo orientato pesato $G = (V, E)$ con sorgente s e funzione peso $\omega : E \to \mathbb{R}$. Se G non contiene cicli di pesi negativo che sono raggiungibili da s, allora l'algoritmo restituisce TRUE, si ha $d[v] = \delta(s, v)$ per tutti i vertici $v \in V$ e il sottografo

dei predecessori G_π è un albero di cammini minimi radicato in s. Se G contiene un ciclo di peso negativo che è raggiungibile da s, allora l'algoritmo restituisce FALSE.

Dimostrazione

Supponiamo che il grafo G non contenga cicli di peso negativo che sono raggiungibili dalla sorgente s. Al termine dell'algoritmo, per tutti gli archi $(u, v) \in E$ si ha

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + \omega(u, v) && \text{(per la disuguaglianza triangolare)} \\ &= d[u] + \omega(u, v) \end{aligned}$$

Supponiamo adesso che il grafo G contenga un ciclo di peso negativo che è raggiungibile dalla sorgente s; indichiamo questo ciclo con $c = \langle v_0, v_1, \ldots, v_k \rangle$, dove v_0 è v_k; allora

$$\sum_{i=1}^{k} \omega(v_{i-1}, v_i) < 0$$

Supponiamo per assurdo che l'algoritmo di Bellman-Ford restituisca TRUE. Quindi $d[v_i] < d[v_{i-1}] + \omega(v_{i-1}, v_i)$ per $i = 1, 2, \ldots, k$. Sommando le disuguaglianze lungo il ciclo c si ottiene

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k} (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} \omega(v_{i-1}, v_i) \end{aligned}$$

Poiché $v_0 = v_k$ ogni vertici di c appare una sola volta in ciascuna delle sommatorie, quindi

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

Pertanto

$$0 \leq \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

che contraddice l'uguaglianza. Concludiamo che l'algoritmo di Bellman-Ford restituisce TRUE se il grafo G non contiene cicli di peso negativo che sono raggiungibili dalla sorgente.

14.4 DAG-SP

Rilassando gli archi di un **DAG**(*Directed acyclic graph*) pesato $G = (V, E)$ secondo un ordine topologico dei suoi vertici, è possibile calcolare i cammini minimi da una sorgente unica nel tempo $\Theta(V + E)$. Nel dag ci possono essere archi di peso negativo, ma non possono esistere cicli di peso negativo. L'algoritmo inizia ordinando topologicamente il dag per imporre un ordinamento lineare ai vertici. Se esiste un cammino dal vertice u al vertice v, allora u precede v nell'ordine topologico. Durante l'elaborazione di un vertice, vengono rilassati tutti gli archi che escono dal vertice.

```
1 DAG-SP(G, ω, s)
2     Usa DFS(G)
3     ∀u ∈ V(G) preso in ordine non decrescente di f
4         ∀v ∈ Adj(u)
5             RELAX(u, v, ω)
```

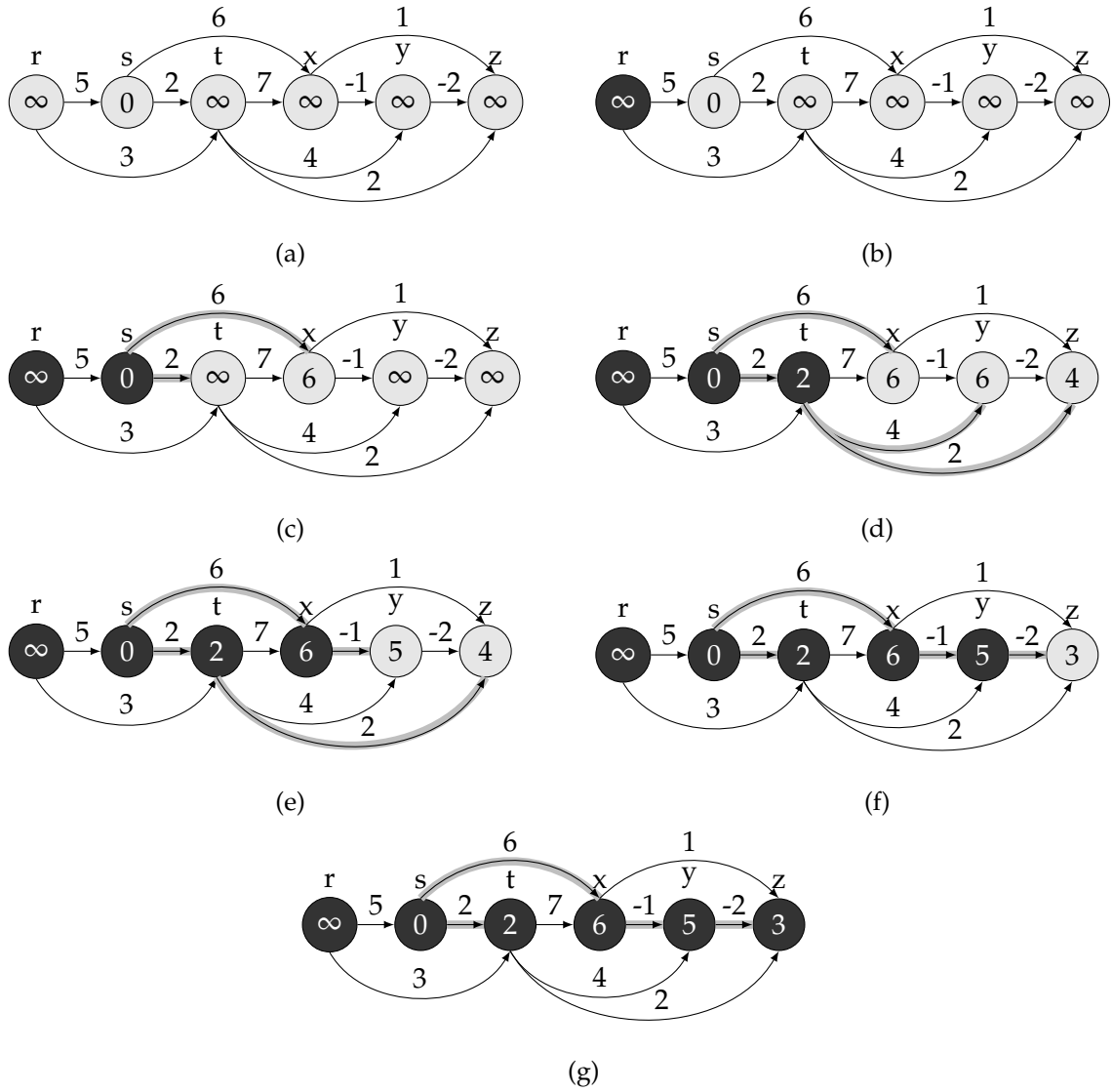


Figura 14.6: L'esecuzione dell'algoritmo per i cammini minimi in un grafo orientato aciclico. I vertici sono in ordine topologico da sinistra a destra. Il vertice sorgente è s. I valori d sono indicati all'interno dei vertici; gli archi ombreggiati indicano i valori π.

L'ordinamento topologico della riga 1 può essere effettuato nel tempo $\Theta(V + E)$. Il ciclo **for** delle righe 3-5 rilassia ciascun arco una sola volta. Poiché ciascuna iterazione del ciclo **for**

interno richiede tempo $\Theta(1)$, il tempo di esecuzione totale è $\Theta(V + E)$, che è lineare nella dimensione di una rappresentazione con liste di adiacenza del grafo.

Teorema 14.4.1. Se un grafo orientato pesato $G = (V, E)$ ha sorgente s e nessun ciclo, allora al termine dell'algoritmo $d[v] = \delta(s, v)$ per tutti i vertici $v \in V$ e il sottografo dei predecessori G_π è un albero di cammini minimi.

Dimostrazione

Se v non è raggiungibile da s, allora $d[v] = \delta(s, v) = \infty$ per la proprietà dell'assenza di un cammino. Se v è raggiungibile da s, allora esiste un cammino minimo $p = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_0 = s$ e $v_k = v$. Poiché i vertici vengono elaborati in ordine topologico, gli archi di p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2) \ldots (v_{k-1}, v_k)$. Per la proprietà del rilassamento del cammino, $d[v_i] = \delta(s, v_i)$ dal termine della procedura. Un **cammino critico** è un *cammino massimo* attraverso il dag, che corrisponde al tempo massimo per eseguire una sequenza ordinata di lavori. Un cammino critico può essere trovato modificando l'algoritmo in uno dei seguenti modi

1. Cambiando il segno dei pesi ed eseguendo DAG-SHORTEST-PATH
2. Eseguendo DAG-SHORTEST-PATH, dopo aver sostituito $>$ con $<$ nella procedura relax.

15 Cammini minimi fra tutte le coppie

15.1 Introduzione

Dato un grafo orientato pesato $G = (V, E)$ con una funzione peso $\omega : E \to \mathbb{R}$ che associa agli archi dei pesi di valore reale, vogliamo trovare, per ogni coppia di vertici $u, v \in V$, un cammino (di peso) minimo da u a v, dove il peso di un cammino è la somma dei pesi degli archi che lo compongono. Per risolvere tale problema si potrebbe eseguire un algoritmo per cammini minimi da sorgente unica $|V|$ volte (una per ogni vertice).

- **Djikstra**. Se non ci sono archi di peso negativo è possibile utilizzare l'algoritmo di Djikstra e otteniamo diverse complessità da come implementiamo la coda:
  1. *array lineare*: $O(V^3 + VE) = O(V^3)$.
  2. *heap binario*: $O(VE \log V)$, che è un miglioramento se il grafo è sparso.
  3. *heap fibonacci*: $O(V^2 \log V + VE)$.
- **Bellman-Ford**. Se il grafo contiene archi di peso negativo.
  1. *Grafo connesso*: $O(V^2E)$.
  2. *Grafo completo*: $O(V^4)$.
  3. *Grafo sparso*: $O(V^3)$.

Per comodità, supponiamo che i vertici abbiano la numerazione $1, 2, \ldots, |V|$; in questo modo, l'input è una matrice W, di dimensione $n \times n$, che rappresenta i pesi degli archi di un grafo orientato $G = (V, E)$ di n vertici. Ovvero $W = (w_{ij})$, dove

$$w_{ij} = \begin{cases} 0 & \text{se } i = \text{j} \\ \text{il peso dell'arco orientato } (i,j) & \text{se } i \neq \text{j e } (i,j) \in E \\ \infty & \text{se } i \neq \text{j e } (i,j) \notin E \end{cases}$$

Sono ammessi gli archi di peso negativo, ma per questi esempi supponiamo che il grafo di input non contenga cicli di peso negativo. L'output è una matrice $D = (d_{ij})$, di dimensione $n \times n$, dove l'elemento d_{ij} contiene il peso di un cammino minimo dal vertice i al vertice j, ovvero $\delta(i, j) = d_{ij}$. Consideriamo inoltre la **matrice dei predecessori** $\Pi = (\pi_{ij})$, dove π_{ij}

$$\pi_{ij} = \begin{cases} NIL & \text{se } i = j \\ NIL & \text{se non esiste cammino da i a j} \\ \text{predecessore di j in qualche cammino minimo da i} & \text{se esiste cammino da i a j} \end{cases}$$

Definiamo G_π come il sottografo dei predecessori, è un albero di cammini minimi per un dato vertice sorgente, la riga $i-esima$ della matrice Π sarà un albero di cammini minimi con radice i. Adottiamo la convenzione di denominare le matrici con le lettere maiuscole, e i loro elementi con lettere minuscole con pedici, per indicare la riga e la colonna. Ad esempio w_{ij} è un elmento a riga i, colonna j della matrice W. Per indicare le iterazioni, alcune matrici hanno gli apici fra parentesi, come $L^{(m)} = \left(l_{ij}^{(m)}\right)$

15.2 Cammini minimi e moltiplicazioni di matrici

15.2.1 Struttura di un cammino minimo

Abbiamo dimostrato che tutti i sottocammini di un cammino minimo sono cammini minimi. Supponiamo che il grafo sia rappresentato da una matrice di adiacenza $W = (w_{ij})$. Consideriamo un cammino minimo p dal vertice i al vertice j e supponiamo che p contenga al più m archi. Se $i = j$, allora p ha peso 0, se $i \neq j$, scomponiamo il cammino p in $i \overset{p'}{\rightsquigarrow} k \to j$, dove p' adesso contiene al più $m-1$ archi. Per i Lemmi enunciati, p' è un cammino minimo da i a k, quindi $\delta(i,j) = \delta(i,k) + w_{kj}$.

15.2.2 Una soluzione ricorsiva per il problema dei cammini minimi fra tutte le coppie

Sia $l_{ij}^{(m)}$ il peso minimo di un cammino qualsiasi dal vertice i al vertice j che contiene al più m archi. Quando $m = 0$, esiste un cammino minimo da i a j senza archi, se e soltanto se $i = j$:

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{se } i \neq j \end{cases}$$

Per $m \geq 1$, calcoliamo $l_{ij}^{(m)}$ come il minimo tra $l_{ij}^{(m-1)}$ (*il peso del cammino minimo da i a j formato da al più m - 1 archi*) e il peso minimo di un cammino qualsiasi da i a j che è formato al più da m archi, che si ottiene considerando tutti i possibili predecessori k di j.

$$l_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + \omega_{kj} \right\}$$

Se il grafo non contiene cicli di peso negativo, allora per ogni coppia di vertici i e j per cui $\delta(i,j) < \infty$, c'è un cammino minimo da i a j che è semplice e quindi contiene al più $n-1$ archi.

15.2.3 Calcolo dei pesi minimi

Prendendo come input la matrice $W = (\omega_{ij})$, calcoliamo una serie di matrici $L^{(0)}, L^{(1)}, \ldots, L^{(m)}$, dove $L^{(m)} = \left(l_{ij}^{(m)}\right)$ per $m = 1, 2, \ldots, n-1$. La matrice finale $L^{(n-1)}$ contiene i pesi effettivi dei cammini minimi. Notare che $L^{(1)} = W$.

```
1 EXTEND–SHORTEST–PATHS(L, W)
2     n = L.rows
3     Sia L' = (l_ij^(m)) una nuova matrice n × n
4     for i ← 1 to n
5         for j ← 1 to n
6             l'_ij = ∞
7             for k ← 1 to n
8                 l'_ij = min(l'_ij, l_ik + ω_kj)
9     return L'
```

La procedura aggiunge un arco ai cammini minimi calcolati fino a quel momento. Data la matrice $L^{(m-1)}$ e W, restituisce la matrice $L^{(m)}$. Il tempo d'esecuzione dell'algoritmo è $\Theta(n^3)$ a causa dei tre cicli for annidati. La relazione con la moltiplicazione di matrici è ben visibile; ricordiamo al lettore che il calcolo del costo di una moltiplicazione tra matrici è dato dall'equazione

$$c_{ij} = \sum_{i=1}^{n} a_{ik} \cdot b_{kj}$$

Se apportassimo le seguente sostituzioni

$$l^{(m-1)} \to a$$
$$\omega \to b$$
$$l^{(m)} \to c$$
$$\min \to +$$
$$+ \to .$$

E le applicassimo alla procedura EXTENDED-SHORTEST-PATH e sostituiamo ∞ (identità per min) con 0 (l'identità per +) otteniamo la stessa procedura con tempo $\Theta(n^3)$. Ritornando al problema dei cammini minimi fra tutte le coppie, dobbiamo eseguire $n-1$ iterazioni di EXTEND-SHORTEST-PATH:

$$L^{(1)} = L^{(0)} \cdot W = W$$
$$L^{(2)} = L^{(1)} \cdot W = W^2$$
$$L^{(3)} = L^{(2)} \cdot W = W^3$$
$$\dots$$
$$L^{(n-1)} = L^{(n-2)} \cdot W = W^{n-1}$$

La procedura quindi esegue in tempo $\Theta(n^4)$.

15.2.4 Migliorare il tempo di esecuzione

Il nostro obbiettivo non è calcolare tutte le matrici $L^{(m)}$, ma siamo interessati soltanto a $L^{(n-1)}$. Possiamo calcolare $L^{(n-1)}$ con soli $\lceil \log(n-1) \rceil$ prodotti di matrici, calcolando la sequenza

$$\begin{aligned} L^{(1)} &= W = W \\ L^{(2)} &= W^2 = W \cdot W \\ L^{(4)} &= W^4 = W^2 \cdot W^2 \\ L^{(8)} &= W^8 = W^4 \cdot W^4 \\ &\dots \\ L^{(2^{\lceil \log(n-1) \rceil})} &= W^{(2^{\lceil \log(n-1) \rceil})} = W^{(2^{\lceil \log(n-1) \rceil - 1})} \cdot W^{(2^{\lceil \log(n-1) \rceil - 1})} \end{aligned}$$

Poiché $2^{\lceil \log(n-1) \rceil} \geq n-1$, il prodotto finale $L^{(2^{\lceil \log(n-1) \rceil})}$ è uguale a $L^{(n-1)}$. La seguente procedura calcola la precedente sequenza di matrici applicando questa tecnita dell'**elevazione al quadrato ripetuta**.

```
1 FASTER–ALL–PAIRS–SHORTEST–PATH(W)
2     n = W.rows
3     L^(1) = W
4     m = 1
5     while m < n − 1
6         Sia L^(2m) una nuova matrice n × n
7         L^(2m) = EXTENDED–SHORTEST–PATHS(L^(m), L^(m))
8         m = 2m
9     return L^(m)
```

In ogni iterazione del ciclo **while** righe 5-8 calcoliamo $L^{(2m)} = \left(L^{(m)}\right)^2$, iniziando con $m = 1$. Alla fine di ogni iterazione , raddoppiamo il valore di m. L'ultima iterazione calcola $L^{(n-1)}$. Il tempo d'esecuzione dell'algoritmo è $\Theta(n^3 \cdot \log n)$ perché ciascuno dei $\lceil \log(n-1) \rceil$ prodotti di matrice richiede un tempo $\Theta(n^3)$

Algoritmo di Floyd Warshall

L'algoritmo considera i vertici "intermedi" di un cammino minimo, dove un **vertice intermedio** di un cammino semplice $p = \langle v_1, v_2, \dots, v_l \rangle$ è un vertice qualsiasi di p diverso da v_1 e v_l, ovvero un vertice qualsiasi dell'insieme $\{v_2, v_3, \dots, v_{l-1}\}$. Supponendo che i vertici di G siano $V = \{1, 2, \dots, n\}$, consideriamo un sottoinsieme $\{1, 2, \dots, k\}$ di vertici per un generico k. Per una coppia qualsiasi di vertici $i, j \in V$, consideriamo tutti i cammini da i a j i cui vertici intermedi sono tutti in $\{1, 2, \dots, k\}$ e sia p un cammino di peso minimo fra di essi (il cammino p è semplice). Da qui abbiamo due casi:

- Se k non è un vertice intermedio del cammino p, tutti i vertici intermedi del cammino p sono nell'insieme $\{1, 2, \dots, k-1\}$. Quindi, un cammino minimo dal vertice i al vertice

j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$. è anche un cammino minimo da i a j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$.

- Se k è un vertice intermedio del cammino p, allora spezziamo p in $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$, p_1 è un cammino minimo da i a k con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$. Poiché il vertice k non è un vertice intermedio del cammino p_1, allora p_1 è un cammino minimo da i a k con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$. Analogamente p_2 è un cammino minimo dal vertice k al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$.

Sia $d_{ij}^{(k)}$ il peso di un cammino minimo dal vertice i al vertice j, i cui vertici intermedi si trovano tutti nell'insieme $\{1, 2, \ldots, k\}$. Se $k = 0$, cammino dal vertice i al vertice j, che non include vertici intermedi con numerazione maggiore di 0. Tale cammino ha al massimo un arco, e quindi $d^{(0)} = \omega_{ij}$. Una definizione ricorsiva al problema è

$$d_{ij}^{(k)} = \begin{cases} \omega_{ij} & \text{se } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{se } k \geq 1 \end{cases}$$

Poiché per un cammino qualsiasi tutti i vertici intermedi si trovano nell'insieme $\{1, 2, \ldots, n\}$, la matrice $D^{(n)} = \left(d_{ij}^{(n)}\right)$ fornisce la solzuione finale: $d_{ij}^{(n)} = \delta(i, j)$ per ogni $i, v \in V$.

15.2.5 Calcolo dei pesi dei cammini minimi secondo lo schema bottom-up

L'input è una matrice W di dimensione $n \times n$ che contiene i pesi degli archi. La procedura restituisce la matrice $D^{(n)}$ dei pesi dei cammini minimi.

```
1 FLOYD–WARSHALL(W)
2     n ← W.rows
3     D^(0) ← W
4     for k ← 1 to n
5         Sia D^(k) ← (d_ij^(k)) una nuova matrice n × n
6         for i ← 1 to n
7             for j ← 1 to n
8                 d_ij^(k) = min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
9     return D^(n)
```

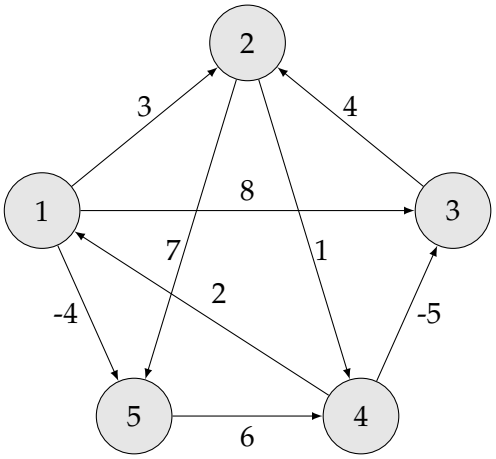


Figura 15.1: Un grafo orientato su cui calcoliamo l'algoritmo Floyd-Warshall

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

15.3 Algoritmo di Johnson per i grafi sparsi

L'algoritmo di Johnson trova i cammini minimi fra tutte le coppie nel tempo $O(V^2 \log V + VE)$ e restituisce una matrice di pesi di cammini minimi per tutte le coppie di vertici oppure segnala la prezenza di un ciclo negativo nel grafo di input. L'algoritmo di Johnson usa la tecnica del **ricalcolo dei pesi** e opera in un grafo $G = (V, E)$ nel seguente modo:

- **archi non negativi**: possiamo trovare cammini minimi fra tutte le coppie di vertici eseguendo l'algoritmo di Dijkstra una volta per ogni vertice; se la coda di min-priorità è implementata con un heap di Fibonacci, il tempo di esecuzione di questo algoritmo per tutte le coppie è $O(V^2 \log V + VE)$.

- **archi negativi, ma privo di cicli di peso negativo**: calcoliamo semplicemente un nuovo insieme di pesi di archi non negativi che ci consente di utilizzare lo stesso metodo.

Il nuovo insieme di pesi $\widehat{\omega}$ deve soddisfare almeno due importanti proprietà:

1. Per ogni coppia di vertici $u, v \in V$, un cammino p è un cammino minimo da u a v con la funzione peso ω, se e soltanto se p è anche un cammino minimo da u a v con la funzione peso $\widehat{\omega}$.
2. Per ogni arco (u, v), il nuovo peso $\widehat{\omega}(u, v)$ non è negativo.

L'elaborazione preliminare di G per determinare la nuova funzione peso $\widehat{\omega}$ può essere eseguita nel tempo $O(VE)$.

Lemma 15.3.0.1 (Il ricalcolo dei pesi non cambia i cammini minimi). Dato un grafo pesato $G = (V, E)$ con funzione peso $\omega : E \mapsto \mathbb{R}$, sia $h : V \mapsto \mathbb{R}$ una funzione qualsiasi che associa i vertici a numeri reali. Per ogni arco $(u, v) \in E$, definiamo

$$\widehat{\omega}(u, v) = \omega(u, v) + h(u) - h(v) \tag{15.1}$$

Sia $p = \langle v_0, v_1, \ldots, v_k \rangle$ un cammino qualsiasi dal vertice v_0 al vertice v_k. Allora p èun cammino minimo da v_0 a v_k con la funzione peso ω, se e soltanto se p è anche un cammino minimo con la funzione peso $\widehat{\omega}$. Ovvero $\omega(p) = \widehat{\omega}(p)$. Inoltre, G ha un ciclo di peso negativo con funzione peso ω, se e soltanto se G ha un ciclo di peso negativo con la funzione $\widehat{\omega}$

Dimostrazione

Iniziamo dimostrando che

$$\widehat{\omega}(p) = \omega(p) + h(v_0) - h(v_k) \tag{15.2}$$

Abbiamo

$$\begin{aligned}\widehat{\omega}(p) &= \sum_{i=1}^{k} \widehat{\omega}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} (\omega(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i))\end{aligned}$$

15.3.1 Produrre pesi non negativi con la tecnica del ricalcolo dei pesi

L'obiettivo successivo è garantire che la seconda proprietà sia valida: vogliamo che il peso $\widehat{\omega}(u, v)$ non sia negativo per ogni arco $(u, v) \in E$. Dato un grafo pesato $G = (V, E)$, creiamo un nuovo grafo $G' = (V', E')$, dove $V' = V \bigcup \{s\}$ per un nuovo vertice $s \notin V$ ed $E' = E \bigcup \{(s, v) : v \in V\}$. Inizializziamo $\omega(s, v) = 0$, per ogni vertice in V. Inoltre G' non ha cicli di peso negativo, se e soltanto se G non ha cicli di peso negativo. Adesso supponiamo che G e G' non abbiano cicli di peso negativo. Definiamo $h(v) = \delta(s, v)$ per ogni $v \in V$, abbiamo per la disuguaglianza triangolare che $h(v) \leq h(u) + \omega(u, v)$ per tutti gli archi $(u, v) \in E'$. Se definiamo i nuovi pesi $\widehat{\omega}$ secondo l'equazione (15.1), abbiamo $\widehat{\omega}(u, v) = \omega(u, v) + h(u) - h(v) \geq 0$.

15.3.2 Calcolo dei cammini minimi fra tutte le coppie

L'algoritmo di Johnson una come subroutine l'algoritmo di Bellman-Ford e l'algoritmo di Dijkstra. Suppone che gli archi siano memorizzati in liste di adiacenza. L'algoritmo restituisce una matrice $D = d_{ij}$, di dimensioni $|V| \times |V|$, dove $d_{ij} = \delta(i, j)$, oppure segnala che il grafo di input contiene un ciclo negativo.

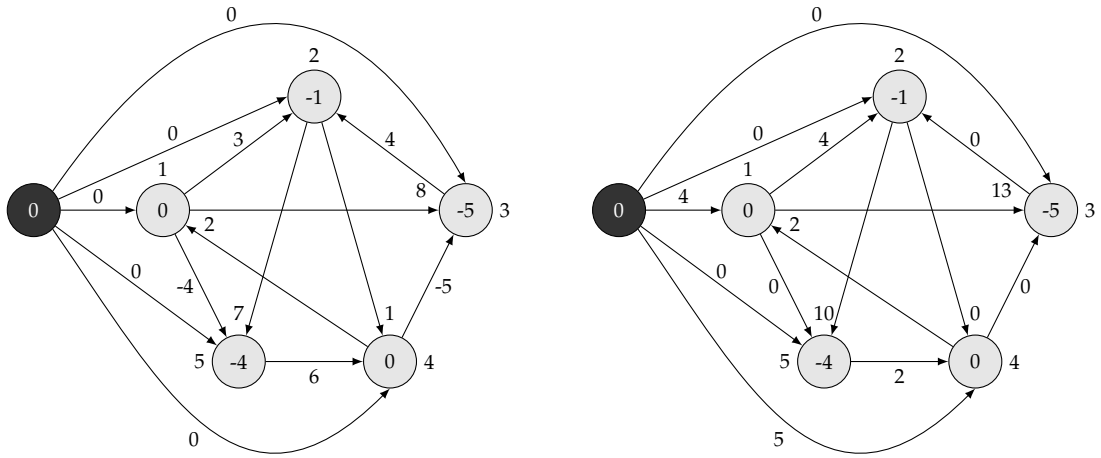


Figura 15.3: L'algoritmo di Johnson per i cammini minimi fra tutte le coppie viene eseguito sul grafo. **(a)** Il grafo G' con la funzione peso originale ω. Il nuovo vertice s è nero. All'interno di ogni vertice v è indicato il valore $h(v) = \delta(s, v)$. **(b)** Il peso di ogni arco (u, v) viene ricalcolato con la funzione peso $\widehat{\omega}(u, v) = \omega(u, v) + h(u) - h(v)$.

Le righe $2 - 4$ produce G', la riga 5 esegue l'algoritmo di Bellman-Ford su G' con la funzione peso ω e il vertice sorgente s. Se G', e quindi G, contiene un ciclo di peso negativo e la riga 6 segnala il problema. Le righe $7 - 14$ assumono che G' non contenga cicli di peso negativo. Le righe $7 - 8$ assegnano a $h(v)$ il peso del cammino minimo $\delta(s, v)$ calcolato dall'algoritmo di Bellman-Ford per ogni vertice $v \in V'$. Le righe $10 - 11$ calcolano i nuovi pesi $\widehat{\omega}$. Per ogni coppia di vertici $u, v \in V$, il ciclo **for** nelle righe $14 - 16$ calcola il peso del cammino minimo $\widehat{\delta}(u, v)$ chiamando l'algoritmo di Dijkstra una volta per ogni vertice in V. La riga 17 memorizza nella posizione d_{uv}, della matrice il peso corretto del cammino minimo $\delta(u, v)$. Infine la riga 18 restituisce la matrice D completa.

```
1 JOHNSON(G, ω)
2       Calcola G', dove G'.V ← G.V ⋃ {s}
3       G'.E ← G.E ⋃ {(s, v) : v ∈ G.V} e
4       ω(s, v) ← 0 ∀v ∈ G.V
```

```
5     if BELLMAN-FORD(G', ω, s) == FALSE
6         "grafo contiene ciclo"
7     else ∀v ∈ G'.V
8         assegna a h(v) il valore δ(s, v)
9             calcolato dall'algoritmo di Bellman-Ford
10        ∀(u, v) ∈ G'.E
11            ω̂ ← ω(u, v) + h(u) − h(v)
12        Sia D ← (d_uv) una nuova matrice n × n
13        ∀u ∈ G.V
14            esegue DJIKSTRA(G, ω̂, u) per calcolare δ̂(u, v)
15            per ogni arco v ∈ G.V
16            ∀v ∈ G.V
17                d_uv ← δ̂(u, v) + h(v) − h(u)
18    return D
```

Se la coda di min-priorità dell'algoritmo di Dijkstra è implementata con un heap di Fibonacci, il tempo di esecuzione dell'algoritmo di Johnson è $O(V^2 \log V + VE)$. L'implementazione più semplice con un min-heap binario genera un tempo di esecuzione di $O(VE \log V)$, che è ancora asintoticamente più veloce dell'algoritmo di Floyd-Warshall se il grafo è sparso.

16 Flusso massimo

16.1 Reti di flusso

Una **rete di flusso** $G = (V, E)$ è un grafo orientato in cui ogni arco $(u, v) \in E$ ha una **capacità** non negativa $c(u, v) \geq 0$. Se l'arco $(u, v) \notin E$ definiamo $c(u, v) = 0$. Distinguiamo due vertici in una rete di flusso: una **sorgente** s e un **pozzo** t, per ogni vertice $v \in V$, esiste un cammino $s \rightsquigarrow v \rightsquigarrow t$ e quindi il graf oè connesso. Sia $G = (V, E)$ una rete di flusso con una funzione capacità c. Sia s la sorgente della rete e sia t il pozzo. Un **flusso** in G è una funzione a valori reali $f : V \times V \to \mathbb{R}$ che soddisfa le seguenti due proprietà:

1. **Vincolo sulle capacità**: per ogni $u, v \in V$, si richiede che $f(u, v) \leq c(u, v)$. Tale vincolo dice semplicmente che il flusso da un vertice all'altro deve essere non negativo e non deve superare la capacità prefissata.
2. **Conservazione del flusso**: per ogni $u \in V - \{s, t\}$ si richiede che

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

   In altre parole, il flusso totale che entra in un vertice, diverso dalla sorgente o dal pozzo, deve essere uguale al flusso totale che esce da quel vertice. (*flusso entrante uguale a quello uscente*).

Infine la quantità $f(u, v)$ è detta **flusso** dal vertice u al vertice v. Il **valore** $|f|$ di un flusso f è definito come

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

ovvero il flusso che esce dalla sorgente meno il flusso che entra nella sorgente.

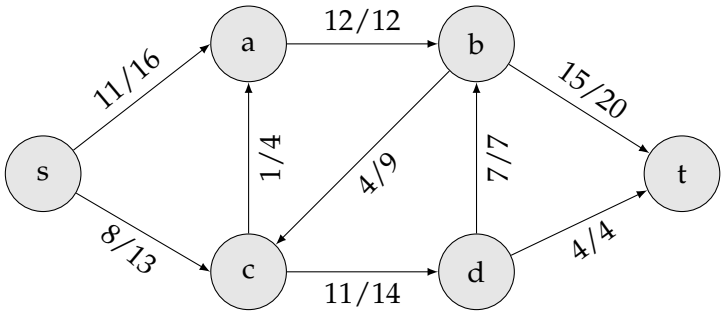


Figura 16.1: Un flusso f in G con valore $|f| = 19$. Ogni arco (u, v) è etichettato con $f(u, v)/c(u, v)$. Il simbolo / è utilizzato semplicemente per separare il flusso dalla capacità; non indica l'operazione di divisione.

16.2 Metodo Ford-Fulkerson

Il metodo Ford-Fulkerson aumenta in modo iterativo il valore del flusso. Iniziamo con $f(u,v) = 0$ per ogni $u, v \in V$, assegnando al flusso iniziale un valore 0. A ogni iterazione, incrementiamo il valore del flusso in G cercando un *cammino aumentante* in una "rete residua". Una volta che conosciamo gli archi di un cammino aumentante in G_f, possiamo facilmente identificare degli archi specifici in G ai quali possiamo modificare il flusso in modo tale da aumentare il valore $|f|$ del flusso. Aumentiamo ripetutamente il flusso finché nella rete residua non ci saranno altri cammini aumentati.

16.2.1 Reti residue

Un arco della rete di flusso può accettare una quantità di flusso aggiuntivo pari alla capacità dell'arco meno il flusso su tale arco. Se tale valore è positivo, inseriamo questo arco in G_f con una *capacità residua* pari a $c_f(u,v) = c(u,v) - f(u,v)$. Gli unici archi di G che si trovano in G_f sono quelli che possono accettare più flusso. Gli archi (u,v) il cui flusso è uguale alla loro capacia hanno $c_f(u,v) = 0$; questi archi non si trovano in G_f. Per rappresentare una possibile riduzione di un flusso positivo $f(u,v)$ in un arco G, poniamo un arco (u,v) in G_f con capacità residua $c_f(v,u) = f(u,v)$ ovvero un arco che può accettare un flusso nella direzione opposta a (u,v), che al più può annullare il flusso in (u,v). Inviare un flusso opposto equivale a *ridurre* il flusso nell'arco. Supponiamo di avere una rete di flusso $G = (V,E)$ con una sorgente s e un pozzo t. Sia f un flusso in G; consideriamo una coppia di vertici $u, v \in V$. Definiamo **capacità residua** $c_f(u,v)$ in questo modo

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{se } (u,v) \in E \\ f(v,u) & \text{se } (v,u) \in E \\ 0 & \text{negli altri casi} \end{cases} \tag{16.1}$$

Come esempio dell'equazione (16.1), se $c(u,v) = 16$ e $f(u,v) = 11$, allora possiamo aumentare $f(u,v)$ fino a $c_f(u,v) = 5$ unità prima di superare il vincolo della capacità nell'arco (u,v). Data una rete di flusso $G = (V,E)$ con un flusso f, la **rete residua** di G indotta da f è $G_f = (V, E_f)$, dove

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\} \tag{16.2}$$

Ogni arco della rete residua, o **arco resiudo**, può ammettere un flusso che è maggiore di 0. Gli archi in E_f o sono archi di E o sono i loro opposti, e quindi

$$|E_f| \leq 2|E|$$

A parte questa differenza, una rete residua ha le stesse proprietà di una rete di flusso, e possiamo definire un flusso nella rete residua come quello che soddisfa la definizione di flusso, ma rispetta alle capacità c_f nella rete G_f.

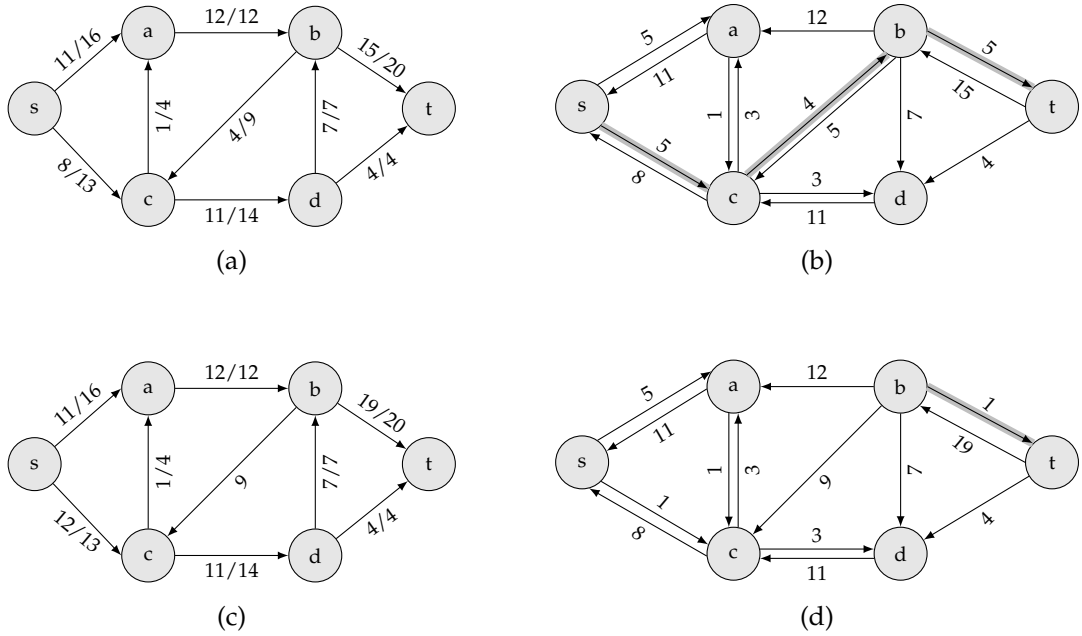


Figura 16.2: **(a)** La rete di flusso G e il flusso f. **(b)** La rete residua G_f con il cammino aumentante p ombreggiato; la sua capacità residua è $c_f(p) = c_f(c, b) = 4$. Gli archi con capacità residua pari a 0, come (a, b), non sono indicati; è questa una convenzione che adotteremo in tutto il paragrafo. **(c)**. Il flusso in G che si ottiene aumentando il flusso lungo il cammino p della sua capacità residua 4. Gli archi senza flusso, come (b, c), sono indicati solo con le loro capacità; questa è un'altra convenzione che adotteremo. **(d)** La rete residua indotta dal flusso in (c).

Se f è un flusso continuo in G ed f' è un flusso nella corrispondente rete residua G_f, definiamo $f \uparrow f'$, l'**aumento** di f' del flusso f, come una funzione $V \times V : \mathbb{R}$

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(u, v) & \text{se } (u, v) \in E \\ 0 & \text{negli altri casi} \end{cases} \tag{16.3}$$

16.2.2 Cammini aumentati

Data una rete di flusso $G = (V, E)$ con un flusso f, un **cammino aumentante** p è un cammino semplice da s a t nella rete residua G_f. Possiamo aumentare il flusso di un arco (u, v) in un cammino aumentante fino a $c_f(u, v)$, senza violare il vincolo sulle capacità. La quantità massima di cui può crescere il flusso in ogni arco di un cammino aumentante p è detta **capacità residua** di p ed è espressa da

$$c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$$

Lemma 16.2.0.1. Sia $G = (V, E)$ una rete di flusso, sia f un flusso in G e sia p un cammino aumentante in G_f. Definiamo una funzione $f_p : V \times V \mapsto \mathbb{R}$ come

$$f_p(u, v) = \begin{cases} c_f(p) & \text{se } (u, v) \in p \\ 0 & \text{negli altri casi} \end{cases} \tag{16.4}$$

Allora f_p è un flusso in G_f con valore $|f_p| = c_f(p) > 0$

Corollario 16.2.0.1. Sia $G = (V, E)$ una rete di flusso, sia f un flusso in G e sia p un cammino aumentante in G_f, supponiamo inoltre di aumentare f di f_p. Allora la funzione $f \uparrow f'$ è un flusso in G con valore $|f \uparrow f'| = |f| + |f_p| > |f|$

16.2.3 Tagli delle reti di flusso

Un **taglio**(S, T) della rete di flusso $G = (V, E)$ è una partizione di V in S e $T = V - S$ tale che $s \in S$ e $t \in T$ (*definizione simile a quella del "taglio" che abbiamo utilizzat oper gli alberi di connessione minimi, con la differenza che qui stiamo tagliando un grafo orientato*). Se f è un flusso, allora il **flusso netto** $f(S, T)$ attraverso il taglio (S, T) è definito come

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

La **capacità** del taglio (S, T) è

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

Un **taglio minimo** di una rete è un taglio la cui capacità è minima rispetto a tutti i tagli della rete. Nel calcolo delle capacità, noi consideriamo soltanto le capacità degli archi che vanno da S a T, ignorando gli archi nella direzione opposta. Per quanto riguarda il flusso, noi consideriamo il flusso che va da S a T meno quello che va nella direzione opposta da T a S.

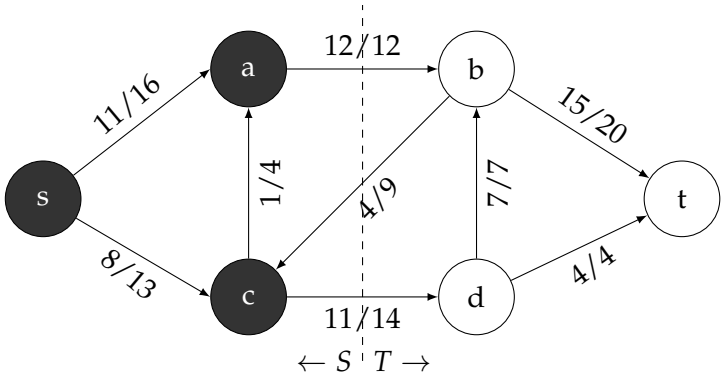


Figura 16.3: Un taglio (S, T) nella rete di flusso dove $S = \{s, a, c\}$ e $T = \{b, d, t\}$. I vertici in S sono neri; i vertici in T sono bianchi. Il flusso netto (S, T) è $f(S, T) = 19(12 + 11 - 4)$ e la capacità è $c(S, T) = 26(12 + 14)$

Lemma 16.2.0.2. Sia f un flusso in una rete di flusso G con sorgente s e pozzo t; sia (S, T) un taglio di G. Allora il flusso netto attraverso (S, T) è $f(S, T) = |f|$

Teorema 16.2.1 (Teorema del flusso massimo/taglio minimo). Se f è un flusso in una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora le seguenti condizioni sono equivalenti:

1. f è un flusso massimo in G.
2. La rete residua G_f non contiene cammini aumentanti.
3. $|f| = c(S, T)$ per qualche taglio (S, T) di G.

Dimostrazione $(1) \rightarrow (2)$: supponiamo per assurdo che f sia un fusso massimo in G; ma che G_f abbia un cammino aumentante p. La somma dei flussi $f + f_p$, dove f_p è il flusso massimo in G_f è un flusso in G con valore strettamente maggiore di $|f|$, contraddicendo l'ipotesi che f sia un flusso massimo.

$(2) \rightarrow (3)$: supponiamo che G_f non abbia cammini aumentanti, ovvero che G_f non contenga un cammino da s a t. Definiamo

$S = \{v \in V$: esiste un cammino da $sav \in G_f\}$

e $T = V - S$. Banalmente si ha $s \in S$ e $t \notin S$ perché non esiste un cammino da s a t in G_f. Consideriamo due vertici $u \in S$ e $v \in T$. Se $(u, v) \in E$ deve essere $f(u, v) = c(u, v)$ altrimenti $(u, v) \in E_f$ e quindi v dovrebbe stare in S. Se $(v, u) \in E$, dobbiamo avere $f(u, v) = 0$, perché altrimenti $c_f(u, v) = f(v, u)$ sarebbe positivo e si avrebbe $(u, v) \in E_f$. Ovviamente, se né $f(u, v)$ né (v, u) si trovano in E, allora $f(u, v) = f(v, u) = 0$; quindi si ha $f(S, T) = c(S, T)$.

16.2.4 Algoritmo di base di Ford-Fulkerson

```
1  FORD–FULKERSON(G, s, t)
2      ∀(u, v) ∈ G.E
3          (u, v).f = 0
4      while esiste un cammino p da s a t in G_f
5          c_f(p) = min{c_f(u, v) : (u, v) ∈ p}
6          ∀(u, v) ∈ p
7              if (u, v) ∈ E
8                  (u, v).f = (u, v).f + c_f(p)
9              else
10                 (v, u).f = (v, u).f − c_f(p)
```

In ogni iterazione del metodo Ford-Fulkerson troviamo *qualche* cammino aumentante p per modificare il flusso f. Sostituiamo f con $f \uparrow f_p$, ottenendo un nuovo flusso il cui valore è $|f| + |f_p|$. L'espressione $c_f(p)$ nel codice è una variabile temporanea che memorizza la capacità residua nel cammino p. Il ciclo **while** delle righe $4 - 10$ trova ripetutamente un cammino aumentante p in G_f e incrementa il flusso f lungo p della capacità residua $c_f(p)$. Le righe $7 - 10$ aggiornano appropriatamente il flusso in ciascun caso, sommando il flusso quando l'arco residuo è un arco originale e sottraendo il flusso nell'altro caso.

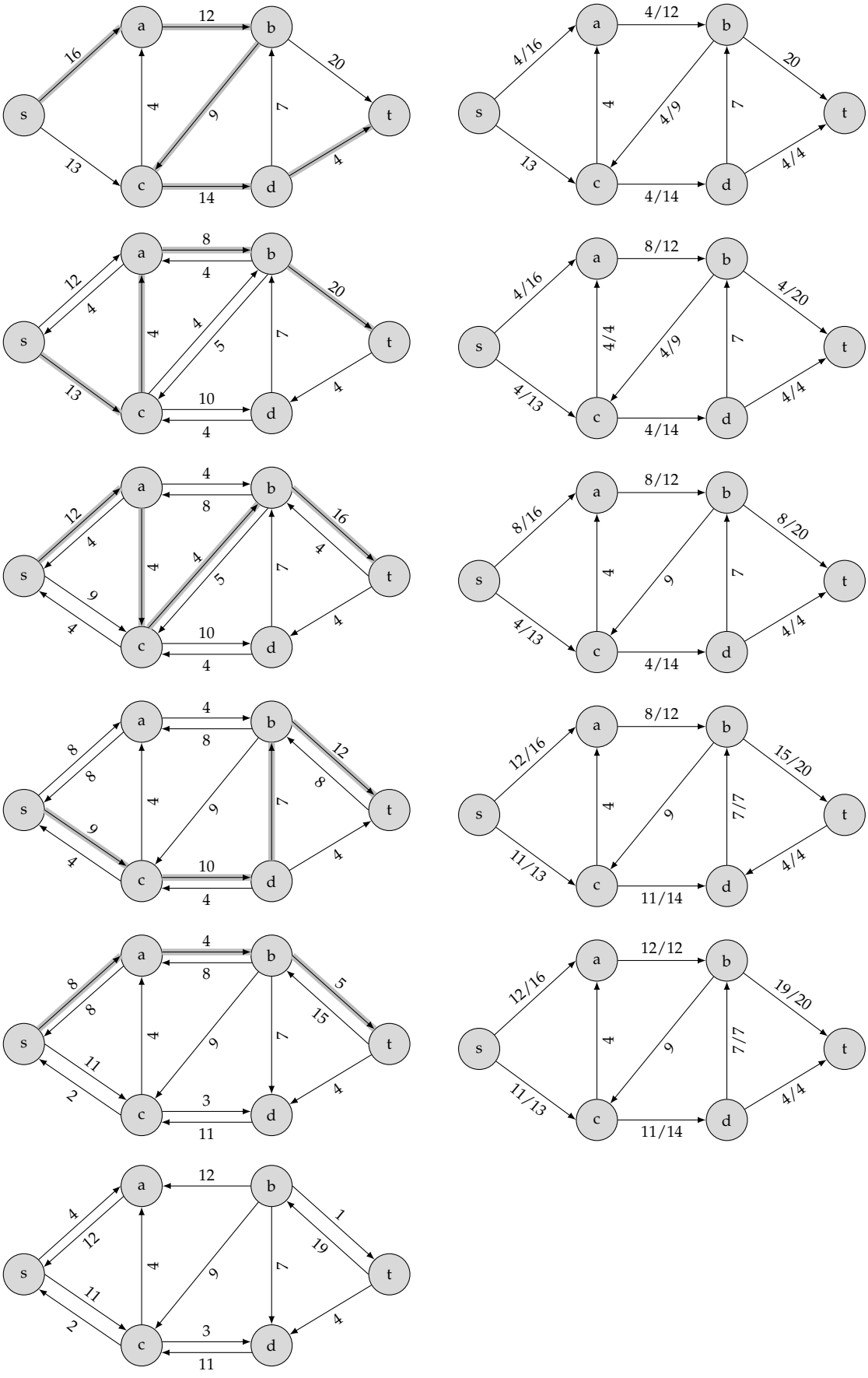


Figura 16.4: L'esecuzione dell'algoritmo di base di Ford-Fulkerson. Il lato sinistro di ciascuna parte mostra la rete residua G_f, il lato destro di ciascuna parte mostra il nuovo flusso f che si ottiene sommando f_p a f.

16.2.5 Analisi dell'algoritmo di Ford-Fulkerson

Se $f*$ indica un flusso massimo nella rete trasformata, allora un'implementazione semplice di FORD-FULKERSON esegue il ciclo **while** al massimo $|f*|$ volte, perché il valore del flusso aumenta di almeno un'unità in ogni iterazione. Il tempo per trovare un cammino in una rete residua è quindi $O(V + E') = O(E)$. Ogni iterazione del ciclo **while** richiede un tempo $O(E)$; ne consegue che il tempo totale di esecuzione di FORD-FULKERSON è $O(E|f*|)$. Se il valore del flusso massimo è piccolo, l'algoritmo FORD-FULKERSON è molto veloce, ma se dovesse iniziare ad essere grande ci impiegherebbe un bel po'.

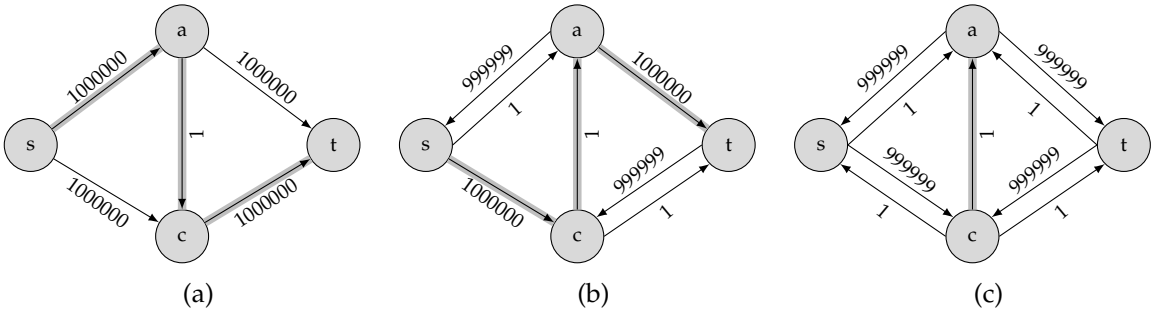


Figura 16.5: **(a)** Una rete di flusso per la quale l'algoritmo FORD-FULKERSON può richiedere un tempo $\Theta(E|f*|)$, dove $f*$ è un flusso massimo. Il camino combreggiato è un cammino aumentante con capacità residua 1. **(b)** La rede residua risultante con un altro cammino aumentante con capacità residua 1. **(c)** La rete residua risultante.

Un flusso massimo in questa rete ha valore 2000000: un milione di unità attraverso il cammino $s \to a \to t$ e un milione di unità di flusso attraverso il cammino $s \to c \to t$. Se il primo cammino aumentante trovato da FORD-FULKERSON è $s \to a \to c \to t$, il flusso ha valore 1 dopo la prima iterazione. Se la seconda iterazione trova il cammino aumentante $s \to c \to a \to t$, il fluss oha valore 2. Possiamo continuare, scegliendo il cammino aumentante $s \to a \to c \to t$ nelle iterazioni dispari e $s \to c \to a \to t$ nelle iterazioni pari. Effettueremo un totale di due milioni di aumenti, con un incremento di una sola unità del valore del flusso in ciascuno aumento.

Algoritmo di Karp

Il limite per la procedura FORD-FULKERSON può essere migliorato se implementiamo il calcolo del cammino aumentante p con una visita in ampiezza, cioè se il cammino aumentante è un *cammino minimo* da s a t nella rete residua, dove ogni arco ha distanza (peso) di valore unitario. L'algoritmo di Karp viene eseguito nel tempo $O(VE^2)$.

Lemma 16.2.1.1. Se l'algoritmo di Karp viene eseguito su una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora per ogni vertice $v \in V - \{s, t\}$, la distanza del cammino minimo $\delta_f(s, v)$ nella rete residua G_f aumenta monotonicamente per ogni aumento di flusso

La dimostrazione è possibile reperirla sul libro.

Teorema 16.2.2. Se l'algoritmo di Karp viene eseguito su una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora il numero totale di aumenti di flusso effettuati dall'algoritmo è $O(VE)$.

Dimostrazione

Diciamo che un arco (u, v) di una rete residua G_f è **critico** in un cammino aumentante p, se la capacità residua di p è la capacità residua di (u, v), cioè se $c_f(p) = c_f(u, v)$. Dimostreremo che ciascuno degli $|E|$ archi può diventare critico al più $|V|/2$ volte. Siano u e v due vertici in V collegati da un arco in E. Quando (u, v) diventa critico per la prima volta si ha

$\delta_f(s, v) = \delta_f(s, u) + 1$

Una volta che il flusso viene aumentato, l'arco (u, v) scompare alla rete residua; non potrà riapparire successivamente in un altro cammino aumentante finché il flusso da u a v non diminuirà, e questo accade soltante se (v, u) appare in un cammino aumentante. Se f' è il flusso in G quando si verifica questo evento, allora si ha

$\delta_{f'}(s, u) = \delta_f(s, u) + 1$

Poiché $\delta_f(s, v) \leq \delta_{f'}(s, v)$, si ha

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 \end{aligned}$$

Di conseguenza, dall'istante in cui (u, v) diventa critico all'istante in cui diventa di nuovo critico, la distanza di u dalla sorgente aumenta di almeno 2 unità. Dopo la prima volta che (u, v) diventa critico esso può ridiventarlo al più altre $(|V| - 2)/2 = |V|/2 - 1$ volte, per un totale di al più $|V|/2$ volte. Poiché ci sono $O(E)$ coppie di vertici che possono essere connessi da un arco in un grafo residuo, il numero totale di archi critici durante l'intera esecuzione dell'algoritmo di Karp è $O(VE)$. Il tempo totale di esecuzione dell'algoritmo di Karp è $O(VE^2)$.

16.3 Abbinamento massimo nei grafi bipartiti

Dato un grafo non orientato $G = (V, E)$, un **abbinamento** è un sottoinsieme di archi $M \subseteq E$ tale che, per ogni vertice v $\in V$, al massimo un arco di M sia incidente su v. Un vertice $v \in V$ è **accoppiato** dall'abbinamento M se qualche arco in M è incidente su v; altrimenti, v è un vertice **non accoppiato**. Un **abbinamento massimo** è un abbinamento con cardinalità massima, cioè un abbinamento M tale che, per qualsiasi abbinamento M', si abbia $|M| \geq |M'|$. Ci concentreremo alla analisi degli abbinamenti massimi nei grafi bipartiti, cioè grafi in cui l'insieme dei vertici può essere partizionato in $V = L \bigcup R$, dove L ed R sono insiemi disgiunti e ogni arco in E collega L ed R.

16.3.1 Trovare un abbinamento massimo nei grafi bipartiti

Definiamo la **rete di flusso associata** $G' = (V', E')$ nel modo seguente. Siano la sorgente s e il pozzo t due nuovi vertici (che non appartengono a V), e poniamo $V' = V \bigcup \{s, t\}$. Se la partizione dei vertici di G è $V = L \bigcup R$, gli archi orientati di G' sono gli archi di E, orientati da L a R, assieme a $|V|$ nuovi archi:

$E' = \{(s, u) : u \in L\} \bigcup \{(u, v) : (u, v) \in E\} \bigcup \{(v, t) : v \in R\}$

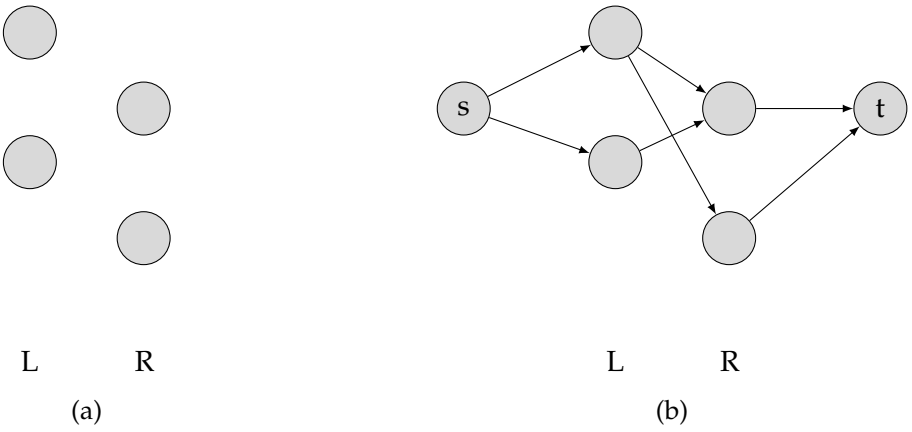


Figura 16.6: Un grafo bipartito $G = (V, E)$ con partizione dei vertici $V = L \bigcup R$. **(c)** La rete di flusso corrispondente G'. Ogni arco ha capacità 1.

Per completare la costruzione, assegnamo una capacità unitaria a ciascun arco in E'. Poiché ogni vertice in V ha almeno un arco incidente, allora $|E| \geq |V|/2$. Quindi $|E| \geq |E'| = |E| + |V| \leq 3|E|$, e dunque $|E'| = \Theta(E)$. Quindi, dato un grafo G non orientato bipartito, possiamo trovare un abbinamento massimo creando la rete di flusso G', eseguendo il metodo di Ford-Fulkerson e ottenendo direttamente un abbinamento massimo. Di conseguenza, possiamo trovare un abbinamento massimo in un grafo bipartito nel tempo $O(VE') = O(VE)$

Algoritmo	Complessità
Breadth-First Search	$O(V+E)$
Deep-First Search	$\Theta(V+E)$
Grafo transposto	$O(V+E)$
Ordinamento topologico	$\Theta(V+E)$
SGG	$O(V+E)$
Kruskal	$O(E \log V)$
Prim	$O(E+\log V)$
Bellman-Ford	$O(VE)$
Directed-Acyclic-Graph Shortest-Path	$\Theta(V+E)$
Djikstra	$O(V \log V + E)$
Faster-all-pairs-shortest-path	$\Theta(n^3)$
Floyd-Warshall	$\Theta(n^3)$
Johnson	$O(V^2 \log V + VE)$

Tabella 16.1: Riassunto algoritmi studiati sui grafi